

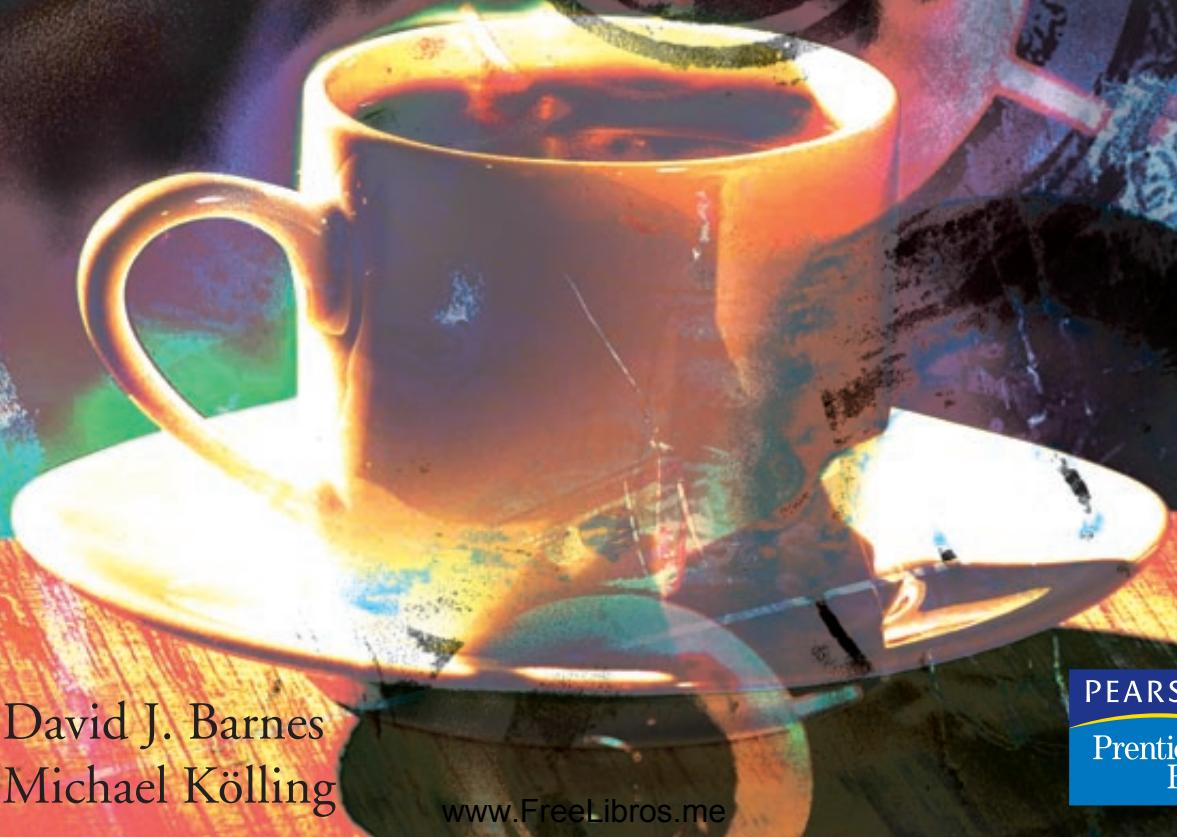


Incluye CD

3<sup>a</sup> edición

# Programación orientada a objetos con Java

Una introducción práctica usando BlueJ

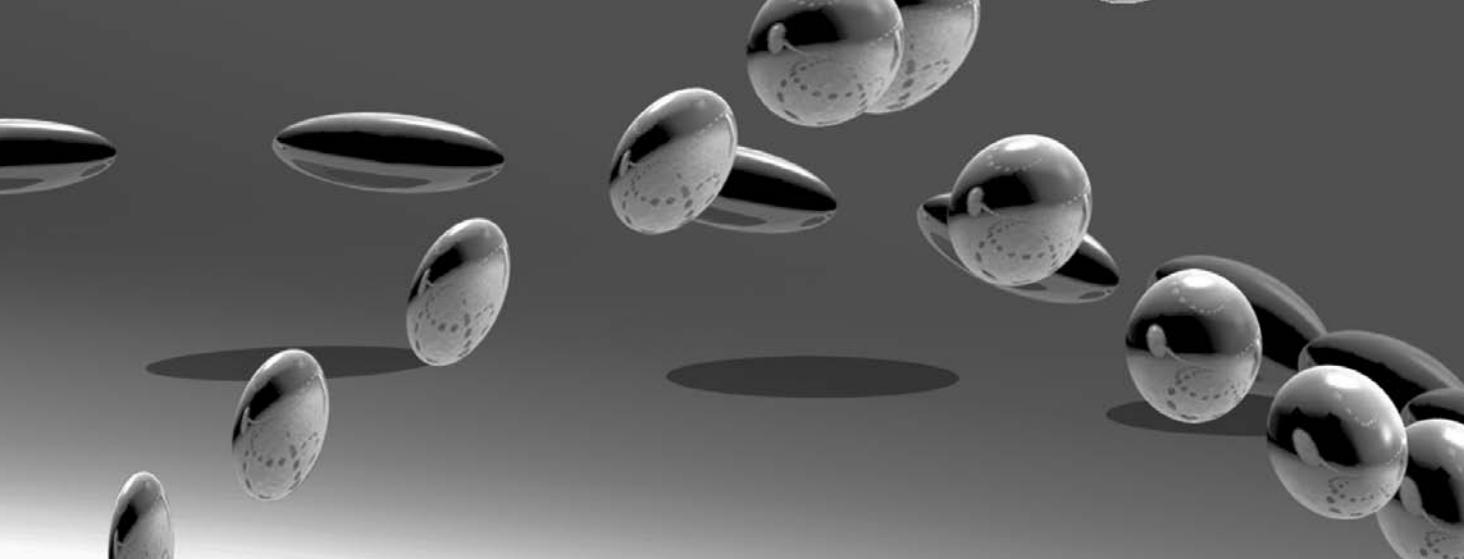


David J. Barnes  
Michael Kölling

[www.FreeLibros.me](http://www.FreeLibros.me)

PEARSON  
Prentice  
Hall





# Programación orientada a objetos con Java

Una introducción práctica usando BlueJ

Tercera edición

**DAVID J. BARNES**

**MICHAEL KÖLLING**

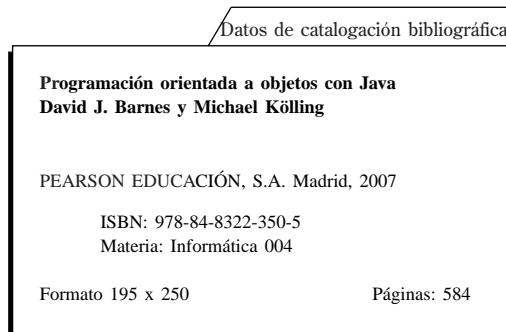
Traducción

**Blanca Irene Brenta**

Departamento de Sistemas Informáticos

Instituto de Tecnología ORT

Argentina



Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra sin contar con autorización de los titulares de propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (*arts. 270 y sgts. Código Penal*).

**DERECHOS RESERVADOS**

© 2007 PEARSON EDUCACIÓN, S.A.  
Ribera del Loira, 28  
28042 Madrid (España)

**PROGRAMACIÓN ORIENTADA A OBJETOS CON JAVA**

**David J. Barnes y Michael Kölling**

**ISBN: 978-84-8322-350-5**

Depósito Legal: M-

This translation of OBJECTS FIRST WITH JAVA A PRACTICAL INTRODUCTION USING BLUEJ 03 Edition is Published by arrangement with Pearson Education Limited, United Kingdom.

©Pearson Education Limited 2003, 2005, 2007

**Equipo editorial:**

Editor: Miguel Martín-Romo

Técnico editorial: Marta Caicoya

**Equipo de producción:**

Director: José Antonio Clares

Técnico: José Antonio Hernán

**Diseño de cubierta:** Equipo de diseño de Pearson Educación, S.A.

**Composición:** COMPOMAR, S.L.

**Impreso por:**

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro ha sido impreso con papel y tintas ecológicos



A mi familia  
Helen, Sarah, Ben, Ana y John  
*db*

A mi familia  
Leah, Sophie y Feena  
*mk*

# Índice de contenido

Prólogo	xix
Prefacio para el profesor	xxi
Proyectos que se discuten en este libro	xxxi
Agradecimientos	xxxv
<b>Parte 1 Fundamentos de programación orientada a objetos</b>	<b>1</b>
Capítulo 1 Objectos y clases	3
1.1 Objetos y Clases	3
1.2 Crear objetos	4
1.3 Invocar métodos	5
1.4 Parámetros	6
1.5 Tipos de dato	7
1.6 Instancias múltiples	8
1.7 Estado	8
1.8 ¿Qué es un objeto?	9
1.9 Interacción entre objetos	10
1.10 Código fuente	11
1.11 Otro ejemplo	13
1.12 Valores de retorno	13
1.13 Objetos como parámetros	14
1.14 Resumen	15
Capítulo 2 Comprender las definiciones de clases	19
2.1 Máquina expendedora de boletos	19
2.1.1 Explorar el comportamiento de una máquina expendedora de boletos ingenua	20
2.2 Examinar una definición de clase	21
2.3 Campos, constructores y métodos	23

2.3.1	Campos	25
2.3.2	Constructores	27
2.4	Pasar datos mediante parámetros	29
2.5	Asignación	30
2.6	Métodos de acceso	31
2.7	Métodos de modificación	33
2.8	Imprimir desde métodos	35
2.9	Resumen de la máquina de boletos simplificada	37
2.10	Reflexión sobre el diseño de la máquina de boletos	38
2.11	Hacer elecciones: la sentencia condicional	39
2.12	Un ejemplo más avanzado de sentencia condicional	43
2.13	Variables locales	44
2.14	Campos, parámetros y variables locales	45
2.15	Resumen de la máquina de boletos mejorada	46
2.16	Ejercicios de revisión	47
2.17	Revisar un ejemplo familiar	48
2.18	Resumen	52
Capítulo 3	Interacción de objetos	57
3.1	El ejemplo reloj	57
3.2	Abstracción y modularización	58
3.3	Abstracción en software	59
3.4	Modularización en el ejemplo reloj	59
3.5	Implementación del visor del reloj	60
3.6	Comparación de diagramas de clases con diagramas de objetos	61
3.7	Tipos primitivos y tipos objeto	64
3.8	El código del VisorDeReloj	64
3.8.1	Clase VisorDeNumeros	64
3.8.2	Concatenación de cadenas	66
3.8.3	El operador módulo	67
3.8.4	La clase VisorDeReloj	68
3.9	Objetos que crean objetos	71
3.10	Constructores múltiples	73
3.11	Llamadas a métodos	73
3.11.1	Llamadas a métodos internos	73
3.11.2	Llamadas a métodos externos	74
3.11.3	Resumen del visor de reloj	75
3.12	Otro ejemplo de interacción de objetos	76
3.12.1	El ejemplo del sistema de correo electrónico	77

3.12.2 La palabra clave <code>this</code>	78
3.13 Usar el depurador	80
3.13.1 Poner puntos de interrupción	80
3.13.2 Paso a paso	82
3.13.3 Entrar en los métodos	82
3.14 Revisión de llamadas a métodos	84
3.15 Resumen	85
 Cáptitulo 4 Agrupar objetos	87
4.1 Agrupar objetos en colecciones de tamaño flexible	87
4.2 Una agenda personal	88
4.3 Una primera visita a las bibliotecas de clases	88
4.3.1 Ejemplo de uso de una biblioteca	89
4.4 Estructuras de objetos con colecciones	91
4.5 Clases genéricas	93
4.6 Numeración dentro de las colecciones	93
4.7 Eliminar un elemento de una colección	94
4.8 Procesar una colección completa	96
4.8.1 El <i>ciclo for-each</i>	97
4.8.2 El <i>ciclo while</i>	98
4.8.3 Recorrer una colección	102
4.8.4 Comparar acceso mediante índices e iteradores	103
4.9 Resumen del ejemplo agenda	103
4.10 Otro ejemplo: un sistema de subastas	104
4.10.1 La clase <code>Lote</code>	105
4.10.2 La clase <code>Subasta</code>	106
4.10.3 Objetos anónimos	109
4.10.4 Usar colecciones	110
4.11 Resumen de colección flexible	112
4.12 Colecciones de tamaño fijo	112
4.12.1 Un analizador de un archivo de registro o archivo «log»	113
4.12.2 Declaración de variables arreglos	115
4.12.3 Creación de objetos arreglo	116
4.12.4 Usar objetos arreglo	118
4.12.5 Analizar el archivo log	118
4.12.6 El <i>ciclo for</i>	119
4.13 Resumen	124

Capítulo 5	Comportamiento más sofisticado	127
5.1	Documentación de las clases de biblioteca	128
5.2	El sistema Soporte Técnico	129
5.2.1	Explorar el sistema Soporte Técnico	129
5.2.2	Lectura de código	131
5.3	Lectura de documentación de clase	135
5.3.1	Comparar interfaz e implementación	136
5.3.2	Usar métodos de clases de biblioteca	137
5.3.3	Comprobar la igualdad de cadenas	139
5.4	Agregar comportamiento aleatorio	139
5.4.1	La clase <code>Random</code>	140
5.4.2	Números aleatorios en un rango limitado	141
5.4.3	Generar respuestas por azar	142
5.4.4	Lectura de documentación de clases parametrizadas	145
5.5	Paquetes y la sentencia <code>import</code>	146
5.6	Usar mapas para las asociaciones	147
5.6.1	Concepto de mapa	148
5.6.2	Usar un <code>HashMap</code>	148
5.6.3	Usar un mapa en el sistema Soporte Técnico	149
5.7	Usar conjuntos	151
5.8	Dividir cadenas	152
5.9	Terminar el sistema de Soporte Técnico	154
5.10	Escribir documentación de clase	156
5.10.1	Usar <code>javadoc</code> en BlueJ	157
5.10.2	Elementos de la documentación de una clase	157
5.11	Comparar público con privado	158
5.11.1	Ocultamiento de la información	159
5.11.2	Métodos privados y campos públicos	160
5.12	Aprender sobre las clases a partir de sus interfaces	161
5.13	Variables de clase y constantes	164
5.13.1	La palabra clave <code>static</code>	164
5.13.2	Constantes	165
5.14	Resumen	166
Capítulo 6	Objetos con buen comportamiento	169
6.1	Introducción	169
6.2	Prueba y depuración	170

6.3	Pruebas de unidad en BlueJ	171
6.3.1	Usar inspectores	175
6.3.2	Pruebas positivas y pruebas negativas	177
6.4	Pruebas automatizadas	178
6.4.1	Prueba de regresión	178
6.4.2	Control automático de los resultados de las pruebas	180
6.4.3	Grabar una prueba	183
6.4.4	Objetos de prueba	185
6.5	Modularización e interfaces	186
6.6	Un escenario de depuración	188
6.7	Comentarios y estilo	189
6.8	Seguimiento manual	190
6.8.1	Un seguimiento de alto nivel	190
6.8.2	Controlar el estado mediante el seguimiento	193
6.8.3	Seguimiento verbal	195
6.9	Sentencias de impresión	195
6.9.1	Activar o desactivar la información de depuración	197
6.10	Elegir una estrategia de prueba	199
6.11	Depuradores	199
6.12	Poner en práctica las técnicas	200
6.13	Resumen	200
Capítulo 7	Diseñar clases	203
7.1	Introducción	204
7.2	Ejemplo del juego <i>world-of-zuul</i>	205
7.3	Introducción al acoplamiento y a la cohesión	207
7.4	Duplicación de código	208
7.5	Hacer extensiones	212
7.5.1	La tarea	212
7.5.2	Encontrar el código relevante	212
7.6	Acoplamiento	214
7.6.1	Usar encapsulamiento para reducir el acoplamiento	215
7.7	Diseño dirigido por responsabilidades	219
7.7.1	Responsabilidades y acoplamiento	219
7.8	Localización de cambios	222
7.9	Acoplamiento implícito	223
7.10	Pensar en futuro	226

7.11	Cohesión	227
7.11.1	Cohesión de métodos	227
7.11.2	Cohesión de clases	228
7.11.3	Cohesión para la legibilidad	229
7.11.4	Cohesión para la reusabilidad	230
7.12	Refactorización	231
7.12.1	Refactorización y prueba	231
7.12.2	Un ejemplo de refactorización	232
7.13	Refactorización para independizarse del idioma	235
7.13.1	Tipos enumerados	236
7.13.2	Más desacoplamiento de la interfaz de comandos	238
7.14	Pautas de diseño	239
7.15	Ejecutar un programa fuera de BlueJ	241
7.15.1	Métodos de clase	241
7.15.2	El método main	242
7.15.3	Limitaciones de los métodos de clase	242
7.16	Resumen	243

## **Parte 2 Estructuras de las aplicaciones** 245

Capítulo 8	Mejorar la estructura mediante herencia	247
8.1	El ejemplo DoME	247
8.1.1	Las clases y los objetos de DoME	248
8.1.2	Código fuente de DoME	251
8.1.3	Discusión de la aplicación DoME	257
8.2	Usar herencia	258
8.3	Jerarquías de herencia	259
8.4	Herencia en Java	260
8.4.1	Herencia y derechos de acceso	261
8.4.2	Herencia e inicialización	262
8.5	DoME: agregar otros tipos de elementos	264
8.6	Ventajas de la herencia (hasta ahora)	266
8.7	Subtipos	266
8.7.1	Subclases y subtipos	268
8.7.2	Subtipos y asignación	268
8.7.3	Subtipos y pasaje de parámetros	270
8.7.4	Variables polimórficas	270

8.7.5 Enmascaramiento de tipos	271
8.8 La clase <code>Object</code>	272
8.9 Autoboxing y clases «envoltorio»	273
8.10 La jerarquía colección	274
8.11 Resumen	275
 Capítulo 9 Algo más sobre herencia	277
9.1 El problema: el método imprimir de DoME	277
9.2 Tipo estático y tipo dinámico	279
9.2.1 Invocar a <code>imprimir</code> desde <code>BaseDeDatos</code>	280
9.3 Sobreescribir	282
9.4 Búsqueda dinámica del método	283
9.5 Llamada a super en métodos	286
9.6 Método polimórfico	287
9.7 Métodos de <code>Object</code> : <code>toString</code>	288
9.8 Acceso protegido	290
9.9 Otro ejemplo de herencia con sobrescritura	292
9.10 Resumen	295
 Capítulo 10 Más técnicas de abstracción	299
10.1 Simulaciones	299
10.2 La simulación de zorros y conejos	300
10.2.1 El proyecto zorros-y-conejos	301
10.2.2 La clase <code>Conejo</code>	303
10.2.3 La clase <code>Zorro</code>	307
10.2.4 La clase <code>Simulador</code> : configuración	310
10.2.5 La clase <code>Simulador</code> : un paso de simulación	314
10.2.6 Camino para mejorar la simulación	316
10.3 Clases abstractas	316
10.3.1 La superclase <code>Animal</code>	316
10.3.2 Métodos abstractos	317
10.3.3 Clases abstractas	320
10.4 Más métodos abstractos	323
10.5 Herencia múltiple	324
10.5.1 La clase <code>Actor</code>	324
10.5.2 Flexibilidad a través de la abstracción	326
10.5.3 Dibujo selectivo	326

10.5.4	Actores dibujables: herencia múltiple	327
10.6	Interfaces	327
10.6.1	La interfaz Actor	327
10.6.2	Herencia múltiple de interfaces	329
10.6.3	Interfaces como tipos	330
10.6.4	Interfaces como especificaciones	331
10.6.5	Otro ejemplo de interfaces	332
10.6.6	¿Clase abstracta o interfaz?	333
10.7	Resumen de herencia	334
10.8	Resumen	334
Capítulo 11	Construir interfaces gráficas de usuario	337
11.1	Introducción	337
11.2	Componentes, gestores de disposición y captura de eventos	338
11.3	AWT y Swing	339
11.4	El ejemplo Visor de Imágenes	339
11.4.1	Primeros experimentos: crear una ventana	340
11.4.2	Agregar componentes simples	342
11.4.3	Agregar menús	344
11.4.4	Manejo de eventos	345
11.4.5	Recepción centralizada de eventos	345
11.4.6	Clases internas	348
11.4.7	Clases internas anónimas	350
11.5	Visor de Imágenes 1.0: primera versión completa	352
11.5.1	Clases para procesar imágenes	352
11.5.2	Agregar la imagen	353
11.5.3	Esquemas de disposición	355
11.5.4	Contenedores anidados	358
11.5.5	Filtros de imagen	360
11.5.6	Diálogos	363
11.6	Visor de Imágenes 2.0: mejorar la estructura del programa	365
11.7	Visor de Imágenes 3.0: más componentes de interfaz	370
11.7.1	Botones	370
11.7.2	Bordes	373
11.8	Otras extensiones	374
11.9	Otro ejemplo: reproductor de sonido	376
11.10	Resumen	379

Capítulo 12	Manejo de errores	383
12.1	El proyecto <i>libreta-de-direcciones</i>	384
12.2	Programación defensiva	389
12.2.1	Interacción cliente-servidor	389
12.2.2	Validar argumentos	390
12.3	Informar de errores del servidor	392
12.3.1	Notificar al usuario	392
12.3.2	Notificar al objeto cliente	393
12.4	Principios del lanzamiento de excepciones	396
12.4.1	Lanzar una excepción	396
12.4.2	Clases Exception	397
12.4.3	El efecto de una excepción	399
12.4.4	Excepciones no comprobadas	399
12.4.5	Impedir la creación de un objeto	401
12.5	Manejo de excepciones	402
12.5.1	Excepciones comprobadas: la cláusula <code>throws</code>	402
12.5.2	Captura de excepciones: la sentencia <code>try</code>	403
12.5.3	Lanzar y capturar varias excepciones	405
12.5.4	Propagar una excepción	407
12.5.5	La cláusula <code>finally</code>	407
12.6	Definir nuevas clases de excepción	408
12.7	Usar aserciones	410
12.7.1	Controlar la consistencia interna	410
12.7.2	La sentencia <code>assert</code>	410
12.7.3	Pautas para usar aserciones	412
12.7.4	Aserciones y el marco de trabajo de unidades de prueba de BlueJ	413
12.8	Recuperarse del error y anularlo	414
12.8.1	Recuperarse del error	414
12.8.2	Anular el error	415
12.9	Estudio de caso: entrada/salida de texto	417
12.9.1	Lectores, escritores y flujos	417
12.9.2	El proyecto <i>libreta-de-direcciones-io</i>	418
12.9.3	Salida de texto con <code>FileWrite</code>	421
12.9.4	Entrada de texto con <code>FileReader</code>	422
12.9.5	Scanner: leer entradas desde la terminal	423
12.9.6	Serialización de objetos	424
12.10	Resumen	425

Capítulo 13	Diseñar aplicaciones	427
13.1	Análisis y diseño	427
13.1.1	El método verbo/sustantivo	428
13.1.2	El ejemplo de reserva de entradas para el cine	428
13.1.3	Descubrir clases	429
13.1.4	Usar tarjetas CRC	430
13.1.5	Escenarios	430
13.2	Diseño de clases	434
13.2.1	Diseñar interfaces de clases	434
13.2.2	Diseño de la interfaz de usuario	436
13.3	Documentación	436
13.4	Cooperación	437
13.5	Prototipos	437
13.6	Crecimiento del software	438
13.6.1	Modelo de cascada	438
13.6.2	Desarrollo iterativo	439
13.7	Usar patrones de diseño	440
13.7.1	Estructura de un patrón	441
13.7.2	Decorador	442
13.7.3	Singleton	442
13.7.4	Método Fábrica	443
13.7.5	Observador	444
13.7.6	Resumen de patrones	445
13.8	Resumen	446
Cápítulo 14	Un estudio de caso	449
14.1	El estudio de caso	449
14.1.1	Descripción del problema	449
14.2	Análisis y diseño	450
14.2.1	Descubrir clases	450
14.2.2	Usar tarjetas CRC	451
14.2.3	Escenarios	452
14.3	Diseño de clases	454
14.3.1	Diseñar las interfaces de las clases	454
14.3.2	Colaboradores	455
14.3.3	El esquema de implementación	455
14.3.4	Prueba	460

14.3.5 Algunos asuntos pendientes	460
14.4 Desarrollo iterativo	460
14.4.1 Pasos del desarrollo	460
14.4.2 La primer etapa	462
14.4.3 Probar la primera etapa	466
14.4.4 Una etapa de desarrollo más avanzada	466
14.4.5 Más ideas para desarrollar	468
14.4.6 Reusabilidad	469
14.5 Otro ejemplo	469
14.6 Para ir más lejos	469
 Apéndices	471
A Trabajar con un proyecto BlueJ	471
B Tipos de dato en Java	473
C Estructuras de control Java	477
D Operadores	483
E Ejecutar Java fuera del entorno BlueJ	487
F Configurar BlueJ	491
G Usar el depurador	493
H Herramienta JUnit de pruebas unitarias	497
I El documentador de Java: javadoc	499
J Guía de estilo de programación	503
K Clases importantes de la biblioteca de Java	507
L Tabla de conversión de términos que aparecen en el CD	511
 Índice analítico	531





# Prólogo

## Por James Gosling, Sun Microsystems

Ver a mi hija Kate y a sus compañeros de escuela, esforzarse para seguir un curso de Java que utilizaba un IDE comercial, fue una experiencia dolorosa. La sofisticación de la herramienta agregaba una considerable complejidad al aprendizaje. Desearía haber comprendido antes lo que estaba ocurriendo. Tal como estaban las cosas, no pude hablar con el instructor sobre el problema hasta que fue demasiado tarde. Este es exactamente el tipo de situación a la que BlueJ se ajusta perfectamente.

BlueJ es un entorno de desarrollo interactivo con una misión: está diseñado para que lo utilicen estudiantes que están aprendiendo a programar. Fue diseñado por instructores que se enfrentaron con este problema en el aula todos los días. Ha sido esclarecedor hablar con la gente que desarrolló BlueJ: tienen una idea muy clara de quienes son sus destinatarios. Las discusiones tendieron a centralizarse más sobre qué omitir, que sobre qué introducir. BlueJ es muy limpio y muy didáctico.

Pese a todo, este no es un libro sobre BlueJ sino sobre programación, y en especial, sobre programación en Java.

En los últimos años, Java ha sido ampliamente usado en la enseñanza de programación y esto se debe a varios motivos. Uno de ellos es que Java tiene muchas características que facilitan la enseñanza: tiene una definición relativamente clara, el compilador realiza extensos análisis estadísticos fáciles de enseñar y tiene un modelo de memoria muy robusto que elimina la mayoría de los errores “misteriosos” que surgen cuando se comprometen los límites de los objetos o el sistema de tipos. Otro motivo es que Java se ha vuelto muy importante comercialmente.

Este libro afronta el concepto más difícil de enseñar: los objetos. Conduce a los estudiantes por un camino que va desde los primeros pasos hasta algunos conceptos muy sofisticados.

Se las arregla para resolver una de las cuestiones más escabrosas al escribir un libro de programación: cómo transmitir la mecánica real de escribir y ejecutar un programa. La mayoría de los libros pasan por alto silenciosamente esta cuestión, o la abordan ligeramente dejando en manos del instructor la forma de solucionar este problema y, de esta manera, lo dejan con la carga de relacionar el material que se enseñará con los pasos que los estudiantes deberán dar para trabajar con los ejercicios. En lugar de seguir este camino, este libro asume el uso de BlueJ y es capaz de integrar las tareas de comprensión de los conceptos con los mecanismos que pueden emplear los estudiantes para explorarlos.

Desearía que este libro hubiera estado al alcance de mi hija el año pasado, tal vez el próximo año...



# Prefacio para el profesor

Este libro es una introducción a la programación orientada a objetos destinada a principiantes. El foco principal del libro es la programación orientada a objetos en general y los conceptos de programación desde la perspectiva de la ingeniería del software.

Los primeros capítulos fueron escritos para estudiantes que no tienen ninguna experiencia en programación, pero los capítulos restantes también se adaptan para estudiantes avanzados o para programadores profesionales. En particular, los programadores que tienen experiencia en lenguajes de programación no orientados a objetos que deseen migrar sus habilidades a la orientación a objetos, también pueden obtener beneficios de este libro.

A lo largo del libro usamos dos herramientas para permitir que se pongan en práctica los conceptos que se presentan: el lenguaje de programación Java y el entorno de desarrollo en Java, BlueJ.

## Java

Se eligió Java porque combina estos dos aspectos: el diseño del lenguaje y su popularidad. El lenguaje de programación Java ofrece en sí una implementación muy limpia de los conceptos más importantes de la orientación a objetos y funciona bien en la enseñanza, como lenguaje introductorio. Su popularidad asegura una inmensa fuente de recursos de apoyo.

En cualquier asignatura, es muy útil contar con una variedad de fuentes de información disponibles tanto para los profesores como para los estudiantes. En particular, para Java, existen innumerables libros, tutoriales, ejercicios, compiladores, entornos y exámenes, de muy diferentes tipos y estilos; muchos de ellos están disponibles online y muchos son gratuitos. La enorme cantidad y la buena calidad del material de apoyo, hace que Java sea una excelente elección para la introducción a la programación orientada a objetos.

Con tanta cantidad de material disponible, ¿hay lugar para decir algo más sobre Java? Pensamos que sí, y la segunda herramienta que usamos es una de las razones...

## BlueJ

La segunda herramienta, BlueJ, merece más comentarios. Este libro es único en cuanto al uso completamente integrado del entorno BlueJ.

BlueJ es un entorno de desarrollo en Java que está siendo desarrollado y mantenido por la University of Southern de Dinamarca, la Deakin University en Australia y la University of Kent en Canterbury, Reino Unido, explícitamente como un entorno para la introducción a la enseñanza de programación orientada a objetos. BlueJ se adapta mejor que otros entornos a la enseñanza introductoria por diversos motivos:

- La interfaz de usuario es sumamente simple. Generalmente, los estudiantes principiantes pueden usar el entorno BlueJ de manera competente después de 20 minutos de introducción. A partir de ahí, la enseñanza se puede concentrar en los conceptos importantes, orientación a objetos y Java, y no es necesario desperdiciar tiempo explicando entornos, sistemas de archivos, rutas de clases, comandos DOS o conflictos con las DLL.
- El entorno cuenta con importantes herramientas de enseñanza que no se disponen en otros entornos. Una de ellas es la visualización de la estructura de las clases. BlueJ muestra automáticamente un diagrama del estilo UML que representa las clases de un proyecto y sus relaciones. La visualización de estos importantes conceptos es una gran ayuda tanto para los profesores como para los estudiantes. ¡Resulta bastante difícil aprehender el concepto de un objeto cuando todo lo que se ve en la pantalla son líneas de código! La notación que se emplea en estos diagramas es un subconjunto simplificado de UML, adaptado a las necesidades de los principiantes, lo que facilita su comprensión, pero también permite migrar al UML completo en una etapa posterior.
- Una de las fortalezas más importantes del entorno BlueJ es que habilita al usuario a crear directamente objetos de cualquier clase y luego interactuar con sus métodos. Esta característica brinda la oportunidad de experimentar de manera directa con los objetos, restando énfasis al entorno. Los estudiantes prácticamente pueden “sentir” lo que significa crear un objeto, invocar un método, pasar un parámetro o recibir un valor de retorno. Pueden probar un método inmediatamente después de haberlo escrito, sin necesidad de escribir código de prueba. Esta facilidad es un objetivo invaluable para la comprensión de los conceptos subyacentes y de los detalles del lenguaje.

BlueJ es un entorno Java completo. No se trata de una versión de Java simplificada o recortada con fines de enseñanza. Se ejecuta sobre el entorno de desarrollo de Java de Sun Microsystems (Java Development Kit) y utiliza el compilador estándar y la máquina virtual. Esto asegura que siempre cumple con la especificación oficial y más actualizada de Java.

Los autores de este libro tienen varios años de experiencia en la enseñanza mediante el entorno BlueJ (y muchos otros años sin este entorno). Ambos hemos experimentado la forma en que BlueJ aumenta el compromiso, la comprensión y la actividad de los estudiantes en nuestros cursos. Uno de los autores también es desarrollador del sistema BlueJ.

## Primero los objetos

Uno de los motivos para seleccionar BlueJ es que permite un abordaje en el que los profesores verdaderamente manejan los conceptos importantes desde el principio. Cómo hacer para comenzar realmente con los objetos ha sido una lamentable batalla para

muchos autores de libros de texto y para algunos profesores durante un tiempo. Desafortunadamente, el lenguaje Java no cumple muy fácilmente con este noble objetivo. Se deben atravesar numerosos temas de sintaxis y detalles antes de que se produzca la primer experiencia de dar vida a un objeto. El menor programa Java que crea e invoca un objeto, incluye típicamente:

- escribir una clase,
- escribir un método “main” que incluye en su firma conceptos tales como métodos estáticos, parámetros y arreglos,
- una sentencia para crear el objeto (“new”),
- una asignación a una variable,
- la declaración de una variable que incluye su tipo,
- una llamada a método que utiliza la notación de punto
- y posiblemente, una lista de parámetros.

Como resultado, los libros de texto generalmente:

- tienen que seguir un camino que atraviesa esta prohibitiva lista y sólo llegan a los objetos aproximadamente en el Capítulo 4, o
- usan un programa del estilo “Hola mundo” con un método main estático y simple como primer ejemplo, pero en el que no se crea ningún objeto.

Con BlueJ, esto no es un problema. ¡Un estudiante puede crear un objeto e invocar sus métodos en su primera actividad! Dado que los usuarios pueden crear e interactuar directamente con los objetos, los conceptos tales como clases, objetos, métodos y parámetros se pueden discutir fácilmente de una manera concreta antes de ver la primera línea en la sintaxis de Java. En lugar de explicar más sobre este punto, sugerimos que el lector curioso se sumerja en el Capítulo 1, y luego las cosas se aclararán rápidamente.

## Un abordaje iterativo

Otro aspecto importante de este libro es que sigue un estilo iterativo. En la comunidad de educación en computación existe un patrón de diseño educativo muy conocido que establece que los conceptos importantes se deben enseñar temprana y frecuentemente.<sup>1</sup> Es muy tentador para los autores de libros de texto tratar y decir absolutamente todo lo relacionado con un tema, en el momento en que se lo introduce. Por ejemplo, es común cuando se introducen los tipos, que se de una lista completa de los tipos de datos que existen, o que se discutan todas las clases de ciclos que existen cuando se introduce el concepto de ciclo.

Estos dos abordajes entran en conflicto: no nos podemos concentrar en discutir primero los conceptos importantes y al mismo tiempo proporcionar una cobertura completa de todos los temas que se encuentran. Nuestra experiencia con los libros de texto

---

<sup>1</sup> El patrón “Early Bird”, en J. Bergin: “Fourteen pedagogical patterns for teaching computer science”, *Proceedings of the Fifth European Conference on Pattern Languages of Programs* (EuroPLop 2000), Irsee, Germany, Julio 2000.

es que la gran cantidad de detalle inicialmente provoca distracción y tiene el efecto de ahogar los temas importantes, por lo que resultan más difíciles de comprender.

En este libro tocamos todos los temas importantes varias veces, ya sea dentro de un mismo capítulo o a lo largo de diferentes capítulos. Los conceptos se introducen generalmente con el nivel de detalle necesario para su comprensión y para su aplicación en tareas concretas. Más tarde se revisitan en un contexto diferente y la comprensión se profundiza a medida que el lector recorre los capítulos. Este abordaje también ayuda a tratar con la frecuente ocurrencia de dependencias mutuas entre los conceptos.

Algunos profesores puede que no estén familiarizados con el abordaje iterativo. Recorriendo los primeros capítulos, los profesores acostumbrados a una introducción más secuencial puede que se sorprendan ante la cantidad de conceptos que se abordan tempranamente. Esto podría parecer una curva de aprendizaje muy empinada.

Es importante comprender que este no es el final de la historia. No se espera que los estudiantes comprendan inmediatamente cada uno de estos conceptos. En cambio, estos conceptos fundamentales se revisitarán nuevamente a lo largo del libro, permitiendo que los estudiantes obtengan cada vez una comprensión más profunda. Dado que su nivel de conocimientos cambia a medida que avanzan, el revisitar luego los temas importantes les permite obtener una comprensión más profunda y más general.

Hemos probado este abordaje con estudiantes varias veces. Pareciera que los estudiantes tienen menos problemas con este abordaje que algunos profesores de larga data. Y recuerde que ¡una curva de aprendizaje empinada no es un problema siempre y cuando se asegure de que sus alumnos puedan escalarla!

## Cobertura incompleta del lenguaje

En relación con nuestro abordaje iterativo está la decisión de no intentar ofrecer una cobertura completa del lenguaje Java dentro del libro.

El foco principal de este libro es transmitir los principios generales de la programación orientada a objetos, no los detalles del lenguaje Java en particular. Los estudiantes que utilicen este libro podrían trabajar como profesionales del software en los próximos 30 o 40 años de sus vidas, por lo que es prácticamente seguro que la mayor parte de sus trabajos no será en Java. Cada libro de texto serio por supuesto que debe intentar prepararlos para algo más importante que el lenguaje de moda del momento.

Por otra parte, son importantes muchos detalles de Java para realizar realmente el trabajo práctico. En este libro cubrimos las construcciones Java con tanto detalle como sea necesario para ilustrar los conceptos que se intentan transmitir y para que puedan implementar el trabajo práctico. Algunas construcciones específicas de Java han sido deliberadamente dejadas fuera del tratamiento.

Somos conscientes de que algunos instructores elegirán trabajar algunos temas que no discutimos detalladamente. Esto es esperable y necesario. Sin embargo, en lugar de tratar de cubrir cada tema posible (y por lo tanto, aumentar el tamaño de este libro a

unas 1500 páginas), trabajamos usando *ganchos*. Estos ganchos son indicadores, con frecuencia bajo la forma de preguntas que disparan el tema y que ofrecen referencias al apéndice o a material externo. Estos ganchos aseguran que se plantee un tema relevante en el momento adecuado y se deja al lector o al profesor la decisión del nivel de detalle con que se tratará el tema. De esta manera, los ganchos funcionan como recordatorios de la existencia del tema y como acomodadores que indican un punto en la secuencia donde puede insertarse.

Los profesores pueden individualmente decidir utilizar el libro de esta manera, siguiendo la secuencia que sugerimos, o ramificarse siguiendo los caminos sugeridos por los ganchos del texto.

Los capítulos también incluyen a menudo varias preguntas relacionadas con el tema, que hacen pensar en el material de la discusión, pero que no se tratan en este libro. Esperamos que los profesores discutan algunas de estas preguntas en clase o que los estudiantes investiguen las respuestas a modo de ejercicios.

## Abordaje por proyectos

La presentación del material en el libro está dirigido por proyectos. El libro discute numerosos proyectos de programación y proporciona cantidad de ejercicios. En lugar de introducir una nueva construcción y luego proporcionar ejercicios de aplicación de esta construcción para resolver una tarea, ofrecemos primero un objetivo y un problema. El análisis del problema determina los tipos de solución que se necesitan. En consecuencia, las construcciones del lenguaje se introducen a medida que se las necesita para resolver los problemas que se presentan.

Al diseñar este libro hemos tratado de usar un buen número y una amplia variedad de proyectos de ejemplo diferentes. Esperamos que esto sirva para capturar el interés del lector, pero también ayuda a ilustrar la variedad de contextos diferentes en los que se pueden aplicar los conceptos. Es difícil encontrar buenos proyectos de ejemplo. Esperamos que nuestros proyectos sirvan para ofrecer a los profesores buenos puntos de comienzo y varias ideas para una amplia variedad de tareas interesantes.

La implementación de todos nuestros proyectos se escribió muy cuidadosamente, de modo que muchas cuestiones periféricas puedan estudiarse leyendo el código fuente de los proyectos. Creemos firmemente en los beneficios de aprender mediante la lectura y la imitación de buenos ejemplos. Sin embargo, para que esto funcione, uno debe asegurarse de que los ejemplos que leen los estudiantes estén bien escritos y sean valiosos de imitar. Hemos tratado de hacerlos de esta manera.

Todos los proyectos se diseñaron como problemas abiertos. Mientras se discuten en detalle una o más versiones de cada problema en el libro, los proyectos están diseñados de modo que los estudiantes puedan agregarles extensiones y mejoras. Se incluye el código completo de todos los proyectos.

## Secuencia de conceptos en lugar de construcciones del lenguaje

Otro aspecto que diferencia este libro de muchos otros es que está estructurado en base a las tareas fundamentales para el desarrollo de software y no necesariamente concuerdan con construcciones particulares del lenguaje Java. Un indicador de esto es el título de los capítulos. En este libro no encontrará muchos de los títulos tradicionales de capítulos tales como “Tipos de dato primitivos” o “Estructuras de control”. El que se estructure alrededor de las tareas fundamentales del desarrollo nos permite ofrecer una introducción mucho más general que no está dirigida por las complejidades del lenguaje de programación utilizado en particular. También creemos que facilita que los estudiantes continúen motivados por la introducción y esto hace que la lectura sea mucho más interesante.

Como resultado de este abordaje, es poco probable que se utilice este libro como un libro de referencia. Los libros de texto introductorios y los libros de referencia tienen objetivos que compiten parcialmente. Hasta cierto punto, un libro puede intentar ser de texto y de referencia al mismo tiempo, pero este compromiso se puede cumplir hasta cierto punto. Nuestro libro está claramente diseñado como un libro de texto y si se presentara un conflicto, el estilo de un libro de texto prevalecerá sobre su uso como libro de referencia.

Sin embargo, proporcionamos apoyo suficiente como para que se lo utilice como libro de referencia enumerando las construcciones de Java que se introducen en cada capítulo en la introducción del mismo.

## Secuencia de capítulos

El Capítulo 1 presenta los conceptos más fundamentales de la orientación a objetos: objetos, clases y métodos. Ofrece una introducción sólida y práctica de estos conceptos sin entrar en los detalles de la sintaxis de Java. También brinda una primer mirada al código. Lo hacemos usando un ejemplo de figuras gráficas que se pueden dibujar interactivamente y un segundo ejemplo de un sistema sencillo de matriculación a un curso laboratorio.

El Capítulo 2 descubre las definiciones de las clases e investiga cómo se escribe código Java para crear el comportamiento de los objetos. Discutimos sobre cómo se definen los campos y cómo se implementan los métodos. En este capítulo también introducimos los primeros tipos de sentencias. El ejemplo principal es la implementación de una máquina de boletos. También retomamos el ejemplo del curso de laboratorio del Capítulo 1 para investigarlo un poco más profundamente.

El Capítulo 3 amplía el panorama al discutir la interacción entre varios objetos. Vemos cómo pueden colaborar los objetos invocando métodos de los otros objetos para llevar a cabo una tarea en común. También discutimos sobre cómo un objeto puede crear otros objetos. Se discute el ejemplo de un reloj digital que utiliza dos objetos visores de números para mostrar las horas y los minutos. Como segundo ejemplo principal del capítulo, examinamos una simulación de un sistema de correo electrónico en el que se pueden enviar mensajes entre los clientes del correo.

En el Capítulo 4 continuamos con la construcción de estructuras de objetos más extensas. Lo más importante es que comenzamos con la utilización de colecciones de objetos. Implementamos una agenda electrónica y un sistema de subastas para introducir las colecciones. Al mismo tiempo tratamos el tema del recorrido de las colecciones y damos una primer mirada a los ciclos. La primer colección que se usa es un `ArrayList`. En la segunda mitad del capítulo introducimos los arreglos como una forma especial de colección y el ciclo `for` como otra forma de ciclo. Discutimos la implementación de un analizador de un registro de conexión a la web como ejemplo para utilizar los arreglos.

El Capítulo 5 se ocupa de las bibliotecas y de las interfaces. Presentamos la biblioteca estándar de Java y discutimos algunas de sus clases más importantes. El punto principal es que explicamos cómo leer y comprender la documentación de la biblioteca. Se discute la importancia de la escritura de la documentación en los proyectos de desarrollo de software y finalizamos practicando cómo se escribe la documentación de nuestras propias clases. `Random`, `Set` y `Map` son ejemplos de las clases que encontramos en este capítulo. Implementamos un sistema de diálogos del estilo *Eliza* y una simulación gráfica del rebote de una pelota para aplicar estas clases.

El Capítulo 6 titulado *Objetos con buen comportamiento* se ocupa de un grupo de cuestiones conectadas con la producción de clases correctas, comprensibles y mantenibles. Cubre cuestiones tales como la escritura de código claro y legible de probar y de depurar que incluyen el estilo y los comentarios. Se introducen las estrategias de prueba y se discuten detalladamente varios métodos de depuración. Usamos el ejemplo de una agenda diaria y la implementación de una calculadora electrónica para discutir estos temas.

En el Capítulo 7 discutimos más formalmente las cuestiones vinculadas con dividir el dominio de un problema en clases con el objetivo de su implementación. Introducimos cuestiones relacionadas con el diseño de clases de buena calidad que incluyen conceptos tales como diseño dirigido por responsabilidades, acoplamiento, cohesión y refactorización. Para esta discusión se usa un juego de aventuras interactivo, basado en texto (*World of Zuul*). Modificamos y ampliamos la estructura interna de las clases del juego mediante un proceso iterativo y finalizamos con una lista de propuestas para que los estudiantes puedan extenderlo como proyectos de trabajo.

En los Capítulo 8 y 9 introducimos herencia y polimorfismo y varios de los detalles que se relacionan con la problemática de estos temas. Discutimos una base de datos sencilla que almacena CD y DVD para ilustrar estos conceptos. Se discuten detalladamente cuestiones tales como el código de la herencia, el subtipoado, la invocación a métodos polimórficos y la sobrescritura.

En el Capítulo 10 implementamos una simulación del modelo predador-presa que sirve para discutir los mecanismos adicionales de abstracción basados en herencia, denominados interfaces y clases abstractas.

El Capítulo 11 presenta dos nuevos ejemplos: un visor de imágenes y un reproductor de sonido. Ambos ejemplos sirven para discutir cómo se construyen las interfaces gráficas de usuario (IGU).

Luego, el Capítulo 12 toma la difícil cuestión del tratamiento de los errores. Se discuten varios problemas y varias soluciones posibles y más detalladamente, el mecanismo de excepciones de Java. Extendemos y mejoramos una libreta de direcciones para ilustrar estos conceptos. Se usa el problema de la entrada y salida de texto como caso de estudio de los errores que se producen.

El Capítulo 13 retoma la discusión más detalladamente del siguiente nivel de abstracción: cómo estructurar en clases y métodos un problema descrito vagamente. En los capítulos anteriores hemos asumido que ya existe una gran parte de las aplicaciones y por lo tanto, realizamos mejoras. Ahora es el momento de discutir cómo comenzar a partir de cero. Esto involucra una discusión detallada sobre cuáles son las clases que debe implementar nuestra aplicación, cómo interactúan y cómo se deben distribuir las responsabilidades. Usamos tarjetas clase/responsabilidades/collaboradores (CRC) para abordar este problema, mientras diseñamos un sistema de reserva de entradas para el cine.

En el Capítulo 14 tratamos de reunir e integrar varios de los temas de los capítulos precedentes del libro. Es un estudio de caso completo que comienza con el diseño de la aplicación, pasa por el diseño de las interfaces de las clases, pasando a discutir detalladamente varias de las características importantes, funcionales y no funcionales. Los temas tratados en los capítulos anteriores (tales como legibilidad, estructuras de datos, diseño de clases, prueba y extensibilidad) se aplican en un nuevo contexto.

### Tercera edición

Esta es la tercera edición de este libro. Se han modificado varias cosas de las versiones anteriores. En la segunda edición se agregó la introducción al JUnit y un capítulo sobre programación de IGU. En esta edición, el cambio más obvio es el uso de Java 5 como lenguaje de implementación. Java 5 introduce nuevas construcciones tales como clases genéricas y tipos enumerados y se cambió casi todo el código de nuestros ejemplos para que utilicen estas nuevas características. También se rescribieron las discusiones en el libro para tenerlas en cuenta, sin embargo, el concepto y el estilo en general de este libro continúa intacto.

La retroalimentación que hemos recibido de los lectores de las ediciones anteriores fue altamente positiva y muchas personas colaboraron en mejorar este libro enviando sus comentarios y sugerencias, encontrando errores y advirtiéndonos sobre ellos, contribuyeron con material para el sitio web del libro, contribuyeron en las discusiones en la lista de correo o en la traducción del libro en diversos idiomas.

Sin embargo, el libro parece estar “funcionando”, de modo que esta tercera edición es un intento de mejorar manteniendo el mismo estilo y no de producir un cambio radical.

### Material adicional

Este libro incluye en un CD todos los proyectos que se usan como ejemplos y ejercicios. El CD también incluye el entorno de desarrollo Java (JDK) y el entorno BlueJ para varios sistemas operativos.

Existe un sitio web de apoyo a este libro en

<http://www.bluej.org/objects-first>

En este sitio se pueden encontrar ejemplos actualizados y proporciona material adicional. Por ejemplo, la guía de estilo de todos los ejemplos del libro está disponible en el sitio web en formato electrónico de modo que los instructores puedan modificarla y adaptarla a sus propios requerimientos.

El sitio web también incluye una sección exclusiva para profesores, protegida por contraseña, que ofrece material adicional.

Se proporciona un conjunto de diapositivas para dar un curso que tenga a este libro como soporte.

## Grupos de discusión

Los autores mantienen dos grupos de discusión por correo electrónico con el propósito de facilitar el intercambio de ideas y el apoyo mutuo entre los lectores de este libro y otros usuarios de BlueJ.

La primera lista, bluej-discuss, es pública (cualquiera se puede suscribir) y tiene un archivo público. Para suscribirse o para leer los archivos, dirigirse a:

<http://lists.bluej.org/mailman/listinfo/bluej-discuss>

La segunda lista, objects-first, es una lista exclusiva para profesores. Se puede utilizar para discutir soluciones, enseñar trucos, exámenes y otras cuestiones relacionadas con la enseñanza. Para obtener instrucciones sobre cómo suscribirse, por favor, diríjase al sitio web del libro.





# Proyectos que se discuten en este libro

## Capítulo 1

*figuras*

Dibuja algunas figuras geométricas sencillas; ilustra la creación de objetos, la invocación de métodos y los parámetros.

## Capítulo 1

*cuadro*

Un ejemplo que usa objetos del proyecto *figuras* para dibujar un cuadro; introduce código fuente, sintaxis de Java y compilación.

## Capítulo 1, Capítulo 2, Capítulo 8

*curso-de-laboratorio*

Un ejemplo sencillo de cursos para estudiantes; ilustra objetos, campos y métodos. Se utiliza nuevamente en el Capítulo 8 para agregar herencia.

## Capítulo 2

*maquina-de-boletos*

La simulación de una máquina que vende boletos para el tren; presenta más detalles sobre campos, constructores, métodos de acceso y de modificación, parámetros y algunas sentencias sencillas.

## Capítulo 2

*agenda*

Almacena los detalles de una agenda. Refuerza las construcciones utilizadas en el ejemplo de la máquina de boletos.

## Capítulo 3

*visor-de-reloj*

La implementación del visor de un reloj digital; ilustra los conceptos de abstracción, modularización y la interacción de objetos.

## Capítulo 3

*sistema-de-correo*

Una simulación sencilla de un sistema de correo electrónico. Se utiliza para demostrar la creación de objetos y la interacción.

## Capítulo 4

*agenda*

La implementación de una agenda electrónica sencilla. Se utiliza para introducir colecciones y ciclos.

**Capítulo 4** *subastas*

Un sistema de subastas. Más sobre colecciones y ciclos, esta vez con iteradores.

**Capítulo 4** *analizador-weblog*

Un programa para analizar los archivos de registro de acceso a un sitio web; introduce arreglos y ciclos for.

**Capítulo 5** *soporte-tecnico*

La implementación de un programa que simula un diálogo al estilo de *Eliza* para proporcionar “soporte técnico” a los clientes; introduce el uso de clases de biblioteca en general y de algunas clases específicas en particular, lectura y escritura de documentación.

**Capítulo 5** *pelotas*

Una simulación gráfica del rebote de pelotas; demuestra la separación entre interfaz e implementación y algunos gráficos sencillos.

**Capítulo 6** *agenda-diaria*

Los primeros estados de una implementación de una agenda diaria para anotar citas; se usa para discutir estrategias de prueba y depuración.

**Capítulo 6** *calculadora*

Una implementación de una calculadora electrónica de escritorio. Este ejemplo refuerza los conceptos introducidos anteriormente y se usa para discutir prueba y depuración.

**Capítulo 6** *ladrillos*

Un ejercicio simple de depuración; modela el armado de palletes de ladrillos mediante cálculos sencillos.

**Capítulo 7, Capítulo 9** *world-of-zuul*

Un juego de aventuras basado en texto. Es un proyecto altamente extendible y puede ser para los estudiantes, un gran proyecto de final abierto. Se utiliza para discutir el diseño de clases de buena calidad, acoplamiento y cohesión. Se utiliza nuevamente en el Capítulo 9 como ejemplo para el uso de herencia.

**Capítulo 8, Capítulo 9** *DoME*

Una base de datos de CD y DVD. Este proyecto se discute y se extiende con mucho detalle para introducir los fundamentos de herencia y polimorfismo.

## Capítulo 10

*zorros-y-conejos*

Una simulación clásica del tipo predador-presa; refuerza los conceptos de herencia y agrega clases abstractas e interfaces.

## Capítulo 11

*visor-de-imagen*

Una aplicación sencilla para visualizar y manipular imágenes. Nos concentraremos principalmente en la construcción de la IGU.

## Capítulo 11

*sonidos-simples*

Una aplicación para reproducir sonidos y otro ejemplo de construcción de IGU.

## Capítulo 12

*libreta-de-direcciones*

La implementación de una libreta de direcciones con una IGU opcional. La búsqueda es flexible: puede buscar las entradas mediante partes del nombre o del número de teléfono. Este proyecto hace uso extenso de las excepciones.

## Capítulo 13

*sistema-de-reserva-de-entradas*

Un sistema que maneja la reserva de entradas para el cine. Este ejemplo se usa en una discusión para descubrir las clases y el diseño de la aplicación. No se proporciona código ya que el ejemplo representa el desarrollo de una aplicación desde una hoja de papel en blanco.

## Capítulo 14

*compania-de-taxis*

Este ejemplo es una combinación del sistema de reservas, un sistema de administración y una simulación. Se utiliza como estudio de caso para reunir muchos de los conceptos y técnicas discutidas a lo largo del libro.



# Agradecimientos

Muchas personas contribuyeron de diferentes maneras con este libro e hicieron posible su creación.

En primer lugar y el más importante de mencionar es John Rosenberg. John es actualmente Deputy Vice-Chancellor en la Deakin University de Australia. Es por una mera coincidencia de circunstancias que John no es uno de los autores de este libro. Fue uno de los que dirigió sus esfuerzos al desarrollo de BlueJ y de algunas ideas y de la pedagogía subyacente en él desde el comienzo, y hemos hablado sobre escribir este libro durante varios años. Gran parte del material de este libro fue desarrollado en las discusiones con John. Simplemente el hecho de que el día tiene sólo 24 horas, y muchas de las cuales ya las tenía asignadas a muchos otros trabajos, le impidieron escribir realmente este libro. John ha contribuido continuamente con este texto mientras lo escribíamos y nos ayudó a mejorarlo de diversas maneras. Apreciamos su amistad y colaboración inmensamente.

Otras varias personas han ayudado a que BlueJ sea lo que es: Bruce Quig, Davin McCall y Andrew Patterson en Australia, y Ian Utting y Poul Henriksen en Inglaterra. Todos trabajaron sobre BlueJ durante varios años, mejorando y extendiendo el diseño y la implementación de manera adicional a sus propios compromisos. Sin su trabajo, BlueJ nunca hubiera logrado alcanzar la calidad y la popularidad que tiene al día de hoy y este libro probablemente, jamás se hubiera escrito.

Otra contribución importante que hizo que la creación de BlueJ y de este libro fuera posible fue el muy generoso aporte de Sun Microsystems. Emil Sarpa, que trabaja para Sun en Palo Alto, CA, ha creído en el proyecto BlueJ desde sus tempranos comienzos. Su apoyo y su sorprendente y nada burocrático modo de cooperación nos ayudó inmensamente a lo largo del camino.

Todas las personas de Pearson Education trabajaron realmente muy duro para lograr la producción de este libro en una agenda muy estrecha y acomodaron varios de nuestros idiosincráticos modos. Gracias a Kate Simon Plumtree que dio a luz esta edición. Gracias también al resto del equipo que incluye a Bridget Allen, Kevin Ancient, Tina Cadle-Bowman, Tim Parker, Veronique Seguin, Fiona Sharples y Owen Knight. Esperamos no habernos olvidado de ninguno y nos disculpamos si así fuera.

El equipo de ventas de Pearson también realizó un tremendo trabajo para que este libro resulte visible, tratando de apartar de cada autor el miedo de que su libro pase inadvertido.

Nuestros revisores también trabajaron muy duro sobre el manuscrito, a menudo en momentos del año de mucho trabajo, y queremos expresar nuestro arecio a Michael Caspersen, Devdatt Dubhashi, Khalid Mughal y Richard Snow por sus críticas estimulantes y constructivas.

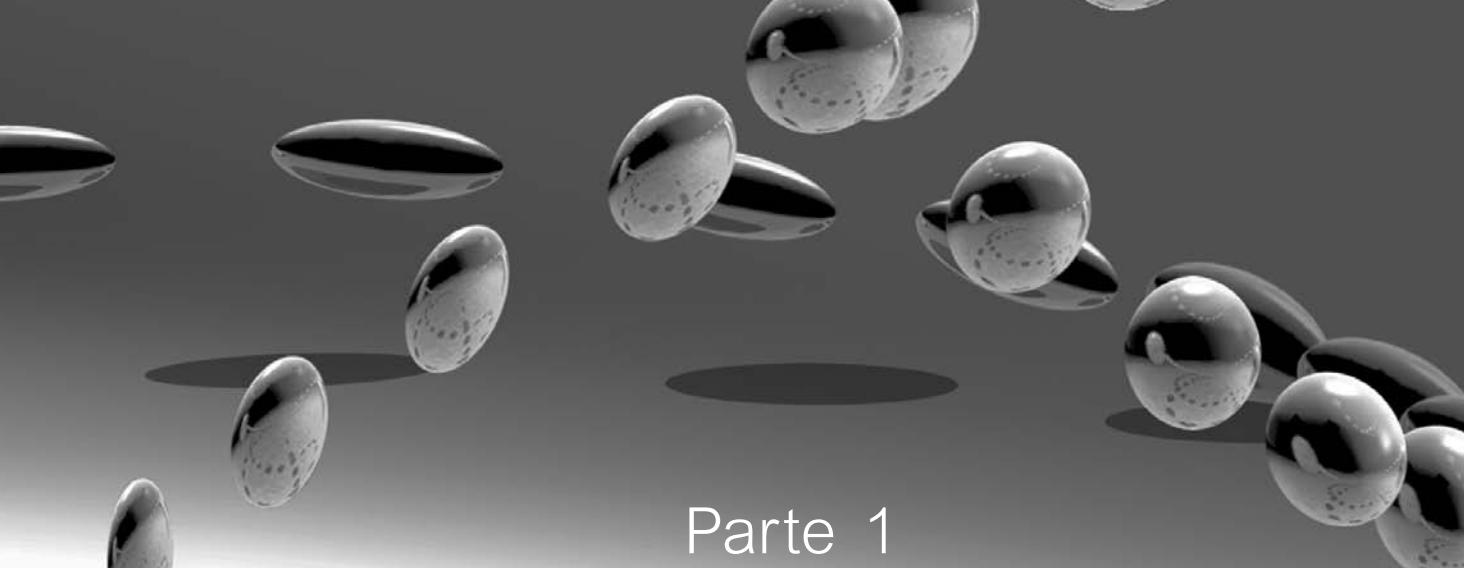
Axel Schmolitzky, quien llevó a cabo la excelente traducción de este libro al alemán, debe haber sido nuestro lector más cuidadoso y escrupuloso; sugirió un buen número de mejoras posibles, a veces sobre puntos muy sutiles.

David desea agregar su agradecimiento tanto a su equipo como a los estudiantes del Computer Science Department de la University of Kent. Ha sido un privilegio enseñar a los estudiantes que tomaron el curso introductorio de OO. Ellos también proporcionaron el estímulo y la motivación esencial que hace que la enseñanza sea mucho más agradable. Sin la invaluable asistencia de colegas y supervisores postgraduados dando las clases, hubiera sido imposible y Simon Thompson proporcionó un firme apoyo en su rol de Head of Department. Fuera de la vida universitaria, varias personas aportaron tiempo de recreación y de vida social para impedir que me dedicara exclusivamente a escribir: gracias a mis mejores amigos, Chris Phillips y Martin Stevens, que me mantuvieron en el aire y a Joe Rotchell, que me ayudó a mantener los pies en la tierra.

Finalmente, quisiera agradecer a mi esposa Helen cuyo amor es muy especial, a mis hijos cuyas vidas son tan preciosas.

Michael desea agradecer a Andrew y a Bruce por las muchas horas de intensa discusión. Aparte del trabajo técnico que dio este resultado, los disfruté inmensamente. Y tengo un buen argumento. John Rosenberg ha sido mi mentor durante varios años desde mis inicios en mi carrera académica. Sin su hospitalidad y apoyo nunca podría haberla hecho en Australia y sin él como supervisor de mi PhD y colega, nunca hubiera llevado a cabo lo mucho que logré hacer. Es un placer trabajar con él y le debo mucho. Gracias a Michael Caspersen quien no sólo es un buen amigo sino que ha influido en mi modo de pensar la enseñanza durante los varios talleres que hemos compartido. Mis colegas del grupo de ingeniería del software del Marsk Institute en Dinamarca, Bent Bruun Kristensen, Palle Nowack, Bo Norregaard Jorgensen, Kasper Hallenborg Pedersen y Daniel May, han tolerado pacientemente cada fecha de entrega mientras escribía este libro y al mismo tiempo, me introdujeron en la vida en Dinamarca.

Finalmente, quisiera agradecer a mi esposa Leah y a mis dos hijitas, Sophie y Feena. Muchas veces tuvieron que tolerar mis largas horas de trabajo a cualquier hora del día mientras escribía este libro. Su amor me da las fuerzas para continuar y hacen que valga la pena.



Parte 1

**Fundamentos  
de programación  
orientada a objetos**

A dark gray background featuring a cluster of metallic, reflective spheres of various sizes and shapes, some with small holes, floating in the upper right quadrant.



## CAPÍTULO

# 1

## Objetos y clases

Principales conceptos que se abordan en este capítulo:

- objetos
- métodos
- clases
- parámetros

Con este capítulo comienza nuestro viaje por el mundo de la programación orientada a objetos. Es aquí donde introducimos los conceptos más importantes que aprenderá: objetos y clases.

Al finalizar el capítulo comprenderá: qué son los objetos y las clases, para qué se usan y cómo interactuar con ellos. Este capítulo sienta las bases para la exploración del resto del libro.

### 1.1

## Objetos y clases

#### Concepto

Los objetos Java modelan objetos que provienen del dominio de un problema.

Cuando escribe un programa de computación en un lenguaje orientado a objetos está creando en su computadora un modelo de alguna parte del mundo real. Las partes con que se construye el modelo provienen de los objetos que aparecen en el dominio del problema. Estos objetos deben estar representados en el modelo computacional que se está creando.

Los objetos pueden ser organizados en categorías y una clase describe, en forma abstracta, todos los objetos de un tipo en particular.

Podemos aclarar estas nociones abstractas mediante un ejemplo. Suponga que desea modelar una simulación de tráfico. Un tipo de entidad con la que tendrá que trabajar es autos. ¿Qué es un auto en nuestro contexto? ¿Es una clase o es un objeto? Algunas preguntas nos ayudarán a tomar una decisión.

#### Concepto

Los objetos se crean a partir de clases. La clase describe la categoría del objeto. Los objetos representan casos individuales de una clase.

¿De qué color es un auto? ¿Cuán rápido puede marchar? ¿Dónde está en este momento?

Observará que no podemos responder estas preguntas a menos que hablemos de un auto específico. La razón es que, en este contexto, la palabra «auto» refiere a la *clase* auto puesto que estamos hablando de los autos en general y no de uno en particular.

Si digo «Mi viejo auto se encuentra estacionado en el garaje de casa», podemos responder todas las preguntas anteriores: este auto es rojo, no marcha demasiado rápido y está en mi garaje. Ahora estoy hablando de un objeto, un ejemplo particular de un auto.

Generalmente, cuando nos referimos a un objeto en particular hablamos de una *instancia*. De aquí en adelante usaremos regularmente el término «instancia». Instancia es casi un sinónimo de objeto. Nos referimos a objetos como instancias cuando queremos enfatizar que son de una clase en particular (como por ejemplo, cuando decimos «este objeto es una instancia de la clase auto»).

Antes de continuar con esta discusión bastante teórica, veamos un ejemplo.

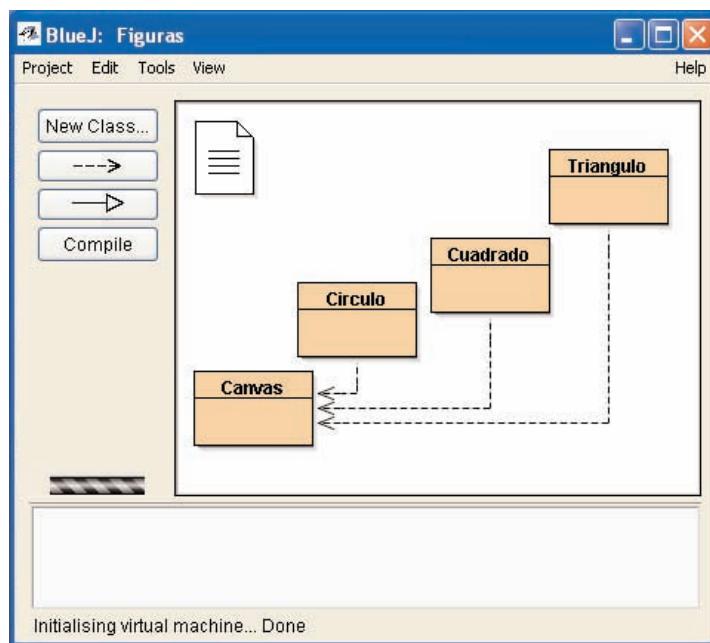
## 1.2

## Crear objetos

Inicie<sup>1</sup> BlueJ y abra el ejemplo que está bajo el nombre *figuras*. Verá una ventana similar a la que se muestra en la Figura 1.1.

**Figura 1.1**

El proyecto *figuras* en BlueJ



En esta ventana aparece un diagrama en el que cada uno de los rectángulos coloreados representa una clase en nuestro proyecto. En este proyecto tenemos las clases de nombre **Circulo**, **Cuadrado**, **Triangulo** y **Canvas**.

Haga clic con el botón derecho del ratón sobre la clase **Circulo** y seleccione el elemento

```
new Circulo()
```

del menú contextual. El sistema solicita el «nombre de la instancia» (*name of the instance*), haga clic en **Ok** ya que, por ahora, el nombre por defecto es suficientemente

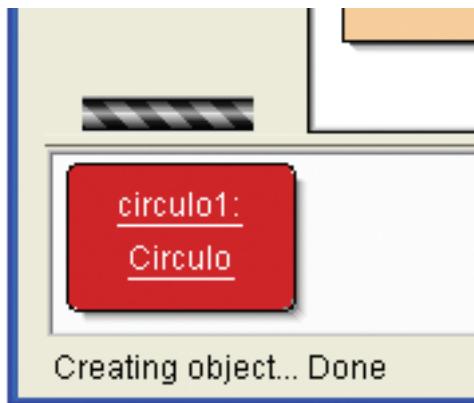
---

<sup>1</sup> Esperamos que mientras lee este libro realice regularmente algunas actividades y ejercicios. En este punto asumimos que sabe cómo iniciar BlueJ y abrir los proyectos de ejemplo. De no ser así, lea primero el Apéndice A.

bueno. Verá un rectángulo rojo ubicado en la parte inferior de la ventana, etiquetado con el nombre «circulo1» (Figura 1.2).

**Figura 1.2**

Un objeto en el banco de objetos



¡Acaba de crear su primer objeto! El ícono rectangular «Circulo» de la Figura 1.1 representa la clase `Circulo` mientras que `circulo1` es un objeto creado a partir de esta clase. La zona de la parte inferior de la ventana en la que se muestran los objetos se denomina *banco de objetos* (*object bench*).

**Convención** Los nombres de las clases comienzan con una letra mayúscula (como `Circulo`) y los nombres de los objetos con letras minúsculas (`circulo1`). Esto ayuda a distinguir de qué elemento estamos hablando.

**Ejercicio 1.1** Cree otro círculo. Luego, cree un cuadrado.

## 1.3

## Invocar métodos

Haga un clic derecho sobre un objeto círculo (¡no sobre la clase!) y verá un menú contextual que contiene varias operaciones. De este menú, seleccione `volverVisible`; esta operación dibujará una representación de este círculo en una ventana independiente. (Figura 1.3.)

Observe que hay varias operaciones en el menú contextual del círculo. Pruebe invocar un par de veces las operaciones `moverDerecha` y `moverAbajo` para desplazar al círculo más cerca del centro de la pantalla. También podría probar `volverInvisible` y `volverVisible` para ocultar y mostrar el círculo.

### Concepto

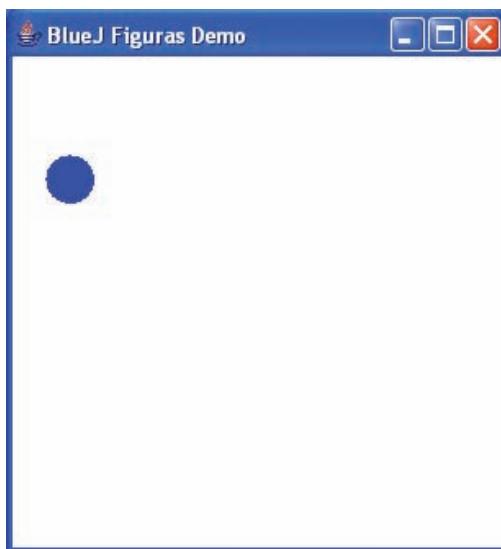
Podemos comunicarnos con los objetos invocando sus métodos. Generalmente, los objetos hacen algo cuando invocamos un método.

**Ejercicio 1.2** ¿Qué ocurre si llama dos veces a `moverAbajo`? ¿O tres veces? ¿Qué pasa si llama dos veces a `volverInvisible`?

Los elementos del menú contextual del círculo representan las operaciones que se pueden usar para manipular el círculo. En Java, estas operaciones se denominan *métodos*. Usando la terminología común, decimos que estos métodos son *llamados* o *invocados*. De aquí en adelante usaremos esta terminología que es más adecuada. Por ejemplo, podríamos pedirle que «invoque el método `moverDerecha` de `circulo1`».

**Figura 1.3**

El dibujo de un círculo

**1.4****Parámetros****Concepto**

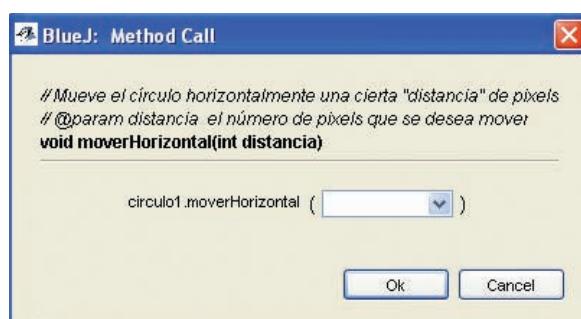
Los métodos pueden tener *parámetros* para proporcionar información adicional para realizar una tarea.

Ahora invoque el método `moverHorizontal`. Aparecerá una caja de diálogo que le solicita que ingrese algunos datos (Figura 1.4). Ingrese el número 50 y haga clic en Ok. Verá que el círculo se mueve 50 píxeles<sup>2</sup> hacia la derecha.

El método `moverHorizontal` que acaba de nombrar, está escrito de modo tal que requiere información adicional para ejecutarse. En este caso, la información requerida es la distancia (cuánto debe moverse el círculo) y esto hace que el método `moverHorizontal` sea más flexible que los métodos `moverDerecha` o `moverIzquierda`. Los últimos métodos mueven siempre al círculo una distancia determinada mientras que `moverHorizontal` permite especificar cuánto se quiere mover.

**Figura 1.4**

Caja de diálogo de una llamada a método



**Ejercicio 1.3** Antes de seguir leyendo, intente invocar los métodos `moverVertical`, `moverLentoVertical` y `cambiarTamaño`. Descubra cómo usar `moverHorizontal` para mover el círculo 70 píxeles hacia la izquierda.

<sup>2</sup> Un pixel es un punto en la pantalla. Toda su pantalla está compuesta por una grilla de simples píxeles.

**Concepto**

El encabezado de un método se denomina su *signatura* y proporciona la información necesaria para invocarlo.

Los valores adicionales que requieren algunos métodos se denominan *parámetros*. Un método indica el tipo de parámetros que requiere. Por ejemplo, cuando invoca al método `moverHorizontal` tal como muestra la Figura 1.4, la caja de diálogo muestra en su parte superior la línea

```
void moverHorizontal(int distancia)
```

Esta línea se denomina *signatura* del método. La signatura proporciona algo de información sobre el método en cuestión. La parte comprendida entre paréntesis (`int distancia`) es la información sobre el parámetro requerido. Para cada parámetro se define un *tipo* y un *nombre*. La signatura anterior establece que el método requiere un parámetro de tipo `int` y de nombre `distancia`. El nombre ofrece alguna pista sobre el significado del dato esperado.

**1.5****Tipos de dato**

Un tipo especifica la naturaleza del dato que debe pasarse a un parámetro. El tipo `int` significa números enteros (en inglés, «*integer numbers*», de aquí su abreviatura «`int`»).

**Concepto**

Los parámetros tienen tipos de dato. El tipo de dato define la clase de valores que un parámetro puede tomar.

En el ejemplo anterior, la signatura del método `moverHorizontal` establece que antes de que el método pueda ejecutarse, necesitamos suministrárle un número entero especificando la distancia a mover. El campo de entrada de datos que se muestra en la Figura 1.4 nos permite ingresar este número.

En los ejemplos que hemos trabajado hasta aquí, el único tipo de dato que hemos visto ha sido `int`. Los parámetros de los métodos `mover` y del método `cambiarTamaño` son todos de ese tipo.

Una mirada más de cerca al menú contextual del objeto nos muestra que los métodos del menú incluyen los tipos de dato de los parámetros. Si un método no tiene parámetros, aparece un par de paréntesis vacíos al final del nombre del método. Si tiene un parámetro, se muestra el tipo de dato del mismo. En la lista de los métodos del círculo podrá ver un método con un tipo de parámetro diferente: el método `cambiarColor` tiene un parámetro de tipo `String`.

El tipo de dato `String` indica que se espera el ingreso de un fragmento de texto (por ejemplo, una palabra o una frase). Llamaremos *cadenas* a estas secciones de texto. Las cadenas van siempre encerradas entre comillas dobles. Por ejemplo, para ingresar la palabra `rojo` como una cadena escribimos

`_rojo_`

La caja de diálogo para la invocación de métodos incluye una sección de texto denominada *comentario* ubicada por encima de la signatura del método. Los comentarios se incluyen para ofrecer información al lector (humano) y se describen en el Capítulo 2. El comentario del método `cambiarColor` describe los nombres de los colores que el sistema reconoce.

**Ejercicio 1.4** Invoque el método `cambiarColor` sobre uno de los objetos círculo e ingrese la cadena «`rojo`». Esta acción debería modificar el color del círculo. Pruebe con otros colores.

**Ejercicio 1.5** Este proyecto es un ejemplo muy simple y no admite demasiados colores. Vea qué ocurre si especifica un color no reconocido por el sistema.

**Ejercicio 1.6** Invoque el método `cambiarColor` y escriba el color sin las comillas, en el campo del parámetro. ¿Qué ocurre?

**Cuidado** Un error muy común entre los principiantes es olvidar las comillas dobles cuando escriben un valor de tipo `String`. Si escribe `verde` en lugar de «`verde`» aparecerá un mensaje de error diciendo «Error: cannot resolve symbol». («no puede resolver el símbolo»).

Java admite otros varios tipos de dato incluyendo, por ejemplo, números decimales y caracteres. No abordaremos todos los tipos ahora, pero volveremos sobre este punto más adelante. Si quiere encontrar información sobre los tipos de datos en Java, vea el Apéndice B.

## 1.6

## Instancias múltiples

**Ejercicio 1.7** Cree en el banco de objetos algunos objetos círculo. Puede hacerlo seleccionando `new Circulo()` del menú contextual de la clase `Circulo`. Vuélvalos visibles y luego desplácelos por la pantalla usando los métodos «mover». Haga que un círculo sea grande y amarillo y que otro sea pequeño y verde. Pruebe también con las otras figuras: cree algunos triángulos y algunos cuadrados. Cambie sus posiciones, tamaños y colores.

Una vez que tiene una clase, puede crear tantos objetos (o instancias) de esa clase como desee. Puede crear muchos círculos a partir de la clase `Circulo`. A partir de la clase `Cuadrado` puede crear muchos cuadrados.

Cada uno de esos objetos tiene su propia posición, color y tamaño. Usted cambia un atributo de un objeto (como su tamaño, por ejemplo) llamando a un método de ese objeto, y esta acción afectará a ese objeto en particular, pero no a los otros.

También puede haber notado un detalle adicional sobre los parámetros. Observe el método `cambiarTamanio` del triángulo. Su signatura es

```
void cambiarTamanio (int nuevoAlto, int nuevoAncho)
```

Este es un ejemplo de un método que tiene más de un parámetro. Este método tiene dos parámetros que están separados por una coma en la signatura. De hecho, los métodos pueden tener cualquier número de parámetros.

## 1.7

## Estado

Se hace referencia al conjunto de valores de todos los atributos que definen un objeto (tales como las posiciones `x` e `y`, el color, el diámetro y el estado de visibilidad para un círculo) como el *estado* del objeto. Este es otro ejemplo de terminología común que usaremos de aquí en adelante.

En BlueJ, el estado de un objeto se puede inspeccionar seleccionando la función *Inspect* del menú contextual del objeto. Cuando se inspecciona un objeto, se despliega una ventana similar a la que se muestra en la Figura 1.5 denominada *Inspector del Objeto (Object Inspector)*.

### Concepto

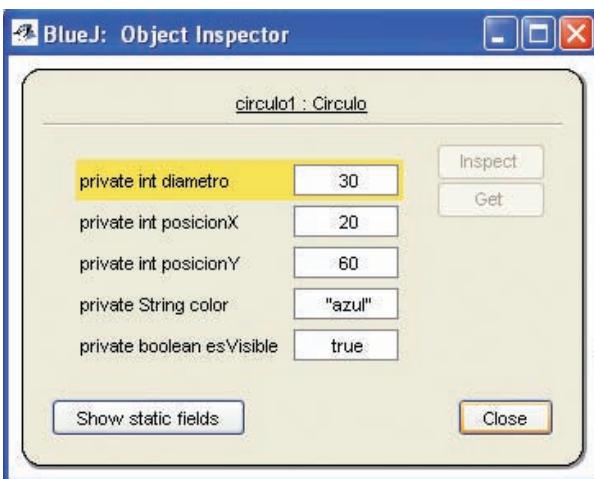
Los objetos tienen un estado. El estado está representado por los valores almacenados en sus campos.

**Ejercicio 1.8** Asegúrese de tener varios objetos en el banco de objetos y luego inspeccione cada uno de ellos. Pruebe cambiar el estado de un objeto (por ejemplo, llamando al método `moverIzquierda`) mientras mantiene abierto el inspector. Debería ver que los valores cambian en el inspector del objeto.

Algunos métodos, cuando son llamados, cambian el estado de un objeto. Por ejemplo, `moverIzquierda` modifica el atributo `posicionX`. Java se refiere a los atributos de los objetos como *campos (fields)*.

**Figura 1.5**

Diálogo de inspección de un objeto



## 1.8

## ¿Qué es un objeto?

Al inspeccionar objetos diferentes observará que todos los objetos de la misma clase tienen los mismos campos; es decir que el número, el tipo de dato y los nombres de los campos de una misma clase son los mismos, mientras que el valor de un campo en particular de cada objeto puede ser distinto. Por el contrario, los objetos de clases diferentes pueden tener diferentes campos. Por ejemplo, un círculo tiene un campo «`diametro`», mientras que un triángulo tiene los campos «`ancho`» y «`alto`».

La razón es que el número, el tipo de dato y el nombre de los campos se definen en una clase, no en un objeto. Por ejemplo, la clase `Circulo` declara que cada objeto `círculo` tendrá cinco campos cuyos nombres son `diametro`, `posicionX`, `posicionY`, `esVisible` y `color`. También define los tipos de dato para cada uno estos campos; es decir, especifica que los tres primeros son de tipo `int`, mientras que `color` es de tipo `String` y la bandera `esVisible` es de tipo `boolean`. (El tipo `boolean` o lógico es un tipo que permite representar sólo dos valores: *verdadero* y *falso* (`true` y `false`), sobre los que hablaremos con más detalle más adelante.)

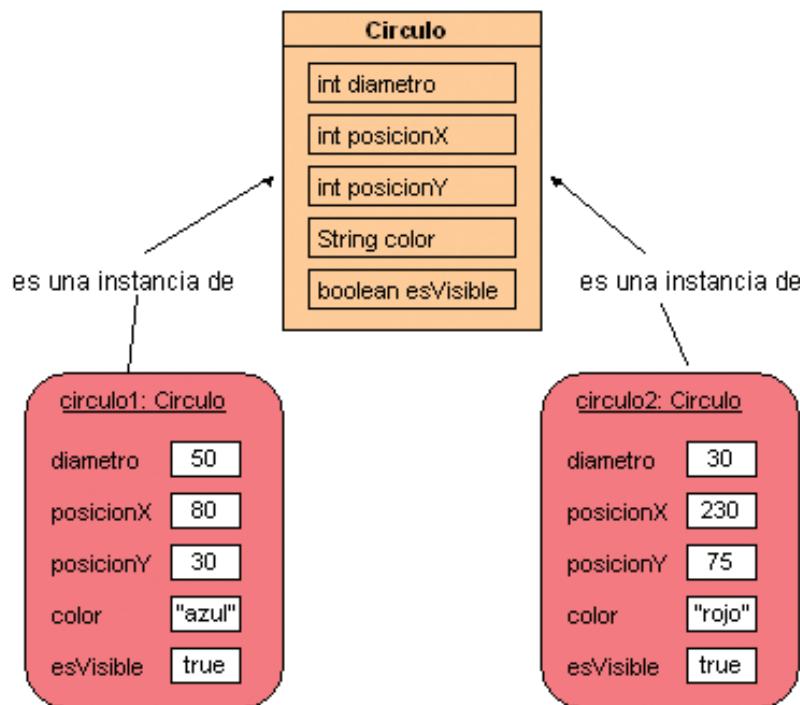
Cuando se crea un objeto de la clase `Circulo`, tendrá automáticamente estos campos. Los valores de estos campos se almacenan en el objeto, lo que asegura, por ejemplo, que cada círculo tiene un color y que cada uno puede tener un color diferente (Figura 1.6).

La historia es similar para los métodos. Los métodos se definen en la clase del objeto. Como resultado, todos los objetos de una clase dada tienen los mismos métodos. Sin embargo, los métodos se invocan desde los objetos, y esto aclara por ejemplo, cuál es el objeto que se modifica cuando se invoca el método `moverDerecha`.

**Ejercicio 1.9** Use las figuras del proyecto *figuras* para crear la imagen de una casa y un sol, similar a la de la Figura 1.7. Mientras lo hace, escriba las acciones que lleva a cabo para lograr el resultado esperado. ¿Podría lograr la misma imagen de diferentes maneras?

**Figura 1.6**

Una clase y sus objetos con campos y valores



## 1.9

## Interacción entre objetos

En la próxima sección trabajaremos con un proyecto de ejemplo diferente. Cierre el proyecto *figuras* si es que todavía lo tiene abierto y abra el proyecto de nombre *cuadro*.

**Ejercicio 1.10** Abra el proyecto *cuadro*. Cree una instancia de la clase Cuadro e invoque su método **dibujar**. Pruebe también los métodos **ponerBlancoYNegro** y **ponerColor**.

**Ejercicio 1.11** ¿Cómo piensa que dibuja la clase Cuadro?

Cuatro de las clases de este proyecto son idénticas a las clases del proyecto *figuras*, pero ahora tenemos una clase adicional: Cuadro. Esta clase está programada para que haga exactamente lo mismo que ya hemos hecho a mano en el Ejercicio 1.9.

En realidad, si queremos que se haga en Java una serie de tareas, normalmente no las hacemos a mano como en el Ejercicio 1.9, sino que creamos una clase que haga dichas tareas por nosotros. Es el caso de la clase Cuadro.

La clase Cuadro está escrita de modo que, cuando se crea una instancia, esta instancia crea dos objetos cuadrado (uno para la pared y otro para la ventana), un triángulo y

un círculo, los mueve y cambia sus colores y tamaño, hasta que el resultado se parezca a la imagen que vemos en la Figura 1.7.

**Figura 1.7**

Una imagen creada a partir de un conjunto de objetos



#### Concepto

**Llamada de métodos.** Los objetos se pueden comunicar entre ellos invocando los métodos de los otros objetos.

El punto importante es que los objetos pueden crear otros objetos y pueden llamar a cada uno de sus métodos. Un programa Java normal puede tener centenares o miles de objetos. El usuario de un programa sólo lo inicia (y por lo general, en el inicio se crea un primer objeto) y todos los otros objetos son creados, directa o indirectamente, por ese objeto.

Ahora, la gran pregunta es: ¿cómo escribimos la clase para un objeto como éste?

## 1.10

## Código fuente

Cada clase tiene algún *código fuente* asociado. El código fuente es un texto que define los detalles de la clase. En BlueJ, se puede visualizar el código fuente de una clase seleccionando la función *Open Editor* del menú contextual de la clase o haciendo doble clic en el ícono de la clase.

**Ejercicio 1.12** Observe nuevamente el menú contextual de la clase **Cuadro**.

Verá una opción etiquetada como *Open Editor*. Selecciónela. Esta acción abre el editor de textos mostrando el código fuente de esta clase.

#### Concepto

El **código fuente** de una clase determina la estructura y el comportamiento (los campos y los métodos) de cada uno de los objetos de dicha clase.

El código fuente (o simplemente el **código**) es un texto escrito en lenguaje de programación Java y define qué campos y métodos tiene la clase y qué ocurre cuando se invoca un método. En el próximo capítulo hablaremos sobre qué contiene exactamente el código de una clase y cómo está estructurado.

Gran parte del aprendizaje del arte de la programación consiste en aprender cómo escribir estas definiciones de clases y para lograrlo, deberemos aprender a usar el lenguaje Java (aunque existen otros lenguajes de programación que se podrían usar para escribir el código).

Cuando realiza algún cambio en el código y cierra el editor<sup>3</sup>, el ícono de esta clase en el diagrama aparece rayado. Las rayas indican que el fuente ha cambiado. En estos casos, la clase necesita ser compilada haciendo clic en el botón *Compile*. (Para más información sobre lo que ocurre cuando se compila una clase puede leer la nota «Acerca de la compilación».) Una vez que una clase ha sido compilada, se pueden crear nuevamente objetos y probar sus cambios.

### Nota: acerca de la compilación

Cuando las personas escriben programas de computación usan generalmente un lenguaje de programación de alto nivel, como por ejemplo Java. El problema que se presenta es que la computadora no puede ejecutar directamente el código Java. Java fue diseñado para ser razonablemente fácil de leer para los humanos, pero no para las computadoras. Internamente, las computadoras trabajan con una representación binaria de un código máquina cuyo aspecto es muy diferente al de Java. Nuestro problema es que este código es tan complejo que no queremos escribirlo directamente, preferimos escribir en Java. ¿Qué podemos hacer?

La solución es un programa denominado compilador. El compilador traduce el código Java a código máquina. Podemos escribir en Java, ejecutar el compilador (que genera el código máquina) y la computadora puede entonces leer el código máquina. Como resultado, cada vez que cambiamos el código debemos ejecutar el compilador antes de poder usar nuevamente la clase para crear un objeto. Por otra parte, no existe la versión del código máquina que necesitan las computadoras.

**Ejercicio 1.13** Busque en el código de la clase **Cuadro** la parte que efectivamente dibuja la imagen. Cambie el código de modo que el sol resulte ser azul en lugar de amarillo.

**Ejercicio 1.14** Agregue un segundo sol a la imagen. Para hacer esto, centre su atención en las declaraciones de campos que están en la parte superior de la clase. Encontrará este código:

```
private Cuadrado pared;
private Cuadrado ventana;
private Triangulo techo;
private Circulo sol;
```

Aquí es donde necesita agregar una línea para el segundo sol, por ejemplo:

```
private Circulo sol2;
```

Escriba el código adecuado para crear el segundo sol.

**Ejercicio 1.15 Desafío** (que sea un ejercicio «desafío» significa que puede que no lo resuelva rápidamente. No esperamos que todos los lectores sean capaces de resolverlo en este momento. Si lo logra, grandioso; de lo contrario, no se preocupe. Las cosas se irán aclarando a medida que siga leyendo. Vuelva

<sup>3</sup> En BlueJ no es necesario grabar explícitamente el texto del editor antes de cerrarlo. Si cierra el editor, el código se graba automáticamente.

a este ejercicio más adelante). Agregue una puesta de sol a la versión de **Cuadro** que tiene un único sol. Es decir, haga que el sol descienda lentamente. Recuerde que el círculo tiene un método `moverLentoVertical` y puede usarlo para lograr que el sol descienda.

**Ejercicio 1.16 Desafío.** Si agregó la puesta de sol al final del método `dibujar` (de modo que el sol baje automáticamente cuando se dibuja la imagen), haga la siguiente modificación. Queremos que la puesta de sol la lleve a cabo un método independiente, de modo que podamos invocar a `dibujar` y ver el sol en lo alto de la imagen, y luego invocar al método `atardecer` (un método independiente!) para hacer que el sol descienda.

## 1.11

### Otro ejemplo

Ya hemos tratado en este capítulo un gran número de conceptos nuevos. Ahora los volveremos a ver en un contexto diferente para ayudarle a comprender estos conceptos. Con este fin usaremos un ejemplo diferente. Cierre el proyecto *cuadro* si es que todavía lo tiene abierto y abra el proyecto *curso-de-laboratorio*.

Este proyecto es una parte de una base de datos de estudiantes simplificada, diseñada para registrar el recorrido de los estudiantes en los cursos de laboratorio e imprimir las listas de alumnos de estos cursos.

**Ejercicio 1.17** Cree un objeto de clase **Estudiante**. Verá que en este caso no sólo se le solicita ingresar el nombre de la instancia sino también el valor de algunos otros parámetros. Complete los datos antes de hacer clic en Ok. (Recuerde que los parámetros de tipo `String` deben escribirse entre comillas dobles.)

## 1.12

### Valores de retorno

Tal como ocurrió anteriormente, puede crear varios objetos, y nuevamente los objetos disponen de métodos que usted puede invocar en sus propios menús contextuales.

#### Concepto

**Resultados.** Los métodos pueden devolver información de algún objeto mediante un **valor de retorno**.

**Ejercicio 1.18** Cree algunos objetos estudiante. Invoque el método `obtenerNombre` de cada objeto. Explique qué ocurre.

Cuando llamamos al método `obtenerNombre` de la clase `Estudiante` notamos algo nuevo: los métodos pueden devolver un valor como resultado. De hecho, la firma de cada método nos informa si devuelve o no un resultado y qué tipo de resultado es. La firma de `obtenerNombre` (tal como muestra el menú contextual del objeto) está definida de la siguiente manera:

```
String obtenerNombre()
```

La palabra `String` que aparece antes del nombre del método especifica el tipo de retorno. En este caso, establece que este método devolverá un resultado de tipo `String` cuando sea invocado. La firma de `cambiarNombre` es:

```
void cambiarNombre(String nuevoNombre)
```

La palabra `void` indica que este método no retorna ningún resultado.

Los métodos que devuelven o retornan valores nos permiten obtener información sobre un objeto mediante una llamada al método. Quiere decir que podemos usar métodos tanto para cambiar el estado de un objeto como para investigar su estado.

## 1.13

### Objetos como parámetros

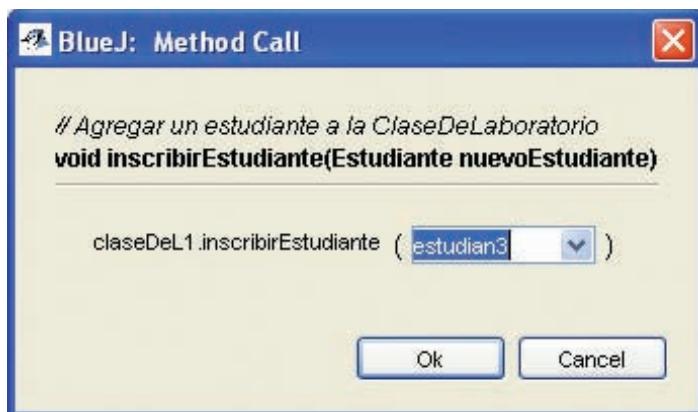
**Ejercicio 1.19** Cree un objeto de clase `CursoDeLaboratorio`. Tal como lo indica su firma, usted necesita especificar el número máximo de estudiantes de ese curso (un entero).

**Ejercicio 1.20** invoque el método `numeroDeEstudiantes` de ese curso. ¿Qué hace este método?

**Ejercicio 1.21** Observe la firma del método `inscribirEstudiante`. Verá que el tipo del parámetro esperado es `Estudiante`. Asegúrese de tener en el banco de objetos dos o tres objetos estudiante y un objeto `CursoDeLaboratorio`; luego invoque el método `inscribirEstudiante` del objeto `CursoDeLaboratorio`. Con el cursor ubicado en el campo de entrada de la caja de diálogo, haga clic sobre uno de los objetos estudiante —esta acción ingresa el nombre del objeto estudiante en el campo del parámetro del método `inscribirEstudiante` (Figura 1.8). Haga clic en Ok, y queda agregado el estudiante al `CursoDeLaboratorio`. También agregue uno o más estudiantes.

**Figura 1.8**

Agregar un estudiante a `CursoDeLaboratorio`



**Ejercicio 1.22** Llame al método `imprimirLista` del objeto `CursoDeLaboratorio`. Verá en la terminal de BlueJ una lista de todos los estudiantes de este curso (Figura 1.9).

Tal como muestra el ejercicio, los objetos pueden ser pasados como parámetros a los métodos de otros objetos. En el caso de que un método espere un objeto como parámetro, el nombre de la clase del objeto que espera se especifica como el tipo de parámetro en la firma de dicho método.

Explore un poco más este proyecto. Pruebe identificar en este contexto los conceptos tratados en el ejemplo *figuras*.

**Figura 1.9**

Salida de la lista del  
CursoDeLaboratorio

```
Curso de Laboratorio Lunes 10:00 a.m.
Instructor: P. Stephenson Aula: U23
Lista de la clase:
Wolfgang Amadeus Mozart (100234)
Lisa Simpson (122044)
Charlie Brown (12003P)
Número de estudiantes: 3
```

**Ejercicio 1.23** Cree tres estudiantes con los detalles siguientes:

*Blanca Nieves*, ID de estudiante: 100234, créditos: 24

*Lisa Simpson*, ID de estudiante: 122044, créditos: 56

*Charlie Brown*, ID de estudiante: 12003P, créditos: 6

Luego inscriba a los tres estudiantes en un curso de laboratorio e imprima una lista en la pantalla.

**Ejercicio 1.24** Use el inspector del `CursoDeLaboratorio` para descubrir los campos que tiene.

**Ejercicio 1.25** Determine el instructor, el aula y el horario de un curso de laboratorio y muestre la lista en la ventana terminal para controlar que aparezcan estos nuevos detalles.

## 1.14

## Resumen

En este capítulo hemos explorado los conceptos básicos de clase y de objeto. Hemos tratado el hecho de que los objetos son especificados por las clases. Las clases representan el concepto general de una cosa, mientras que los objetos representan instancias concretas de una clase. Podemos tener varios objetos de cualquier clase.

Los objetos tienen métodos que podemos usar para comunicarnos con ellos. Podemos usar un método para modificar al objeto o para obtener información acerca de él. Los métodos pueden tener parámetros y los parámetros tienen tipos. Los métodos pueden tener tipos de retorno que especifican el tipo de dato que devuelven. Si el tipo de retorno es `void`, el método no devuelve nada.

Los objetos almacenan datos en sus campos (que también tienen tipos). Se hace referencia al conjunto de todos los datos de un objeto como el estado del objeto.

Los objetos se crean a partir de las definiciones de una clase que deben escribirse en un lenguaje particular de programación. Gran parte de la programación en Java consiste en aprender a escribir definiciones de clases. Un programa grande escrito en Java puede contener muchas clases, cada una de ellas con muchos métodos que se invocan unos a otros de muchas maneras diferentes.

Para aprender a desarrollar programas en Java necesitamos aprender cómo escribir las definiciones de clase, incluyendo los campos y los métodos, y también, cómo reunir todas estas clases. El resto de este libro trata estas cuestiones.

## Términos introducidos en este capítulo

**objeto, clase, instancia, método, firma, parámetro, tipo, estado, código fuente, valor de retorno, compilador**

### Resumen de conceptos

- **objeto** Los objetos Java modelan los objetos del dominio de un problema.
- **clase** Los objetos se crean a partir de las clases. La clase describe la categoría del objeto; los objetos representan instancias individuales de la clase.
- **método** Podemos comunicarnos con los objetos invocando sus métodos. Generalmente, los objetos hacen algo cuando invocamos un método.
- **parámetro** Los métodos pueden tener parámetros para aportar información adicional para realizar una tarea.
- **signatura** El encabezado de un método se denomina su signatura. Proporciona la información necesaria para invocar dicho método.
- **tipo** Los parámetros tienen tipos. El tipo define la clase de valor que un parámetro puede tomar.
- **instancias múltiples** Se pueden crear muchos objetos similares a partir de una sola clase.
- **estado** Los objetos tienen un estado. El estado está representado por los valores almacenados en los campos.
- **llamar métodos** Los objetos se pueden comunicar invocando los métodos de cada uno de los otros objetos.
- **código fuente** El código de una clase determina la estructura y el comportamiento (los campos y los métodos) de cada uno de los objetos de dicha clase.
- **resultado** Los métodos pueden devolver información de un objeto mediante valores de retorno.

**Ejercicio 1.26** En este capítulo hemos mencionado los tipos de dato `int` y `String`. Java tiene más tipos de datos predefinidos. Averigüe cuáles son y para qué se usan. Para hacerlo puede recurrir al Apéndice B o buscar en

otro libro de Java o en un manual online sobre lenguaje Java. Uno de estos manuales es:

<http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html>

**Ejercicio 1.27** ¿Cuál es el tipo de los siguientes valores?

0  
\_holo\_  
101  
-1  
true  
\_33\_  
3.1415

**Ejercicio 1.28** Para agregar a un objeto círculo un nuevo campo, por ejemplo de nombre `nombre`, ¿qué debe hacer?

**Ejercicio 1.29** Escriba la firma de un método de nombre `enviar` que tiene un parámetro de tipo `String` y que no retorna ningún valor.

**Ejercicio 1.30** Escriba la firma de un método de nombre `promedio` que tiene dos parámetros, ambos de tipo `int`, y que retorna un valor `int`.

**Ejercicio 1.31** Vea el libro que está leyendo en este momento, ¿es un objeto o una clase? Si es una clase, mencione algunos objetos; si es un objeto, mencione su clase.

**Ejercicio 1.32** ¿Puede un objeto provenir de diferentes clases? Discútalo.



## CAPÍTULO

# 2

# Comprender las definiciones de clases

Principales conceptos que se abordan en este capítulo:

- campos
- métodos (de acceso y de modificación)
- constructores
- asignación y sentencia condicional
- parámetros

Construcciones Java que se abordan en este capítulo

campo, constructor, comentario, parámetro, asignación (=), bloque, sentencia return, void, operadores de asignación compuestos (+ =,- =), sentencia if

En este capítulo nos internamos por primera vez en el código fuente de una clase. Discutiremos sobre los elementos básicos de las definiciones de una clase: *campos*, *constructores* y *métodos*. Los métodos contienen sentencias, e inicialmente vemos métodos que sólo contienen sentencias aritméticas sencillas y sentencias de impresión. Más adelante introducimos las *sentencias condicionales* que permiten realizar elecciones entre las diferentes acciones que llevan a cabo los métodos.

Comenzaremos examinando un nuevo proyecto que contiene una cantidad adecuada de detalle. Este proyecto representa una implementación simplificada de una máquina expendedora de boletos automatizada. Cuando empecemos a introducir la mayoría de las características básicas de las clases, encontraremos rápidamente que esta implementación es deficiente de diversas maneras, de modo que luego procederemos a describir una versión más sofisticada de la máquina expendedora de boletos, que representa una mejora significativa de la misma. Finalmente, y con el objetivo de reforzar los conceptos introducidos en este capítulo, daremos una mirada al interior del ejemplo *curso-de-laboratorio* que ya encontramos en el Capítulo 1.

## 2.1

### Máquina expendedora de boletos

Las estaciones de tren a menudo tienen máquinas que imprimen un boleto cuando un cliente introduce en ella el dinero correspondiente a su tarifa. En este capítulo definiremos una clase que modela algo similar a estas máquinas. Como estaremos entrando en el interior de nuestras primeras clases de ejemplo en Java, para comenzar manten-

dremos nuestra simulación lo suficientemente simple, lo que nos dará la oportunidad de hacer algunas preguntas sobre cómo estos modelos difieren de las versiones del mundo real y cómo podríamos cambiar nuestras clases para que los objetos que ellas crean se parezcan más a las cosas reales.

Nuestras máquinas trabajan con clientes que introducen dinero en ella y luego le solicitan que imprima un boleto. La máquina mantiene un registro de la cantidad de dinero que ha recaudado durante todo su funcionamiento. En la vida real, es frecuente que la máquina expendedora de boletos ofrezca un conjunto de boletos de diferentes tipos y los clientes escogen entre ellos, sólo el que desean. Nuestra máquina simplificada imprime sólo boletos de un único precio. Resulta significativamente más complicado programar una clase que sea capaz de emitir boletos de diferentes valores que si tienen un único precio. Por otra parte, con programación orientada a objetos es muy fácil crear varias instancias de la clase, cada una con su propio precio, para cumplir con la necesidad de diferentes tipos de boletos.

### 2.1.1 Explorar el comportamiento de una máquina expendedora de boletos ingenua

Abra en BlueJ el proyecto *maquina-de-boletos-simple*. Este proyecto contiene sólo una clase, *MaquinaDeBoletos*, que podrá explorar de manera similar a los ejemplos discutidos en el Capítulo 1. Cuando cree una instancia de *MaquinaDeBoletos*, le pedirá que ingrese un número que corresponde al precio de los boletos que emitirá esta máquina en particular. Este número refleja la cantidad de centavos del precio, por lo que resulta apropiado como valor para trabajar un número entero positivo, por ejemplo 500.

**Ejercicio 2.1** Cree un objeto *MaquinaDeBoletos* en el banco de objetos y observe sus métodos. Podrá ver los siguientes métodos: *obtenerSaldo*, *obtenerPrecio*, *ingresarDinero* e *imprimirBoleto*. Pruebe el método *obtenerPrecio*. Verá un valor de retorno que contiene el precio de los boletos que se determinó cuando se creó este objeto. Use el método *ingresarDinero* para simular que coloca una cantidad de dinero en la máquina y luego use *obtenerSaldo* para controlar que la máquina registró la cantidad introducida. Puede ingresar sucesivamente varias cantidades de dinero en la máquina, como si colocara varias monedas o billetes en una máquina real. Pruebe ingresar la cantidad exacta de dinero requerida para un boleto. Como esta es una máquina simplificada, el boleto no se imprimirá automáticamente, de modo que una vez que haya ingresado dinero suficiente, llame al método *imprimirBoleto*. Se emitirá en la ventana terminal de BlueJ un facsímil del boleto.

**Ejercicio 2.2** ¿Qué valor aparece si controla el saldo de la máquina después de que se imprimió el boleto?

**Ejercicio 2.3** Experimente ingresando diferentes cantidades de dinero antes de emitir los boletos. ¿Observa algo extraño en el comportamiento de la máquina? ¿Qué ocurre si ingresa demasiado dinero en la máquina? ¿Recibe algún reintegro? ¿Qué ocurre si no coloca dinero suficiente y luego prueba emitir un boleto?

**Ejercicio 2.4** Trate de comprender bien el comportamiento de la máquina interactuando con ella en el banco de objetos antes de comenzar a ver cómo está implementada la clase `MaquinaDeBoletos` en la próxima sección.

**Ejercicio 2.5** Cree otra máquina que opere con boletos de un precio diferente. Compre un boleto a esta máquina. El boleto que emite, ¿tiene un aspecto diferente del anterior?

## 2.2

## Examinar una definición de clase

El examen del comportamiento de los objetos `MaquinaDeBoletos` en BlueJ revela que sólo se comportan de la manera que esperamos si ingresamos la cantidad exacta de dinero que corresponde al precio de un boleto. Podremos comenzar a ver por qué ocurre esto, cuando exploremos los detalles internos de la clase en esta sección.

Entre al código de la clase `MaquinaDeBoletos` haciendo doble clic sobre su ícono en el diagrama de clases. Verá algo similar a la Figura 2.1.

**Figura 2.1**

Ventana del editor de BlueJ

```

* Tambien asume que los usuarios ingresan cantidades que tienen sentido.
*
* @author David J. Barnes and Michael Kolling
* @version 2006.03.30
*/
public class MaquinaDeBoletos
{
    // El precio de un boleto de esta máquina.
    private int precio ;
    // La cantidad de dinero ingresada hasta ahora por un cliente.
    private int saldo;
    // La cantidad total de dinero recolectada por esta máquina.
    private int total;

    /**
     * Crea una máquina que vende boletos de un determinado precio.
     * Observe que el precio debe ser mayor que cero y que no hay
     * controles que aseguren esto.
     */
    public MaquinaDeBoletos(int precioDelBoleto)
    {
        precio = precioDelBoleto;
        saldo = 0;
    }
}

```

El texto completo de la clase se muestra en Código 2.1. Viendo el texto de la definición de la clase parte por parte podremos analizar algunos de los conceptos de orientación a objetos sobre los que hemos hablado en el Capítulo 1.

**Código 2.1**

La clase  
MaquinaDeBoletos

```
/*
 * MaquinaDeBoletos modela una máquina de boletos
 * simplificada
 * e ingenua que trabaja con boletos de tarifa plana.
 * El precio de un boleto se especifica mediante el
 * constructor.
 * Es una máquina ingenua en el sentido de que confía en
 * que los
 * usuarios introducen la cantidad de dinero necesaria antes
 * de imprimir un boleto.
 * También asume que los usuarios ingresan cantidades que
 * tienen
 * sentido.
 *
 * @author David J. Barnes and Michael Kolling
 * @version 2006.03.30
 */
public class MaquinaDeBoletos
{
    // El precio de un boleto de esta máquina.
    private int precio ;
    // La cantidad de dinero ingresada hasta ahora por un
    // cliente.
    private int saldo;
    // La cantidad total de dinero recolectada por esta
    // máquina.
    private int total;
    /**
     * Crea una máquina que vende boletos de un
     * determinado precio.
     * Observe que el precio debe ser mayor que cero y
     * que no hay
     * controles que aseguren esto.
     */
    public MaquinaDeBoletos(int precioDelBoleto)
    {
        precio = precioDelBoleto;
        saldo = 0;
        total = 0;
    }
    /**
     * Devuelve el precio de un boleto.
     */
    public int obtenerPrecio()
    {
        return precio;
    }
}
```

**Código 2.1  
(continuación)**

La clase  
MaquinaDeBoletos

```

        * Devuelve la cantidad de dinero que ya se ingresó
        para
        * el siguiente boleto.
    */
public int obtenerSaldo()
{
    return saldo;
}
/**
 * Recibe del cliente una cantidad de dinero en
centavos.
*/
public void ingresarDinero(int cantidad)
{
    saldo = saldo + cantidad;
}
/**
 * Imprime un boleto.
 * Actualiza el total de dinero recolectado y
 * pone el saldo en cero.
*/
public void imprimirBoleto()
{
    // Simula la impresión de un boleto.
    System.out.println("#####");
    System.out.println("# Línea Blue ");
    System.out.println("# Boleto");
    System.out.println("# " + precio + " cvos.");
    System.out.println("#####");
    System.out.println();
    // Actualiza el total recaudado con el saldo.
    total = total + saldo;
    // Limpia el saldo.
    saldo = 0;
}

```

## 2.3

## Campos, constructores y métodos

El código de la mayoría de las clases puede descomponerse en dos partes principales: una envoltura exterior pequeña que simplemente da nombre a la clase y una parte interna mucho más grande que hace todo el trabajo. En este caso, la envoltura exterior es la siguiente:

```

public class MaquinaDeBoletos
{
    Se omite la parte interna de la clase
}

```

La envoltura exterior de las diferentes clases es muy parecida, su principal finalidad es proporcionar un nombre a la clase.

**Ejercicio 2.6** Escriba la envoltura exterior de las clases `Estudiante` y `CursoDeLaboratorio` tal como piense que deberían ser, no se preocupe por la parte interna.

**Ejercicio 2.7** ¿Tiene importancia si escribimos

```
public class MaquinaDeBoletos
o
class Public MaquinaDeBoletos
```

en la parte exterior de la clase?

Edite el código de la clase `MaquinaDeBoletos` para probar las dos formas anteriores y cierre la ventana del editor. ¿Observa algún cambio en el diagrama de clases?

¿Qué mensaje de error aparece cuando presiona el botón *Compile*? ¿Considera que este mensaje explica claramente cuál es el error?

**Ejercicio 2.8** Verifique si es posible quitar la palabra `public` de la parte exterior de la clase `MaquinaDeBoletos`.

La parte interna de la clase es el lugar en el que definimos los campos, los constructores y los métodos que dan a los objetos de la clase sus características particulares y su comportamiento. Podemos resumir las características esenciales de estos tres componentes de una clase como sigue:

- Los campos almacenan datos para que cada objeto los use.
- Los constructores permiten que cada objeto se prepare adecuadamente cuando es creado.
- Los métodos implementan el comportamiento de los objetos.

En Java existen muy pocas reglas sobre el orden que se puede elegir para definir los campos, los constructores y los métodos dentro de una clase. En la clase `MaquinaDeBoletos` hemos elegido listar primero los campos, segundo los constructores y por último los métodos (Código 2.2). Este es el orden que seguiremos en todos nuestros ejemplos. Otros autores eligen adoptar diferentes estilos y esto es, mayormente, una cuestión de preferencia. Nuestro estilo no es necesariamente mejor que el de otros. Sin embargo, es importante elegir un estilo y luego usarlo de manera consistente, porque de este modo las clases serán más fáciles de leer y de comprender.

### Código 2.2

Nuestro orden de campos, constructores y métodos

```
public class NombreDeClase
{
    Campos
    Constructores
    Métodos
}
```

**Ejercicio 2.9** Como consecuencia de su temprana experimentación en BlueJ con los objetos de la máquina expendedora de boletos, probablemente recuerde los nombres de algunos de los métodos, por ejemplo `imprimirBoleto`. Observe

la definición de clase en el Código 2.1 y utilice el conocimiento que ha adquirido junto con la información adicional sobre el orden que hemos dado, para hacer una lista de los nombres de los campos, los constructores y los métodos de la clase `MaquinaDeBoletos`. *Pista:* hay un solo constructor en la clase.

**Ejercicio 2.10** ¿Observa algún aspecto del constructor que lo haga significativamente diferente de los otros métodos de la clase?

### 2.3.1 Campos

#### Concepto

Los **campos** almacenan datos para que un objeto los use. Los campos también son conocidos como *variables de instancia*.

La clase `MaquinaDeBoletos` tiene tres campos: `precio`, `saldo` y `total`. Los campos también son conocidos como *variables de instancia*. Los hemos definido al comienzo de la definición de la clase (Código 2.3). Todos los campos están asociados a los temas monetarios con los que trabaja la máquina expendedora de boletos:

- El campo `precio` almacena el precio de un boleto.
- El campo `saldo` almacena la cantidad de dinero ingresada por el usuario en la máquina antes de pedir la impresión de un boleto.
- El campo `total` guarda un registro de la cantidad total de dinero ingresado en la máquina por todos los usuarios desde que el objeto máquina fue construido.

#### Código 2.3

Los campos de la clase  
`MaquinaDeBoletos`

```
public class MaquinaDeBoletos
{
    private int precio;
    private int saldo;
    private int total;
```

*Se omitieron el constructor y los métodos.*

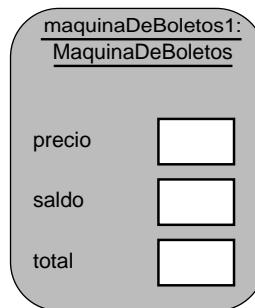
Los campos son pequeñas cantidades de espacio dentro de un objeto que pueden usarse para almacenar valores. Todo objeto, una vez creado, dispondrá de un espacio para cada campo declarado en su clase. La Figura 2.2 muestra un diagrama que representa un objeto máquina de boletos con sus tres campos. Los campos aún no tienen valores asignados; una vez que los tengan, podemos escribir cada valor dentro de la caja que representa al campo. La notación es similar a que se usa en BlueJ para mostrar los objetos en el banco de objetos, excepto que aquí mostramos un poco más de detalle. En BlueJ, por razones de espacio, los campos no se muestran en el ícono del objeto, sin embargo, podemos verlos abriendo la ventana del inspector de objetos.

Cada campo tiene su propia declaración en el código. En la definición de la clase, arriba de cada una de estas líneas hemos agregado una línea de texto, *un comentario*, para beneficio de los lectores humanos:

```
// El precio de un boleto de esta máquina.
private int precio;
```

**Figura 2.2**

Un objeto de la clase  
MaquinaDeBoletos

**Concepto**

Los **comentarios** se insertan en el código de una clase para proporcionar explicaciones a los lectores humanos. No tienen ningún efecto sobre la funcionalidad de la clase.

Se introduce una sola línea de comentario mediante los dos caracteres «//» que se escriben sin espacios entre ellos. Los comentarios más detallados, que frecuentemente ocupan varias líneas, se escriben generalmente en la forma de comentarios multilínea: comienzan con el par de caracteres «/\*» y terminan con el par «\*/». Hay un buen ejemplo de este tipo de comentarios antes del encabezado de la clase en el Código 2.1.

Las definiciones de los tres campos son bastante similares:

- Todas las definiciones indican que son campos *privados* (*private*) del objeto; hablaremos más sobre su significado en el Capítulo 5, pero por el momento, simplemente diremos que siempre definimos los campos como privados.
- Los tres campos son de tipo *int*. Esto indica que cada campo puede almacenar un número entero, cuestión que resulta razonable dado que deseamos que almacenen números que representan cantidades de dinero en centavos.

Puesto que los campos pueden almacenar valores que pueden variar a lo largo del tiempo, se les conoce como *variables*. El valor almacenado en un campo puede ser cambiado, si se desea. Por ejemplo, cuando se introduce más dinero en la máquina queremos que se modifique el valor almacenado en el campo *saldo*. En las siguientes secciones encontraremos otras categorías de variables además de los campos.

Los campos *precio*, *saldo* y *total* son todos los datos que necesita el objeto máquina para cumplir su rol de recibir dinero de un cliente, emitir boletos y mantener actualizado el total de dinero que ha sido introducido en ella. En las siguientes secciones veremos cómo el constructor y los métodos usan estos campos para implementar el comportamiento de la máquina expendedora de boletos ingenua.

**Ejercicio 2.11** ¿De qué tipo considera que es cada uno de los siguientes campos?

```

private int cantidad;
private Estudiante representante;
private Servidor host;
  
```

**Ejercicio 2.12** ¿Cuáles son los nombres de los siguientes campos?

```

private boolean vive;
private Persona tutor;
private Juego juego;
  
```

**Ejercicio 2.13** En la siguiente declaración de campo que está en la clase `MaquinaDeBoletos`

```
private int precio;
```

¿Tiene importancia el orden en que aparecen las tres palabras? Edite la clase `MaquinaDeBoletos` para probar los diferentes órdenes. Cierre el editor después de cada cambio. La apariencia del diagrama de clases después de cada cambio, ¿le da alguna clave sobre cuáles son los órdenes posibles? Verifique su respuesta presionando el botón *Compile* para ver si existe algún mensaje de error.

¡Asegúrese de reinstalar la versión original después de sus experimentaciones!

**Ejercicio 2.14** ¿Es necesario que cada declaración de campo siempre finalice con un punto y coma? Experimente una vez más usando el editor. La regla que aprenderá aquí es muy importante, por lo que asegúrese de recordarla.

**Ejercicio 2.15** Escriba la declaración completa de un campo cuyo tipo es `int` y cuyo nombre es `estado`.

### 2.3.2 Constructores

#### Concepto

Los **constructores** permiten que cada objeto sea preparado adecuadamente cuando es creado.

Los constructores de una clase tienen un rol especial que cumplir: su responsabilidad es poner cada objeto de esa clase en un estado adecuado para que pueda ser usado una vez que haya sido creado. Esta operación se denomina *inicialización*. El constructor inicializa el objeto en un estado razonable. El Código 2.4 muestra el constructor de la clase `MaquinaDeBoletos`.

Uno de los rasgos distintivos de los constructores es que tienen el mismo nombre que la clase en la que son definidos, en este caso `MaquinaDeBoletos`.

#### Código 2.4

El constructor de la clase `MaquinaDeBoletos`

```
public class MaquinaDeBoletos
{
    Se omitieron los campos
    /**
     * Crea una máquina que vende boletos de un
     * determinado precio.
     * Observe que el precio debe ser mayor que cero
     * y que no hay
     * controles que aseguren esto.
    */
    public MaquinaDeBoletos (int precioDelBoleto)
    {
        precio = precioDelBoleto;
        saldo = 0;
        total = 0;
    }
    Se omitieron los métodos
}
```

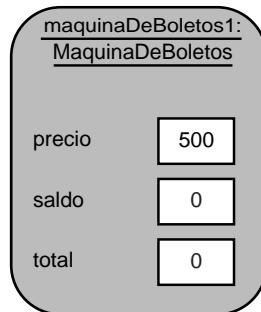
Los campos del objeto se inicializan en el constructor. A algunos campos, tales como `saldo` y `total`, se les puede poner un valor inicial que tenga sentido asignando un valor constante, en este caso, cero. Con otros campos, tal como ocurre con el precio del boleto, no resulta tan simple elegir este valor inicial ya que no conocemos el precio de los boletos de una máquina en particular hasta que la máquina esté construida: recuerde que deseamos crear varios objetos máquina para vender boletos de diferentes precios, por lo que no será correcto para todos los casos ningún precio inicial. Recordará que al experimentar en BlueJ con la creación de objetos `MaquinaDeBoletos` tuvo que ingresar el costo del boleto cada vez que creaba una nueva máquina. Un punto importante para destacar aquí es que el precio de un boleto se determina, en un principio, *fuera* de la máquina, y luego debe ser *pasado dentro* del objeto máquina. En BlueJ usted decide el valor del boleto y lo ingresa en una caja de diálogo. Una tarea del constructor es recibir este valor y almacenarlo en el campo `precio` de la nueva máquina creada de modo que la máquina pueda recordar dicho valor sin que usted tenga que tenerlo en mente. Podemos ver que uno de los papeles más importantes de un campo es recordar información, de modo que esté disponible para un objeto durante toda la vida del mismo.

La Figura 2.3 muestra un objeto máquina de boletos después de que se haya ejecutado su constructor. Los valores han sido asignados a los campos. A partir de este diagrama podemos decir que la máquina fue creada al pasar el número 500 como el valor del precio del boleto.

En la próxima sección hablaremos sobre cómo hace un objeto para recibir estos valores desde el exterior.

**Figura 2.3**

Un objeto  
`MaquinaDeBoletos`  
después de su  
inicialización (creado  
para boletos de 500  
centavos)



**Nota:** en Java, todos los campos son inicializados automáticamente con un valor por defecto, si es que no están inicializados explícitamente. El valor por defecto para los campos enteros es 0. Por lo que hablando estrictamente, podríamos trabajar sin asignar el valor 0 a los campos `saldo` y `total`, confiando en que el valor por defecto o predefinido dará el mismo resultado. Sin embargo, preferimos escribir explícitamente las asignaciones. No hay ninguna desventaja en hacer esto y sirve para documentar lo que está ocurriendo realmente. No esperamos que el lector de la clase conozca cuál es el valor por defecto y documentamos que realmente queremos que este valor sea 0 y no que hemos olvidado inicializarlo.

## 2.4

# Pasar datos mediante parámetros

La manera en que los constructores y los métodos reciben valores es mediante sus *parámetros*. Recuerde que hemos hablado brevemente sobre los parámetros en el Capítulo 1. Los parámetros se definen en el encabezado de un constructor o un método:

```
public MaquinaDeBoletos (int precioDelBoleto)
```

Este constructor tiene un solo parámetro, *precioDelBoleto*, que es de tipo *int*, del mismo tipo que el campo *precio* que se usará para determinar el precio del boleto. La Figura 2.4 ilustra cómo se pasan los valores mediante parámetros. En este caso, un usuario de BlueJ ingresa un valor en la caja de diálogo cuando crea una nueva máquina (se muestra a la izquierda), y ese valor luego es copiado dentro del parámetro *precioDelBoleto* del constructor de la nueva máquina (se ilustra con la flecha A). La caja que presenta el objeto máquina de la Figura 2.4, titulada «*MaquinaDeBoletos (constructor)*» es el espacio adicional para el objeto, que se crea solamente cuando se ejecuta el constructor: lo llamaremos el *espacio del constructor* del objeto (o *espacio del método* cuando hablemos sobre métodos en lugar de constructores, ya que la situación es la misma). El espacio del constructor se usa para proporcionar lugar para almacenar los valores de los parámetros del constructor (y todas las variables que vendrán más adelante).

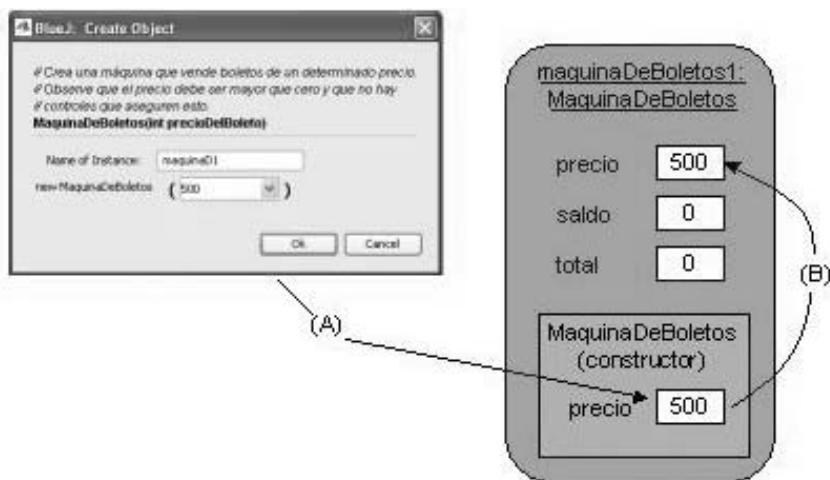
### Concepto

El **alcance** de una variable define la sección de código en la que la variable puede ser accedida.

Distinguimos entre nombres de los parámetros dentro de un constructor o un método, y valores de los parámetros fuera de un constructor o un método: hacemos referencia a los nombres como *parámetros formales* y a los valores como *parámetros actuales*. Por lo tanto *precioDelBoleto* es un parámetro formal y el valor ingresado por el usuario, por ejemplo 500, es un parámetro actual. Puesto que permiten almacenar valores, los parámetros formales constituyen otra clase de variables. En nuestros diagramas, todas las variables se representan mediante cajas blancas.

**Figura 2.4**

(A) Pasaje de parámetro y  
(B) asignación



Un parámetro formal está disponible para un objeto sólo dentro del cuerpo del constructor o del método que lo declara. Decimos que el *alcance* de un parámetro está restringido al cuerpo del constructor o del método en el que es declarado. En cambio, el alcance de un campo es toda la clase y puede ser accedido desde cualquier lugar en la misma clase.

**Concepto**

El **tiempo de vida** de una variable describe cuánto tiempo continuará existiendo la variable antes de ser destruida.

Un concepto relacionado con el alcance de una variable es el *tiempo de vida* de la variable. El tiempo de vida de un parámetro se limita a una sola llamada de un constructor o método. Una vez que completó su tarea, los parámetros formales desaparecen y se pierden los valores que contienen. En otras palabras, cuando un constructor termina su ejecución, se elimina el espacio del constructor (véase Figura 2.4) junto con las variables parámetro que contiene.

Por el contrario, el tiempo de vida de un campo es el mismo tiempo de vida que el del objeto al que pertenece. En conclusión, si queremos recordar el costo de los boletos contenido por el parámetro `precioDelBoleto`, debemos guardar su valor en algún lugar más persistente, esto es, en el campo `precio`.

**Ejercicio 2.16** ¿A qué clase pertenece el siguiente constructor?

```
public Estudiante (String nombre)
```

**Ejercicio 2.17** ¿Cuántos parámetros tiene el siguiente constructor y cuáles son sus tipos?

```
public Libro (String titulo, double precio)
```

**Ejercicio 2.18** ¿Puede suponer de qué tipo serán algunos de los campos de la clase `Libro`? ¿Puede asumir algo respecto de los nombres de estos campos?

**2.5**

## Asignación

En la sección anterior destacamos la necesidad de almacenar el valor de corta vida de un parámetro dentro de algún lugar más permanente, un campo. Para hacer esto, el cuerpo del constructor contiene la siguiente *sentencia de asignación*:

```
precio = precioDelBoleto;
```

**Concepto**

Las **sentencias de asignación** almacenan el valor representado por el lado derecho de la sentencia en una variable nombrada a la izquierda.

Se reconocen las sentencias de asignación por la presencia de un operador de asignación, como es el signo `<=` en el ejemplo anterior. Las sentencias de asignación funcionan tomando el valor de lo que aparece del lado derecho del operador y copiando dicho valor en una variable ubicada en el lado izquierdo. En la Figura 2.4 ilustramos esta operación con la flecha B. La parte de la derecha se denomina una *expresión*: las expresiones son cosas que la computadora puede evaluar. En este caso, la expresión consiste en una sola variable pero veremos más adelante en este capítulo algunos ejemplos de expresiones más complicadas que contienen operaciones aritméticas. Una regla sobre las sentencias de asignación es que el tipo de una expresión debe coincidir con el tipo de la variable a la que es asignada. Hasta ahora hemos encontrado tres tipos diferentes: `int`, `String` y muy brevemente, `boolean`. Esta regla significa que, por ejemplo, no tenemos permitido almacenar una expresión de tipo entero en una variable de tipo cadena. La misma regla se aplica también entre los parámetros formales y los parámetros actuales: el tipo de una expresión de un parámetro actual debe coincidir con el tipo de una variable parámetro formal. Por ahora, podemos decir que ambos parámetros deben ser del mismo tipo, aunque veremos en capítulos posteriores que esto no es totalmente cierto.

**Ejercicio 2.19** Suponga que la clase `Mascota` tiene un campo denominado nombre de tipo `String`. Escriba una sentencia de asignación en el cuerpo del siguiente constructor, de modo que el campo nombre se inicialice con el valor del parámetro del constructor.

```
public Mascota (String nombreMascota)
{
    ...
}
```

**Ejercicio 2.20** Desafío ¿Cuál es el error en la siguiente versión del constructor de la clase `MaquinaDeBoletos`?

```
public MaquinaDeBoletos(int precioDelBoleto)
{
    int precio = precioDelBoleto;
    saldo = 0;
    total = 0;
}
```

Una vez que haya resuelto el problema, pruebe esta versión en el proyecto `maquina-de-boletos-simple`. ¿Compila esta versión? Cree un objeto e inspeccione sus campos. ¿Observa algún error en el valor del campo `precio`? ¿Qué explicación puede dar?

## 2.6

## Métodos de acceso

La clase `MaquinaDeBoletos` tiene cuatro métodos: `obtenerPrecio`, `obtenerSaldo`, `ingresarDinero` e `imprimirBoleto`. Comenzaremos por ver el código de los métodos considerando el método `obtenerPrecio` (Código 2.5).

### Código 2.5

El método  
`obtenerPrecio`

```
public class MaquinaDeBoletos
{
    Se omitieron los campos.
    Se omitieron los constructores.
    /**
     * Devuelve el precio de un boleto.
     */
    public int obtenerPrecio()
    {
        return precio;
    }
    Se omitieron los restantes métodos.
}
```

Los métodos tienen dos partes: un encabezado y un cuerpo. A continuación mostramos el encabezado del método `obtenerPrecio`:

```
/**
 * Devuelve el precio de un boleto.
```

**Concepto**

Los métodos se componen de dos partes: un encabezado y un cuerpo.

```
 */
public int obtenerPrecio()
```

Las tres primeras líneas conforman un comentario que describe qué hace el método. La cuarta línea es conocida también como la *signatura del método*<sup>1</sup>. Es importante distinguir entre signatura del método y declaración de campos porque son muy parecidos. Podemos decir que `obtenerPrecio` es un método y no un campo porque está seguido de un par de paréntesis: «( « y »)». Observe también que no hay un punto y coma al final de la signatura.

El cuerpo del método es la parte restante del método, que aparece a continuación del encabezado. Está siempre encerrado entre llaves: «{ « y »}». Los cuerpos de los métodos contienen las *declaraciones* y las *sentencias* que definen qué ocurre dentro de un objeto cuando es invocado ese método. En nuestro ejemplo anterior, el cuerpo del método contiene una sola sentencia, pero veremos rápidamente muchos ejemplos en los que el cuerpo del método consta de varias líneas de declaraciones y sentencias.

Cualquier conjunto de declaraciones y sentencias, ubicado entre un par de llaves, es conocido como un *bloque*. Por lo que el cuerpo de la clase `MaquinaDeBoletos` y los cuerpos de todos los métodos de la clase son bloques.

Existen, por lo menos, dos diferencias significativas entre las signaturas del constructor `MaquinaDeBoletos` y del método `obtenerPrecio`:

```
public MaquinaDeBoletos (int precioDelBoleto)
public int obtenerPrecio()
```

- El método tiene un *tipo de retorno* `int` pero el constructor no tiene tipo de retorno. El tipo de retorno se escribe exactamente antes del nombre del método.
- El constructor tiene un solo parámetro formal, `precioDelBoleto`, pero el método no tiene ninguno, sólo un par de paréntesis vacíos.

Es una regla de Java que el constructor no puede tener ningún tipo de retorno. Por otro lado, tanto los constructores como los métodos pueden tener cualquier número de parámetros formales, inclusive pueden no tener ninguno.

En el cuerpo de `obtenerPrecio` hay una sola sentencia:

```
return precio;
```

Esta es una *sentencia return* y es la responsable de devolver un valor entero que coincide con el tipo de retorno `int` de la signatura del método. Cuando un método contiene una sentencia `return`, siempre es la última sentencia del mismo porque una vez que se ejecutó esta sentencia no se ejecutarán más sentencias en el método.

El tipo de retorno `int` de `obtenerPrecio` es una forma de prometer que el cuerpo del método hará algo que resulte finalmente un valor entero que haya sido calculado y retornado como resultado del método. Podría pensar en la llamada a un método como si fuera una manera de preguntar algo a un objeto, y el valor de retorno del método sería la respuesta del objeto a dicha pregunta. En este caso, cuando se invoque el método `obtenerPrecio` de una máquina de boletos, la pregunta equivalente es, ¿cuál es el costo del boleto? Una máquina de boletos no necesita realizar ningún cálculo

---

<sup>1</sup> Esta definición difiere ligeramente de la definición más formal de la especificación del lenguaje Java donde la signatura no incluye al modificador de acceso ni al tipo de retorno.

para ser capaz de responder esta pregunta porque mantiene la respuesta en su campo **precio**, por lo tanto, el método responde devolviendo justamente el valor de esa variable. A medida que desarrollemos clases más complejas encontraremos inevitablemente preguntas más complejas que requieren más trabajo para brindar sus respuestas.

### Concepto

Los **métodos de acceso** devuelven información sobre el estado de un objeto.

Frecuentemente describimos a métodos tales como los dos métodos obtener de la **MaquinaDeBoletos** (**obtenerPrecio** y **obtenerSaldo**) como *métodos de acceso*. El motivo de mencionarlos de esta manera es que devuelven información al invocador sobre el estado de un objeto, es decir, proporcionan acceso a dicho estado. Un método de acceso contiene generalmente una sentencia **return** para devolver información como un valor en particular.

**Ejercicio 2.21** Compare el método **obtenerSaldo** con el método **obtenerPrecio**. ¿Cuáles son las diferencias entre ellos?

**Ejercicio 2.22** Si una llamada a **obtenerPrecio** puede ser caracterizada por la pregunta ¿cuánto cuesta el boleto?, ¿cómo podría caracterizar una llamada a **obtenerSaldo**?

**Ejercicio 2.23** Si se cambia el nombre de **obtenerSaldo** por **obtenerDineroIngresado**, ¿es necesario modificar la sentencia **return** en el cuerpo del método? Pruebe este cambio en **BlueJ**.

**Ejercicio 2.24** Defina un método de acceso, **obtenerTotal**, que devuelva el valor del campo **total**.

**Ejercicio 2.25** Pruebe eliminar la sentencia **return** del cuerpo del método **obtenerPrecio**. ¿Qué mensaje de error aparece cuando trata de compilar la clase?

**Ejercicio 2.26** Compare las signaturas de los métodos **obtenerPrecio** e **imprimirBoleto** que se muestran en el Código 2.1. Además de sus nombres, ¿cuál es la principal diferencia entre ellas?

**Ejercicio 2.27** Los métodos **ingresarDinero** e **imprimirBoleto**, ¿tienen sentencias **return**? ¿Por qué considera que es así? ¿Observa algo en sus encabezados que podría sugerir el porqué no requieren sentencias **return**?

## 2.7

## Métodos de modificación

### Concepto

Los **métodos de modificación** cambian el estado de un objeto.

Los métodos **obtener** de la máquina de boletos realizan, todos ellos, tareas similares: devuelven el valor de uno de los campos del objeto. El resto de los métodos, **ingresarDinero** e **imprimirBoleto**, tienen un papel más significativo, principalmente porque *modifican* el valor de uno o más campos del objeto máquina cada vez que son invocados. A los métodos que modifican el estado de su objeto los llamamos *métodos de modificación* (o sólo *modificadores*).

De la misma manera en que pensamos en los métodos de acceso como solicitantes de información (preguntas), podemos pensar en los métodos de modificación como solicitudes a un objeto para que cambie su estado.

Un efecto distintivo de un modificador es que un objeto exhibirá con frecuencia un comportamiento ligeramente diferente antes y después de ser llamado. Podemos ilustrar esto con el siguiente ejercicio.

**Ejercicio 2.28** Cree una máquina de boletos con un precio de su elección. Primeramente llame a su método `obtenerSaldo`. Luego llame al método `ingresarDinero` (Código 2.6) e ingrese como parámetro actual una cantidad de dinero positiva y distinta de cero. Llame nuevamente a `obtenerSaldo`. Las dos llamadas a `obtenerSaldo` debieran tener diferente salida puesto que la llamada a `ingresarDinero` tuvo el efecto de cambiar el estado de la máquina mediante su campo `saldo`.

La firma de `ingresarDinero` tiene tipo de retorno `void` y un solo parámetro formal, `cantidad`, de tipo `int`. Un tipo de retorno `void` significa que el método no devuelve ningún valor cuando es llamado; es significativamente diferente de todos los otros tipos de retorno. En BlueJ, la diferencia es más notable porque después de una llamada a un método `void` no se muestra ninguna caja de diálogo con el valor devuelto. En el cuerpo de un método `void`, esta diferencia se refleja en el hecho de que no hay ninguna sentencia `return`<sup>2</sup>.

#### Código 2.6

El método

`ingresarDinero`

```
/**
 * Recibe de un cliente una cantidad de dinero en centavos.
 */
public void ingresarDinero(int cantidad)
{
    saldo = saldo + cantidad;
}
```

En el cuerpo de `ingresarDinero` hay una sola sentencia que es otra forma de sentencia de asignación. Siempre consideraremos las sentencias de asignación examinando primero los cálculos que aparecen a la parte derecha del símbolo de asignación. En este caso, el efecto es calcular un valor que es la suma del número del parámetro `cantidad` con el número del campo `saldo`. Este valor es calculado y luego asignado al campo `saldo`, por lo que el efecto de esta sentencia es incrementar el valor de `saldo` en el valor de `cantidad`<sup>3</sup>.

**Ejercicio 2.29** ¿Qué elementos del encabezado de `ponerPrecio` nos indican que es un método y no un constructor?

```
public void ponerPrecio (int precioDelBoleto)
```

<sup>2</sup> En realidad, Java permite que los métodos `void` contengan una forma especial de sentencia de retorno en la que no se devuelve ningún valor. Esta sentencia toma la forma  
`return;`

y simplemente hace que el método finalice sin ejecutar ninguna línea más de código.

<sup>3</sup> El sumar una cantidad al valor de una variable es algo tan común que existe un **operador de asignación compuesto**, especial para hacerlo: «`+=`». Por ejemplo:  
`saldo += cantidad;`

**Ejercicio 2.30** Complete el cuerpo del método `ponerPrecio` de modo que asigne el valor de su parámetro al campo `precio`.

**Ejercicio 2.31** Complete el cuerpo del siguiente método cuyo propósito es sumar el valor de su parámetro al campo de nombre `puntaje`.

```
/**
 * Incrementa el puntaje en un número de puntos dado
 */
public void incrementar (int puntos)
{
    ...
}
```

**Ejercicio 2.32** Complete el siguiente método cuyo propósito es restar el valor de su parámetro del campo de nombre `precio`.

```
/**
 * Disminuye el precio en una cantidad dada
 */
public void descuento (int cantidad)
{
    ...
}
```

#### **Nota: convenciones Java sobre métodos de acceso y de modificación7**

En Java, los nombres de los métodos de acceso suelen comenzar con la palabra «get» en lugar de la palabra «obtener» y los nombres de los métodos de modificación, con la palabra «set» en lugar de «poner».

Por ejemplo:

`getPrecio`, `getSaldo` son métodos de acceso a las variables `precio` y `saldo`.

`setPrecio`, `setSaldo` son métodos de modificación de las variables `precio` y `saldo`.

De aquí en adelante, usaremos esta convención para los nombres de los métodos de modificación y de acceso.

## 2.8

## Imprimir desde métodos

El Código 2.7 muestra el método más complejo de la clase, `imprimirBoleto`. Para ayudarle a comprender la siguiente discusión, asegúrese de haber invocado este método en una máquina de boletos. Debiera ver algo similar a la siguiente ventana terminal de BlueJ.

```
#####
# Línea BlueJ
# Boleto
# 500 cvos.
#####
```

**Código 2.7**

El método  
imprimirBoleto

```
/*
 * Imprime un boleto y pone el saldo actual en cero
 */
public void imprimirBoleto()
{
    // Simula la impresión de un boleto.
    System.out.println("#####");
    System.out.println("# Línea BlueJ");
    System.out.println("# Boleto");
    System.out.println("# " + precio + " cvos.");
    System.out.println("#####");
    System.out.println();
    // Actualiza el total recaudado con el saldo.
    total = total + saldo;
    // Limpia el saldo.
    saldo = 0;
}
```

Este es el método más largo que hemos visto hasta ahora, por lo que lo dividiremos en partes más manejables:

- La firma indica que el método tiene un tipo de retorno `void` y que no tiene parámetros.
- El cuerpo contiene ocho sentencias además de los comentarios asociados.
- Las primeras seis sentencias son las responsables de imprimir lo que se ve en la terminal de BlueJ.
- La séptima sentencia suma el dinero ingresado por el cliente (a través de llamadas previas a `ingresarDinero`) al total del dinero recolectado por la máquina desde que fue creada.
- La octava sentencia vuelve el saldo al valor 0 con una sentencia básica de asignación, y prepara la máquina para el próximo cliente que introducirá dinero en ella.

Comparando la salida que aparece con las sentencias que la producen, es fácil ver que una sentencia como

```
System.out.println("# Línea BlueJ");
```

imprime literalmente la cadena que aparece entre el par de comillas dobles. Todas estas sentencias de impresión son invocaciones al método `println` del objeto `System.out` que está construido dentro del lenguaje Java. En la cuarta sentencia, el parámetro actual de `println` es un poco más complicado:

```
System.out.println("# " + precio + " cvos.");
```

Se usan los dos operadores “+” para construir un solo parámetro de tipo cadena a partir de tres componentes:

- la cadena literal «# » (observe el carácter espacio luego del numeral);
- el valor del campo `precio` (observe que no hay comillas alrededor del nombre del campo);
- la cadena literal «cvos.» (observe el carácter espacio antes de la palabra `cvos`).

**Concepto**

El método  
**System.out.println** imprime su  
parámetro en la  
terminal de texto.

Cuando se usa el símbolo «+» entre una cadena y cualquier otra cosa, este símbolo es un operador de concatenación de cadenas (es decir, concatena o reúne cadenas para crear una nueva cadena) en lugar de ser el operador aritmético de suma.

Observe que la última llamada a `println` no contiene ningún parámetro de tipo cadena. Esto está permitido y el resultado de la llamada será dejar una línea en blanco entre esta salida y cualquier otra que le siga. Podrá ver fácilmente la línea en blanco si imprime un segundo boleto.

**Ejercicio 2.33** Agregue un método de nombre `mensaje` a la clase `MaquinaDeBoletos`, su tipo de retorno debe ser `void` y no debe tomar ningún parámetro. El cuerpo del método debe imprimir algo parecido a:

Por favor, ingrese la cantidad de dinero correcta.

**Ejercicio 2.34** Agregue un método `mostrarPrecio` a la clase `MaquinaDeBoletos`. Será un método con tipo de retorno `void` y sin parámetros. El cuerpo del método deberá imprimir algo similar a:

El precio del boleto es de xyz centavos.

Donde `xyz` deberá ser reemplazado por el valor que contenga el campo `precio` cuando el método sea llamado.

**Ejercicio 2.35** Cree dos máquinas con diferentes precios de boletos. Sus respectivas llamadas al método `mostrarPrecio` ¿producen la misma salida o es diferente? ¿Cómo explica este efecto?

**Ejercicio 2.36** Si se altera la cuarta sentencia de `imprimirBoleto` de modo que el `precio` también esté entre comillas, ¿qué piensa que se imprimirá?

`System.out.println("# " + "precio" + " cvos.");`

**Ejercicio 2.37** ¿Qué piensa sobre la siguiente versión?

`System.out.println("# precio cvos.");`

**Ejercicio 2.38** ¿Podría usarse en máquinas diferentes alguna de las dos últimas versiones anteriores para mostrar los precios de sus boletos? Explique su respuesta.

## 2.9

## Resumen de la máquina de boletos simplificada

Hemos examinado hasta ahora con cierto detalle la estructura interna de la clase máquina de boletos simplificada. Hemos visto que la clase tiene una pequeña capa exterior que le asigna un nombre y un cuerpo interno más sustancial que contiene los campos, un constructor y varios métodos. Los campos se usan para almacenar datos que permiten a los objetos mantener un estado. Los constructores se usan para preparar un estado inicial cuando se crea un objeto. Tener un estado inicial permitirá que un objeto responda apropiadamente a las llamadas a métodos inmediatamente después de su creación. Los métodos implementan el comportamiento definido para los objetos de la clase. Los métodos de acceso brindan información sobre el estado de un objeto y los de modificación cambian el estado de un objeto.

Hemos visto que los constructores se diferencian de los métodos por tener el mismo nombre que la clase en la que están definidos. Tanto los constructores como los métodos pueden tener parámetros, pero solamente los métodos pueden tener tipo de retorno. Los tipos de retorno que no son `void` nos permiten pasar un resultado hacia el exterior de un método. Un método que no tiene un tipo de retorno `void` debe tener una sentencia `return` como la última sentencia de su cuerpo. Las sentencias `return` se aplican solamente en los métodos porque los constructores nunca tienen tipo de retorno de ninguna naturaleza, ni siquiera `void`.

*Antes de que intente hacer estos ejercicios, asegúrese de haber comprendido bien cómo se comporta la máquina de boletos y cómo está implementado ese comportamiento a través de los campos, el constructor y los métodos de la clase.*

**Ejercicio 2.39** Modifique el constructor de la `MaquinaDeBoletos` de modo que no tenga ningún parámetro. En su lugar, el precio de los boletos debiera fijarse en 1 000 centavos. ¿Qué efecto tendrá esta modificación cuando se construyan objetos máquina de boletos en BlueJ?

**Ejercicio 2.40** Implemente un método `vaciar`, que simule el efecto de quitar todo el dinero de la máquina. Este método debe tener un tipo de retorno `void` y su cuerpo simplemente pone en cero el valor del campo `total`. ¿Necesita tener algún parámetro? Pruebe su método creando una máquina, ingrese algo de dinero, emita algunos boletos, verifique el total y luego vacíe la máquina. ¿Es un método de modificación o de acceso?

**Ejercicio 2.41** Implemente un método, `ponerPrecio`, que permita modificar el precio de los boletos con un nuevo valor. El nuevo precio se pasa al método mediante un parámetro. Pruebe su método creando una máquina, mostrando el precio de los boletos, cambiando el precio y luego mostrando el nuevo precio. ¿Es un método de modificación?

**Ejercicio 2.42** Provea a la clase de dos constructores: uno debe tomar un solo parámetro que especifique el precio del boleto, y el otro no debe tener parámetros y debe establecer el precio como un valor fijo por defecto, el que usted elija. Pruebe su implementación creando máquinas mediante los dos constructores diferentes.

## 2.10

### Reflexión sobre el diseño de la máquina de boletos

En las próximas secciones examinaremos la implementación de una clase mejorada para la máquina de boletos, que trate de remediar algunas de las restricciones que presenta la implementación simplificada.

A partir de nuestro análisis del interior de la clase `MaquinaDeBoletos` se puede apreciar lo inadecuada que sería esta implementación en el mundo real. Es deficiente por varios motivos:

- No verifica si el cliente ingresó dinero suficiente como para pagar el boleto.
- No devuelve nada de dinero si el cliente pagó de más por el boleto.

- No controla si el cliente ingresa cantidades de dinero que tienen sentido: experimentalmente, por ejemplo, qué ocurre si ingresa una cantidad negativa.
- No verifica si tiene sentido el precio del boleto pasado a su constructor.

Si pudiésemos remediar estos problemas entonces tendríamos una pieza de software mucho más funcional que podría servir como base para operar una máquina de boletos del mundo real. Dado que vemos que podemos mejorar la versión existente, abra el proyecto *maquina-de-boletos-mejorada*. Tal como en el caso anterior, el proyecto contiene una sola clase, *MaquinaDeBoletos*. Antes de ver los detalles internos de la clase, experimente con ella creando algunas instancias y vea si observa alguna diferencia en el comportamiento entre la versión previa simplificada y ésta. Una diferencia específica es que la nueva versión tiene un método adicional, *reintegrarSaldo*. Más adelante, en este capítulo, usaremos este método para introducir un aspecto adicional de Java, de modo que vea qué ocurre cuando lo invoca.

## 2.11

## Hacer elecciones: la sentencia condicional

El Código 2.8 muestra los detalles internos de la definición de clase de la máquina de boletos mejorada. Muchas de estas definiciones ya son familiares a partir del análisis de la máquina de boletos simplificada. Por ejemplo, la envoltura exterior que nombra a la clase es la misma porque hemos elegido dar el mismo nombre a esta clase; además, contiene los mismos tres campos para mantener el estado del objeto y han sido declarados de la misma manera; el constructor y los dos métodos *get* también son los mismos que los anteriores.

### Código 2.8

Una máquina de boletos más sofisticada

```
/*
 * MaquinaDeBoletos modela una máquina de boletos que trabaja
 * con tarifa plana.
 * El precio de un boleto se especifica a través del
constructor.
 * Implementa controles para asegurar que un usuario ingrese
 * sólo cantidades de dinero con sentido y sólo se imprimirá
 * un boleto si el dinero ingresado alcanza.
 *
 * @author David J. Barnes and Michael Kolling
 * @version 2006.03.30
 */
public class MaquinaDeBoletos{
    // El precio de un boleto de esta máquina.
    private int precio;
    // La cantidad de dinero ingresada hasta ahora por un
cliente.
    private int saldo;
    // El total del dinero recolectado por esta máquina.
    private int total;
    /**
```

**Código 2.8  
(continuación)**

Una máquina de boletos más sofisticada

```
* Crea una máquina que vende boletos de un precio
determinado.
*/
public MaquinaDeBoletos(int precioDelBoleto)
{
    precio = precioDelBoleto;
    saldo = 0;
    total = 0;
}
/**
 * Devuelve el precio de un boleto.
 */
public int getPrecio()
{
    return precio;
}
/**
 * Devuelve la cantidad de dinero que ya se ingresó
para
 * el siguiente boleto.
 */
public int getSaldo()
{
    return saldo;
}
/**
 * Recibe del cliente una cantidad de dinero en
centavos.
 * Controla que la cantidad tenga sentido.
 */
public void ingresarDinero(int cantidad)
{
    if(cantidad > 0) {
        saldo = saldo + cantidad;
    }
    else {
        System.out.println("Debe ingresar una cantidad
positiva: " +
                           cantidad);
    }
}
/**
 * Imprime un boleto si la cantidad de dinero ingresada
 * alcanza y disminuye el saldo actual en el precio
 * del boleto. Imprime un mensaje de error si se
 * requiere más dinero.
 */
public void imprimirBoleto()
{
```

**Código 2.8  
(continuación)**

Una máquina de boletos más sofisticada

```

        if(saldo >= precio) {
            // Simula la impresión de un boleto.
            System.out.println("#####");
            System.out.println("# Línea BlueJ ");
            System.out.println("# Boleto");
            System.out.println("# " + precio + "
cvos.");
            System.out.println("#####");
            System.out.println();
            // Actualiza el total recolectado con el
precio.
            total = total + precio;
            // Disminuye el saldo en el valor del precio.
            saldo = saldo - precio;
        }
        else {
            System.out.println("Debe ingresar como mínimo: "
+ (precio - saldo) + "
cvos más.");
        }
    }
    /**
     * Devuelve el valor del saldo.
     * Se limpia el saldo.
     */
    public int reintegrarSaldo()
    {
        int cantidadAReintegar;
        cantidadAReintegar = saldo;
        saldo = 0;
        return cantidadAReintegar;
    }
}

```

Encontramos el primer cambio significativo en el método `ingresarDinero`. Hemos reconocido que el principal problema de la máquina de boletos simplificada era su falta de control sobre ciertas condiciones. Una de esas faltas de control era sobre la cantidad de dinero introducida por un cliente, de modo que resultaba posible ingresar una cantidad de dinero negativa. Hemos remediado esa falla haciendo uso de una *sentencia condicional* que controla que el monto ingresado sea un valor mayor que cero:

```

if(cantidad > 0) {
    saldo = saldo + cantidad;
}
else {
    System.out.println("Debe ingresar una cantidad positiva: " +
cantidad);
}

```

**Concepto**

Una **sentencia condicional** realiza una de dos acciones posibles basándose en el resultado de una prueba.

Las sentencias condicionales también son conocidas como *sentencias if* debido a la palabra usada en la mayoría de los lenguajes de programación que las introducen. Una sentencia condicional nos permite hacer una de dos acciones posibles basándose en el resultado de una verificación o prueba: si el resultado es verdadero entonces hacemos una cosa, de lo contrario hacemos algo diferente. Una sentencia condicional tiene la forma general descrita en el siguiente *pseudo-código*:

```
if(se lleva a cabo alguna prueba que da un resultado
verdadero o falso) {
    Si la prueba dio resultado verdadero, ejecutar estas sentencias
}
else {
    Si el resultado dio falso, ejecutar estas sentencias
}
```

Es importante apreciar que después de la evaluación de la prueba se llevará a cabo sólo uno de los conjuntos de sentencias que están a continuación de la prueba. Por lo que, en el ejemplo del método `ingresarDinero`, a continuación de la prueba sobre la cantidad de dinero introducida, sólo sumaremos la cantidad al saldo o bien mostraremos el mensaje de error. La prueba usa el *operador mayor que <>* para comparar el valor de cantidad con cero. Si el valor es mayor que cero entonces se sumará al saldo. Si no es mayor que cero, se muestra un mensaje de error. En efecto, usando una sentencia condicional podemos proteger la modificación del saldo del caso en que el parámetro no represente una cantidad válida.

**Concepto**

Las **expresiones booleanas** tienen sólo dos valores posibles: verdadero o falso. Se las encuentra comúnmente controlando la elección entre los dos caminos posibles de una sentencia condicional.

La prueba que se usa en una sentencia condicional es un ejemplo de una *expresión booleana*. Anteriormente en este capítulo introdujimos expresiones aritméticas que producen resultados numéricos. Una expresión booleana tiene sólo dos valores posibles, verdadero(true) o falso(false): una de dos, el valor de cantidad es mayor que cero (verdadero) o no es mayor que cero (falso). Una sentencia condicional hace uso de esos dos posibles valores para elegir entre dos acciones diferentes.

**Ejercicio 2.43** Controle que el comportamiento del que hemos hablado es correcto creando una instancia de `MaquinaDeBoletos` e invocando a `ingresarDinero` con varios valores diferentes en el parámetro actual. Controle el saldo antes y después de invocar a `ingresarDinero`. En los casos en que se muestra un mensaje de error, ¿cambia el valor del saldo? Trate de predecir qué ocurriría si ingresa como parámetro el valor cero y luego compruebe la verdad de su predicción.

**Ejercicio 2.44** Prediga qué cree que ocurrirá si cambia el control de `ingresarDinero` usando el *operador mayor o igual que*.

```
if(cantidad >= 0)
```

Verifique sus predicciones ejecutando algunas pruebas. ¿Qué diferencia produce este cambio en el comportamiento del método?

**Ejercicio 2.45** En el proyecto *figuras* que vimos en el Capítulo 1 usamos un campo boolean para controlar un aspecto de los objetos círculo. ¿Cuál es ese aspecto? ¿Estaba bien hecho el control mediante un tipo que tiene sólo dos valores diferentes?

## 2.12

# Un ejemplo más avanzado de sentencia condicional

El método `imprimirBoleto` contiene un ejemplo más avanzado de una sentencia condicional. Aquí está su esquema:

```
if(saldo >= precio) {  
    Se omitieron los detalles de impresión.  
    // Actualiza el total recaudado con el precio.  
    total = total + precio;  
    // Decrementa el saldo en el valor del precio.  
    saldo = saldo - precio;  
}  
else {  
    System.out.println("Debe ingresar como mínimo: "  
        + (precio - saldo) + "  
    céntimos.");  
}
```

Queremos remediar el hecho de que la versión simplificada no controla que un cliente haya introducido dinero suficiente para que se emita un boleto. Esta versión verifica que el valor del campo `saldo` es como mínimo tan grande como el valor del campo `precio`. De ser así está bien que se emita un boleto; de lo contrario, en lugar del boleto mostramos un mensaje de error.

**Ejercicio 2.46** En esta versión de `imprimirBoleto` también hacemos algo ligeramente diferente con los campos `total` y `saldo`. Compare la implementación del método en el Código 2.1 con la del Código 2.8 para ver si puede encontrar cuáles son esas diferencias. Luego compruebe su comprensión experimentando en BlueJ.

El método `imprimirBoleto` disminuye el valor del saldo en el valor del precio. En consecuencia, si un cliente ingresa más dinero que el precio del boleto, quedará algo de dinero en `saldo` que podrá usarse para conformar el precio de un segundo boleto. Alternativamente, el cliente puede pedir el reintegro del dinero sobrante y esto es lo que hace el método `reintegrarSaldo` tal como veremos en la próxima sección.

**Ejercicio 2.47** Después de emitido un boleto, si se resta el `precio` del campo `saldo` ¿Puede este último campo tener un valor negativo? Justifique su respuesta.

**Ejercicio 2.48** Hasta ahora hemos introducido dos operadores aritméticos, `+` y `-`, que pueden usarse en *expresiones aritméticas* en Java. Vea el Apéndice D para encontrar qué otros operadores están disponibles en Java.

**Ejercicio 2.49** Escriba una sentencia de asignación que almacene el resultado de multiplicar dos variables, `precio` y `descuento`, en una tercera variable, `ahorro`.

**Ejercicio 2.50** Escriba una sentencia de asignación que divida el valor de **total** por el valor de **cantidad** y almacene el resultado en la variable **promedio**.

**Ejercicio 2.51** Escriba una sentencia **if** que compare el valor de **precio** con el valor de **presupuesto**. Si el **precio** es mayor que el **presupuesto** imprimir el mensaje «Muy caro», de lo contrario imprimir el mensaje «El precio es justo».

**Ejercicio 2.52** Modifique su respuesta al ejercicio anterior de modo que el mensaje que se emite, cuando el precio es demasiado alto, incluya el valor de su presupuesto.

## 2.13

### Variables locales

#### Concepto

Una **variable local** es una variable que se declara y se usa dentro de un solo método. Su alcance y tiempo de vida se limitan a los del método.

El método **reintegrarSaldo** contiene tres sentencias y una declaración. La declaración ilustra una nueva clase de variable:

```
public int reintegrarSaldo()
{
    int cantidadAReintegar;
    cantidadAReintegar = saldo;
    saldo = 0;
    return cantidadAReintegar;
}
```

¿Qué clase de variable es **cantidadAReintegar**? Sabemos que no es un campo porque los campos se definen fuera de los métodos. Tampoco es un parámetro porque siempre se definen en el encabezado del método. La variable **cantidadAReintegar** es lo que se conoce como una *variable local* porque está definida *dentro* de un método. Es muy común inicializar variables locales cuando se las declara, por lo que podríamos abreviar las dos primeras sentencias de **reintegrarSaldo** de la siguiente manera:

```
int cantidadAReintegar = saldo;
```

Las declaraciones de las variables locales son muy similares a las declaraciones de los campos pero las palabras **private** o **public** nunca forman parte de ellas. Tal como con los parámetros formales, las variables locales tienen un alcance que está limitado a las sentencias del método al que pertenecen. Su tiempo de vida es el tiempo de la ejecución del método: se crean cuando se invoca un método y se destruyen cuando el método termina. Los constructores también pueden tener variables locales.

Las variables locales se usan frecuentemente como lugares de almacenamiento temporal para ayudar a un método a completar su tarea. En este método se usa **cantidadAReintegar** para guardar el valor del **saldo** inmediatamente antes de ponerlo en cero; el método retorna entonces el viejo valor del **saldo**. Los siguientes ejercicios lo ayudarán a comprender la necesidad de usar una variable local para escribir el método **reintegrarSaldo**.

**Ejercicio 2.53** ¿Por qué la siguiente versión de **reintegrarSaldo** no da el mismo resultado que el original?

```
public int reintegrarSaldo()
{
    saldo = 0;
    return saldo;
}
```

¿Qué pruebas podría ejecutar para demostrar la diferencia entre los resultados?

**Ejercicio 2.54** ¿Qué ocurre si trata de compilar la clase MaquinaDeBoletos con la siguiente versión de reintegrarSaldo?

```
public int reintegrarSaldo()
{
    return saldo;
    saldo = 0;
}
```

¿Qué conocimiento tiene sobre las sentencias return que lo ayudaría a explicar por qué esta versión no compila?

Ahora que ha visto cómo se usan las variables locales, vuelva al Ejercicio 2.20 y verifique que lo comprende: allí, una variable local evita que un campo con el mismo nombre sea accedido.

**Cuidado** Una variable local del mismo nombre que un campo evitara que el campo sea accedido dentro de un método. Vea la Sección 3.12.2 para otra manera de prevenir el acceso, cuando sea necesario.

## 2.14

## Campos, parámetros y variables locales

Con la introducción de cantidadAReintegrar en el método reintegrarSaldo hemos visto tres tipos diferentes de variables: campos, parámetros formales y variables locales. Es importante comprender las similitudes y diferencias entre estos tipos de variables. A continuación hacemos un resumen de sus características:

- Las tres clases de variables pueden almacenar un valor acorde a su definición de tipo de dato. Por ejemplo, una variable definida como de tipo `int` permite almacenar un valor entero.
- Los campos se definen fuera de los constructores y de los métodos.
- Los campos se usan para almacenar datos que persisten durante la vida del objeto, de esta manera mantienen el estado actual de un objeto. Tienen un tiempo de vida que finaliza cuando termina el objeto.
- El alcance de los campos es la clase: la accesibilidad de los campos se extiende a toda la clase y por este motivo pueden usarse dentro de cualquier constructor o método de clase en la que estén definidos.
- Como son definidos como privados (`private`), los campos no pueden ser accedidos desde el exterior de la clase.
- Los parámetros formales y las variables locales persisten solamente en el lapso durante el cual se ejecuta un constructor o un método. Su tiempo de vida es tan

largo como una llamada, por lo que sus valores se pierden entre llamadas. Por este motivo, actúan como lugares de almacenamiento temporales antes que permanentes.

- Los parámetros formales se definen en el encabezado de un constructor o de un método. Reciben sus valores desde el exterior, se inicializan con los valores de los parámetros actuales que forman parte de la llamada al constructor o al método.
- Los parámetros formales tienen un alcance limitado a su definición de constructor o de método.
- Las variables locales se declaran dentro del cuerpo de un constructor o de un método. Pueden ser inicializadas y usadas solamente dentro del cuerpo de las definiciones de constructores o métodos. Las variables locales deben ser inicializadas antes de ser usadas en una expresión, no tienen un valor por defecto.
- Las variables locales tienen un alcance limitado al bloque en el que son declaradas. No son accesibles desde ningún lugar fuera de ese bloque.

**Ejercicio 2.55** Agregue un nuevo método, `vaciarMaquina`, diseñado para simular el quitar todo el dinero de la máquina. Debe retornar el valor de `total` y poner `total` nuevamente en cero.

**Ejercicio 2.56** El método `vaciarMaquina`, ¿es un método de acceso, de modificación, o ambos?

**Ejercicio 2.57** Escriba nuevamente el método `imprimirBoleto` de modo que declare una variable local, `cantidadRestanteAPagar` que debe ser inicializada para que contenga la diferencia entre el `precio` y el `saldo`. Rescriba la prueba de la sentencia condicional para controlar el valor de `cantidadRestanteAPagar`: si su valor es menor o igual que cero se deberá imprimir un boleto, de lo contrario se emitirá un mensaje de error mostrando la cantidad de dinero que falta para pagar el boleto. Pruebe su versión para asegurarse de que se comporta exactamente de la misma manera que la versión original.

**Ejercicio 2.58 Desafío.** Suponga que queremos que un único objeto `MaquinaDeBoletos` disponga de boletos de diferentes precios: por ejemplo, los usuarios podrían presionar un botón de la máquina real para seleccionar un boleto de un precio en particular. ¿Qué otros métodos o campos necesitaría agregar a la `MaquinaDeBoletos` para lograr esta funcionalidad? ¿Considera que varios de los métodos existentes debieran también ser cambiados?

Grabe el proyecto `maquina-de-boletos-mejorada` bajo un nuevo nombre e implemente sus cambios en el nuevo proyecto.

## 2.15

## Resumen de la máquina de boletos mejorada

En vías de desarrollar una versión más sofisticada de la clase `MaquinaDeBoletos`, hemos sido capaces de encontrar los mayores inconvenientes de la versión simplificada. Al hacerlo, hemos introducido dos nuevas construcciones del lenguaje: la sentencia condicional y las variables locales.

- Una sentencia condicional nos da la posibilidad de realizar una prueba, y en base a su resultado llevar a cabo una u otra de dos acciones distintas.
- Las variables locales nos permiten calcular y almacenar temporalmente valores dentro de un constructor o un método. Contribuyen al comportamiento que implementan las definiciones de sus métodos, pero sus valores se pierden una vez que el constructor o el método finaliza su ejecución.

Puede encontrar más detalles sobre las sentencias condicionales y las formas que pueden asumir sus pruebas en el Apéndice C.

## 2.16

## Ejercicios de revisión

En este capítulo hemos sentado bases nuevas y hemos introducido una gran cantidad de conceptos nuevos. Seguiremos construyéndolos en capítulos posteriores de modo que es importante que se sienta familiarizado con ellos. Pruebe hacer los siguientes ejercicios en lápiz y papel como una forma de verificar que ha comenzado a usar la terminología que hemos introducido en este capítulo. No se moleste por el hecho de que sugerimos hacerlos en papel en lugar de hacerlos en BlueJ, será una buena práctica el intentar resolver los ejercicios sin que medie un compilador.

**Ejercicio 2.59** Determine el nombre y el tipo de retorno de este método:

```
public String getCodigo()  
{  
    return codigo;  
}
```

**Ejercicio 2.60** Indique el nombre de este método y el nombre y el tipo de su parámetro.

```
public void setCreditos(int cantidadDeCreditos)  
{  
    creditos = cantidadDeCreditos;  
}
```

**Ejercicio 2.61** Escriba la envoltura exterior de una clase de nombre Persona. Recuerde incluir las llaves al comienzo y al final del cuerpo de la clase; pero, por otra parte, deje el cuerpo vacío.

**Ejercicio 2.62** Escriba las declaraciones de los siguientes campos:

- Un campo denominado **nombre** y de tipo **String**
- Un campo de tipo **int** y de nombre **edad**
- Un campo de tipo **String** denominado **codigo**
- Un campo de nombre **creditos** de tipo **int**

**Ejercicio 2.63** Escriba un constructor para la clase **Modulo**. El constructor tendrá un solo parámetro de tipo **String** denominado **codigoDelModulo**. El cuerpo del constructor deberá asignar el valor de su parámetro a un campo de nombre **codigo**. No tiene que incluir la declaración de **codigo**, sólo el texto del constructor.

**Ejercicio 2.64** Escriba un constructor para una clase de nombre `Persona`.

El constructor deberá tener dos parámetros: el primero de tipo `String` y denominado `miNombre`, y el segundo de tipo `int` y de nombre `miEdad`. Use el primer parámetro para establecer el valor de un campo denominado `nombre`, y el segundo para preparar un campo de nombre `edad`. No tiene que incluir las definiciones de estos campos, sólo el texto del constructor.

**Ejercicio 2.65** Corrija el error de este método:

```
public void getEdad()
{
    return edad;
}
```

**Ejercicio 2.66** Escriba un método de acceso de nombre `getNombre` que retorna el valor de un campo denominado `nombre`, cuyo tipo es `String`.**Ejercicio 2.67** Escriba un método de modificación de nombre `setEdad` que tenga un único parámetro de tipo `int` y que cambie el valor del campo de nombre `edad`.**Ejercicio 2.68** Escriba un método de nombre `imprimirDetalles` para una clase que tiene un campo de tipo `String` denominado `nombre`. El método `imprimirDetalles` debe mostrar en la terminal de texto, la cadena «El nombre de esta persona es» seguida del valor del campo `nombre`. Por ejemplo, si el valor del campo `nombre` es «Elena», el método imprimiría:

El nombre de esta persona es Elena

Si se las arregló para completar la mayoría o todos estos ejercicios, entonces parece que está en condiciones de intentar crear en BlueJ un nuevo proyecto y llevar a cabo su propia definición de una clase `Persona`. Por ejemplo, la clase debería tener campos para registrar el nombre y la edad de una persona. Si se sintió inseguro para completar cualquiera de estos ejercicios, vuelva a las secciones anteriores de este capítulo y al código de la clase `MaquinaDeBoletos` para revisar las cuestiones que aún no le quedan claras. En la próxima sección ofrecemos algún material más para realizar la revisión.

**2.17****Revisar un ejemplo familiar**

Al llegar a este punto del capítulo ha encontrado una gran cantidad de conceptos nuevos. Para ayudar a reforzar esos conceptos los revisaremos en un contexto familiar pero diferente. Abra el proyecto *curso-de-laboratorio* que trabajamos en el Capítulo 1 y luego examine la clase `Estudiante` en el editor (Código 2.9).

**Código 2.9**

La clase  
`Estudiante`

```
/**
 * La clase Estudiante representa un estudiante en un
 * sistema
 * administrativo de estudiantes. Contiene los detalles
 * relevantes
```

**Código 2.9  
(continuación)**

La clase  
Estudiante

```
* en nuestro contexto.  
*  
* @author Michael Kolling y David Barnes  
* @version 2006.03.30  
*/  
public class Estudiante  
{  
    // nombre completo del estudiante  
    private String nombre;  
    // ID (identificador) del estudiante  
    private String id;  
    // la cantidad de créditos que tiene hasta ahora  
    private int creditos;  
    /**  
     * Crea un nuevo estudiante con un determinado nombre  
     * y con  
     * un determinado número de identificación.  
     */  
    public Estudiante(String nombreCompleto, String  
IdEstudiante)  
    {  
        nombre = nombreCompleto;  
        id = IdEstudiante;  
        creditos = 0;  
    }  
    /**  
     * Devuelve el nombre completo de este estudiante.  
     */  
    public String getNombre()  
    {  
        return nombre;  
    }  
    /**  
     * Asigna un nuevo nombre a este estudiante.  
     */  
    public void cambiarNombre(String nuevoNombre)  
    {  
        nombre = nuevoNombre;  
    }  
    /**  
     * Devuelve el Id de este estudiante.  
     */  
    public String getIdEstudiante()  
    {  
        return id;  
    }  
    /**  
     * Suma algunos puntos a los créditos acumulados del  
     * estudiante.  
    */
```

**Código 2.9  
(continuación)**

La clase  
Estudiante

```
    */
    public void sumarCreditos(int puntosAdicionales)
    {
        creditos += puntosAdicionales;
    }
    /**
     * Devuelve el número de créditos que el estudiante
     ha acumulado.
     */
    public int getCreditos()
    {
        return creditos;
    }
    /**
     * Devuelve el nombre de usuario del estudiante.
     * El nombre de usuario es una combinación de los
     cuatro primeros
     * caracteres del nombre del estudiante y los tres
     primeros
     * caracteres del número del ID de estudiante.
     */
    public String getNombreDeUsuario()
    {
        return nombre.substring(0,4) + id.substring(0,3);
    }

    /**
     * Imprime el nombre y el número de ID del
     estudiante en la
     * terminal de salida.
     */
    public void imprimir()
    {
        System.out.println(nombre + " (" + id + ")");
    }
}
```

La clase contiene tres campos: `nombre`, `id` y `creditos`. Cada uno de ellos es inicializado en un único constructor. Los valores iniciales de los primeros dos campos están determinados por los valores pasados al constructor mediante parámetros. Cada uno de estos campos tiene un método de acceso `get` pero solamente los campos `nombre` y `creditos` tienen asociados métodos de modificación: esto significa que el valor de un campo `id` permanece fijo una vez que se ha construido el objeto.

El método `getNombreDeUsuario` ilustra una nueva característica que será fuertemente explorada:

```
public String getNombreDeUsuario()
{
    return nombre.substring(0,4) + id.substring(0,3);
}
```

Tanto `nombre` como `id` son cadenas, y la clase `String` tiene un método de acceso, `substring`, con la siguiente firma en Java:

```
/**
 * Return a new string containing the characters from
 * beginIndex to (endIndex-1) from this string.
 */
public String substring(int beginIndex, int endIndex)
```

El valor cero del índice representa el primer carácter de una cadena, de modo que `getNombreDeUsuario` toma los primeros cuatro caracteres de la cadena `nombre`, los primeros tres caracteres de la cadena `id` y los concatena formando una nueva cadena que, en definitiva, es el resultado que devuelve el método. Por ejemplo, si `nombre` es la cadena «Leonardo da Vinci» y el `id` es la cadena «468366», entonces este método devuelve la cadena «Leon468» .

**Ejercicio 2.69** Dibuje una figura similar a la que muestra la Figura 2.3 para representar el estado inicial de un objeto `Estudiante` después de su construcción, con los siguientes valores para sus parámetros actuales.

```
new Estudiante("Benjamín Jonson", "738321")
```

**Ejercicio 2.70** Si el nombre de un estudiante es «Henry Moore» y su `id` es «557214» ¿Qué debiera retornar el método `getNombreDeUsuario`?

**Ejercicio 2.71** Cree un estudiante de nombre “djb” y con `id` “859012”. ¿Qué ocurre cuando se invoca `getNombreDeUsuario` sobre este objeto estudiante? ¿Por qué considera que es así?

**Ejercicio 2.72** La clase `String` define el método de acceso `length` con la siguiente firma

```
/**
 * Return the number of characters in this string.
 */
public int length()
```

Es decir que el método de acceso `length` de la clase `String` de Java devuelve la cantidad de caracteres de una cadena.

Agregue una sentencia condicional al constructor de `Estudiante` para emitir un mensaje de error si el largo del parámetro `nombre` es menor de cuatro caracteres o el largo del parámetro `idEstudiante` es menor de tres caracteres. Sin embargo, el constructor debe seguir usando esos parámetros para preparar los campos `nombre` e `idEstudiante`, aun cuando se imprima el mensaje de error. *Pista:* use sentencias `if` de la siguiente forma (sin su parte «`else`») para imprimir los mensajes de error.

```
if(se realiza la prueba sobre uno de los parámetros) {
```

```
Si la prueba dio resultado verdadero, imprimir un mensaje  
de error.  
}
```

Si es necesario, vea el Apéndice C para encontrar más detalles sobre los diferentes tipos de sentencias `if`.

**Ejercicio 2.73** Desafío. Modifique el método `getNombreDeUsuario` de la clase `Estudiante` de modo que siempre genere un nombre de usuario, aun cuando alguno de sus campos `nombre` o `id` no tengan la longitud necesaria. Para las cadenas más cortas que las del largo requerido, use la cadena completa.

## 2.18

## Resumen

En este capítulo hemos sentado las bases para crear una definición de clase. Las clases contienen campos, constructores y métodos que definen el estado y el comportamiento de los objetos. Dentro de los constructores y de los métodos, una secuencia de sentencias define cómo un objeto cumple con las tareas diseñadas. Hemos abordado las sentencias de asignación y las sentencias condicionales y agregaremos otros tipos más de sentencias en capítulos posteriores.

Términos introducidos en este capítulo

**campo, variable de instancia, constructor, método, firma del método, cuerpo del método, parámetro, método de acceso, método de modificación, declaración, inicialización, bloque, sentencia, sentencia de asignación, sentencia condicional, sentencia return, tipo de retorno, comentario, expresión, operador, variable, variable local, alcance, tiempo de vida**

## Resumen de conceptos

- **campo** Los campos almacenan datos para que un objeto los use. Los campos se conocen como variables de instancia.
- **comentario** Los comentarios se insertan dentro del código de una clase para brindar explicaciones a los lectores humanos. No tienen efecto sobre la funcionalidad de la clase.
- **constructor** Los constructores permiten que cada objeto sea preparado adecuadamente cuando es creado.
- **alcance** El alcance de una variable define la sección de código desde donde la variable puede ser accedida.
- **tiempo de vida** El tiempo de vida de una variable describe el tiempo durante el cual la variable continúa existiendo antes de ser destruida.
- **asignación** Las sentencias de asignación almacenan el valor representado del lado derecho de la sentencia en la variable nombrada en el lado izquierdo.

- **método** Los métodos están compuestos por dos partes: un encabezado y un cuerpo.
- **método de acceso** Los métodos de acceso devuelven información sobre el estado de un objeto.
- **métodos de modificación** Los métodos de modificación cambian el estado de un objeto.
- **println** El método `System.out.println(...)` imprime su parámetro en la terminal de texto.
- **condicional** Una sentencia condicional realiza una de dos acciones posibles basándose en el resultado de una prueba.
- **expresión booleana** Las expresiones booleanas tienen sólo dos valores posibles: verdadero y falso. Se las encuentra comúnmente controlando la elección entre los dos caminos de una sentencia condicional.
- **variable local** Las variables locales son variables que se declaran y usan dentro de un único método. Su alcance y tiempo de vida están limitados por el método.

Los siguientes ejercicios están diseñados para ayudarlo a experimentar con los conceptos de Java que hemos discutido en este capítulo. Creará sus propias clases que contienen elementos tales como campos, constructores, métodos, sentencias de asignación y sentencias condicionales.

**Ejercicio 2.74** Debajo de este ejercicio se encuentra el esquema de la clase `Libro` que se encuentra en el proyecto `ejercicio-libro`. El esquema ya declara dos campos y un constructor para inicializar dichos campos. En este ejercicio y en algunos de los siguientes, agregará más aspectos al esquema de la clase.

Agregue a la clase dos métodos de acceso, `getAutor` y `getTitulo`, que devuelven los campos `autor` y `titulo` como sus respectivos resultados. Pruebe su clase creando algunas instancias y llamando a estos métodos.

```
/**  
 * Una clase que registra información sobre un libro.  
 * Puede formar parte de una aplicación más grande  
 * como por ejemplo, un sistema de biblioteca.  
 *  
 * @author (Escriba su nombre aquí.)  
 * @version (Escriba la fecha aquí.)  
 */  
public class Libro  
{  
    // Los campos.  
    private String autor;  
    private String titulo;  
    /**  
     * Inicializa los campos autor y titulo cuando  
     * se construya este objeto  
     */
```

```

public Libro(String autorDelLibro, String tituloDelLibro)
{
    autor = autorDelLibro;
    titulo = tituloDelLibro;
}
// Agregue los métodos aquí...
}

```

**Ejercicio 2.75** Agregue al esquema de la clase `Libro` dos métodos, `imprimirAutor` e `imprimirTitulo`, que impriman respectivamente, los campos del autor y del título del libro en la ventana terminal.

**Ejercicio 2.76** Agregue un campo más, `paginas`, a la clase `Libro` para almacenar la cantidad de páginas. Este campo debe ser de tipo `int` y su valor inicial debe ser pasado al único constructor, junto con las cadenas para el autor y el título. Incluya un método de acceso adecuado para este campo, `getPaginas`.

**Ejercicio 2.77** Agregue a la clase `Libro` el método `imprimirDetalles` para imprimir los detalles del autor, el título y la cantidad de páginas en la ventana terminal. Los detalles sobre el formato de esta salida quedan a su libre elección. Por ejemplo, los tres elementos pueden imprimirse en una sola línea o bien se puede imprimir cada elemento en una línea independiente. También puede incluir algún texto explicativo para ayudar al usuario a saber cuál es el autor y cuál es el título. Por ejemplo:

```
Titulo: Robinson Crusoe, Autor: Daniel Defoe, Paginas: 232
```

**Ejercicio 2.78** Agregue otro campo a la clase, `numeroDeReferencia`. Este campo puede almacenar, por ejemplo, un número de referencia para una biblioteca. Debe ser de tipo `String` y ser inicializado en el constructor con una cadena de longitud cero (" ") cuando su valor inicial no sea pasado al constructor mediante el parámetro. Defina un método de modificación para este campo con la siguiente firma:

```
public void setNumeroDeReferencia(String ref)
```

El cuerpo de este método debe asignar el valor del parámetro al campo `numeroDeReferencia`. Agregue el método de acceso correspondiente para ayudar a controlar que el método de modificación funciona correctamente.

**Ejercicio 2.79** Modifique su método `imprimirDetalles` para que incluya la impresión del número de referencia. Sin embargo, el método imprimirá el número de referencia solamente si el campo `numeroDeReferencia` contiene una cadena de longitud distinta de cero. Si no es así, en su lugar imprima la cadena «ZZZ». *Pista:* use una sentencia condicional cuya prueba invoque al método `length` sobre la cadena `numeroDeReferencia`.

**Ejercicio 2.80** Modifique su método `setNumeroDeReferencia` de modo que cambie el contenido del campo `numeroDeReferencia` sólo si el parámetro es una cadena de tres caracteres como mínimo. Si es menor que tres, imprima un mensaje de error y deje este campo sin cambios.

**Ejercicio 2.81** Agregue a la clase `Libro` un nuevo campo entero, `prestado`. Este campo representa un contador del número de veces que un libro ha sido prestado. Agregue un método de modificación a la clase, `prestar`, que incremente el campo `prestado` en 1 cada vez que es llamado. Incluya un método de acceso, `getPrestado`, que retorne el valor de este nuevo campo como su resultado. Modifique `imprimirDetalles` para que incluya el valor de este campo con algún texto explicativo.

**Ejercicio 2.82** Desafío. Cree un nuevo proyecto en BlueJ: `ejercicio-calentador`. Escriba los detalles del proyecto en el *descriptor del proyecto*, la nota de texto que se ve en el diagrama. Cree una clase `Calentador` que contenga un solo campo entero: `temperatura`. Defina un constructor sin parámetros. El campo `temperatura` debe ser preparado en el constructor con el valor 15. Defina los métodos de modificación `calentar` y `enfriar` cuyo efecto es aumentar o disminuir el valor de la temperatura en 5° respectivamente. Defina un método de acceso que retorne el valor de la `temperatura`.

**Ejercicio 2.83** Desafío. Modifique su clase `Calentador` agregando tres nuevos campos enteros: `min`, `max` e `incremento`. Los valores iniciales de `min` y `max` deben establecerse mediante parámetros del constructor. El valor inicial del `incremento` en el constructor es 5. Modifique las declaraciones de `calentar` y `enfriar` de modo que usen el valor del incremento en lugar del valor explícito 5. Antes de avanzar con este ejercicio, controle que todo funcione bien. Luego modifique el método `calentar` para que no permita que la `temperatura` pueda recibir un valor mayor que `max`. De manera similar modifique `enfriar` para que no permita que la `temperatura` tome un valor menor que `min`. Controle que la clase funcione adecuadamente. Luego agregue un método, `setIncremento`, que tiene un solo parámetro entero que se usa para establecer el valor del `incremento`. Nuevamente controle que la clase funcione tal como se espera creando algunos objetos `Calentador` en BlueJ. Si se pasa un valor negativo al método `setIncremento`, ¿sigue funcionando todo tal como se esperaba? Agregue un control para que este método no permita que se asigne un valor negativo al `incremento`.



## CAPÍTULO

# 3

## Interacción de objetos

Principales conceptos que se abordan en este capítulo:

- abstracción
- modularización
- creación de objetos
- diagramas de objetos
- llamadas a métodos
- depuradores

Construcciones Java que se abordan en este capítulo

clases como tipos, operadores lógicos (`&&`, `||`), concatenación de cadenas, operador módulo (%), construcción de objetos (`new`), llamadas a métodos (notación de punto), palabra clave `this`

En los capítulos anteriores hemos examinado qué son los objetos y cómo se los implementa; en particular, cuando analizamos las definiciones de las clases, hablamos sobre campos, constructores y métodos.

Ahora, iremos un paso más adelante. Para construir aplicaciones interesantes no es suficiente construir objetos que trabajan individualmente. En realidad, los objetos deben estar combinados de tal manera que cooperen entre ellos al llevar a cabo una tarea en común. En este capítulo construiremos una pequeña aplicación a partir de tres objetos y trabajaremos con métodos que invocan a otros métodos para lograr su objetivo.

### 3.1

#### El ejemplo reloj

El proyecto que usaremos para discutir sobre la interacción de objetos modela un visor para un reloj digital. El visor muestra las horas y los minutos separados por dos puntos (Figura 3.1). Para este ejercicio, construiremos primeramente un reloj con un visor de 24 horas, de estilo europeo, por lo que muestra la hora desde las 00:00 (medianoche) hasta las 23:59 (un minuto antes de medianoche). Debido a que es un poco más difícil de construir un reloj de 12 horas, dejaremos este modelo para el final de este capítulo.

**Figura 3.1**

El visor de un reloj digital

11:03

## 3.2

# Abstracción y modularización

Una primera idea podría ser implementar totalmente el visor del reloj en una sola clase. Después de todo, esto es lo que hemos visto hasta ahora: cómo construir clases para hacer un trabajo.

Sin embargo, abordaremos este problema de una manera un poco diferente. Veremos si podemos identificar en el problema, componentes que se puedan convertir en clases independientes; la razón de proceder así radica en la complejidad del problema. A medida que avancemos en este libro, los ejemplos que usamos y los programas que construimos se volverán más y más complejos. Tareas triviales tales como la de la máquina de boletos pueden ser resueltas como si fueran un único problema: se puede ver la tarea completa y divisar una solución usando una sola clase. En los problemas más complejos, esta es una visión demasiado simplista. Cuando un problema se agranda, se vuelve más difícil mantener todos los detalles al mismo tiempo.

### Concepto

La **abstracción** es la habilidad de ignorar los detalles de las partes para centrar la atención en un nivel más alto de un problema.

La solución que usamos para tratar el problema de la complejidad es la *abstracción*: dividimos el problema en subproblemas, luego en sub-subproblemas y así sucesivamente, hasta que los problemas resultan suficientemente fáciles de tratar. Una vez que resolvemos uno de los subproblemas no pensamos más sobre los detalles de esa parte, pero tratamos la solución hallada como un bloque de construcción para nuestro siguiente problema. Esta técnica se conoce como la técnica del *divide y reinará*.

Veamos todo lo dicho con un ejemplo. Imaginemos a los ingenieros de una fábrica de coches diseñando un nuevo coche. Pueden pensar en partes del coche tales como: su forma exterior, el tamaño y ubicación del motor, el número y el tamaño de los asientos en la zona de los pasajeros, la cantidad exacta de espacio entre las ruedas, etc. Por otro lado, otro ingeniero (en realidad, este es un equipo de ingenieros pero lo simplificamos un poco en función del ejemplo), cuyo trabajo es diseñar el motor, piensa en las partes que componen un motor: los cilindros, el mecanismo de inyección, el carburador, la electrónica, etc. Piensa en el motor no como una única entidad sino como un trabajo compuesto por varias partes, una de esas partes podría ser una bujía.

Por lo tanto, hay un ingeniero (quizás en una fábrica diferente) que diseña las bujías. Piensa en las bujías como un artefacto compuesto por varias partes. Puede haber hecho estudios complejos para determinar exactamente la clase de metal que debe usar en los contactos o el tipo de material y el proceso de producción a emplear para su aislamiento.

El mismo razonamiento es válido para muchas otras partes del coche. Un diseñador del nivel más alto pensará una rueda como una única parte; otro ingeniero ubicado mucho más abajo en la cadena de diseño pasará sus días pensando sobre la composición química para producir el mejor material para construir los neumáticos. Para el ingeniero de los neumáticos, el neumático es algo complejo. La fábrica de coches comprará los neumáticos a una fábrica por lo que los verá como una única entidad y esto es la abstracción.

Por ejemplo, el ingeniero de la fábrica de coches hace *abstracción* de los detalles de la fabricación de los neumáticos para concentrarse en los detalles de la construcción de una rueda. El diseñador que se ocupa de la forma del coche se abstraerá de los detalles técnicos de las ruedas y del motor para concentrarse en el diseño del cuerpo del coche (se interesará por el tamaño del motor y de las ruedas).

El mismo argumento es cierto para cualquier otro componente. Mientras que algunas personas se ocupan de diseñar el espacio interior del coche, otros trabajan en desarrollar el tejido que usarán para cubrir los asientos.

**Concepto**

La **modularización** es el proceso de dividir un todo en partes bien definidas que pueden ser construidas y examinadas separadamente, las que interactúan de maneras bien definidas.

El punto es: si miramos detalladamente un coche, está compuesto de tantas partes que es imposible que una sola persona conozca todos los detalles de todas las partes al mismo tiempo. Si esto fuera necesario, jamás se hubiera construido un coche.

La razón de que los coches se construyen exitosamente es que los ingenieros usan *modularización* y abstracción: dividen el coche en módulos independientes (rueda, motor, asiento, caja de cambios, etc.) y asignan grupos de gente para trabajar en cada módulo por separado. Cuando construyen un módulo usan abstracción: ven a ese módulo como un componente único que se utiliza para construir componentes más complejos.

La modularización y la abstracción se complementan mutuamente. La modularización es el proceso de dividir cosas grandes (problemas) en partes más pequeñas, mientras que la abstracción es la habilidad de ignorar los detalles para concentrarse en el cuadro más grande.

### 3.3

## Abstracción en software

Los mismos principios de modularización y de abstracción discutidos en la sección anterior se aplican en el desarrollo de software. En el caso de programas complejos, para mantener una visión global del problema tratamos de identificar los componentes que podemos programar como entidades independientes, y luego intentamos utilizar esos componentes como si fueran partes simples sin tener en cuenta su complejidad interna.

En programación orientada a objetos, estos componentes y subcomponentes son objetos. Si estuviéramos tratando de construir un programa que modele un coche mediante un lenguaje orientado a objetos, intentaríamos hacer lo mismo que hacen los ingenieros: en lugar de implementar el coche en un único objeto monolítico, primeramente podríamos construir objetos independientes para un motor, una caja de cambios, una rueda, un asiento, etc., y luego ensamblar el objeto coche a partir de esos objetos más pequeños.

No siempre resulta fácil identificar qué clases de objetos debe tener un sistema de software que resuelve determinado problema. Más adelante en este libro, tendremos mucho más para decir sobre este tema, pero por el momento, comenzaremos con un ejemplo relativamente simple. Y ahora volvamos a nuestro reloj digital.

### 3.4

## Modularización en el ejemplo reloj

Demos una mirada más de cerca al ejemplo *visor-del-reloj*. Usando los conceptos de abstracción de los que hemos hablado, simplemente queremos intentar encontrar la mejor manera de ver este ejemplo, para que podamos escribir algunas clases para implementarlo. Una forma de verlo, es considerarlo como compuesto por un único visor con cuatro dígitos (dos dígitos para la hora y dos para los minutos). Si nos abstraemos nuevamente de ese nivel tan bajo, podemos ver que también se podría considerar el visor como compuesto por dos visores de dos dígitos: un visor de dos dígitos para las horas y otro visor de dos dígitos para los minutos. Un par de dígitos comienza en cero, aumenta en uno cada hora y vuelve a ponerse en cero después de alcanzar su límite 23. El otro par de dígitos se vuelve a poner en cero después de alcanzar su límite 59. La similitud del comportamiento de estos dos visores podría llevarnos a abstraer nuevamente y ver más allá del visor de las horas y del visor de los minutos de manera independiente. Podríamos, en cambio, pensar en ellos como objetos que pueden mostrar valores desde cero hasta un determinado límite. El valor puede ser incrementado,

pero si alcanza el límite vuelve al valor cero. Parece que hemos encontrado un nivel de abstracción adecuado que nos permite representar la situación mediante una sola clase: la clase del visor de dos dígitos.

Para nuestro visor del reloj programaremos primero una clase para representar un visor de un número de dos dígitos (Figura 3.2); le pondremos un método de acceso para tomar su valor y dos métodos modificadores: uno para establecer el valor del límite y otro para incrementarlo. Una vez que tengamos definida esta clase, podremos crear dos objetos de esta clase con diferentes límites para construir el visor del reloj completo.

**Figura 3.2**

El visor de un número de dos dígitos



## 3.5

### Implementación del visor del reloj

Tal como lo mencionamos anteriormente, en vías de construir el visor del reloj construiremos primero un visor que muestra un número de dos dígitos. Este visor necesita almacenar dos valores: uno es el límite hasta el que puede incrementarse el valor antes de volver a cero y el otro es el valor actual. Representaremos ambos en nuestra clase mediante campos enteros (Código 3.1).

#### Código 3.1

Clase para el visor de un número de dos dígitos

```
public class VisorDeNumeros
{
    private int limite;
    private int valor;
    Se omitieron los constructores y los métodos.
}
```

#### Concepto

**Las clases definen tipos.** El nombre de una clase puede ser usado como el tipo de una variable. Las variables cuyo tipo es una clase pueden almacenar objetos de dicha clase.

Más adelante, veremos los restantes detalles de esta clase. Primero, asumimos que podemos construir la clase `VisorDeNumeros` y pensar un poco más sobre el visor del reloj completo como un objeto que internamente está compuesto por dos visores de números (uno para las horas y otro para los minutos). Cada uno de estos visores de números puede ser un campo en el visor del reloj (Código 3.2). Hacemos uso aquí de un detalle que no hemos mencionado antes: las *clases definen tipos*.

#### Código 3.2

La clase `VisorDeReloj` contiene dos `VisorDeNumeros`

```
public class VisorDeReloj
{
    private VisorDeNumeros horas;
    private VisorDeNumeros minutos;
    Se omitieron los constructores y los métodos.
}
```

Cuando hablamos sobre los campos en el Capítulo 2, dijimos que la palabra «**private**» en la declaración del campo va seguida de un tipo y de un nombre para dicho campo. En este caso en particular, usamos la clase **VisorDeNumeros** como el tipo de los campos de nombre **horas** y **minutos**, lo que muestra que los nombres de clase pueden usarse como tipos.

El tipo de un campo especifica la naturaleza del valor que puede almacenarse en dicho campo. Si el tipo es una clase, el campo puede contener objetos de esa clase.

## 3.6

# Comparación de diagramas de clases con diagramas de objetos

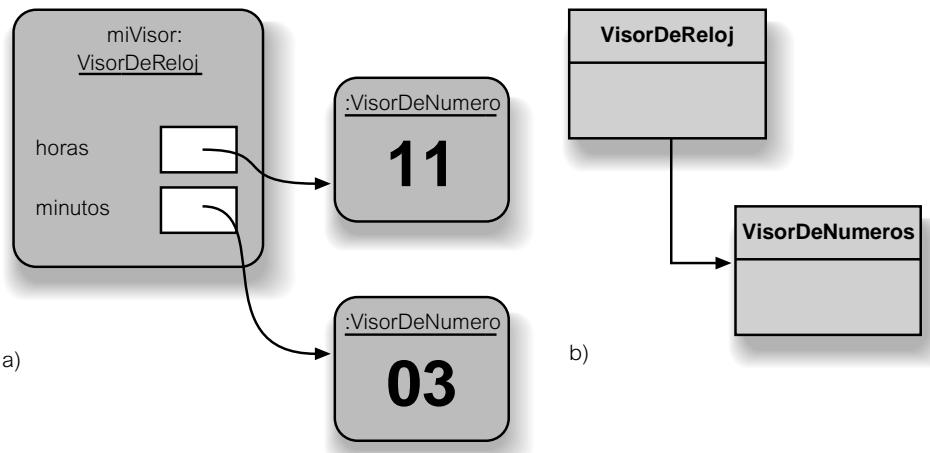
La estructura que hemos descrito en las secciones anteriores (un objeto **VisorDeReloj** que contiene dos objetos **VisorDeNumeros**) puede visualizarse en un *diagrama de objetos* tal como se muestra en la Figura 3.3a. En este diagrama puede ver que estamos trabajando con tres objetos. La Figura 3.3b muestra el diagrama de clases de la misma situación.

**Figura 3.3**

Diagrama de objetos y diagrama de clases del **VisorDeReloj**

### Concepto

El **diagrama de clases** muestra las clases de una aplicación y las relaciones entre ellas. Da información sobre el código. Representa la vista estática de un programa.



Observe que el diagrama de clases muestra solamente dos clases mientras que el diagrama de objetos muestra tres objetos, cuestión que tiene que ver con el hecho de que podemos crear varios objetos de la misma clase. En este caso, creamos dos objetos **VisorDeNumeros** a partir de la clase **VisorDeNumeros** (Código 3.3).

Estos dos diagramas ofrecen vistas diferentes de la misma aplicación. El diagrama de clases muestra una vista estática. Representa lo que tenemos en el momento de escribir el programa: tenemos dos clases y la flecha indica que la clase **VisorDeReloj** hace uso de la clase **VisorDeNumeros** (esto quiere decir que la clase **VisorDeNumeros** es mencionada en el código de la clase **VisorDeReloj**). También decimos que **VisorDeReloj depende de VisorDeNumeros**.

Para iniciar el programa, crearemos un objeto a partir de la clase **VisorDeReloj**. Nosotros programaremos el visor del reloj de modo que cree automáticamente, en el mismo momento en que se inicie el programa, dos objetos **VisorDeNumeros**. Por lo tanto, el diagrama de objetos muestra la situación en *tiempo de ejecución* (cuando la aplicación se está ejecutando); por este motivo, este diagrama se suele llamar *vista dinámica*.

### Concepto

El **diagrama de objetos** muestra los objetos y sus relaciones en un momento dado de la ejecución de una aplicación. Da información sobre los objetos en tiempo de ejecución. Representa la vista dinámica de un programa.

El diagrama de objetos también muestra otro detalle importante: cuando una variable almacena un objeto, éste no es almacenado directamente en la variable sino que en la variable sólo se almacena una *referencia al objeto*. En el diagrama, la variable se muestra como una caja blanca y la referencia al objeto se muestra mediante una flecha. El objeto al que se hace referencia se almacena fuera del objeto que hace la referencia, y la referencia al objeto enlaza la caja de la variable con la caja del objeto.

**Concepto**

**Referencia a un objeto.** Las variables de **tipo objeto** almacenan referencias a los objetos.

Es muy importante comprender las diferencias entre estos dos diagramas y sus respectivas vistas. BlueJ muestra solamente la vista estática: en la ventana principal se visualiza el diagrama de clases. Con la idea de planificar y comprender los programas Java, usted necesita poder construir diagramas de objetos en papel o en su mente. Cuando pensamos sobre qué hará nuestro programa, pensaremos sobre las estructuras de objetos que crean y cómo interactúan esos objetos. Es esencial comenzar a ser capaz de visualizar las estructuras de los objetos.

**Código 3.3**

Implementación de la clase VisorDeNumeros

```
/*
 * La clase VisorDeNumeros representa un visor digital de
números que
 * puede mostrar valores desde cero hasta un determinado
límite.
 * Se puede especificar el límite cuando se crea el
visor. El rango de
 * valores va desde cero (inclusive) hasta el límite-1.
Por ejemplo,
 * si se usa el visor para los segundos de un reloj
digital, el límite
 * podría ser 60, y como resultado se mostrarán los
valores desde 0 hasta 59.
 * Cuando se incrementa el valor, el visor vuelve
automáticamente al
 * valor 0 al alcanzar el valor límite.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 2006.03.30
 */
public class VisorDeNumeros
{
    private int limite;
    private int valor;
    /**
     * Constructor de objetos de la clase VisorDeNumeros
     */
    public VisorDeNumeros(int limiteMaximo)
    {
        limite = limiteMaximo;
        valor = 0;
    }
    /**
     * Devuelve el valor actual.
     */
}
```

**Código 3.3****(continuación)**

Implementación  
de la clase  
VisorDeNumeros

```

        public int getValor()
    {
        return valor;
    }
    /**
     * Configura el valor del visor con el nuevo valor
     * especificado. Si el
     * nuevo valor es menor que cero o si se pasa del
     * límite, no hace nada.
    */
    public void setValor(int nuevoValor)
    {
        if((nuevoValor >= 0) && (nuevoValor < limite))
            valor = nuevoValor;
    }
    /**
     * Devuelve el número del visor (es decir, el valor
     * actual, como una
     * cadena de dos dígitos. Si el valor es menor que
     * 10, se completa con
     * un cero).
    */
    public String getValorDelVisor()
    {
        if(valor < 10)
            return "0" + valor;
        else
            return "" + valor;
    }
    /**
     * Incrementa el valor del visor en uno, lo vuelve
     * a cero si
     * alcanza el valor límite.
    */
    public void incrementar()
    {
        valor = (valor + 1) % limite;
    }
}

```

**Ejercicio 3.1** Piense nuevamente en el proyecto *curso-de-laboratorio* que hemos trabajado en los capítulos 1 y 2. Imagine que crea un objeto **CursoDeLaboratorio** y tres objetos **Estudiante** y luego inscribe a los tres estudiantes en el curso. Intente dibujar un diagrama de clases y un diagrama de objetos para esta situación. Identifique y explique las diferencias entre ellos.

**Ejercicio 3.2** ¿En qué momento puede cambiar un diagrama de clases?  
¿Cómo se cambia?

**Ejercicio 3.3** ¿En qué momento puede cambiar un diagrama de objetos?  
¿Cómo se cambia?

**Ejercicio 3.4** Escriba la declaración de un campo de nombre `tutor` que pueda contener referencias a objetos de tipo `Instructor`.

## 3.7

### Tipos primitivos y tipos objeto

#### Concepto

Los **tipos primitivos** en Java son todos los tipos que no son objetos. Los tipos primitivos más comunes son los tipos `int`, `boolean`, `char`, `double` y `long`. Los tipos primitivos no poseen métodos.

Java reconoce dos clases de tipos muy diferentes: los *tipos primitivos* y los *tipos objeto*. Los tipos primitivos están todos predefinidos en el lenguaje Java; incluyen los tipos `int` y `boolean`. En el Apéndice B se ofrece una lista completa de los tipos primitivos de Java. Los tipos objeto son aquellos que se definen mediante clases. Algunas clases están definidas por el sistema Java estándar (como por ejemplo, la clase `String`), otras son las clases que escribimos nosotros mismos. Tanto los tipos primitivos como los tipos objeto pueden ser usados como tipos, pero existen situaciones en las que se comportan de manera muy diferente. Una diferencia radica en cómo se almacenan los valores. Como podemos ver en nuestros diagramas, los valores primitivos se almacenan directamente en una variable (hemos escrito los valores directamente en una caja de variable, como por ejemplo, en el Capítulo 2, Figura 2.3). Por otro lado, los objetos no se almacenan directamente en una variable sino que se almacena una referencia al objeto (dibujada en los diagramas como una flecha, Figura 3.3a).

Más adelante veremos otras diferencias entre los tipos primitivos y los tipos objeto.

## 3.8

### El código del VisorDeReloj

Antes de comenzar a analizar el código, le será de ayuda explorar el ejemplo por sí mismo.

**Ejercicio 3.5** Inicie BlueJ, abra el ejemplo `visor-de-reloj` y experimente con él.

Para usarlo, cree un objeto `VisorDeReloj` y abra la ventana del inspector. invoque los métodos del objeto manteniendo abierta la ventana del inspector. Observe el campo `cadVisor` en el inspector. Lea el documento del proyecto para obtener más información (haciendo doble clic sobre la nota de texto en la ventana principal).

#### 3.8.1

#### Clase VisorDeNumeros

Ahora analizaremos la implementación completa de esta tarea. El proyecto `visor-de-reloj` en los ejemplos adjuntados a este libro contiene la solución. Primeramente veremos la implementación de la clase `VisorDeNumeros`; el Código 3.3 muestra el código completo de esta clase. En su conjunto, esta es una clase bastante clara; tiene dos campos de los que hemos hablado con anterioridad (Sección 3.5), un constructor y cuatro métodos (`setValor`, `getValor`, `getValorDelVisor` e `incrementar`).

El constructor recibe mediante un parámetro, el límite para volver el valor a cero. Por ejemplo, si se pasa 24 como límite, el visor volverá a cero cuando alcance dicho valor. Por lo que el rango de valores para el visor será desde cero hasta 23. Esta característica nos permite usar esta clase tanto para el visor de horas como para el visor de minutos. Para el visor de horas creamos un `VisorDeNumeros` con límite 24, para el visor de minutos creamos otro con límite 60.

Entonces, el constructor almacena en un campo el límite para volver a cero y pone en cero el valor actual del visor.

A continuación, se presenta un método de acceso para el valor del visor actual (`getValor`). Este método permite que otros objetos lean el valor actual del visor.

El siguiente método, `setValor`, es un método de modificación y es más interesante.

```
public void setValor(int nuevoValor)
{
    if((nuevoValor >= 0) && (nuevoValor < limite))
        valor = nuevoValor;
}
```

Pasamos al método el nuevo valor para el visor mediante un parámetro. Sin embargo, antes de asignar su valor, tenemos que verificar si es válido. El rango de validez para este valor, tal como lo discutimos anteriormente, va desde cero hasta uno menos que el valor del límite. Usamos una sentencia condicional (`if`) para controlar que el valor sea válido antes de asignarlo. El símbolo «`&&`» es el operador lógico «y»; obliga a que la condición de la sentencia condicional sea verdadera cuando ambas condiciones a ambos lados del símbolo «`&&`» sean verdaderas. Para más detalles, vea la nota *Operadores Lógicos* que está a continuación. El Apéndice D muestra una tabla completa de los operadores lógicos de Java.

### Operadores Lógicos

Los operadores lógicos operan con valores booleanos (verdadero o falso) y producen como resultado un nuevo valor booleano. Los tres operadores lógicos más importantes son «y», «o» y «no». En Java se escriben:

<b>&amp;&amp;</b>	(y)
<b>  </b>	(o)
<b>!</b>	(no)

La expresión

**a && b**

es verdadera si tanto **a** como **b** son verdaderas, en todos los otros casos es falsa.

La expresión

**a || b**

es verdadera si alguna de las dos es verdadera, puede ser **a** o puede ser **b** o pueden ser las dos; si ambas son falsas el resultado es falso. La expresión

**!a**

es verdadera si **a** es falso, y es falsa si **a** es verdadera.

**Ejercicio 3.6** ¿Qué ocurre cuando se invoca el método `setValor` con un valor no válido? ¿Es una solución buena? ¿Puede pensar una solución mejor?

**Ejercicio 3.7** ¿Qué ocurre si en la condición reemplaza el operador «`>=`» por el operador «`>`»? Es decir:

```
if((nuevoValor > 0) && (nuevoValor < limite))
```

**Ejercicio 3.8** ¿Qué ocurriría si en la condición reemplaza el operador «`&&`» por el operador «`||`»?, de modo que:

```
if((nuevoValor > 0) || (nuevoValor < limite))
```

**Ejercicio 3.9** ¿Cuáles de las siguientes expresiones resultan verdaderas?

```
! (4 < 5)
! false
(2 > 2) || ((4 == 4) && (1 < 0))
(2 > 2) || (4 == 4) && (1 < 0)
(34 != 33) && ! false
```

**Ejercicio 3.10** Escriba una expresión usando las variables booleanas **a** y **b** que dé por resultado verdadero cuando una de las dos sea verdadera o cuando ambas sean falsas.

**Ejercicio 3.11** Escriba una expresión usando las variables booleanas **a** y **b** que dé por resultado verdadero solamente cuando una de las dos sea verdadera, y que dé falso cuando ambas sean falsas o cuando ambas sean verdaderas (esta operación se suele llamar «o exclusivo» o *disyunción excluyente*).

**Ejercicio 3.12** Considere la expresión `(a && b)`. Escriba una expresión equivalente sin utilizar el operador `&&`. (Es decir, una expresión que se evalúe como verdadera sólo cuando ambas sean verdaderas.)

El siguiente método, `getValorDelVisor`, también devuelve el valor del visor pero en un formato diferente. La razón es que queremos mostrar el valor con una cadena de dos dígitos. Es decir, si la hora actual es 3:05, queremos mostrar «03:05» y no «3:5». Para hacer esto más fácilmente hemos implementado el método `getValorDelVisor`. Este método devuelve el valor actual del visor como una cadena y agrega un cero si el valor es menor que 10. Aquí presentamos el fragmento de código que resulta relevante:

```
if(valor < 10)
    return "0" + valor;
else
    return "" + valor;
```

Observe que el cero («0») está escrito entre comillas dobles. Es por este motivo que hemos hablado de la *cadena 0* y no del número entero 0. Luego la expresión

```
"0" + valor
```

«suma» una cadena y un entero (ya que el tipo de `valor` es entero). Pero en este caso el operador «más» representa nuevamente una concatenación de cadenas, tal como lo explicamos en la Sección 2.8. Antes de continuar, veamos más de cerca la concatenación de cadenas.

### 3.8.2 Concatenación de cadenas

El operador suma (+) tiene diferentes significados dependiendo del tipo de sus operandos. Si ambos operandos son números, el operador + representa la adición tal como esperamos. Por lo tanto,

```
42 + 12
```

suma esos dos números y su resultado es 54. Sin embargo, si los operandos son cadenas, el significado del signo más es la concatenación de cadenas y el resultado es una única cadena compuesta por los dos operandos. Por ejemplo, el resultado de la expresión

```
"Java" + "con BlueJ"
```

es una sola cadena que es

```
"Javacon BlueJ"
```

Observe que el sistema no agrega automáticamente espacios entre las cadenas. Si quiere tener un espacio entre ellas debe incluirlo usted mismo dentro de una de las cadenas a concatenar.

Si uno de los operandos del operador más es una cadena y el otro no, el operando que no es cadena es convertido automáticamente en una cadena y luego se realiza la concatenación correspondiente. Por ejemplo:

```
"respuesta: " + 42
```

da por resultado la cadena

```
"respuesta: 42"
```

Esta conversión funciona para todos los tipos. Cualquier tipo que se «sume» con una cadena, automáticamente es convertido a una cadena y luego concatenado.

Volviendo al código del método `getValorDelVisor`, si `valor` contiene, por ejemplo un 3, la sentencia

```
return "0" + valor;
```

devolverá la cadena «03». Para el caso en que el valor sea mayor que 9, hemos usamos el siguiente truco:

```
return "" + valor;
```

En la última sentencia concatenamos `valor` con una cadena vacía. El resultado es que `valor` será convertido en una cadena sin agregar ningún carácter delante de él. Usamos el operador suma con el único propósito de forzar la conversión de un valor entero a un valor de tipo `String`.

**Ejercicio 3.13** ¿Funciona correctamente el método `getValorDelVisor` en todas las circunstancias? ¿Qué supuestos se han hecho? ¿Qué ocurre si, por ejemplo, crea un visor de números con un límite 800?

**Ejercicio 3.14** ¿Existe alguna diferencia entre los resultados al escribir las sentencias siguientes en el método `getValorDelVisor`? Es decir, al escribir

```
return valor + "";
```

en lugar de

```
return "" + valor;
```

### 3.8.3 El operador módulo

El último método de la clase `VisorDeNumeros` incrementa el valor del visor en 1 y cuida que el valor vuelva a ser cero cuando alcanza el límite:

```
public void incrementar()
{
    valor = (valor + 1) % limite;
}
```

Este método usa el operador *módulo* (%). El operador módulo calcula el resto de una división entera. Por ejemplo, el resultado de la división

$27/4$

puede expresarse en números enteros como

```
resultado = 6, resto = 3
```

La operación módulo justamente devuelve el resto de la división, por lo que el resultado de la expresión  $(27\%4)$  será 3.

**Ejercicio 3.15** Explique cómo funciona el operador módulo. Puede ocurrir que necesite consultar más recursos para encontrar los detalles (recursos online del lenguaje Java, libros de Java, etc.).

**Ejercicio 3.16** ¿Cuál es el resultado de la expresión  $(8\%3)$ ?

**Ejercicio 3.17** Si **n** es una variable entera, ¿cuáles son todos los resultados posibles de la expresión  $(n\%5)$ ?

**Ejercicio 3.18** Si **n** y **m** son variables enteras, ¿cuáles son todos los posibles resultados de la expresión  $(n\%m)$ ?

**Ejercicio 3.19** Explique detalladamente cómo trabaja el método `incrementar`.

**Ejercicio 3.20** Rescriba el método `incrementar` sin el operador módulo, usando, en cambio, una sentencia condicional. ¿Cuál de las soluciones es mejor?

**Ejercicio 3.21** Usando el proyecto *visor-de-reloj* en BlueJ, pruebe la clase `VisorDeNumeros` creando algunos objetos `VisorDeNumeros` e invocando sus métodos.

### 3.8.4 La clase `VisorDeReloj`

Ahora que hemos visto cómo podemos construir una clase que define un visor para un número de dos dígitos, podremos ver con más detalle la clase `VisorDeReloj`, la clase que creará dos visores de números para crear un visor con la hora completa. En Código 3.4 se muestra el código de la clase `VisorDeReloj` completa.

Tal como lo hicimos con la clase `VisorDeNumeros`, discutiremos brevemente sobre todos sus campos, constructores y métodos.

#### Código 3.4

Implementación de la clase  
`VisorDeReloj`

```
/**
 * La clase VisorDeReloj implementa un visor para un reloj
digital
 * de estilo europeo de 24 horas. El reloj muestra horas
y minutos.
```

**Código 3.4****(continuación)**

Implementación de la clase  
VisorDeReloj

```
* El rango del reloj va desde las 00:00 (medianoche)
hasta las 23:59
* (un minuto antes de medianoche)
*
* El visor del reloj recibe "tics " en cada
minuto(mediante el método
* ticTac) y reacciona incrementando el visor. Esto es lo
que hacen los
* relojes modernos: se incrementa la hora cuando los
minutos vuelven
* a cero.
*
* @author Michael Kölling and David J. Barnes
* @version 2006.03.30
*/
public class VisorDeReloj
{
    private VisorDeNumeros horas;
    private VisorDeNumeros minutos;
    private String cadvisor; // simula el visor actual
del reloj

    /**
     * Constructor de objetos VisorDeReloj. Este constructor
     * crea un nuevo reloj puesto en hora con el valor 00:00.
     */
    public VisorDeReloj()
    {
        horas = new VisorDeNumeros(24);
        minutos = new VisorDeNumeros(60);
        actualizarVisor();
    }
    /**
     * Constructor de objetos VisorDeReloj. Este constructor
     * crea un nuevo reloj puesto en hora con el valor
especificado
     * por sus parámetros.
     */
    public VisorDeReloj(int hora, int minuto)
    {
        horas = new VisorDeNumeros(24);
        minutos = new VisorDeNumeros(60);
        ponerEnHora(hora, minuto);
    }
    /**
     * Este método debe invocarse una vez por cada
minuto; hace
     * que el visor avance un minuto.
     */
    public void ticTac()
    {
        minutos.incrementar();
```

**Código 3.4  
(continuación)**

Implementación de la clase VisorDeReloj

```

        if(minutos.getValor() == 0) { // ¡alcanzó el
            limite!
                horas.incrementar();
            }
            actualizarVisor();
        }
    /**
     * Pone en hora el visor con la hora y los minutos
     * especificados
     */
    public void ponerEnHora(int hora, int minuto)
    {
        horas.setValor(hora);
        minutos.setValor(minuto);
        actualizarVisor();
    }
    /**
     * Devuelve la hora actual del visor en el formato
     * HH:MM.
     */
    public String getHora()
    {
        return cadVisor;
    }

    /**
     * Actualiza la cadena interna que representa al visor.
     */
    private void actualizarVisor()
    {
        cadVisor = horas.getValorDelVisor() + ":" +
minutos.getValorDelVisor();
    }
}

```

En este proyecto usamos el campo `cadVisor` para simular el dispositivo visor del reloj (como habrá podido ver en el Ejercicio 3.5). Si este software se ejecutara en un reloj real, presentaríamos los cambios en su visor en lugar de representarlo mediante una cadena. De modo que, en nuestro programa de simulación, esta cadena funciona como el dispositivo de salida del reloj.

Para lograr esta simulación usamos un campo cadena y un método:

```

public class VisorDeReloj
{
private String cadVisor;
Se omitieron otros campos y métodos.
/**
 * Actualiza la cadena interna que representa al visor.
 */
private void actualizarVisor()

```

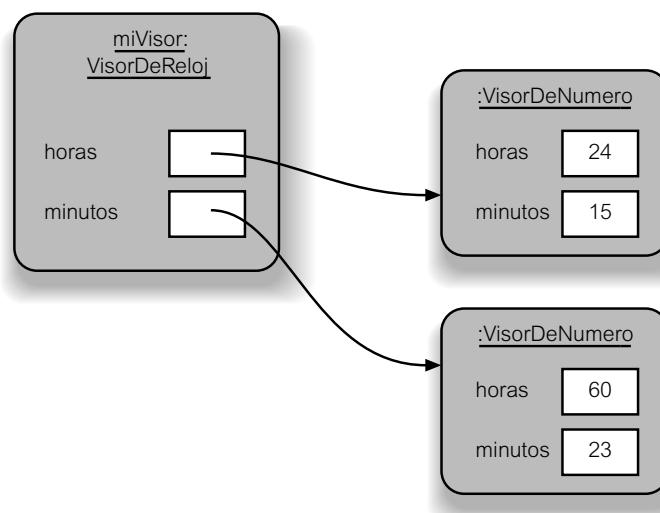
```
{
Se omitió la implementación del método.
}
}
```

Cada vez que queremos que el visor del reloj cambie, llamaremos al método interno `actualizarVisor`. En nuestra simulación, este método cambiará la cadena del visor (a continuación, examinaremos el código que lleva a cabo esta tarea). En un reloj real, este método también existiría y cambiaría su visor.

Además de la cadena para el visor, la clase `VisorDeReloj` tiene sólo dos campos más: `horas` y `minutos`. Cada uno de estos campos puede contener un objeto de tipo `VisorDeNumeros`. El valor lógico del visor del reloj (es decir, la hora actual) se almacena en estos objetos `VisorDeNumeros`. La Figura 3.4 muestra un diagrama de objetos de esta aplicación cuando la hora actual es 15:23.

**Figura 3.4**

Diagrama de objetos del visor del reloj



## 3.9

## Objetos que crean objetos

La primera pregunta que nos hacemos es: ¿de dónde provienen estos tres objetos? Cuando queremos usar un visor de un reloj debemos crear un objeto `VisorDeReloj`, por lo tanto, asumimos que nuestro reloj muestra horas y minutos. Es decir, que con sólo crear un visor de reloj esperamos que implícitamente se creen dos visores de números, uno para las horas y otro para los minutos.

### Concepto

**Creación de objetos.** Los objetos pueden crear otros objetos usando el operador `new`.

Como escritores de la clase `VisorDeReloj` tenemos que lograr que ocurra esto y para ello, simplemente escribimos código en el constructor del `VisorDeReloj` que crea y almacena dos objetos `VisorDeNumeros`. Dado que el constructor se ejecuta automáticamente cuando se crea un nuevo objeto `VisorDeReloj`, los objetos `VisorDeNumeros` serán creados automáticamente al mismo tiempo. A continuación, está el código del constructor de `VisorDeReloj` que lleva a cabo este trabajo:

```
public class VisorDeReloj
{
    private VisorDeNumeros horas;
```

```
private VisorDeNumeros minutos;
```

*Se omitieron los restantes campos.*

```
public VisorDeReloj()
{
    horas = new VisorDeNumeros(24);
    minutos = new VisorDeNumeros(60);
    actualizarVisor();
}
```

*Se omitieron los métodos.*

}

Cada una de las dos primeras líneas del constructor crea un nuevo objeto `VisorDeNumeros` y lo asigna a una variable. La sintaxis de una operación para crear un objeto nuevo es:

```
new NombreDeClase (lista-de-parámetros)
```

La operación `new` hace dos cosas:

1. Crea un nuevo objeto de la clase nombrada (en este caso, `VisorDeReloj`).
2. Ejecuta el constructor de dicha clase.

Si el constructor de la clase tiene parámetros, los parámetros actuales deben ser proporcionados en la sentencia `new`. Por ejemplo, el constructor de la clase `VisorDeNumeros` fue definido para esperar un parámetro de tipo entero:

```
public VisorDeNumeros (int limiteMaximo)
```

A callout line points from the word "parámetro formal" to the parameter "limiteMaximo" in the code.

Por lo tanto, la operación `new` sobre la clase `VisorDeNumeros` que invoca a este constructor, debe proveer un parámetro actual de tipo entero para que coincida con el encabezado que define el constructor:

```
new VisorDeNumeros (24);
```

A callout line points from the word "parámetro actual" to the value "24" in the code.

Esta es la misma cuestión de la que hablamos sobre los métodos en la Sección 2.4. Con este constructor hemos logrado lo que queríamos: si alguien crea un nuevo objeto `VisorDeReloj`, se ejecutará automáticamente su constructor y éste, a su vez, creará dos objetos `VisorDeNumeros`, dejando al visor de reloj listo para funcionar.

**Ejercicio 3.22** Cree un objeto `VisorDeReloj` seleccionando el siguiente constructor:

```
new VisorDeReloj()
```

Llame a su método `getHora` para encontrar la hora con que inicia el reloj. ¿Puede explicar por qué comienza con esa hora en particular?

**Ejercicio 3.23** Sobre un objeto `VisorDeReloj` recién creado, ¿cuántas veces necesita invocar al método `ticTac` para que llegue a la hora 01:00? ¿Qué otra cosa podría hacer para que muestre la misma hora?

**Ejercicio 3.24** Escriba la signatura de un constructor que se ajuste a la siguiente instrucción de creación de un objeto:

```
new Editor ("leeme.txt", -1)
```

**Ejercicio 3.25** Escriba sentencias Java que definan una variable de nombre `ventana` y de tipo `Rectangulo`; luego cree un objeto rectángulo y asígnelo a dicha variable. El constructor del rectángulo tiene dos parámetros de tipo `int`.

## 3.10

### Constructores múltiples

Al crear objetos `visorDeReloj`, seguramente habrá notado que el menú contextual ofrece dos formas de hacerlo:

```
new VisorDeReloj()
new VisorDeReloj(hora, minuto)
```

#### Concepto

**Sobrecarga.** Una clase puede contener más de un constructor o más de un método con el mismo nombre, siempre y cuando tengan distintos conjuntos de parámetros que se diferencien por sus tipos.

Es así porque la clase contiene dos constructores que proveen formas alternativas de inicializar un objeto `VisorDeReloj`. Si se usa el constructor que no tiene parámetros, la primer hora que se mostrará en el reloj será 00:00. Por otra parte, si desea tener una hora inicial diferente, puede establecerla usando el segundo constructor. Es común que las declaraciones de clases contengan versiones alternativas de constructores o métodos que proporcionan varias maneras de llevar a cabo una tarea en particular mediante diferentes conjuntos de parámetros. Este punto se conoce como *sobrecarga* de un constructor o método.

**Ejercicio 3.26** Busque en el código de `VisorDeReloj` el segundo constructor. Explique qué hace y cómo lo hace.

**Ejercicio 3.27** Identifique las similitudes y las diferencias entre los dos constructores. ¿Por qué no hay una llamada al método `actualizarVisor` en el segundo constructor?

## 3.11

### Llamadas a métodos

#### 3.11.1

#### Llamadas a métodos internos

La última línea del primer constructor de `VisorDeReloj` es la sentencia

```
actualizarVisor();
```

Esta sentencia es una *llamada a un método*. Como hemos visto anteriormente, la clase `VisorDeReloj` tiene un método con la siguiente signatura:

```
private void actualizarVisor()
```

La llamada a método que mostramos en la línea anterior, justamente invoca a este método. Dado que este método está ubicado en la misma clase en que se produce su

**Concepto**

Los métodos pueden llamar a otros métodos de la misma clase como parte de su implementación. Esto se denomina **llamada a método interno**.

llamada, decimos que es una *llamada a un método interno*. Las llamadas a métodos internos tienen la siguiente sintaxis:

```
nombreDelMétodo (lista-de-parámetros)
```

En nuestro ejemplo, el método no tiene ningún parámetro por lo que la lista de parámetros queda vacía: este es el significado de los dos paréntesis sin nada entre ellos.

Cuando se encuentra una llamada a un método, se ejecuta este último, y luego de su ejecución se vuelve a la llamada al método y se continúa con la sentencia que sigue a la invocación. Para que la llamada a un método coincida con la firma del mismo, deben coincidir tanto el nombre del método como su lista de parámetros. En este caso, ambas listas de parámetros están vacías, por lo tanto, coinciden. Esta necesidad de que coincidan tanto el nombre del método como la lista de parámetros es importante porque, si el método está sobrecargado, podría haber más de un método con el mismo nombre en una clase.

En nuestro ejemplo, el propósito de este método es actualizar la cadena del visor. Después de que se crean los dos visores de números, la cadena del visor se configura para mostrar la hora indicada por dichos objetos. A continuación, discutiremos la implementación del método `actualizarVisor`.

### 3.11.2 Llamadas a métodos externos

Ahora, examinaremos el siguiente método: `ticTac`. Su definición es:

```
public void ticTac()
{
    minutos.incrementar();
    if(minutos.getValor() == 0) { // ¡alcanzó el
        límite!
        horas.incrementar();
    }
    actualizarVisor();
}
```

**Concepto**

Los métodos pueden llamar a métodos de otros objetos usando la notación de punto: se denomina **llamada a método externo**.

Si este visor se conectara a un reloj real, este método sería invocado una vez cada 60 segundos por el temporizador electrónico del reloj. Por ahora, lo llamamos nosotros para probar el visor.

Cuando es llamado, el método `ticTac` ejecuta primero la sentencia

```
minutos.incrementar();
```

Esta sentencia llama al método `incrementar` del objeto `minutos`. Cuando se llama a uno de los métodos del objeto `VisorDeReloj`, este método a su vez llama a un método de otro objeto para colaborar en la tarea. Una llamada a método desde un método de otro objeto se conoce como *llamada a un método externo*. La sintaxis de una llamada a un método externo es

```
objeto.nombreDelMétodo (lista-de-parámetros)
```

Esta sintaxis se conoce con el nombre de «*notación de punto*». Consiste en un nombre de objeto, un punto, el nombre del método y los parámetros para la llamada. Es particularmente importante apreciar que usamos aquí el nombre de un objeto y no el

nombre de una clase: usamos el nombre `minutos` en lugar del nombre `VisorDeNumeros`.

A continuación, el método `ticTac` tiene una sentencia condicional que verifica si deben ser incrementadas las horas. Forma parte de la condición de la sentencia `if` una llamada a otro método del objeto `minutos`: `getValor` que devuelve el valor actual de los minutos. Si este valor es cero, sabemos entonces que el visor alcanzó su límite y que debemos incrementar las horas, y esto es exactamente lo que hace este fragmento de código.

Si el valor de los minutos no es cero, no tenemos que hacer ningún cambio en las horas, por lo tanto, la sentencia `if` no necesita de su parte `else`.

Ahora estamos en condiciones de comprender los restantes tres métodos de la clase `VisorDeReloj` (véase Código 3.4). El método `setHora` tiene dos parámetros, la hora y los minutos, y pone el reloj en la hora especificada. Observando el cuerpo del método, podemos ver que realiza esta tarea llamando a los métodos `setValor` de ambos visores de números, uno para las horas y otro para los minutos; luego invoca al método `actualizarValor` para actualizar la cadena del visor acorde con los nuevos valores, tal como lo hace el constructor.

El método `getHora` es trivial, sólo devuelve la cadena actual del visor. Dado que mantenemos siempre la cadena del visor actualizada, es todo lo que hay que hacer.

Finalmente, el método `actualizarVisor` es responsable de actualizar la cadena del visor para que refleje correctamente la hora representada por los dos objetos visores de números. Se lo llama cada vez que cambia la hora del reloj y trabaja invocando los métodos `getValorDelVisor` de cada uno de los objetos `VisorDeNumeros`. Estos métodos devuelven el valor de cada visor de números por separado y luego se usa la concatenación de cadenas para unir estos dos valores con dos puntos entre medias de ellos y dar por resultado una única cadena.

**Ejercicio 3.28** Sea la variable

`Impresora p1;`

que actualmente contiene un objeto impresora, y dos métodos dentro de la clase `Impresora` con los siguientes encabezados

```
public void imprimir (String nombreDeArchivo, boolean  
dobleFaz)  
public int consultarEstado (int espera)
```

Escriba dos llamadas posibles a cada uno de estos métodos.

### 3.11.3 Resumen del visor de reloj

Es importante que nos tomemos un minuto para ver la manera en que este ejemplo hace uso de la abstracción para dividir el problema en partes más pequeñas. Viendo el código de la clase `VisorDeReloj` notará que sólo creamos un objeto `VisorDeNumeros` sin interesarnos demasiado en lo que este objeto hace internamente. Sólo hemos llamado a los métodos de ese objeto (`incrementar` y `getValor`) para que hagan el trabajo por nosotros. En este nivel, asumimos que el método `incrementar` aumentará correctamente el valor en el visor del reloj sin tener en cuenta cómo lo logra.

En los proyectos reales, las diferentes clases son frecuentemente escritas por diferentes personas. Como habrá notado, estas dos personas deben acordar las firmas que tendrán las clases y lo que pueden hacer estas clases. Después del acuerdo, una persona se puede concentrar en implementar los métodos mientras que la otra puede hacer uso de ellos.

El conjunto de métodos de un objeto que está disponible para otros objetos se denomina su *interfaz*. Trataremos más adelante en este libro las interfaces con más detalle.

**Ejercicio 3.29** *Desafío.* Modifique el reloj de 24 horas por un reloj de 12 horas.

Tenga cuidado, no es tan fácil como parece a primera vista. En un reloj de 12 horas, después de la medianoche y después del mediodía no se muestra la hora como 00:30 sino como 12:30. Por lo tanto, los minutos varían desde 0 hasta 59 mientras que las horas que se muestran varían desde 1 hasta 12.

**Ejercicio 3.30** Hay por lo menos dos maneras de construir un reloj de 12 horas. Una posibilidad es almacenar la hora con valores desde 1 hasta 12. Por otro lado, puede dejar que el reloj trabaje internamente como un reloj de 24 horas pero modificar la cadena del visor para que muestre, por ejemplo, 4:23 o 4:23 pm cuando el valor interno sea 16:23. Implemente ambas versiones. ¿Qué opción es la más fácil? ¿Cuál es la mejor? ¿Por qué?

## 3.12

### Otro ejemplo de interacción de objetos

Examinaremos ahora los mismos conceptos pero con un ejemplo diferente y usando otras herramientas. Estamos empeñados en comprender cómo los objetos crean otros objetos y en cómo los objetos llaman a los métodos de otros objetos. En la primera mitad de este capítulo hemos usado la técnica más fundamental para analizar un programa: la lectura de código. La habilidad para leer y comprender código es una de las habilidades más esenciales de un desarrollador de software y necesitaremos aplicarla en cada proyecto en que trabajemos. Sin embargo, algunas veces resulta beneficioso usar herramientas adicionales que nos ayudan a comprender más profundamente cómo se ejecuta un programa. Una de estas herramientas que veremos ahora es el *depurador* (*debugger*).

#### Concepto

Un **depurador** es una herramienta de software que ayuda a examinar cómo se ejecuta una aplicación. Puede usarse para encontrar problemas.

Un depurador es un programa que permite que los programadores ejecuten una aplicación de a un paso por vez. Generalmente, ofrece funciones para detener y comenzar la ejecución de un programa en un punto seleccionado del código y para examinar los valores de las variables.

Los depuradores varían mucho en cuanto a su complejidad. Los que usan los desarrolladores profesionales tienen un gran número de funciones útiles para realizar análisis sofisticados de varias fases de una aplicación. BlueJ tiene un depurador que es mucho más sencillo; podemos usarlo para detener un programa, recorrer el código línea por línea y examinar los valores de nuestras variables. A pesar de su falta de sofisticación, nos alcanza para obtener gran cantidad de información.

Antes de comenzar a experimentar con el depurador, veremos un ejemplo con el que lo usaremos: una simulación de un sistema de correo electrónico.

### El término «debugger»

Los errores en los programas de computación se conocen comúnmente como «bugs», por lo que los programas que ayudan a eliminar dichos errores se conocen como «debuggers».

No está muy claro el origen del término «bug». Hay un caso famoso que se conoce como «El primer bug en computación», se trata de un insecto real (una polilla, en realidad) detectado por Grace Murray Hopper, una de las pioneras en computación, dentro de la computadora Mark II, en el año 1945. En el Smithsonian Institute del National Museum of American History existe un libro de registros que muestra una cita, con la polilla pegada con cinta en el libro, que reza «el primer caso real de un bug encontrado». La redacción, sin embargo, sugiere que el término «bug» se utilizaba antes de la aparición del insecto que causó el problema en la Mark II.

Para encontrar más información sobre este caso, búsqelo en un sitio web como «first computer bug» y ¡encontrará hasta las imágenes de esta polilla!

## 3.12.1 El ejemplo del sistema de correo electrónico

Comenzamos investigando la funcionalidad del proyecto *sistema-de-correo*. En este momento, para comprender mejor las tareas que realiza este proyecto, no es tan importante leer su código sino que es más conveniente ejecutarlo.

**Ejercicio 3.31** Abra el proyecto *sistema-de-correo* que puede encontrar en el material de soporte de este libro. La idea de este proyecto es simular las acciones de usuarios que se envían correos electrónicos entre ellos. Un usuario utiliza un cliente de correo para enviar mensajes a un servidor que se encarga de despacharlos al cliente de correo de otro usuario. Primero, cree un objeto **ServidorDeCorreo**. Luego cree un objeto **ClienteDeCorreo** para cada uno de los usuarios. En el momento de crear un cliente necesitará aportar la instancia de **ServidorDeCorreo** como un parámetro: utilice el que ha creado al principio. También necesitará especificar un nombre de usuario para el cliente de correo electrónico. A continuación, cree un segundo **ClienteDeCorreo** de manera similar al anterior pero con otro nombre de usuario.

Experimente con los objetos **ClienteDeCorreo** que pueden usarse para enviar mensajes de un cliente de correo a otro (mediante el método **enviarMensaje**) y para recibir mensajes (mediante los métodos **getMensajeSiguiente** o **imprimirMensajeSiguiente**).

Examinando el proyecto sistema de correo electrónico verá que:

- Tiene tres clases: **ServidorDeCorreo**, **ClienteDeCorreo** y **Mensaje**.
- Debe crearse un objeto servidor de correo que es usado por todos los clientes de correo y maneja el intercambio de los mensajes.
- Se pueden crear varios objetos clientes de correo. Cada cliente tiene un nombre de usuario asociado.
- Los mensajes pueden enviarse desde un cliente a otro mediante un método de la clase cliente de correo.

- Un cliente puede recibir los mensajes desde el servidor de a uno por vez, usando un método del cliente de correo.
- La clase `Mensaje` jamás es instanciada explícitamente por el usuario. Se usa internamente en los clientes de correo y en el servidor para almacenar e intercambiar los mensajes.

**Ejercicio 3.32** Dibuje un diagrama de objetos para la situación que se tiene después de crear un servidor de correo y tres clientes. Los diagramas de objetos fueron tratados en la Sección 3.6.

Las tres clases tienen diferente grado de complejidad. La clase `Mensaje` es bastante trivial. Aquí discutiremos solamente sobre un pequeño detalle de esta clase y dejamos al lector que investigue el resto de la misma por su propia cuenta. La clase `ServidorDeCorreo` es, en este punto, muy compleja ya que usa conceptos que trataremos mucho más adelante en este libro. No analizaremos esta clase en detalle ahora, sólo confiamos en que hace bien su trabajo (otro ejemplo del modo en que se usa la abstracción para ocultar detalles que no se necesitan para seguir adelante).

La clase `ClienteDeCorreo` es la más interesante y la examinaremos con más detalle.

### 3.12.2 La palabra clave `this`

La única sección de código de la clase `Mensaje` que analizaremos es el constructor, que usa una construcción de Java que no hemos hallado anteriormente. El código de esta clase se muestra en Código 3.5.

#### Código 3.5

Los campos y el constructor de la clase `Mensaje`

```
public class Mensaje
{
    // El remitente del mensaje.
    private String de;
    // El destinatario del mensaje.
    private String para;
    // El texto del mensaje.
    private String texto;
    /**
     * Crea un mensaje de correo del remitente para un
     * destinatario
     * dado, que contiene el texto especificado.
     * @param de    El remitente de este mensaje.
     * @param para  El destinatario de este mensaje.
     * @param texto El texto del mensaje que será enviado.
     */
    public Mensaje(String de, String para, String texto)
    {
        this.de = de;
        this.para = para;
        this.texto = texto;
    }
    Se omitieron los métodos.
}
```

La nueva característica de Java que aparece en este fragmento de código es el uso de la palabra clave `this`:

```
this.de = de;
```

La línea en su totalidad es una sentencia de asignación: asigna el valor del lado derecho (`de`) a la variable que está del lado izquierdo (`this.de`) del símbolo igual (`=`).

El motivo por el que se usa esta construcción radica en que tenemos una situación que se conoce como *sobrecarga de nombres*, y significa que el mismo nombre es usado por entidades diferentes. La clase contiene tres campos de nombres `de`, `para` y `texto`. ¡Y el constructor tiene tres parámetros con los mismos tres nombres: `de`, `para` y `texto`!

De modo que, mientras se está ejecutando el constructor, ¿cuántas variables existen? La respuesta es seis: tres campos y tres parámetros. Es importante comprender que los campos y los parámetros son variables que existen independientemente unas de otras, aun cuando comparten nombres similares. Un parámetro y un campo que comparten un nombre no representan un problema para Java.

Por lo tanto, el problema que tenemos es cómo hacer referencia a las seis variables de modo que se pueda distinguir entre los dos conjuntos. Si usamos en el constructor simplemente el nombre de variable «`de`» (por ejemplo, en una sentencia `System.out.println(de)`), ¿qué variable se usará, el parámetro o el campo?

La especificación de Java responde a esta pregunta: Java especifica que siempre se usará la declaración más cercana encerrada en un bloque. Dado que el parámetro `de` está declarado en el constructor y el campo `de` está declarado en la clase, se usará el parámetro pues su declaración es la más cercana a la sentencia que lo usa.

Ahora, todo lo que necesitamos es un mecanismo para acceder a un campo cuando existe una variable con el mismo nombre declarada más cerca de la sentencia que la usa. Este mecanismo es justamente lo que significa la palabra clave `this`. La expresión `this` hace referencia al objeto actual. Al escribir `this.de` estamos haciendo referencia al campo del objeto actual, por lo que esta construcción nos ofrece una forma de referirnos a los campos en lugar de a los parámetros cuando tienen el mismo nombre. Ahora podemos leer la sentencia de asignación nuevamente:

```
this.de = de;
```

Como podemos ver, esta sentencia tiene el mismo efecto que la siguiente:

```
campo de nombre "de" = parámetro de nombre "de";
```

En otras palabras, asigna el valor del parámetro `de` al campo del mismo nombre y por supuesto, esto es exactamente lo que necesitamos hacer para inicializar el objeto adecuadamente.

Resta una última pregunta: ¿por qué hacemos todo esto? El problema se podría resolver fácilmente dando nombres diferentes a los campos y a los parámetros. La razón radica en la legibilidad del código.

Algunas veces, hay un nombre que describe perfectamente el uso de una variable y encaja tan bien que no queremos inventar un nombre diferente para ella. Por lo tanto, queremos usar este nombre para el parámetro, lugar donde sirve para indicarle al invocador qué elemento necesita pasarse, y también queremos usarla como campo, donde resulta útil como recordatorio para el implementador de la clase, indicando para qué

se usa este campo. Si un nombre describe perfectamente la finalidad, resulta razonable usarlo como nombre de parámetro y de campo y eliminar los conflictos de nombres usando la palabra clave `this` en la asignación.

### 3.13

## Usar el depurador

La clase más interesante del ejemplo sistema de correo electrónico es la que corresponde al cliente. Ahora investigaremos esta clase con más detalle usando un depurador. El cliente de correo tiene tres métodos: `getMensajeSiguiente`, `imprimirMensajeSiguiente` y `enviarMensaje`. Analizaremos en primer lugar el método `imprimirMensajeSiguiente`.

Antes de comenzar con el depurador, configuremos un escenario que podamos usar para la investigación (Ejercicio 3.33).

**Ejercicio 3.33** Establezca un escenario para la investigación: cree un servidor de correo y luego dos clientes para los usuarios «Sofía» y «Juan» (también podría nombrar las instancias como «sofia» y «juan» para que las pueda distinguir en el banco de objetos). Luego, envíe un mensaje para Juan mediante el método `enviarMensaje` de Sofía. No lea aún el mensaje.

Después de realizar el Ejercicio 3.33 tenemos una situación en la que hay un mensaje para Juan almacenado en el servidor, esperando ser recogido. Hemos visto que el método `imprimirMensajeSiguiente` toma este mensaje y lo muestra en la terminal. Ahora queremos investigar exactamente cómo funciona.

### 3.13.1 Poner puntos de interrupción

Para comenzar nuestra investigación establecemos un punto de interrupción (Ejercicio 3.34). Un punto de interrupción es una bandera que se adjunta a la línea de código en la que se detendrá la ejecución de un método cuando encuentre dicho punto. En BlueJ, este punto de interrupción se representa mediante una pequeña señal de parada (ícono de «stop») (Figura 3.5).

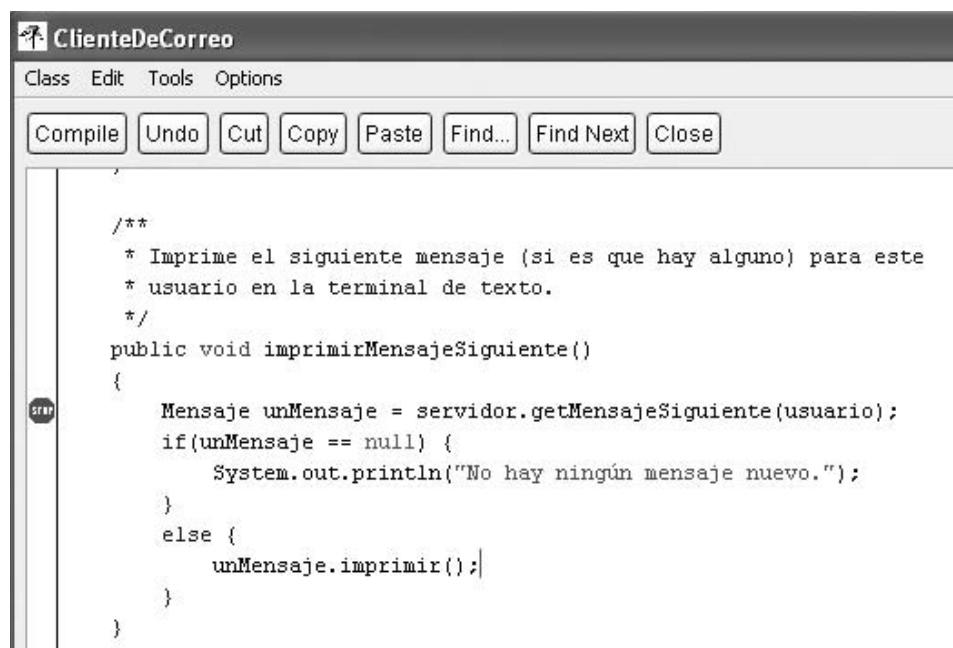
Puede poner un punto de interrupción abriendo el editor de BlueJ, seleccionando la línea apropiada (en nuestro caso, la primera línea del método `imprimirMensajeSiguiente`) y seleccionando *Set Breakpoint* del menú *Tools*. También puede, simplemente, hacer clic en la zona situada a la izquierda de las líneas de código en la que aparece el símbolo de parada para agregar o quitar puntos de interrupción. Observe que la clase debe estar compilada para poder ubicar puntos de interrupción y que al compilar se eliminan los puntos establecidos.

**Ejercicio 3.34** Abra el editor para visualizar el código de la clase `ClienteDeCorreo` y establezca un punto de interrupción en la primera línea del método `imprimirMensajeSiguiente`, tal como muestra la Figura 3.5.

Una vez que colocó un punto de interrupción, invoque el método `imprimirMensajeSiguiente` desde el cliente de correo de Juan. Se abren la ventana del editor de la clase `ClienteDeCorreo` y la ventana del depurador (Figura 3.6).

**Figura 3.5**

Un punto de interrupción en el editor de BlueJ



En la parte inferior de la ventana del depurador hay algunos botones de comandos que se pueden usar para continuar o interrumpir la ejecución del programa. (Para una explicación más detallada sobre los comandos del depurador, vea el Apéndice G.)

**Figura 3.6**

La ventana del depurador, la ejecución se detuvo en un punto de interrupción



Por otra parte, la ventana del depurador tiene tres áreas para mostrar las variables tituladas: *static variables* (variables estáticas), *instance variables* (variables de instancia) y *local variables* (variables locales). Por ahora, ignoraremos la zona de las variables estáticas ya que más adelante hablaremos sobre este tipo de variables y además, esta clase no posee ninguna.

Vemos que este objeto tiene dos variables de instancia (o campos): *servidor* y *usuario* y también podemos ver sus valores actuales. La variable *usuario* almacena la cadena «Sofía» y la variable *servidor* almacena una referencia a otro objeto. La referencia a un objeto la hemos representado anteriormente, mediante una flecha en los diagramas de objetos.

Observe que aún no hay ninguna variable local y se debe a que la ejecución del código se detuvo justo antes de la línea en la que se encuentra el punto de interrupción. Dado que la línea con el punto de interrupción contiene la declaración de la única variable local y que esta línea aún no se ha ejecutado, no existen variables locales en este momento.

El depurador no sólo nos permite interrumpir la ejecución del programa e inspeccionar las variables sino que también podemos recorrer lentamente el código.

### 3.13.2 Paso a paso

Cuando la ejecución se detiene en un punto de interrupción, al hacer clic sobre el botón *Step* se ejecuta una sola línea de código y luego se detiene nuevamente.

**Ejercicio 3.35** Avance una línea en la ejecución del método *imprimirMensajeSigui-*  
*ente* haciendo clic sobre el botón *Step*.

El resultado de ejecutar la primer línea del método *imprimirMensajeSigui-*  
*ente* se muestra en la Figura 3.7. Podemos ver que la ejecución se desplazó una sola línea (aparece una pequeña flecha negra cerca de la línea de código que indica la posición actual), y la lista de variables locales en la ventana del depurador indica que se ha creado una variable local y que se ha asignado un objeto a ella.

**Ejercicio 3.36** Prediga qué línea se marcará como la siguiente a ser ejecutada, después de dar un paso más. Luego ejecute otro paso y verifique su predicción. ¿Fue correcta su respuesta? Explique qué ocurrió y por qué.

Ahora podemos usar el botón *Step* reiteradamente hasta el final del método, lo que nos permite visualizar la ruta que toma la ejecución. Este recorrido paso a paso es especialmente interesante cuando hay sentencias condicionales: podemos ver claramente cómo se ejecuta una de las ramas de la sentencia condicional y visualizar si se satisfacen nuestras expectativas.

**Ejercicio 3.37** Invoque nuevamente al mismo método (*imprimirMensaje-*  
*Siguiente*). Recorra nuevamente el método tal como lo hizo antes, de a un paso por vez. ¿Qué observa? Explique por qué ocurre esto.

### 3.13.3 Entrar en los métodos

Cuando recorrimos el método *imprimirMensajeSigui-*  
*ente* hemos visto dos llamadas a métodos de objetos de nuestras propias clases. La línea:

```
Mensaje unMensaje = servidor.getMensajeSigui-
```

**Figura 3.7**

Detenido nuevamente después de un paso



incluye una llamada al método `getMensajeSiguiente` del objeto `servidor`. Al controlar las declaraciones de variables de instancia podemos ver que el objeto `servidor` fue declarado de clase `ServidorDeCorreo`.

La línea

```
unMensaje.imprimir();
```

invoca al método `imprimir` del objeto `unMensaje`. Podemos ver en la primera línea del método `imprimirMensajeSiguiente` que `unMensaje` fue declarado de clase `Mensaje`.

Utilizando el comando *Step* del depurador hemos usado abstracción: hemos visto al método `imprimir` de la clase `Mensaje` como si fuera una instrucción simple y pudimos observar que su efecto es imprimir los detalles del mensaje (remitente, destinatario y texto del mensaje).

Si estuviéramos interesados en ver más detalles, podemos entrar en el proceso y ver el método `imprimir` en sí mismo ejecutándolo paso a paso. Hacemos esto usando el comando *Step Into* del depurador en lugar del comando *Step*. *Step Into* entra en el código del método invocado y se detiene en la primera línea de código dentro de dicho método.

**Ejercicio 3.38** Configure la misma situación que hemos construido antes, es decir, enviar un mensaje de Sofía para Juan. Luego invoque nuevamente el método `imprimirMensajeSiguiente` del cliente de correo de Juan. Recorra el código como lo hizo antes, pero esta vez, cuando encuentre la línea

```
unMensaje.imprimir()
```

utilice el comando *Step Into* en lugar del comando *Step*. Asegúrese de que puede ver la ventana terminal de texto cada vez que avanza. ¿Qué observa? Explique lo que ve.

## 3.14

## Revisión de llamadas a métodos

En los experimentos de la Sección 3.13 hemos visto otro ejemplo de interacción de objetos similar al que vimos anteriormente: objetos que llaman a métodos de otros objetos. En el método `imprimirMensajeSiguiente`, el objeto `ClienteDeCorreo` hizo una llamada al objeto `ServidorDeCorreo` para tomar el próximo mensaje. Este método (`getMensajeSiguiente`) devolvió un valor: un objeto de tipo `Mensaje`. Luego hubo una llamada al método `imprimir` del mensaje. Usando abstracción, podemos ver al método `imprimir` como un comando único o bien, si estamos interesados en ver más detalles, podemos descender un nivel más de abstracción y mirar dentro del método `imprimir`.

Con un estilo similar, podemos usar el depurador para observar cuando un objeto crea otro objeto. El método `enviarMensaje` de la clase `ClienteDeCorreo` muestra un buen ejemplo. En este método, se crea un objeto `Mensaje` en la primer línea de código:

```
Mensaje elemento = new Mensaje(usuario, para, texto);
```

La idea aquí es que el elemento de correo es usado para encapsular el mensaje de correo electrónico. El elemento contiene información sobre el remitente, el destinatario y el mensaje en sí mismo. Cuando se envía un mensaje, un cliente de correo crea un elemento con toda esta información y luego almacena este elemento en el servidor de correo, desde donde es recogido más tarde por el cliente de correo que indica su dirección.

En la línea de código de arriba vemos que se ha usado la palabra clave `new` para crear un nuevo objeto y también vemos cómo se pasan los parámetros al constructor. (Recuerde que al construir un objeto se hacen dos cosas: se crea el objeto y se ejecuta su constructor.) La llamada al constructor funciona en forma muy similar a las llamadas a métodos y puede observarse usando el comando *Step Into* en la línea en que se construye el objeto.

**Ejercicio 3.39** Ubique un punto de interrupción en la primera línea del método `enviarMensaje` de la clase `ClienteDeCorreo` y luego invoque este método. Use la función *Step Into* para entrar en el código del constructor del mensaje. En la ventana del depurador, se muestran las variables de instancia y las variables locales del objeto `Mensaje` y puede ver que tienen los mismos nombres, tal como lo hablamos en la Sección 3.12.2. Dé algunos pasos más para ver cómo se inicializan las variables de instancia.

**Ejercicio 3.40** Combine la lectura del código, la ejecución de métodos, los puntos de interrupción y el recorrer código paso a paso para familiarizarse con las clases `Mensaje` y `ClienteDeCorreo`. Tenga en cuenta que aún no hemos analizado la implementación de la clase `ServidorDeCorreo` como para que usted la pueda comprender en su totalidad, de modo que por ahora, ignórela. (Por supuesto que puede sentir como una aventura el entrar en el código de esta clase, pero no se sorprenda si encuentra cosas algo «raras».) Explique por

escrito cómo interactúan las clases `ClienteDeCorreo` y `Mensaje`. Incluya en su explicación un diagrama de objetos.

## 3.15

## Resumen

En este capítulo hemos hablado sobre cómo se puede dividir un problema en subproblemas. Podemos tratar de identificar componentes en aquellos objetos que queremos modelar y podemos implementar estos componentes como clases independientes. Hacer esto ayuda a reducir la complejidad de implementación de aplicaciones grandes dado que nos permite implementar, probar y mantener clases individualmente.

Hemos visto que esta modalidad de trabajo da por resultado estructuras de objetos que trabajan juntos para resolver una tarea en común. Los objetos pueden crear otros objetos y se pueden invocar sus métodos unos con otros. Comprender estas interacciones de objetos es esencial al planificar, implementar y depurar aplicaciones.

Podemos usar diagramas en papel y lápiz, leer código y usar depuradores para investigar cómo se ejecuta una aplicación o corregir los errores que aparezcan.

Términos introducidos en este capítulo

**abstracción, modularización, divide y reinarás, diagrama de clases, diagrama de objetos, referencia a un objeto, sobrecarga, llamada a método interno, llamada a método externo, notación de punto, depurador, punto de interrupción**

### Resumen de conceptos

- **abstracción** La abstracción es la habilidad de ignorar los detalles de las partes para enfocar la atención en un nivel más alto de un problema.
- **modularización** La modularización es el proceso de dividir una totalidad en partes bien definidas que podemos construir y examinar separadamente y que interactúan de maneras bien definidas.
- **las clases definen tipos** Puede usarse un nombre de clase para el tipo de una variable. Las variables que tienen una clase como su tipo pueden almacenar objetos de dicha clase.
- **diagrama de clases** Los diagramas de clases muestran las clases de una aplicación y las relaciones entre ellas. Dan información sobre el código. Representan la vista estática de un programa.
- **diagrama de objetos** Los diagramas de objetos muestran los objetos y sus relaciones en un momento dado, durante el tiempo de ejecución de una aplicación. Dan información sobre los objetos en tiempo de ejecución. Representan la vista dinámica de un programa.
- **referencias a objetos** Las variables de tipo objeto almacenan referencias a los objetos.

- **tipo primitivo** Los tipos primitivos en Java no son objetos. Los tipos `int`, `boolean`, `char`, `double` y `long` son los tipos primitivos más comunes. Los tipos primitivos no tienen métodos.
- **creación de objetos** Los objetos pueden crear otros objetos usando el operador `new`.
- **sobrecarga** Una clase puede contener más de un constructor o más de un método con el mismo nombre, siempre y cuando tengan un conjunto de tipos de parámetros que los distinga.
- **llamada a método interno** Los métodos pueden llamar a otros métodos de la misma clase como parte de su implementación. Esto se denomina llamada a método interno.
- **llamada a método externo** Los métodos pueden llamar a métodos de otros objetos usando la notación de punto. Esto se denomina llamada a método externo.
- **depurador** Un depurador es una herramienta de software que ayuda a examinar cómo se ejecuta una aplicación. Puede usarse para encontrar errores.

**Ejercicio 3.41** Use el depurador para investigar el proyecto *visor-de-reloj*. Ponga puntos de interrupción en el constructor de `VisorDeReloj` y en cada uno de los métodos y luego recórralos paso a paso. El comportamiento, ¿es el que esperaba? ¿Le aporta nuevos conocimientos? ¿Cuáles son?

**Ejercicio 3.42** Use el depurador para investigar el método `ingresarDinero` del proyecto *maquina-de-boletos-mejorada* del Capítulo 2. Implemente pruebas que provoquen que se ejecute el código de cada una de las ramas de la sentencia condicional.

**Ejercicio 3.43** Agregue una línea de asunto para los mensajes del proyecto *sistema-de-correo*. Asegúrese de que al imprimir los mensajes, también se imprima el asunto. Modifique el cliente de correo de forma coherente con esta modificación.

**Ejercicio 3.44** Dada la siguiente clase (de la que solamente se muestra un fragmento):

```
public class Pantalla
{
    public Pantalla (int resX, int resY)
    {...}
    public int numeroDePixels()
    {...}
    public void limpiar (boolean invertido)
    {...}
}
```

Escriba algunas líneas en código Java que creen un objeto `Pantalla` y luego invocan a su método `limpiar` si (y sólo si) su número de píxeles es mayor que dos millones. (No se preocupe aquí sobre la lógica, el objetivo es sólo escribir algo que sea sintácticamente correcto, por ejemplo, que pueda compilar si lo tipeamos en el editor.)

## CAPÍTULO

# 4

## Agrupar objetos

Principales conceptos que se abordan en este capítulo

- colecciones              ■ iteradores
- ciclos                    ■ arreglos

Construcciones Java que se abordan en este capítulo

`ArrayList`, `Iterator`, ciclo while, null, objetos anónimos, arreglo, ciclo for, ciclo for-each, ++

El foco principal de este capítulo es introducir algunas maneras en que pueden agruparse los objetos para formar colecciones. En particular, se trata a la clase `ArrayList` como un ejemplo de colecciones de tamaño flexible y al uso de los vectores o arreglos de objetos como colecciones de tamaño fijo. Íntimamente relacionada con las colecciones, aparece la necesidad de recorrer o iterar los elementos que ellas contienen y con este propósito, introducimos tres estructuras de control nuevas: dos versiones del ciclo «for» y el ciclo «while».

### 4.1

## Agrupar objetos en colecciones de tamaño flexible

Cuando escribimos programas, frecuentemente necesitamos agrupar los objetos en colecciones. Por ejemplo:

- Las agendas electrónicas guardan notas sobre citas, reuniones, fechas de cumpleaños, etc.
- Las bibliotecas registran detalles de los libros y revistas que poseen.
- Las universidades mantienen registros de la historia académica de los estudiantes.

Una característica típica de estas situaciones es que el número de elementos almacenados en la colección varía a lo largo del tiempo. Por ejemplo, en una agenda electrónica se agregan nuevas notas para registrar eventos futuros y se borran aquellas notas de eventos pasados en la medida en que ya no son más necesarios; en una biblioteca,

el inventario cambia cuando se compran libros nuevos y cuando algunos libros viejos se archivan o se descartan.

Hasta ahora, no hemos hallado en Java ninguna característica que nos permita agrupar un número arbitrario de elementos. Podríamos definir una clase con una gran cantidad de campos individuales, suficiente como para almacenar un número muy grande pero fijo de elementos. Sin embargo, generalmente los programas necesitan una solución más general que la citada. Una solución adecuada sería aquella que no requiera que conozcamos anticipadamente la cantidad de elementos que queremos agrupar o bien, establecer un límite mayor que dicho número.

En las próximas secciones, usaremos el ejemplo de una agenda personal para ilustrar una de las maneras en que Java nos permite agrupar un número arbitrario de objetos en un único objeto contenedor.

## 4.2

### Una agenda personal

Planeamos modelar una aplicación que represente una agenda personal con las siguientes características básicas:

- Permite almacenar notas.
- El número de notas que se pueden almacenar no tiene límite.
- Mostrará las notas de manera individual.
- Nos informará sobre la cantidad de notas que tiene actualmente almacenadas.

Encontraremos que podemos implementar todas estas características muy fácilmente si tenemos una clase que sea capaz de almacenar un número arbitrario de objetos (las notas). Una clase como ésta ya está preparada y disponible en una de las *bibliotecas* que forman parte del entorno estándar de Java.

Antes de analizar el código necesario para hacer uso de esta clase, es útil explorar el comportamiento del ejemplo agenda.

**Ejercicio 4.1** Abra el proyecto *agenda1* en BlueJ y cree un objeto *Agenda*. Almacene algunas notas (que son simplemente cadenas) y luego verifique que el número que devuelve *numeroDeNotas* coincida con el número de notas que guardó. Cuando use el método *mostrarNota* necesitará un parámetro con valor 0 (cero) para imprimir la primera nota, de valor 1 para imprimir la segunda nota y así sucesivamente. Explicaremos el motivo de esta numeración oportunamente.

## 4.3

### Una primera visita a las bibliotecas de clases

#### Concepto

Las **colecciones de objetos** son objetos que pueden almacenar un número arbitrario de otros objetos.

Una de las características de los lenguajes orientados a objetos que los hace muy potentes es que frecuentemente están acompañados de *bibliotecas de clases*. Estas bibliotecas contienen, comúnmente, varios cientos o miles de clases diferentes que han demostrado ser de gran ayuda para los desarrolladores en un amplio rango de proyectos diferentes. Java cuenta con varias de estas bibliotecas y seleccionaremos clases de varias de ellas a lo largo del libro. Java denomina a sus bibliotecas como *paquetes (packages)*; trabajaremos con los paquetes más detalladamente en los próximos capítulos. Podemos usar las clases

de las bibliotecas exactamente de la misma manera en que usamos nuestras propias clases: las instancias se construyen usando la palabra `new` y las clases tienen campos, constructores y métodos. En la clase `Agenda` haremos uso de la clase `ArrayList` que está definida en el paquete `java.util`; mostraremos cómo hacerlo en la sección siguiente. `ArrayList` es un ejemplo de una clase *colección*. Las colecciones pueden almacenar un número arbitrario de elementos en el que cada elemento es otro objeto.

### 4.3.1 Ejemplo de uso de una biblioteca

El Código 4.1 muestra la definición completa de la clase `Agenda` que usa la clase de biblioteca `ArrayList`.

#### Código 4.1

La clase Agenda

```
import java.util.ArrayList;
/*
 * Una clase para mantener una lista arbitrariamente larga
 * de notas.
 * Las notas se numeran para referencia externa de un
 * usuario
 * humano.
 * En esta versión, la numeración de las notas comienzan
 * en 0.
 * @author David J. Barnes and Michael Kölling.
 * @version 2006.03.30
 */
public class Agenda
{
    // Espacio para almacenar un número arbitrario de
    // notas.
    private ArrayList<String> notas;
    /**
     * Realiza cualquier inicialización que se requiera
     * para la
     * agenda.
     */
    public Agenda()
    {
        notas = new ArrayList<String>();
    }
    /**
     * Almacena una nota nueva en la agenda.
     * @param nota La nota que se almacenará.
     */
    public void guardarNota(String nota)
    {
        notas.add(nota);
    }
    /**
     * @return El número de notas que tiene actualmente
     * la agenda.
    }
```

**Código 4.1  
(continuación)**

La clase Agenda

```

        */
    public int numeroDeNotas()
    {
        return notas.size();
    }
    /**
     * Muestra una nota.
     * @param numeroDeNota El número de la nota que se
mostrará.
    */
    public void mostrarNota(int numeroDeNota)
    {
        if(numeroDeNota < 0) {
            // No es un número de nota válido, por lo
tanto no se hace nada.
        }
        else if(numeroDeNota < numeroDeNotas()) {
            // Es un número válido de nota, por lo
tanto se la puede mostrar.
            System.out.println(notas.get(numeroDeNota));
        }
        else {
            // No es un número válido de nota, por
lo tanto no se hace nada.
        }
    }
}

```

La primera línea de esta clase muestra el modo en que obtenemos el acceso a una clase de una biblioteca de Java mediante la *sentencia import*:

```
import java.util.ArrayList;
```

Esta sentencia hace que la clase `ArrayList` del paquete `java.util` esté disponible para nuestra clase. Las sentencias `import` deben ubicarse en el texto de la clase, siempre antes del comienzo de la declaración de la clase. Una vez que el nombre de una clase ha sido importado desde un paquete de esta manera, podemos usar dicha clase tal como si fuera una de nuestras propias clases, de modo que usamos `ArrayList` al principio de la definición de la clase `Agenda` para declarar el campo `notas`:

```
private ArrayList<String> notas;
```

Aquí vemos una nueva construcción: la mención de `String` entre símbolos de menor (`<`) y de mayor (`>`): `<String>`.

Cuando usamos colecciones, debemos especificar dos tipos: el tipo propio de la colección (en este caso, `ArrayList`) y el tipo de los elementos que planeamos almacenar en la colección (en este caso, `String`). Podemos leer la definición completa del tipo como «*ArrayList de String*». Usamos esta definición de tipo como el tipo de nuestra variable `notas`.

En el constructor de la agenda, creamos un objeto de tipo `ArrayList<String>` y guardamos dentro de él nuestro campo `notas`. Observe que necesitamos especificar nuevamente el tipo completo con el tipo de elemento entre los símbolos de menor y de mayor, seguido de los paréntesis para la lista de parámetros (vacía):

```
notas = new ArrayList<String>();
```

Las clases similares a `ArrayList` que se parametrizan con un segundo tipo se denominan *clases genéricas*; hablaremos sobre ellas con más detalles muy rápidamente.

La clase `ArrayList` declara muchos métodos pero en este momento, sólo usaremos tres de ellos para implementar la funcionalidad que requerimos: `add`, `size` y `get`.

Los dos primeros se ilustran en los métodos relativamente claros `guardarNota` y `numeroDeNotas` respectivamente. El método `add` de un `ArrayList` almacena un objeto en la lista y el método `size` devuelve la cantidad de elementos que están almacenados realmente en ella.

## 4.4

## Estructuras de objetos con colecciones

Para comprender cómo opera una colección de objetos tal como `ArrayList` resulta útil examinar un diagrama de objetos. La Figura 4.1 ilustra cómo se presentaría un objeto `Agenda` que contiene dos notas. Compare la Figura 4.1 con la Figura 4.2 en la que se almacenó una tercera nota.

Existen por lo menos tres características importantes de la clase `ArrayList` que debería observar:

- Es capaz de aumentar su capacidad interna tanto como se requiera: cuando se agregan más elementos, simplemente hace suficiente espacio para ellos.
- Mantiene su propia cuenta privada de la cantidad de elementos que tiene actualmente almacenados. Su método `size` devuelve el número de objetos que contiene actualmente.
- Mantiene el orden de los elementos que se agregan, por lo que más tarde se pueden recuperar en el mismo orden.

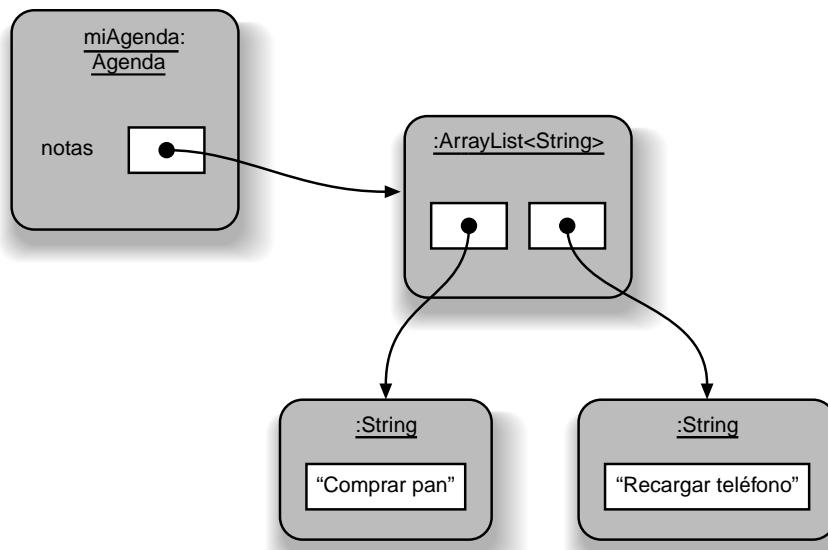
Vemos que el objeto `Agenda` tiene un aspecto muy simple: tiene sólo un campo que almacena un objeto de tipo `ArrayList<String>`. Parece que todo el trabajo difícil lo hace el objeto `ArrayList`, y esta es una de las grandes ventajas de usar clases de bibliotecas: alguien invirtió tiempo y esfuerzo para implementar algo útil y nosotros tenemos acceso prácticamente libre a esta funcionalidad usando esa clase.

En esta etapa, no necesitamos preocuparnos por cómo fue implementada la clase `ArrayList` para que tenga estas características; es suficiente con apreciar lo útil que resulta su capacidad. Esto significa que podemos utilizarla para escribir cualquier cantidad de clases diferentes que requieran almacenar un número arbitrario de objetos.

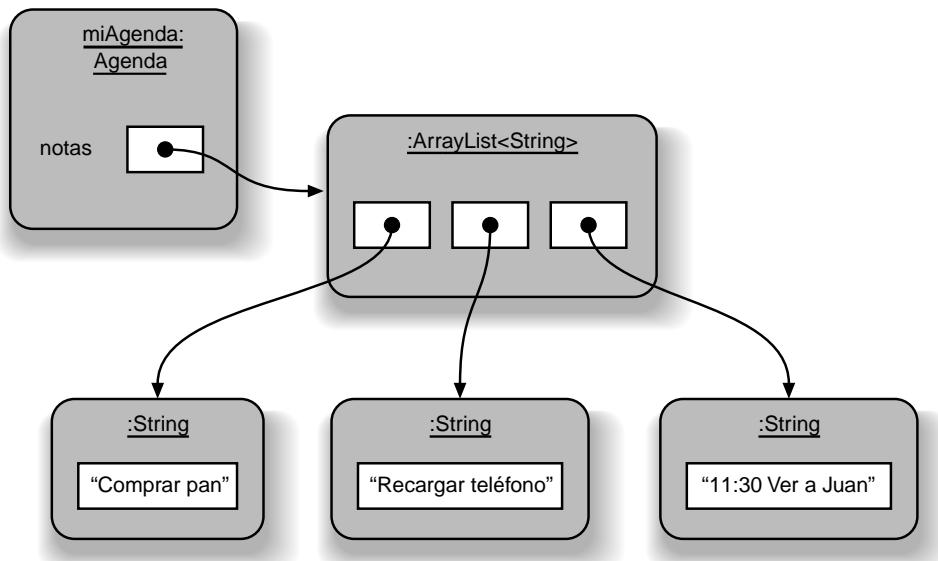
En la segunda característica, el objeto `ArrayList` mantiene su propia cuenta de la cantidad de objetos insertados, tiene consecuencias importantes en el modo en que implementamos la clase `Agenda`. A pesar de que la agenda tiene un método `numeroDeNotas`, no hemos definido realmente un campo específico para guardar esta infor-

**Figura 4.1**

Una Agenda que contiene dos notas

**Figura 4.2**

Una Agenda que contiene tres notas



mación. En su lugar, la agenda delega la responsabilidad de mantener el número de elementos a su objeto `ArrayList`, quiere decir que la agenda no duplica información que esté disponible desde cualquier otro objeto. Si un usuario solicita información a la agenda sobre el número de notas que tiene guardadas, la agenda pasará la pregunta al objeto `notas` y luego devolverá cualquier respuesta que obtenga de él.

La duplicación de información o del comportamiento es algo sobre lo que tendremos que trabajar muy duro para evitarla. La duplicación puede representar esfuerzos desperdiados y puede generar inconsistencias cuando dos objetos que debieran brindar idéntica respuesta no lo hacen.

## 4.5

## Clases genéricas

La nueva notación que utiliza los símbolos de menor y de mayor que hemos visto con anterioridad merece un poco más de discusión. El tipo de nuestro campo `notas` fue declarado como:

```
ArrayList<String>
```

La clase que estamos usando aquí se denomina justamente `ArrayList`, pero requiere que se especifique un segundo tipo como parámetro cuando se usa para declarar campos u otras variables. Las clases que requieren este tipo de parámetro se denominan *clases genéricas*. Las clases genéricas, en contraste con las otras clases que hemos visto hasta ahora, no definen un tipo único en Java sino potencialmente muchos tipos. Por ejemplo, la clase `ArrayList` puede usarse para especificar un *ArrayList de Strings*, un *ArrayList de Personas*, un *ArrayList de Rectángulos*, o un `ArrayList` de cualquier otra clase que tengamos disponible. Cada `ArrayList` en particular es un tipo distinto que puede usarse en declaraciones de campos, parámetros y tipos de retorno. Podríamos, por ejemplo, definir los siguientes dos campos:

```
private ArrayList<Persona> miembros;  
private ArrayList<MaquinaDeBoletos> misMaquinas;
```

Estas declaraciones establecen que `miembros` contiene un `ArrayList` que puede almacenar objetos `Persona`, mientras que `misMaquinas` puede contener un `ArrayList` que almacena objetos `MaquinaDeBoletos`. Tenga en cuenta que `ArrayList<Persona>` y `ArrayList<MaquinaDeBoletos>` son tipos diferentes. Los campos no pueden ser asignados uno a otro, aun cuando sus tipos deriven de la misma clase.

**Ejercicio 4.2** Escriba la declaración de un campo privado de nombre `biblioteca` que pueda contener un `ArrayList`. Los elementos del `ArrayList` son de tipo `Libro`.

## 4.6

## Numeración dentro de las colecciones

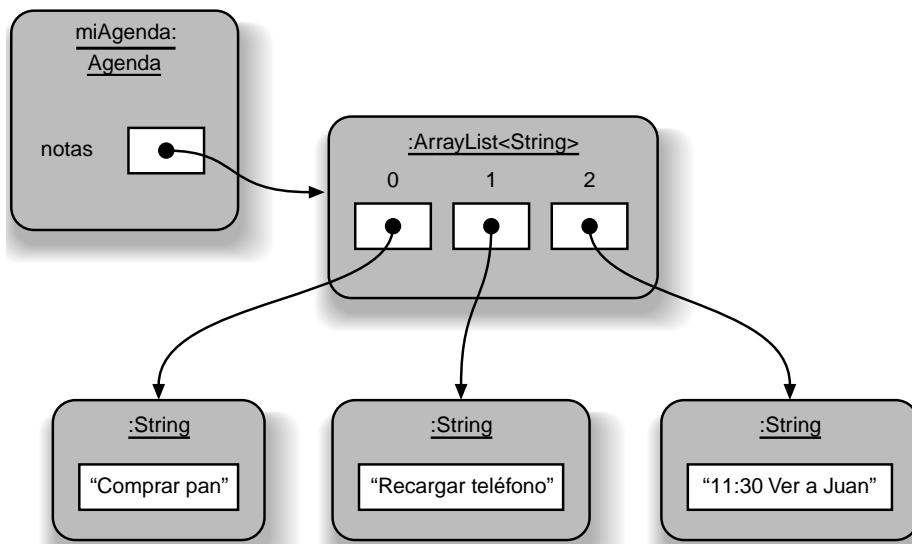
Mientras explorábamos el proyecto `agenda1` en el Ejercicio 4.1 observamos que para imprimir las notas era necesario usar valores numéricos a partir de cero para el parámetro. La razón que subyace detrás de este requerimiento es que los elementos almacenados en las colecciones tienen una numeración implícita o posicionamiento que comienza a partir de cero. La posición que ocupa un objeto en una colección es conocida más comúnmente como su *índice*. El primer elemento que se agrega a una colección tiene por índice al número 0, el segundo tiene al número 1, y así sucesivamente. La Figura 4.3 ilustra la misma situación que antes, pero se muestran los números índice del objeto `ArrayList`.

El método `mostrarNota` en el Código 4.1 ilustra la manera en que se usa un índice para obtener un elemento desde el `ArrayList` mediante su método `get`. La mayor parte del código del método `mostrarNota` es la concerniente a controlar que el valor del parámetro esté en el rango de valores válidos `[0.. (size-1)]` antes de llamar al método `get`.

Es importante tener en cuenta que `get` no elimina un elemento de la colección.

**Figura 4.3**

Índices de los elementos de una colección



**Cuidado:** si usted no es cuidadoso, podría intentar acceder a un elemento de una colección que está fuera de los índices válidos del `ArrayList`. Cuando lo haga, obtendrá un mensaje del error denominado *desbordamiento*. En Java, verá un mensaje que dice `IndexOutOfBoundsException`.

**Ejercicio 4.3** Si una colección almacena 10 objetos, ¿qué valor devolverá una llamada a su método `size`?

**Ejercicio 4.4** Escriba una llamada al método `get` para devolver el quinto objeto almacenado en una colección de nombre `elementos`.

**Ejercicio 4.5** ¿Cuál es el índice del último elemento almacenado en una colección de 15 objetos?

**Ejercicio 4.6** Escriba una llamada para agregar el objeto contenido en la variable `cita` a una colección de nombre `notas`.

## 4.7

### Eliminar un elemento de una colección

Sería muy útil tener la capacidad de eliminar las notas viejas de la Agenda cuando ya no nos interesen más. En principio, hacer esto es fácil porque la clase `ArrayList` tiene un método `remove` que toma como parámetro el índice de la nota que será eliminada. Cuando un usuario quiera eliminar una nota de la agenda, podemos lograrlo con sólo invocar al método `remove` del objeto `notas`. El Código 4.2 ilustra el método `remove` que podríamos agregar a la clase `Agenda`.

**Código 4.2**

Eliminar una nota de la agenda

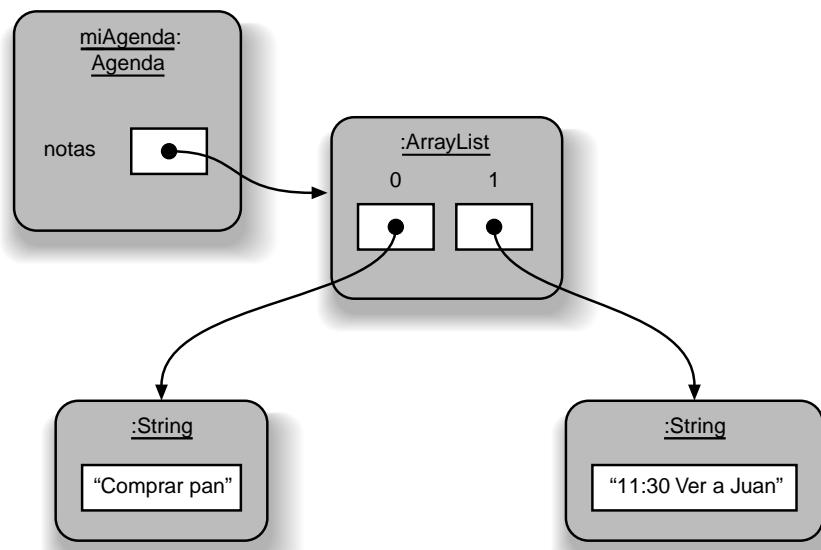
```
public void eliminarNota(int numeroDeNota)
{
    if(numeroDeNota < 0) {
        // No es un número de nota válido, no
        se hace nada.
    }
    else if(numeroDeNota < numeroDeNotas()) {
        // Número de nota válido, se la puede
        borrar.
        notas.remove(numeroDeNota);
    }
    else {
        // No es un número válido de nota,
        entonces no se hace nada.
    }
}
```

Una complicación del proceso de eliminación es que se modifican los valores de los índices de las restantes notas que están almacenadas en la colección. Si se elimina una nota que tiene por índice un número muy bajo, la colección desplaza todos los siguientes elementos una posición a la izquierda para llenar el hueco; en consecuencia, sus índices disminuyen en 1.

La Figura 4.4 muestra la forma en que se modifican algunos índices de los elementos de un `ArrayList` debido a la eliminación de un elemento en medio de ella. Comenzando con la situación ilustrada en la Figura 4.3, la nota número 1 (“Recargar teléfono”) ha sido eliminada, y como resultado, el índice de la nota que originalmente tenía el número de índice 2 (“11:30 Ver a Juan”) ha cambiado al valor 1 mientras que la nota que tiene número índice 0 permanece sin cambios.

**Figura 4.4**

Los índices se modifican después de la eliminación de un elemento



Más adelante veremos que también es posible insertar elementos en un `ArrayList` en otros lugares distintos que el final de la colección. Esto significa que los elementos que ya están en la lista deben incrementar sus índices cuando se agrega un nuevo elemento. Los usuarios deben ser conscientes de estos cambios en los índices cuando agregan o eliminan notas.

**Ejercicio 4.7** Escriba una llamada a método para eliminar el tercer objeto almacenado en una colección de nombre `notas`.

**Ejercicio 4.8** Suponga que un objeto está almacenado en una colección bajo el índice 6. ¿Cuál será su índice inmediatamente después de que se eliminen los objetos de las posiciones 0 y 9?

**Ejercicio 4.9** Implemente un método `eliminarNota` en su agenda.

## 4.8

## Procesar una colección completa

Si agregar y eliminar notas significa que los índices pueden cambiar con el tiempo, sería de gran ayuda tener un método en la clase `Agenda` que pueda listar todas las notas con sus índices actuales. Podemos establecer de otra manera lo que podría hacer el método diciendo que queremos obtener cada número de índice válido y mostrar la nota que está almacenada en ese número. Antes de seguir leyendo, intente realizar el siguiente ejercicio para ver si se puede escribir fácilmente un método como el descrito con los conocimientos de Java que tenemos.

**Ejercicio 4.10** ¿Cómo debiera ser el encabezado del método `listarTodasLasNotas`? ¿Cuál debe ser su tipo de retorno? ¿Debe tener algún parámetro?

**Ejercicio 4.11** Sabemos que la primera nota está almacenada en la posición 0 del `ArrayList`. ¿Podríamos escribir el cuerpo de `listarTodasLasNotas` mediante las siguientes líneas?

```
System.out.println(notas.get(0));  
System.out.println(notas.get(1));  
System.out.println(notas.get(2));
```

etc.

¿Cuántas sentencias `println` requeriría la versión completa del método `listarTodasLasNotas` descrito en el Ejercicio 4.11?

Probablemente ya se habrá dado cuenta de que realmente no es posible responder esta pregunta porque depende de cuántas notas haya en la agenda en el momento en que sean listadas. Si hay tres notas se requieren tres sentencias `println`, si hay cuatro notas entonces necesitaríamos cuatro sentencias, y así sucesivamente. Los métodos `mostrarNota` y `eliminarNota` ilustran que el rango de números de índice válidos en cualquier momento es `[0...(size)-1]`, por lo que el método `listarTodasLasNotas` también debería tener este tamaño dinámico en algún contador en vías de realizar su trabajo.

Aquí tenemos la necesidad de hacer algo numerosas veces, pero el número de veces depende de circunstancias que pueden variar. Encontraremos esta clase de problemas en muchos programas de diferente naturaleza y la mayoría de los lenguajes de programación tienen varias maneras de resolver tales problemas. La solución que elegimos usar en esta etapa es introducir una de las *sentencias de ciclo* de Java: el *ciclo for-each*.

### 4.8.1 El *ciclo for-each*

#### Concepto

Un **ciclo** puede usarse para ejecutar repetidamente un bloque de sentencias sin tener que escribirlas varias veces.

Un *ciclo for-each* es una forma de llevar a cabo repetidamente un conjunto de acciones, sin tener que escribir esas acciones más de una vez. Podemos resumir las acciones de un *ciclo for-each* en el siguiente pseudocódigo:

```
for (TipoDelElemento elemento : colección) {
    cuerpo del ciclo
}
```

La nueva pieza principal de Java es la palabra **for**. El lenguaje Java tiene dos variantes del ciclo **for**: uno es el *ciclo for-each* del que estamos hablando ahora, el otro se denomina simplemente *ciclo for* y lo discutiremos un poco más adelante en este capítulo.

Un *ciclo for-each* consta de dos partes: un encabezado de ciclo (la primer línea del ciclo) y un cuerpo a continuación del encabezado. El cuerpo contiene aquellas sentencias que deseamos llevar a cabo una y otra vez.

El *ciclo for-each* toma su nombre a partir de la manera en que podemos leerlo: si leemos la palabra clave **For** como «para cada» y los dos puntos en la cabecera del ciclo como las palabras «en la», entonces la estructura del código que mostramos anteriormente comenzaría a tener más sentido, tal como aparece en este pseudocódigo:

```
Para cada elemento en la colección hacer: {
    cuerpo del ciclo
}
```

Cuando compare esta versión con el pseudocódigo original de la primera versión, observará que **elemento** se escribió de manera similar a una declaración de variable: **TipoDelElemento elemento**. Esta sección realmente declara una variable que luego se usa a su vez, para cada elemento de la colección. Antes de avanzar en la discusión, veamos un ejemplo de código Java.

#### Código 4.3

Uso de un ciclo para imprimir las notas

```
/**
 * Imprime todas las notas de la agenda
 */
public void imprimirNotas()
{
    for(String nota : notas) {
        System.out.println(nota);
    }
}
```

En este *ciclo for-each*, el cuerpo del ciclo (que consiste en una sola sentencia `System.out.println`) se ejecuta repetidamente, una vez para cada elemento del `ArrayList` `notas`. Por ejemplo: si en la lista de notas hubiera cuatro cadenas, la sentencia de impresión se ejecutaría cuatro veces.

En cada vuelta, antes de que la sentencia se ejecute, la variable `notas` se configura para contener uno de los elementos de la lista: primero el del índice 0, luego el del índice 1, y así sucesivamente. Por lo tanto, cada elemento de la lista logra ser impreso.

Permítanos disecar el ciclo un poco más detalladamente. La palabra clave `for` introduce el ciclo. Está seguida por un par de paréntesis en los que se definen los detalles del ciclo. El primero de estos detalles es la declaración `String nota`, que define una nueva variable local `nota` que se usará para contener los elementos de la lista. Llamamos a esta variable *variable de ciclo*. Podemos elegir el nombre de esta variable de la misma manera que el de cualquier otra variable, no tiene porqué llamarse «`nota`». El tipo de la variable de ciclo debe ser el mismo que el tipo del elemento declarado para la colección que estamos usando, en nuestro caso `String`.

A continuación aparecen dos puntos y la variable que contiene la colección que deseamos procesar. Cada elemento de esta colección será asignado en su turno a la variable de ciclo, y para cada una de estas asignaciones el cuerpo del ciclo se ejecutará una sola vez. Luego, podemos usar en el cuerpo del ciclo la variable de ciclo para hacer referencia a cada elemento.

Para poner a prueba su comprensión sobre cómo operan los ciclos, intente resolver los siguientes ejercicios.

**Ejercicio 4.12** Implemente el método `imprimirNotas` en su versión del proyecto `agenda`. (En el proyecto `agenda2` se ofrece una solución con este método implementado, pero para mejorar su comprensión del tema, le recomendamos que escriba el método por su propia cuenta.)

**Ejercicio 4.13** Cree una Agenda y almacene algunas notas en ella. Utilice el método `imprimirNotas` para mostrarlas por pantalla y verificar que el método funciona como debiera.

**Ejercicio 4.14** Si lo desea, podría utilizar el depurador para ayudarlo a comprender cómo se repiten las sentencias del cuerpo del ciclo. Fije un punto de interrupción justo antes del ciclo y ejecute el método paso a paso hasta que el ciclo haya procesado todos los elementos y finalice.

**Ejercicio 4.15** Modifique los métodos `mostrarNota` y `eliminarNota` para que impriman un mensaje de error si el número ingresado no fuera válido.

Ahora, ya hemos visto cómo podemos usar el *ciclo for-each* para llevar a cabo algunas operaciones (el cuerpo del ciclo) sobre cada elemento de una colección. Este es un gran paso hacia adelante, pero no resuelve todos nuestros problemas. Algunas veces necesitamos un poco más de control y Java ofrece una construcción de ciclo diferente que nos permite hacerlo: el ciclo `while`.

## 4.8.2 El *ciclo while*

Un *ciclo while* es similar en su estructura y propósito que el *ciclo for-each*: consiste en un encabezado de ciclo y un cuerpo, y el cuerpo puede ejecutarse repeti-

damente. Sin embargo, los detalles son diferentes. Aquí está la estructura de un *ciclo while*:

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

Observamos que el *ciclo while* comienza con la palabra clave *while*, seguida de una condición. Este ciclo es más flexible que el *ciclo for-each*. En lugar de recorrer todos los elementos de una colección, puede recorrer un número variable de elementos de la colección, dependiendo de la condición del ciclo.

La condición es una expresión lógica que se usa para determinar si el cuerpo debe ejecutarse por lo menos una vez. Si la condición se evalúa verdadera, se ejecuta el cuerpo del ciclo. Cada vez que se ejecuta el cuerpo del ciclo, la condición se vuelve a controlar nuevamente. Este proceso continúa repetidamente hasta que la condición resulta falsa, que es el punto en el que se salta del cuerpo del ciclo y la ejecución continúa con la sentencia que esté ubicada inmediatamente después del cuerpo.

Podemos escribir un *ciclo while* que imprima todas las notas de nuestra lista, tal como lo hemos hecho anteriormente mediante un *ciclo for-each*. La versión que usa un *ciclo while* se muestra en Código 4.4.

#### Código 4.4

Uso de un *ciclo while* para mostrar todas las notas

```
int indice = 0;
while(indice < notas.size()) {
    System.out.println(notas.get(indice));
    indice++;
}
```

Este *ciclo while* es equivalente al *ciclo for-each* que hemos discutido en la sección anterior. Son relevantes algunas observaciones:

- En este ejemplo, el *ciclo while* resulta un poco más complicado. Tenemos que declarar fuera del ciclo una variable para el índice e iniciarla por nuestros propios medios en 0 para acceder al primer elemento de la lista.
- Los elementos de la lista no son extraídos automáticamente de la colección y asignados a una variable. En cambio, tenemos que hacer esto nosotros mismos usando el método *get* del *ArrayList*. También tenemos que llevar nuestra propia cuenta (*índice*) para recordar la posición en que estábamos.
- Debemos recordar incrementar la variable contadora (*índice*) por nuestros propios medios.

La última sentencia del cuerpo del *ciclo while* ilustra un operador especial para incrementar una variable numérica en 1:

```
indice ++;
```

esto es equivalente a:

```
index = index + 1;
```

Hasta ahora, el *ciclo for-each* es claramente bueno para nuestro objetivo, fue menos complicado de escribir y es más seguro porque garantiza que siempre llegará a un final.

En nuestra versión del *ciclo while* es posible cometer errores que den por resultado un *ciclo infinito*. Si nos olvidamos de incrementar la variable índice (la última línea del cuerpo del ciclo) la condición del ciclo nunca podría ser evaluada como falsa y el ciclo se repetiría indefinidamente. Este es un error típico de programación y hace que el programa continúe ejecutándose eternamente. En tal situación, si el ciclo no contiene una sentencia de corte, el programa aparecerá como «colgado»: parece que no está haciendo nada y no responde a ningún clic del ratón o a pulsar una tecla. En realidad, el programa está haciendo mucho: ejecuta el ciclo una y otra vez, pero no podemos ver ningún efecto de esto y parece que el programa hubiera muerto.

Por lo tanto, ¿cuáles son los beneficios de usar un *ciclo while* en lugar de un *ciclo for-each*?

Existen dos fundamentos: primeramente, el *ciclo while* no necesita estar relacionado con una colección (podemos reciclar cualquier condición que necesitemos); en segundo lugar, aun si usáramos el ciclo para procesar la colección, no necesitamos procesar cada uno de sus elementos, en cambio, podríamos querer frenar el recorrido tempranamente.

Veremos primero un ejemplo simple de un *ciclo while* que no está relacionado con una colección. El siguiente ciclo imprime en la pantalla todos los números pares hasta 30:

```
int numero = 0;
while (numero <= 30) {
    System.out.println(numero);
    numero = numero + 2;
}
```

Para poner a prueba su comprensión sobre los *ciclos while* intente realizar los siguientes ejercicios.

**Ejercicio 4.16** Escriba un *ciclo while* (por ejemplo, en un método de nombre `prueba`) que muestre en la pantalla todos los múltiplos de 5 comprendidos entre 10 y 95.

**Ejercicio 4.17** Escriba un método de nombre `sumar` con un *ciclo while* que sume todos los números comprendidos entre dos números **a** y **b**. Los valores de **a** y **b** pueden ser pasados al método `sumar` como parámetros.

**Ejercicio 4.18 Desafío.** Escriba un método `esPrimo(int n)` que devuelva el valor verdadero si el parámetro **n** es un número primo, y falso en caso contrario. Para implementar el método puede escribir un *ciclo while* que divida **n** por todos los números comprendidos entre 2 y (**n**-1) y controlar si el resultado de la división es un número entero. Puede escribir esta verificación usando el operador módulo (%) para controlar que el resto de la división entera sea 0 (véase la discusión sobre el operador módulo en la Sección 3.8.3).

Ahora podemos usar el *ciclo while* para escribir un ciclo que busque en nuestra colección un elemento específico y se detenga cuando lo encuentre. Para ser precisos, queremos un método de nombre `buscar` que tenga un parámetro `String` de nombre `cadABuscar` y luego imprima en pantalla la primer nota de la agenda que contenga la cadena de búsqueda. Se puede llevar a cabo esta tarea con la siguiente combinación del *ciclo while* con una sentencia condicional:

```
int indice = 0;
boolean encontrado = false;
```

```

while (indice < notas.size() && !encontrado) {
    String nota = notas.get(indice);
    if (nota.contains(cadABuscar)) {
        encontrado = true;
    }
    else {
        indice++;
    }
}

```

Estudie este fragmento de código hasta que logre comprenderlo (es importante). Verá que la condición está escrita de tal manera que el ciclo se detiene bajo cualquiera de estas dos condiciones: si efectivamente se encuentra la cadena buscada, o cuando hemos controlado todos los elementos y no se encontró la cadena buscada.

Este código necesita completarse para agregar la salida del método. Lo hacemos en el siguiente ejercicio.

**Ejercicio 4.19** Implemente el método `buscar` en la clase `Agenda` tal como se describió anteriormente. El código que se muestra en el ejemplo anterior es parte de este método, pero no está completo. Necesita agregar código a continuación del ciclo para mostrar si se encontró la nota o bien la cadena «No se encontró el elemento buscado». Asegúrese de controlar su método dos veces como mínimo, buscando una cadena que sabe que está en la lista y una que sabe que no está.

**Ejercicio 4.20** Modifique el método `imprimirNotas` de modo que muestre al comienzo de cada nota un número que corresponda a su índice en el `ArrayList`. Por ejemplo:

```

0: Comprar pan.
1: Recargar teléfono.
2: 11:30: Ver a Juan.

```

Este listado hace que sea mucho más fácil ingresar el índice correcto en el momento de eliminar una nota de la agenda.

**Ejercicio 4.21** En una ejecución del método `buscar`, se le pregunta repetidamente a la colección `notas` cuántas notas contiene actualmente. Se lleva a cabo cada vez que se evalúa la condición del ciclo. ¿Varía el valor que retorna `size` en cada verificación? Si considera que la respuesta es no, escriba el método `buscar` de modo que el tamaño de la colección `notas` se determine una única vez y se almacene en una variable local, antes de la ejecución del ciclo. Luego utilice la variable local en la condición del ciclo en lugar de una invocación a `size`. Pruebe que esta versión produce el mismo resultado que la versión anterior. Si tiene problemas al completar este ejercicio, intente usar el depurador para detectar cuáles son los errores.

**Ejercicio 4.22** Modifique su agenda de modo que las notas se numeren a partir de 1 y no de 0. Recuerde que el objeto `ArrayList` continuará usando el índice a partir de cero, pero usted puede presentar las notas numeradas a partir de 1 en su listado. Asegúrese de modificar adecuadamente los métodos `mostrarNota` y `eliminarNota`.

### 4.8.3 Recorrer una colección

Antes de avanzar, discutiremos una tercer variante para recorrer una colección, que está entre medio de los ciclos *while* y *for-each*. Usa un ciclo while para llevar a cabo el recorrido y un *objeto iterador* en lugar de una variable entera como índice del ciclo para mantener el rastro de la posición en la lista.

#### Concepto

Un **iterador** es un objeto que proporciona funcionalidad para recorrer todos los elementos de una colección.

Examinar cada elemento de una colección es una actividad tan común que un *ArrayList* proporciona una forma especial de *recorrer o iterar* su contenido. El método *iterator* de *ArrayList* devuelve un objeto *Iterator*<sup>1</sup>. La clase *Iterator* también está definida en el paquete *java.util* de modo que debemos agregar una segunda sentencia import a la clase *Agenda* para poder usarla.

```
import java.util.ArrayList;
import java.util.Iterator;
```

Un *Iterator* provee dos métodos para recorrer una colección: *hasNext* y *next*. A continuación describimos en pseudocódigo la manera en que usamos generalmente un *Iterator*:

```
Iterator<TipoDelElemento> it = miColeccion.iterator();
while (it.hasNext()) {
    Invocar it.next() para obtener el siguiente elemento
    Hacer algo con dicho elemento
}
```

En este fragmento de código usamos primero el método *iterator* de la clase *ArrayList* para obtener un objeto iterador. Observe que *Iterator* también es de tipo genérico y por lo tanto, lo parametrizamos con el tipo de los elementos de la colección. Luego usamos dicho iterador para controlar repetidamente si hay más elementos (*it.hasNext()*) y para obtener el siguiente elemento (*it.next()*). Un punto a destacar es que le pedimos al iterador que devuelva el siguiente elemento y no la colección.

Podemos escribir un método que usa un iterador para listar por pantalla todas las notas, tal como se muestra en el Código 4.5. En efecto, el iterador comienza en el inicio de la colección y trabaja progresivamente, de a un objeto por vez, cada vez que se invoca su método *next*.

#### Código 4.5

Uso de un Iterador para recorrer la lista de notas

```
/**
 * Listar todas las notas de la agenda.
 */
public void listarTodasLasNotas()
{
    Iterator<String> it = notas.iterator();
    while (it.hasNext()) {
        System.out.println(it.next());
    }
}
```

<sup>1</sup> Preste especial atención en distinguir las diferentes capitalizaciones de las letras del método *iterator* y de la clase *Iterator*.

Tómese algún tiempo para comparar esta versión con las dos versiones del método `listarTodasLasNotas` que se muestran en el Código 4.3 y en el Código 4.4. Un punto para resaltar de la última versión es que usamos explícitamente un *ciclo while*, pero no necesitamos tomar precauciones respecto de la variable `indice`. Es así porque el `Iterator` mantiene el rastro de lo que atravesó de la colección por lo que sabe si quedan más elementos en la lista (`hasNext`) y cuál es el que debe retornar (`next`), si es que hay alguno.

#### 4.8.4

#### Comparar acceso mediante índices e iteradores

Hemos visto en las últimas dos secciones que tenemos por lo menos tres maneras diferentes de recorrer un `ArrayList`. Podemos usar un *ciclo for-each* (tal como lo hemos visto en la Sección 4.8.1), el método `get` con un índice (Sección 4.8.2) o podemos usar un objeto `Iterator` (Sección 4.8.3).

Por lo que sabemos hasta ahora, todos los abordajes parecen iguales en calidad. El primero es un poco más fácil de comprender.

El primer abordaje, usando el *ciclo for-each*, es la técnica estándar que se usa si deben procesarse todos los elementos de una colección porque es el más breve para este caso. Las últimas dos versiones tienen el beneficio de que la iteración puede ser detenida más fácilmente en el medio de un proceso, de modo que son preferibles para cuando se procesa sólo parte de una colección.

Para un `ArrayList`, los dos últimos métodos (usando *ciclos while*) son buenos aunque no siempre es así. Java provee muchas otras clases de colecciones además de `ArrayList`. Veremos algunas otras en los capítulos siguientes. Para algunas colecciones, es imposible o muy ineficiente acceder a elementos individuales mediante un índice. Por lo que nuestra primera versión del *ciclo while* es una solución particular para la colección `ArrayList` y puede que no funcione para otros tipos de colecciones.

La segunda solución, usando un iterador, está disponible para todas las colecciones de las clases de las bibliotecas de Java y es un patrón importante que usaremos nuevamente en posteriores proyectos.

## 4.9

## Resumen del ejemplo agenda

En el ejemplo agenda hemos visto cómo podemos usar un objeto `ArrayList`, creado a partir de una clase extraída de una biblioteca de clases, para almacenar un número arbitrario de objetos en una colección. No tenemos que decidir anticipadamente cuántos objetos deseamos almacenar y el objeto `ArrayList` mantiene automáticamente el registro de la cantidad de elementos que contiene.

Hemos hablado sobre cómo podemos usar un ciclo para recorrer todos los elementos de una colección. Java tiene varias construcciones para ciclos; las dos que hemos usado en este lugar son el *ciclo for-each* y el *ciclo while*.

En un `ArrayList` podemos acceder a sus elementos por un índice o podemos recorrerla completamente usando un objeto `Iterator`.

**Ejercicio 4.23** Use el proyecto `club` para realizar los siguientes ejercicios. El proyecto proporciona un esquema de la clase `Club`; su tarea consiste en completar el código de esta clase. La clase `Club` tiene la finalidad de almacenar objetos `Socios` en una colección.

Dentro de **Club** declare un campo de tipo **ArrayList**. Escriba una sentencia **import** adecuada para este campo y considere cuidadosamente el tipo de la lista. En el constructor, cree el objeto colección y asígnelo al campo. Asegúrese de que todos los archivos del proyecto compilen correctamente antes de pasar al próximo ejercicio.

**Ejercicio 4.24** Complete el método **numeroDeSocios** para devolver el tamaño actual de la colección. Antes de que tenga un método para agregar objetos en la colección, por supuesto que este método devolverá siempre cero, pero estará listo para ser probado más adelante.

**Ejercicio 4.25** Se representa un socio del club mediante una instancia de la clase **Socio**. El proyecto *club* provee una versión completa de la clase **Socio** que no requiere ninguna modificación. Una instancia contiene los detalles del nombre, el mes y el año en que la persona se asoció al club. Todos los detalles de los socios se completan cuando se crea una instancia. Se agrega un objeto **Socio** a la colección del objeto **Club** mediante el método **asociar** del objeto **Club** que tiene la siguiente descripción:

```
/**
 * Agrega un nuevo socio a la colección socios del club.
 * @param socio El objeto Socio que se agregará.
 */
public void asociar(Socio socio)
```

Complete el método **asociar**.

Cuando quiera agregar un objeto **Socio** al objeto **Club** en el banco de objetos, hay dos maneras de hacerlo: crear un objeto **Socio** en el banco de objetos, invocar el método **asociar** del objeto **Club** y hacer clic en el objeto **Socio** para pasarlo como parámetro o bien, invocar al método **asociar** del objeto **Club** y escribir en la caja de diálogo del parámetro del constructor:

```
new Socio("nombre del socio", mes, anio)
```

Cada vez que agregue un socio, utilice el método **numeroDeSocios** para verificar que el método **asociar** lo agregó a la colección y que el método **numeroDeSocios** da el resultado correcto.

Continuaremos explorando este proyecto más adelante en este capítulo mediante más ejercicios.

## 4.10

## Otro ejemplo: un sistema de subastas

En esta sección continuaremos con algunas de las ideas nuevas que hemos introducido en este capítulo, pero nuevamente las veremos en un contexto diferente.

El proyecto *subasta* modela parte de la operación de un sistema de subastas *online*. La idea central es que una subasta consiste en un conjunto de elementos que se ofrecen para su venta. Estos elementos se denominan «lotes» y a cada lote se le asigna un número único que lo identifica. Una persona trata de comprar el lote que desea ofreciendo cierta cantidad de dinero por él. Nuestras subastas son ligeramente diferentes

de otras porque ofrecen todos los lotes por un tiempo limitado<sup>2</sup>. Al finalizar este tiempo, se cierra la subasta y se considera compradora del lote a la persona que ofertó la mayor cantidad de dinero. Si, al cierre de la subasta el lote no tiene ofertantes, se lo considera no vendido y estos lotes pueden ser ofrecidos en posteriores subastas.

El proyecto *subastas* contiene las siguientes clases: Subasta, Oferta, Lote y Persona. Ni la clase Oferta, ni la clase Persona desarrollan actividad alguna dentro del sistema por lo que aquí no las vemos en detalle: la clase Persona simplemente almacena el nombre de un ofertante y la clase Oferta almacena los detalles del valor de dicha oferta y quién la efectuó. El estudio de estas clases queda como un ejercicio para el lector, nosotros nos concentraremos en las clases Lote y Subasta.

#### 4.10.1

#### La clase Lote

La clase Lote almacena la descripción del lote, el número que lo identifica y los detalles de la mayor oferta recibida hasta el momento. La parte más complicada de la clase es el método *ofertarPara* (Código 4.6) que interviene cuando una persona realiza una oferta para ese lote. Cuando se realiza una oferta, es necesario controlar que su nuevo valor sea mayor que el valor de cualquier oferta existente para dicho lote; si resulta mayor, entonces se almacena dentro del lote la nueva oferta como la mayor oferta actual.

Primero verificamos si la oferta actual es mayor que la oferta máxima. Esto será cierto en el caso en que no se haya realizado ninguna oferta o si la oferta actual supera a la mejor oferta hecha hasta el momento. La primera parte del control involucra la siguiente prueba:

```
ofertaMaxima == null
```

Esta sentencia prueba, en realidad, si la variable *ofertaMaxima* está haciendo o no referencia a un objeto. La palabra clave *null* es un valor especial en Java que significa «no hay objeto». Si observa el constructor de la clase *Lote* verá que no se asigna explícitamente un valor inicial a este campo de lo que resulta que contiene el valor por defecto para las variables que hacen referencias a objetos que es *null*. De modo que, hasta que no se reciba una oferta para este lote, el campo *ofertaMaxima* contendrá el valor *null*.

#### Concepto

Se usa la palabra reservada *null* de Java para significar que «no hay objeto» cuando una variable objeto no está haciendo referencia realmente a un objeto en particular. Un campo que no haya sido inicializado explícitamente contendrá el valor por defecto *null*.

#### Código 4.6

El manejo de una oferta para un lote

```
public class Lote
{
    // La mayor oferta actual para este lote.
    private Oferta ofertaMaxima;
    Se omitieron los otros campos y el constructor

    /**
     * Intento de ofertar para este lote. Una oferta
     * exitosa debe tener un valor mayor que cualquier
```

<sup>2</sup> En vías de la simplicidad, no se implementa la característica «tiempo límite» de las subastas dentro de las clases que consideramos en este proyecto.

**Código 4.6  
(continuación)**

El manejo de una oferta para un lote

```

        * oferta existente.
        * @param oferta Una nueva oferta.
        * @return true si es exitosa, falso en caso contrario
        */
    public boolean ofertarPara(Oferta oferta)
    {
        if((ofertaMaxima == null) ||
           (oferta.getValor() >
            ofertaMaxima.getValor())) {
            // Esta oferta es mejor que la oferta
            actual.
            ofertaMaxima = oferta;
            return true;
        }
        else {
            return false;
        }
    }

Se omitieron los otros métodos.
}

```

#### 4.10.2 La clase Subasta

La clase Subasta (Código 4.7) proporciona una ilustración más detallada de los conceptos *ArrayList* y *ciclo for-each* tratados anteriormente en este capítulo.

El campo *lotes* es un *ArrayList* que se usa para contener los lotes ofrecidos en esta subasta. Los lotes se ingresan en la subasta pasando sólo una descripción al método *ingresarLote*. Se crea un nuevo lote pasando al constructor de *Lote* la descripción y un número de identificación. El nuevo objeto *Lote* se agrega a la colección. Las siguientes secciones tratan algunas características adicionales ilustradas en la clase Subasta.

**Código 4.7**

La clase Subasta

```

import java.util.ArrayList;
/**
 * Un modelo simplificado de una subasta.
 * La subasta mantiene una lista de lotes, de longitud
arbitraria.
 * @author David J. Barnes and Michael Kölling
 * @version 2006.03.30
 */
public class Subasta {
    // La lista de lotes de esta subasta.
    private ArrayList<Lote> lotes;
    // El número que se le dará al próximo lote que
    // ingrese a esta subasta.
    private int numeroDeLoteSiguiente;
}

```

**Código 4.7  
(continuación)**

La clase Subasta

```
    /**
     * Crea una nueva subasta.
     */
    public Subasta()
    {
        lotes = new ArrayList<Lote>();
        numeroDeLoteSiguiente = 1;
    }
    /**
     * Ingresa un nuevo lote a la subasta.
     * @param descripcion La descripción del lote.
     */
    public void ingresarLote(String descripcion)
    {
        lotes.add(new Lote(numeroDeLoteSiguiente,
descripcion));
        numeroDeLoteSiguiente++;
    }
    /**
     * Muestra la lista de todos los lotes de esta
subasta.
     */
    public void mostrarLotes()
    {
        for(Lote : lotes)
            System.out.println(lote.toString());
    }

    /**
     * Ofertar para un lote.
     * Emite un mensaje que indica si la oferta es
exitosa o no.
     * @param numeroDeLote El número de lote al que se
oferta.
     * @param ofertante      La persona que hace la
oferta.
     * @param valor           El valor de la oferta.
     */
    public void ofertarPara(int numeroDeLote, Persona
ofertante, long valor)
{
    Lote loteElegido = getLote(numeroDeLote);
    if(loteElegido != null) {
        boolean exito = loteElegido.ofertarPara(
new
Oferta(ofertante, valor);
        if(exito) {
            System.out.println("La oferta para
el lote número " +
```

**Código 4.7  
(continuación)**

La clase Subasta

```
numeroDeLote + " resultó exitosa.");
    }
    else {
        // Informa cuál es la mayor oferta
        Oferta ofertaMaxima =
loteElegido.getOfertaMaxima();
        System.out.println("El lote número: " + numeroDeLote +
                           " ya
tiene una oferta de: " +
ofertaMaxima.getValor());
    }
}
/***
 * Devuelve el lote de un determinado número.
Devuelve null
 * si no existe un lote con este número.
 * @param numeroDeLote El número del lote a
retornar.
 */
public Lote getLote(int numeroDeLote)
{
    if((numeroDeLote >= 1) && (numeroDeLote <
NumeroDeLoteSiguiente)) {
        // El número parece ser razonable.
        Lote loteElegido = lotes.get(numeroDeLote -
1);
        // Incluye un control confidencial para
asegurar que
        // el lote es el correcto
        if(loteElegido.getNumero() != numeroDeLote)
{
            System.out.println("Error interno: se
retornó el lote Nro. " +
LoteElegido.getNumero() +
                           " en
lugar del Nro. " +
                           + numeroDeLote);
        }
        return loteElegido;
    }
    else {
        System.out.println("El lote número: " +
numeroDeLote +
                           " no existe.");
    }
}
```

**Código 4.7  
(continuación)**

La clase Subasta

```
        return null;
    }
}
```

### 4.10.3 Objetos anónimos

El método `ingresarLote` de la clase `Subasta` ilustra un idioma común, los objetos anónimos y lo vemos en la siguiente sentencia:

```
lotes.add(new Lote(numeroDeLoteSiguiente, descripcion));
```

Aquí estamos haciendo dos cosas:

- Creamos un nuevo objeto `Lote` y
- Pasamos este nuevo objeto al método `add` de `ArrayList`.

Podríamos haber escrito lo mismo en dos líneas de código para producir el mismo efecto pero en pasos separados y más explícitos:

```
Lote nuevoLote = new Lote(numeroDeLoteSiguiente, descripcion);
lotes.add(nuevoLote);
```

Ambas versiones son equivalentes, pero si la variable `nuevoLote` no se usa más dentro del método, la primera versión evita declarar una variable que tenga un uso tan limitado. En efecto, creamos un objeto anónimo, un objeto sin nombre, pasándolo directamente al método que lo utiliza.

**Ejercicio 4.26** Agregue un método `cerrar` a la clase `Subasta`. Este método deberá recorrer la colección de lotes e imprimir los detalles de todos los lotes. Para hacerlo, puede usar tanto un *ciclo for-each* como un *ciclo while*. Cualquier lote que haya recibido por lo menos una oferta es considerado vendido. Para los lotes vendidos, los detalles incluyen el nombre del ofertante ganador y el valor de la oferta ganadora; para los lotes no vendidos, mostrar un mensaje que indique este hecho.

**Ejercicio 4.27** Agregue un método `getNoVendidos` a la clase `Subasta` con la siguiente firma:

```
public ArrayList getNoVendidos()
```

Este método debe recorrer el campo `lotes` y almacenar los no vendidos en un nuevo `ArrayList` que será una variable local. Al finalizar, el método devuelve la lista de los lotes no vendidos.

**Ejercicio 4.28** Suponga que la clase `Subasta` incluye un método que posibilita la eliminación de un lote de la subasta. Asuma que el resto de los lotes no cambian el valor de sus campos `loteNúmero` cuando se elimina un lote. ¿Qué efecto producirá eliminar un lote sobre el método `getLote`?

**Ejercicio 4.29** Escriba el método `getLote` de modo que no se fíe de que un lote con un número en particular sea almacenado en el índice (`loteNúmero-1`) de la colección. Por ejemplo, si se elimina el lote número 2, entonces

el lote número 3 se moverá del índice 2 al índice 1 y todos los números de lote mayores que 2 también cambiarán una posición. Puede asumir que los lotes se almacenan en orden creciente por su número de lote.

**Ejercicio 4.30** Agregue un método `eliminarLote` a la clase `Subasta`, con la siguiente firma:

```
/** Elimina el lote que tiene determinado número
 * @param numero El número del lote a eliminar
 * @return El Lote con el número dado, o null
 *         si no existe dicho lote.
 */
public Lote eliminarLote (int numero)
```

Este método no debe asumir que un lote con un número determinado esté almacenado en una posición particular de la colección.

**Ejercicio 4.31** La clase `ArrayList` se encuentra en el paquete `java.util`, que incluye también una clase de nombre `LinkedList`. Busque toda la información que pueda sobre esta última clase y compare sus métodos con los de `ArrayList`. ¿Qué métodos tienen en común y cuáles son diferentes?

#### 4.10.4 Usar colecciones

La clase colección `ArrayList` (y otras similares) constituyen una herramienta de programación importante porque muchos problemas requieren trabajar con colecciones de objetos de tamaño variable. Antes de continuar con el resto del capítulo es importante que se familiarice y se sienta cómodo trabajando con las colecciones; los siguientes ejercicios pueden ayudarlo.

**Ejercicio 4.32** Continúe trabajando con el proyecto `club` del Ejercicio 4.23. Defina un método en la clase `Club` con la siguiente descripción:

```
/** Determina el número de socios que se asociaron
 * en determinado mes.
 * @param mes El mes que nos interesa
 * @return el número de socios.
 */
public int asociadosEnMes(int mes)
```

Si el parámetro `mes` está fuera del rango válido 1-12, muestra un mensaje de error y devuelve el valor 0.

**Ejercicio 4.33** Defina un método en la clase `Club` con la siguiente descripción:

```
/** Elimina de la colección, todos los socios
 * que se hayan asociado en un mes determinado y
 * los devuelve en otro objeto colección.
 * @param mes El mes en que ingresó el socio
 * @param año El año de ingreso del socio
```

```
* @return Los socios que se asociaron en el mes  
dado  
*/  
public ArrayList<Socio> purgar(int mes, int anio)
```

Si el parámetro mes está fuera del rango válido 1-12, muestra un mensaje de error y devuelve un objeto colección vacío.

**Nota:** el método `purgar` es significativamente más difícil de escribir que los restantes métodos de esta clase.

**Ejercicio 4.34** Abra el proyecto `productos` y complete la clase `AdministradorDeStock` mediante este ejercicio y los que le siguen. La clase `AdministradorDeStock` usa un objeto `ArrayList` para almacenar los `Productos`. Su método `agregarProducto` ya agrega un producto en la colección, pero es necesario completar los siguientes métodos: `recibirProducto`, `buscarProducto`, `mostrarDetallesDeProductos` y `cantidadEnStock`.

Cada producto que vende la empresa se representa mediante una instancia de la clase `Producto` que registra su ID, su nombre y la cantidad que hay en stock. La clase `Producto` declara el método `aumentarCantidad` para registrar los incrementos de los niveles de stock de dicho producto. El método `venderUno` registra la venta de una unidad de dicho producto y disminuye en 1 el nivel del campo `cantidad`. El proyecto proporciona la clase `Producto` que no requiere ninguna modificación.

Comience por implementar el método `mostrarDetallesDeProductos` para asegurarse de que puede recorrer la colección de productos. Sólo imprima los detalles de cada `Producto` retornado invocando su método `toString`.

**Ejercicio 4.35** Implemente el método `buscarProducto` que busca en la colección un producto cuyo campo `ID` coincide con el argumento `ID` del método. Si encuentra un producto que coincide, lo devuelve como resultado del método; de lo contrario devuelve `null`.

Este método difiere de `mostrarDetallesDeProductos` en que no necesariamente hay que examinar todos los productos de la colección para encontrar una coincidencia. Por ejemplo, si el primer producto de la colección coincide con el `ID` del producto buscado, finaliza el recorrido y se devuelve el primer producto. Por otro lado, es posible que no haya ninguna coincidencia en la colección, en cuyo caso se examinará la colección completa y no habrá ningún producto para devolver, por lo que retornará el valor `null`.

Cuando busque una coincidencia necesitará invocar al método `getID` sobre un `Producto`.

**Ejercicio 4.36** Implemente el método `cantidadEnStock` que debe ubicar un producto en la colección que coincide con su `ID` y devolver como resultado, la cantidad en stock del mismo; si no coincide con el `ID` de ningún producto, retorna cero. Este es un proceso relativamente simple de implementar una vez que haya completado el método `buscarProducto`. Por ejemplo, `cantida-`

dEnStock puede invocar al método `buscarProducto` para hacer la búsqueda y luego invocar sobre el resultado al método `getCantidad`. Tenga cuidado con los productos que no se encuentran, piense.

**Ejercicio 4.37** Implemente el método `recibirProducto` usando un enfoque similar al usado en `cantidadEnStock`. Puede buscar en la lista de productos el producto con un ID dado y luego invocar al método `aumentarCantidad`.

**Ejercicio 4.38** *Desafíos.* Implemente un método en `AdministradorDeStock` para mostrar los detalles de todos los productos cuyos niveles de stock están por debajo de un nivel determinado (que se pasa al método mediante un parámetro).

Modifique el método `agregarProducto` de modo que impida que se agregue en la lista un nuevo producto con un ID ya existente.

Agregue un método en `AdministradorDeStock` que busque un producto por su nombre en lugar de por su ID:

```
public Producto buscarProducto(String nombre)
```

Para implementar este método necesita saber que dos objetos String s1 y s2 pueden compararse para ver si son iguales mediante la expresión lógica:

```
s1.equals(s2)
```

Encontrará más detalles sobre este tema en el Capítulo 5.

## 4.11

### Resumen de colección flexible

Hemos visto que clases tales como `ArrayList` nos permiten crear colecciones que contienen un número arbitrario de objetos. La biblioteca de Java contiene más colecciones similares a esta y veremos algunas otras en los próximos capítulos. Encontrará que usar estas colecciones confidencialmente es una habilidad importante para escribir programas interesantes.

Existe apenas una aplicación, que veremos a partir de ahora, que no usa colecciones de este estilo. Sin embargo, antes de investigar otras variantes de colecciones flexibles de la biblioteca estudiaremos primero las colecciones de tamaño fijo.

## 4.12

### Colecciones de tamaño fijo

Las colecciones de tamaño flexible son muy potentes porque no necesitamos conocer anticipadamente la cantidad de elementos que se almacenarán y porque es posible variar el número de los elementos que contienen. Sin embargo, algunas aplicaciones son diferentes por el hecho de que conocemos anticipadamente cuántos elementos deseamos almacenar en la colección y este número permanece invariable durante la vida de la colección. En tales circunstancias, tenemos la opción de elegir usar una colección de objetos de tamaño fijo, especializada para almacenar los elementos.

Una colección de tamaño fijo se denomina *array* o *arreglo*. A pesar del hecho de que los arreglos tengan un tamaño fijo puede ser una desventaja, se obtienen por lo menos

**Concepto**

Un arreglo es un tipo especial de colección que puede almacenar un número fijo de elementos.

dos ventajas en compensación, con respecto a las clases de colecciones de tamaño flexible:

- El acceso a los elementos de un arreglo es generalmente más eficiente que el acceso a los elementos de una colección de tamaño flexible.
- Los arreglos son capaces de almacenar objetos o valores de tipos primitivos. Las colecciones de tamaño flexible sólo pueden almacenar objetos<sup>3</sup>.

Otra característica distintiva de los arreglos es que tienen una sintaxis especial en Java, el acceso a los arreglos utiliza una sintaxis diferente de las llamadas a los métodos habituales. La razón de esta característica es mayormente histórica: los arreglos son las estructuras de colección más antiguas en los lenguajes de programación y la sintaxis para tratarlos se ha desarrollado durante varias décadas. Java utiliza la misma sintaxis establecida en otros lenguajes de programación para mantener las cosas simples para los programadores que todavía usan arreglos, aunque no sea consistente con el resto de la sintaxis del lenguaje.

En las siguientes secciones mostraremos cómo se pueden usar los arreglos para mantener una colección de tamaño fijo. También introducimos una nueva estructura de ciclo que, con frecuencia, se asocia fuertemente con los arreglos, el *ciclo for*. (Tenga en cuenta que el ciclo *for* es diferente del ciclo *for-each*.)

#### 4.12.1 Un analizador de un archivo de registro o archivo «log»

Los servidores web, típicamente mantienen archivos de registro de los accesos de los clientes a las páginas web que almacenan. Dadas las herramientas convenientes, estos archivos de registro permiten a los administradores de servicios web extraer y analizar información útil tal como:

- Cuáles son las páginas más populares que proveen.
- Si se rompieron los enlaces de otros sitios con estas páginas web.
- La cantidad de datos entregada a los clientes.
- Los períodos de mayor cantidad de accesos durante un día, una semana o un mes.

Esta información puede permitir a los administradores, por ejemplo, determinar si necesitan actualizar sus servidores para que resulten más potentes o establecer los períodos de menor actividad para realizar las tareas de mantenimiento.

El proyecto *analizador-weblog* contiene una aplicación que lleva a cabo un análisis de los datos de un servidor web. El servidor graba una línea de registro en un archivo cada vez que se realiza un acceso. En la carpeta del proyecto hay un ejemplo de un archivo de registro denominado *weblog.txt*. Cada línea registra la fecha y hora del acceso en el siguiente formato:

año mes día hora minutos

---

<sup>3</sup> Una construcción de Java denominada «autoboxing» (que encontraremos más adelante en este libro) proporciona un mecanismo que nos permite almacenar valores primitivos en colecciones de tamaño flexible. Sin embargo, es cierto que sólo los arreglos pueden almacenar directamente tipos primitivos.

Por ejemplo, la línea siguiente registra un acceso hecho a las 3:45 am del 7 de junio de 2006:

2006 06 07 03 45

El proyecto consta de cuatro clases: AnalizadorLog, LectorDeArchivoLog, EntradaDeLog y SeparadorDeLineaLog. Invertiremos la mayor parte de nuestro tiempo en ver la clase AnalizadorLog porque contiene ejemplos de creación y uso de un arreglo (Código 4.8). Más tarde, en los ejercicios, instamos al lector a examinar y modificar la clase EntradaDeLog porque también usa un arreglo. Las clases restantes utilizan características del lenguaje Java que aún no hemos tratado, de modo que no las exploraremos en detalle.

#### Código 4.8

El analizador de archivo log

```
/*
 * Lee los datos de un servidor web y analiza
 * los modelos de acceso de cada hora.
 *
 * @author David J. Barnes and Michael Kölling.
 * @version 2006.03.30
 */
public class AnalizadorLog
{
    // Arreglo para almacenar la cantidad de accesos por
    // hora.
    private int[] contadoresPorHora;
    // Usa un LectorDeArchivoLog para acceder a los datos
    private LectorDeArchivoLog lector;
    /**
     * Crea un objeto para analizar los accesos a la
     * web en cada hora.
     */
    public AnalizadorLog()
    {
        // Crea un objeto arreglo para guardar la
        // cantidad
        // de accesos por hora.
        contadoresPorHora = new int[24];
        // Crea el lector para obtener los datos.
        lector = new LectorDeArchivoLog();
    }
    /**
     * Analiza los accesos por hora a partir de los
     * datos del archivo log.
     */
    public void analizarPorHora()
    {
        while(lector.hayMasDatos()) {
            EntradaLog entrada =
            lector.siguienteEntrada();
```

```
        int hora = entrada.getHora();
        contadoresPorHora[hora]++;
    }
}
/** 
 * Imprime las cantidades de accesos hora por hora.
 * Debe ser rellenado previamente mediante un
llamado a analizarPorHora
 */
public void imprimirContadoresPorHora()
{
    System.out.println("Hora: Cantidad");
    for(int hora = 0; hora <
contadoresPorHora.length; hora++) {
        System.out.println(hora + ":" + 
contadoresPorHora[hora]);
    }
}

/**
 * Imprime las líneas de datos leídas por el
LectorDeArchivoLog
 */
public void imprimirDatos()
{
    lector.imprimirDatos();
}
}
```

El analizador utiliza realmente sólo una parte de los datos almacenados en una línea de un archivo log de un servidor. Proporciona información que nos podría permitir determinar en qué horas del día el servidor tiende, en promedio, a estar más ocupado o desocupado y lo hace contando la cantidad de accesos que se realizaron en cada hora, durante el período cubierto por el archivo log.

**Ejercicio 4.39** Explore el proyecto *anificador-weblog*. Para ello cree un objeto *AnificadorLog* e invoque su método *analizarPorHora*. A continuación llame al método *imprimirContadoresPorHora*. ¿Qué resultados muestra el analizador? ¿Cuáles son las horas del día en que se realizaron más accesos?

En las próximas secciones examinaremos la forma en que esta clase utiliza un arreglo para cumplir con su tarea.

#### 4.12.2 Declaración de variables arreglos

La clase *AnificadorLog* contiene un campo que es de tipo arreglo:

```
private int[ ] contadoresPorHora;
```

La característica distintiva de la declaración de una variable de tipo arreglo es un par de corchetes que forman parte del nombre del tipo: `int[ ]`. Este detalle indica que la variable `contadoresPorHora` es de tipo *arreglo de enteros*. Decimos que `int` es el tipo base de este arreglo en particular. Es importante distinguir entre una declaración de una variable arreglo y una declaración simple ya que son bastante similares:

```
int hora;
int[ ] contadoresPorHora;
```

En este caso, la variable `hora` es capaz de almacenar un solo valor entero mientras que la variable `contadoresPorHora` se usará para hacer referencia a un objeto arreglo, una vez que dicho objeto se haya creado. La declaración de una variable arreglo no crea en sí misma un objeto arreglo, sólo reserva un espacio de memoria para que en un próximo paso, usando el operador `new`, se cree el arreglo tal como con los otros objetos.

Merece la pena que miremos nuevamente la rara sintaxis de esta notación por un momento. Podría ser una sintaxis de aspecto más convencional tal como `Array<int>` pero, tal como lo mencionamos anteriormente, las razones de esta notación son más históricas que lógicas. Deberá acostumbrarse a leer los arreglos de la misma forma que las colecciones, como un «*arreglo de enteros*».

**Ejercicio 4.40** Escriba una declaración de una variable arreglo de nombre `gente` que podría usarse para referenciar un arreglo de objetos `Persona`.

**Ejercicio 4.41** Escriba una declaración de una variable arreglo `vacante` que hará referencia a un arreglo de valores lógicos.

**Ejercicio 4.42** Lea la clase `AnalizadorLog` e identifique todos los lugares en los que aparece la variable `contadoresPorHora`. En esta etapa, no se preocupe sobre el significado de todos sus usos dado que se explicará en las siguientes secciones. Observe la frecuencia con que se utiliza un par de corchetes con esta variable.

**Ejercicio 4.43** Encuentre los errores de las siguientes declaraciones y corríjalas.

```
[ ] contadores;
boolean [5000] ocupado;
```

#### 4.12.3 Creación de objetos arreglo

La próxima cuestión por ver es la manera en que se asocia una variable arreglo con un objeto arreglo.

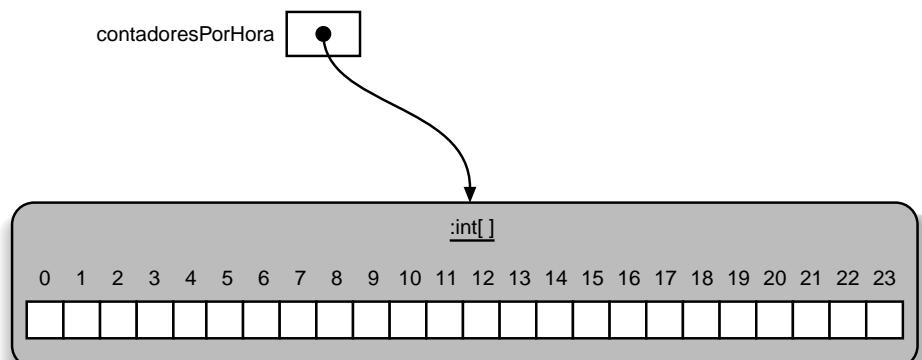
El constructor de la clase `AnalizadorLog` incluye una sentencia para crear un arreglo de enteros:

```
contadoresPorHora = new int[24];
```

Esta sentencia crea un objeto arreglo que es capaz de almacenar 24 valores enteros y hace que la variable arreglo `contadoresPorHora` haga referencia a dicho objeto. La Figura 4.5 muestra el resultado de esta asignación.

**Figura 4.5**

Un arreglo de 24 enteros



La forma general de la construcción de un objeto arreglo es:

```
new tipo/expresión-entera]
```

La elección del tipo especifica de qué tipo serán todos los elementos que se almacenarán en el arreglo. La expresión entera especifica el tamaño del arreglo, es decir, un número fijo de elementos a almacenar.

Cuando se asigna un objeto arreglo a una variable arreglo, el tipo del objeto arreglo debe coincidir con la declaración del tipo de la variable. La asignación a `contadoresPorHora` está permitida porque el objeto arreglo es un arreglo de enteros y la variable `contadoresPorHora` es una variable arreglo de enteros. La línea siguiente declara una variable arreglo de cadenas que hace referencia a un arreglo que tiene capacidad para 10 cadenas:

```
String [ ] nombres = new String[10];
```

Es importante observar que la creación del arreglo asignado a `nombres` no crea realmente 10 cadenas. En realidad, crea una colección de tamaño fijo que es capaz de almacenar 10 cadenas en ella. Probablemente, estas cadenas serán creadas en otra parte de la clase a la que pertenece `nombres`. Inmediatamente después de su creación, un objeto arreglo puede pensarse como vacío. En la próxima sección veremos la forma en que se almacenan los elementos en los arreglos y la forma de recuperar dichos elementos.

**Ejercicio 4.44** Dadas las siguientes declaraciones de variables:

```
double [ ] lecturas;
String[ ] urls;
MaquinaDeBoletos[ ] maquinas;
```

Escriba sentencias que lleven a cabo las siguientes tareas: a) la variable `lecturas` hace referencia a un arreglo capaz de contener 60 valores de tipo `double`; b) la variable `urls` hace referencia a un arreglo capaz de contener 90 objetos `String`; c) la variable `maquinas` hace referencia a un arreglo capaz de contener cinco objetos `MaquinaDeBoletos`.

**Ejercicio 4.45** ¿Cuántos objetos `String` se crean mediante la siguiente declaración?

```
String [ ] etiquetas = new String[20];
```

**Ejercicio 4.46** Detecte el error que presenta la siguiente declaración de arreglo y corríjalo.

```
double [ ] precios = new double(50);
```

#### 4.12.4 Usar objetos arreglo

Se accede a los elementos individuales de un objeto arreglo mediante un *índice*. Un índice es una expresión entera escrita entre un par de corchetes a continuación del nombre de una variable arreglo. Por ejemplo:

```
etiquetas[6];
maquinas[0];
gente[x + 10 - y];
```

Los valores válidos para una expresión que funciona como índice depende de la longitud del arreglo en el que se usarán. Los índices de los arreglos siempre comienzan por cero y van hasta el valor uno menos que la longitud del arreglo. Por lo que los índices válidos para el arreglo `contadoresPorHora` son desde 0 hasta 23, inclusive.

**Cuidado:** dos errores muy comunes al trabajar con arreglos: uno es pensar que los índices válidos de un arreglo comienzan en 1 y otro, usar el valor de la longitud del arreglo como un índice. Usar índices fuera de los límites de un arreglo trae aparejado un error en tiempo de ejecución denominado `ArrayIndexOutOfBoundsException`.

Las expresiones que seleccionan un elemento de un arreglo se pueden usar en cualquier lugar que requiera un valor del tipo base del arreglo. Esto quiere decir que podemos usarlas, por ejemplo, en ambos lados de una asignación. Aquí van algunos ejemplos de uso de expresiones con arreglos en diferentes lugares:

```
etiqueta[5] = "Salir";
double mitad = lecturas[0] / 2;
System.out.println(gente[3].getNombre());
maquinas[0] = new MaquinaDeBoletos(500);
```

El uso de un índice de un arreglo en el lado izquierdo de una asignación es equivalente a un método de modificación (o método *set*) del arreglo porque cambiará el contenido del mismo. Los restantes usos del índice son equivalentes a los métodos de acceso (o métodos *get*).

#### 4.12.5 Analizar el archivo log

El arreglo `contadoresPorHora` creado en el constructor de `AnalizadorLog` se usa para almacenar un análisis de los datos sobre el acceso. Los datos se almacenan en el arreglo dentro del método `analizarPorHora` y los datos del arreglo se muestran en el método `imprimirContadoresPorHora`. Como la tarea del método `analizar` es contar cuantos accesos se hicieron durante cada período de una hora, el arreglo necesita 24 posiciones, una para cada período de una hora del día. El analizador delega la tarea de leer el archivo a la clase `LectorDeArchivoLog`.

La clase `LectorDeArchivoLog` es un poco más complicada y sugerimos que no invierta demasiado tiempo en investigar su implementación. Su rol es realizar la tarea de tomar cada línea del archivo de registros y separar los valores de los datos, pero nos podemos abstraer de los detalles de su implementación considerando sólo el encabezado de dos de sus métodos:

```
public boolean hayMasDatos()
public EntradaLog siguienteEntrada()
```

El método `hayMasDatos` le dice al analizador si existe por lo menos una entrada más en el archivo de registros y el método `siguienteEntrada` retorna un objeto `EntradaLog` que contiene los valores de la siguiente línea del archivo. Estos dos métodos imitan el estilo de los métodos `hasNext` y `next` de la clase `Iterator` dado que puede haber un número arbitrario de entradas en un archivo log en particular.

Para cada `EntradaLog`, el método `anализarPorHora` del analizador obtiene el valor del campo hora:

```
int hora = entrada.getHora();
```

Sabemos que el valor almacenado en la variable local `hora` se mantendrá siempre en el rango 0 a 23 que coincide exactamente con los valores del rango de los índices del arreglo `contadoresPorHora`. Cada posición del arreglo se usa para representar un contador de accesos para la hora correspondiente. De modo que, cada vez que se lee un valor de hora queremos actualizar el contador de esa hora en 1. Lo hemos escrito así:

```
contadoresPorHora[hora]++;
```

Las siguientes alternativas son equivalentes a esta, pues usamos un elemento de un arreglo exactamente de la misma forma en que lo podemos hacer con una variable común:

```
contadoresPorHora[hora] = contadoresPorHora[hora] + 1;
contadoresPorHora[hora] += 1;
```

Al final del método `anализарPorHora` tenemos un conjunto completo de valores en los contadores para cada hora del período del archivo de registros.

En la próxima sección veremos el método `imprimirContadoresPorHora` como medio para presentar una nueva estructura de control que encaja perfectamente con el recorrido de un arreglo.

#### 4.12.6 El *ciclo for*

Java define dos variantes para el *ciclo for*, ambas se indican mediante la palabra clave `for`. En la Sección 4.8 hemos presentado la primer variante, el *ciclo for-each*, como una manera conveniente de recorrer una colección flexible. La segunda variante, el *ciclo for*, es una estructura de control repetitiva<sup>4</sup> alternativa que resulta particularmente adecuada cuando:

- queremos ejecutar un conjunto de sentencias un número exacto de veces

---

<sup>4</sup> A veces, cuando la gente quiere distinguir más claramente entre el *ciclo for* y el *ciclo for-each*, nombran al primero como «*ciclo for* de estilo antiguo», ya que pertenece al lenguaje Java mucho antes que el *ciclo for-each*. A veces se hace referencia al *ciclo for-each* como «*ciclo for* mejorado».

- necesitamos una variable dentro del ciclo cuyo valor cambie en una cantidad fija, generalmente en 1, en cada iteración.

Por ejemplo, es común el uso del *ciclo for* cuando queremos hacer algo con cada elemento de un arreglo tal como imprimir el contenido de cada elemento. Esto encaja con el criterio de que el número fijo de veces se corresponde con la longitud del arreglo y la variable es necesaria para incrementar el índice del arreglo.

Un *ciclo for* tiene la siguiente forma general:

```
for (inicialización; condición; acción modificadora) {
    setencias a repetir
}
```

El siguiente ejemplo concreto está tomado del método `imprimirContadoresPorHora` del analizador del archivo log:

```
for(int hora = 0; hora < contadoresPorHora.length; hora++)
{
    System.out.println(hora + ":" + 
contadoresPorHora[hora]);
}
```

El resultado de este ciclo será que el valor de cada elemento del arreglo se imprime en pantalla precedido por su correspondiente número de hora. Por ejemplo:

```
0: 149
1: 149
2: 148
...
23: 166
```

Cuando comparamos este *ciclo for* con el *ciclo for-each*, observamos que la diferencia sintáctica aparece en la sección entre paréntesis del encabezado del ciclo. En este *ciclo for*, los paréntesis contienen tres secciones distintas separadas por símbolos de punto y coma (;).

Desde el punto de vista de un lenguaje de programación, habría sido mejor utilizar dos palabras claves diferentes para estos ciclos, que podrían ser *for* y *for-each*. El motivo por el que se utiliza la palabra clave *for* en ambos ciclos es nuevamente histórica y accidental. Las viejas versiones del lenguaje Java no contenían al *ciclo for-each*, y cuando finalmente se le introdujo, los diseñadores prefirieron no agregar una nueva palabra clave en esa etapa pues hacerlo podría causar dificultades en los programas existentes. De modo que decidieron usar la misma palabra clave *for* para ambos ciclos. Esto hace que nos sea relativamente más difícil distinguir entre estos dos ciclos, pero nos acostumbraremos a reconocerlos por las estructuras diferentes de sus encabezados.

Podemos ilustrar la forma en que se ejecuta un *ciclo for* escribiendo su forma general mediante un *ciclo while* equivalente:

```
inicialización;
while (condición) {
    setencias a repetir
    condición modificadora
}
```

Por lo que la forma alternativa del cuerpo de `imprimirContadoresPorHora` sería:

```
int hora = 0;
```

```

while (hora < contadoresPorHora.length) {
    System.out.println(hora + ":" + contadoresPorHora[hora]);
    hora++
}

```

En estas dos versiones podemos ver que la acción modificadora no se ejecuta realmente hasta que no se hayan ejecutado las sentencias del cuerpo del ciclo, por este motivo aparece como la última sección en el encabezado del *ciclo for*. Además, podemos ver que la inicialización se ejecuta una sola vez, inmediatamente antes de evaluar la condición por primera vez.

En ambas versiones observe que aparece la condición

```
hora < contadoresPorHora.length
```

Esto ilustra dos puntos importantes:

- Todos los arreglos contienen un campo `length` que contiene el valor del tamaño del arreglo. El valor de este campo coincide siempre con el valor entero usado para crear el objeto arreglo. Por lo que, el valor de `length` será 24.
- La condición usa el operador menor que «<» para controlar el valor de hora respecto de la longitud del arreglo. Por lo que en este caso, el ciclo continuará siempre que la hora sea menor que 24. En general, cuando deseamos acceder a cada elemento de un arreglo, el encabezado del ciclo *for* tendrá la siguiente forma:

```
for (int indice = 0; indice < arreglo.length; indice ++)
```

Esto es correcto porque no queremos usar un valor para el índice igual a la longitud del arreglo pues tal elemento no existe nunca.

¿Podríamos escribir también el *ciclo for* mostrado anteriormente mediante un *ciclo for-each*? La respuesta es: casi siempre. Aquí hay un intento:

```

for(int valor : contadoresPorHora) {
    System.out.println(": " + valor);
}

```

Este código compilará y se ejecutará. ¡Pruébelo! En este fragmento de código podemos ver que los arreglos pueden, de hecho, usarse con *ciclos for-each* tal como lo hicimos con las otras colecciones. Sin embargo, tenemos un problema: no podemos imprimir fácilmente la hora delante de los dos puntos. Es así porque el *ciclo for-each* no proporciona acceso a la variable `contadora` del ciclo, que necesitamos en este caso para imprimir la hora.

Para arreglar este código necesitaríamos definir nuestra propia variable `contadora` (de manera similar al ejemplo con *ciclo while*). En lugar de hacer esto, preferimos usar el *ciclo for* de estilo antiguo ya que es más conciso.

**Ejercicio 4.47** Verifique qué ocurre si en la condición del *ciclo for* se usa incorrectamente el operador «`<=`» en el método `imprimirContadoresPorHora`:

```
for(int hora = 0; hora <= contadoresPorHora.length; hora++)
```

**Ejercicio 4.48** Rescriba el cuerpo de `imprimirContadoresPorHora` de modo que reemplace al *ciclo for* por un *ciclo while* equivalente. Invoque el método resrito para comprobar que imprime los mismos resultados que antes.

**¿Qué ciclo debo usar?** Hemos hablado sobre tres ciclos diferentes: el *ciclo for*, el *ciclo for-each* y el *ciclo while*. Como habrá visto, en muchas situaciones el programador debe seleccionar el uso de alguno de estos ciclos para resolver una tarea. Generalmente, un ciclo puede ser resrito mediante otro ciclo. De modo que, ¿cómo puede hacer para decidir qué ciclo usar en una situación? Ofrecemos algunas líneas guías:

- Si necesita recorrer todos los elementos de una colección, el *ciclo for-each* es, casi siempre, el ciclo más elegante para usar. Es claro y conciso (pero no provee una variable contadora de ciclo).
- Si tiene un ciclo que no está relacionado con una colección (pero lleva a cabo un conjunto de acciones repetidamente), el *ciclo for-each* no resulta útil. En este caso, puede elegir entre el *ciclo for* y el *ciclo while*. El *ciclo for-each* es sólo para colecciones.
- El *ciclo for* es bueno si conoce anticipadamente la cantidad de repeticiones necesarias (es decir, cuántas vueltas tiene que dar el ciclo). Esta información puede estar dada por una variable, pero no puede modificarse durante la ejecución del ciclo. Este ciclo también resulta muy bueno cuando necesita usar explícitamente una variable contadora.
- El *ciclo while* será el preferido si, al comienzo del ciclo, no se conoce la cantidad de iteraciones que se deben realizar. El fin del ciclo puede determinarse previamente mediante alguna condición (por ejemplo, lee una línea de un archivo (repetidamente) hasta que alcanza el fin del archivo).

**Ejercicio 4.49** Corrija todos los errores que encuentre en el siguiente método.

```
/**
 * Imprime todos los valores del arreglo marcas
 * que son mayores que el promedio.
 * @param marcas Un arreglo que contiene valores de marcas
 * @param promedio El promedio de las marcas
 */
public void imprimirMayores (double marcas, double promedio)
{
    for(indice = 0; indice <= marcas.length; indice++){
        if (marcas[indice] > promedio) {
            System.out.println(marcas[indice]);
        }
    }
}
```

**Ejercicio 4.50** Rescriba el siguiente método de la clase *Agenda* que aparece en el proyecto *agenda2*, para que use un *ciclo for* en lugar de un *ciclo while*.

```
/**
 * Lista todas las notas de la agenda
 */
public void listarNotas()
```

```

{
    int indice = 0;
    while (indice < notas.size()) {
        System.out.println(notas.get(indice));
        indice++;
    }
}

```

**Ejercicio 4.51** Rescriba nuevamente el mismo método anterior, pero utilizando un *ciclo for-each*.

**Ejercicio 4.52** Complete el método `numeroDeAccesos` que se da a continuación, para contar el total de accesos grabados en el archivo de registros. Complételo usando un *ciclo for* para recorrer `contadoresPorHora`.

```

/**
 * Devuelve el número de accesos grabados en el archivo log
 */
public int numeroDeAccesos()
{
    int total = 0;
    // Sumar el valor de cada elemento de
    // contadoresPorHora a total
    ...
    return total;
}

```

**Ejercicio 4.53** Agregue el método `numeroDeAccesos` a la clase `AnalizadorLog` y compruebe si da el resultado correcto. *Pista:* puede simplificar su verificación haciendo que el analizador lea archivos de registros que contengan pocas líneas de datos. De esta manera, podrá determinar fácilmente si el método da el resultado correcto. La clase `LectorDeArchivoLog` tiene un constructor con la siguiente firma para leer un archivo en particular:

```

/**
 * Crea un LectorDeArchivoLog para traer los datos
 * desde un archivo de registros en particular
 * @param nombreDeArchivo El archivo con los datos sobre
 * los accesos.
 */
public LectorDeArchivoLog (String nombreDeArchivo)

```

**Ejercicio 4.54** Agregue un método `horaMasOcupada` al `AnalizadorLog` que devuelva la hora de mayor cantidad de accesos del día. Puede llevar a cabo esta tarea recorriendo el arreglo `contadoresPorHora` para encontrar el elemento que contiene el mayor número. *Pista:* ¿necesita probar cada elemento para ver si ha encontrado la hora más ocupada? De ser así, use un *ciclo for* o un *ciclo for-each*. ¿Qué ciclo resulta mejor para este caso?

**Ejercicio 4.55** Agregue un método `horaMasTranquila` al `AnalizadorLog` que devuelva el número de la hora con menos cantidad de accesos. *Nota:* este problema suena idéntico al ejercicio anterior pero tiene una pequeña trampa. Asegúrese de controlar su método con algún conjunto de datos en el que todos los contadores tengan valores distintos de cero.

**Ejercicio 4.56** ¿Qué hora retorna el método horaMasOcupada si existe más de una hora con el mismo nivel de accesos?

**Ejercicio 4.57** Agregue un método al AnalizadorLog que encuentre el período de dos horas en el que se presenta la mayor cantidad de accesos. Retorna el valor de la primer hora de este período.

**Ejercicio 4.58** *Desafío.* Grabe el proyecto analizador-weblog con un nombre diferente de modo que pueda desarrollar una nueva versión que realice un análisis más extensivo de los datos disponibles. Por ejemplo, sería útil conocer qué días tienden a ser más tranquilos que otros. Por ejemplo, los siete días ¿constituyen un modelo cíclico? Para poder realizar un análisis diario, mensual o anual necesitará hacer algunos cambios en la clase EntradaLog. Esta clase almacena todos los valores que provienen de una línea del archivo log, pero sólo están disponibles las horas y los minutos mediante métodos de acceso. Agregue métodos para hacer que los restantes campos estén disponibles de la misma manera. Luego agregue un conjunto de métodos adicionales de análisis en el analizador.

**Ejercicio 4.59** *Desafío.* Si completó el ejercicio anterior podría extender el formato del archivo log con campos numéricos adicionales. Por ejemplo, los servidores comúnmente almacenan un código numérico que indica si un acceso resultó o no exitoso. Se establece el valor 200 para un acceso exitoso; 403 quiere decir que se prohibió el acceso al documento y 404 significa que no se pudo encontrar el documento. Provea al analizador de información sobre el número de accesos exitosos y fallidos. Este ejercicio es realmente un desafío pues requiere que realice cambios en cada clase del proyecto.

## 4.13

### Resumen

En este capítulo hemos hablado sobre los mecanismos para almacenar colecciones de objetos en lugar de objetos únicos en diferentes campos. Hemos visto en detalle dos tipos de colecciones diferentes: el ArrayList como un ejemplo de una colección de tamaño flexible y los arreglos como colecciones de tamaño fijo.

El uso de colecciones como estas será muy importante en todos los proyectos de aquí en adelante. Verá que la mayoría de las aplicaciones tienen en algún punto la necesidad de usar una colección. Las colecciones son fundamentales para escribir programas.

Cuando se usan colecciones, aparece la necesidad de recorrer todos sus elementos para hacer uso de los objetos que tiene almacenados. Con este propósito hemos visto el uso de los ciclos y de los iteradores.

Los ciclos son un concepto fundamental en computación que se usará en cada proyecto de aquí en adelante. Asegúrese de que se ha familiarizado lo suficiente con la escritura de ciclos porque no podrá ir muy lejos sin ellos.

En paralelo hemos mencionado la biblioteca de clases de Java; una gran colección de clases útiles que podemos usar para dar más potencia a nuestras clases. Necesitaremos estudiar la biblioteca más detalladamente para ver qué otras cosas debiéramos saber sobre ella. Este será el tema del próximo capítulo.

Términos introducidos en este capítulo

**colección, arreglo, iterador, ciclo for-each, ciclo while, ciclo for, índice, sentencia import, biblioteca, paquete, objeto anónimo**

## Resumen de conceptos

- **colecciones** Las colecciones de objetos son objetos que pueden almacenar un número arbitrario de otros objetos.
- **ciclo** Un ciclo se usa para ejecutar un bloque de sentencias repetidamente sin tener que escribirlas varias veces.
- **iterador** Un iterador es un objeto que proporciona funcionalidad para recorrer todos los elementos de una colección.
- **null** Se usa la palabra reservada de Java **null** para indicar que «no hay objeto» cuando una variable objeto no está haciendo referencia a un objeto en particular. Un campo que no ha sido inicializado explícitamente contendrá por defecto el valor **null**.
- **arreglo** Un arreglo es un tipo especial de colección que puede almacenar un número fijo de elementos.

**Ejercicio 4.60** En el proyecto *curso-de-laboratorio* que hemos trabajado en capítulos anteriores, la clase *CursoDeLaboratorio* incluye un campo **estudiantes** para mantener una colección de objetos *Estudiante*. Lea el código de *CursoDeLaboratorio* para reforzar los conceptos que hemos tratado en este capítulo.

**Ejercicio 4.61** La clase *CursoDeLaboratorio* impone un límite al número de estudiantes que se pueden inscribir en un grupo en particular. Teniendo esto en vista, para el campo **estudiantes** ¿considera que sería más apropiado usar un arreglo de tamaño fijo en lugar de una colección de tamaño flexible? Señale motivos a favor y en contra de estas alternativas.

**Ejercicio 4.62** Java proporciona otro tipo de ciclo: el *ciclo do-while*. Averigüe como funciona este ciclo y descríbalo. Escriba un ejemplo de *ciclo do-while* que imprima todos los números del 1 al 10. Para obtener información sobre este ciclo busque una descripción del lenguaje Java (por ejemplo en

<http://java.sun.com/docs/books/tutorial/java/nutsandbolts/>

en la sección Control Flow Statements).

**Ejercicio 4.63** Rescriba el método *listarNotas* de la agenda que utilice un *ciclo do-while*.

**Ejercicio 4.64** Busque información sobre la sentencia *switch-case* de Java. ¿Cuál es su finalidad? ¿Cómo se la usa? Escriba un ejemplo. (Esta sentencia es otra sentencia de control de flujo, de modo que podrá encontrar información en direcciones similares a las que usó para encontrar el *ciclo do-while*.)



## CAPÍTULO

# 5

## Comportamiento más sofisticado

Principales conceptos que se abordan en este capítulo

- uso de clases de biblioteca
- lectura de documentación
- escritura de documentación

Construcciones Java que se abordan en este capítulo

`String, ArrayList, Random, HashMap, HashSet, Iterator, Arrays, static, final`

En el Capítulo 4 hemos introducido la clase `ArrayList` de la biblioteca de clases de Java y vimos la forma en que nos permite hacer algo que, con otros medios, sería muy complicado de implementar; en este caso, almacenar un número arbitrario de objetos.

Este fue sólo un ejemplo sencillo de la utilidad de una clase de la biblioteca de Java. La biblioteca está compuesta por miles de clases, muchas de las cuales son generalmente muy útiles para nuestro trabajo (y muchas de las cuales probablemente no las usemos nunca).

Es esencial para un buen programador Java ser capaz de trabajar con la biblioteca de Java y de realizar elecciones informadas de las clases a usar. Una vez que comience a trabajar con la biblioteca, verá rápidamente que le permite llevar a cabo muchas tareas más fácilmente que si no la usara. El tópico central de este capítulo es aprender a trabajar con las clases de la biblioteca.

Presentaremos y discutiremos varias clases diferentes de la biblioteca. A lo largo de este capítulo trabajaremos en la construcción de una aplicación sencilla (el sistema Soporte Técnico) que hace uso de varias clases distintas de la biblioteca. La implementación completa que contiene todas las ideas y el código fuente que se discute aquí, así como varias versiones intermedias, se incluyen en el CD y en el sitio web de este libro. Ya que esto le permitirá estudiar la solución completa, le sugerimos seguir el camino a través de todos los ejercicios de este capítulo. Luego de una mirada breve al programa completo, comenzaremos con una versión inicial muy simple del proyecto y luego iremos desarrollando e implementando gradualmente la solución completa del sistema.

La aplicación hace uso de varias clases de biblioteca nuevas y las técnicas que cada una requiere tal como números aleatorios, mapas de hashing, conjuntos y explosión de cadenas. Hacemos una advertencia, este capítulo no es para leer y comprender en un solo día sino que contiene numerosas secciones que merecen algunos días de estudio. Al finalizar el capítulo y luego de haber manejado los conceptos para implementar las soluciones de los ejercicios, habrá aprendido una buena variedad de temas importantes.

## 5.1

### Documentación de las clases de biblioteca

#### Concepto

**Biblioteca Java.** La biblioteca de clases estándar de Java contiene muchas clases que son muy útiles. Es importante saber cómo se usa la biblioteca.

La biblioteca de Java es enorme. Consiste en miles de clases, cada una de las cuales tiene muchos métodos, con y sin parámetros, y con y sin tipo de retorno. Es imposible memorizarlas todas y recordar todos los detalles que contienen. En lugar de memorizarlas, un buen programador Java debiera conocer:

- algunas de las clases más importantes de la biblioteca por su nombre (una de ellas es `ArrayList`) y
- la forma de encontrar otras clases y buscar sus detalles (tales como sus métodos y parámetros).

En este capítulo presentaremos algunas de las clases importantes de la biblioteca de clases y otras vendrán más adelante en este libro. Pero lo más importante es que mostraremos la forma en que usted puede explorar y comprender la biblioteca por sus propios medios. Esto le permitirá escribir programas mucho más interesantes. Afortunadamente, la biblioteca de Java está muy bien documentada. Esta documentación está disponible en formato HTML (de modo que puede leerla en un navegador) y es lo que usaremos para hallar información sobre las clases de la biblioteca.

La primer parte de nuestra introducción a las clases de biblioteca apuntan a poder leer y comprender la documentación. Luego daremos un paso más y veremos cómo preparar nuestras propias clases de modo que otras personas puedan usarlas de la misma manera en que se usan las clases de la biblioteca estándar. Este es un punto verdaderamente importante para el desarrollo real de software en el que los equipos deben lidiar con proyectos muy grandes y mantener el software actualizado.

Una de las cosas que puede haber notado sobre la clase `ArrayList` es que la hemos utilizado sin mirar su código fuente. No hemos controlado cómo fue implementada; no fue necesario para usar su funcionalidad. Todo lo que necesitamos saber fue el nombre de la clase, los nombres de los métodos, los parámetros y los tipos de retorno de los métodos y saber exactamente qué hacen estos métodos. No nos importó cómo realizan el trabajo. Este es un punto típico del uso de clases de biblioteca.

La misma cuestión es cierta para otras clases en proyectos de software grandes. Generalmente, algunas personas trabajan juntas en un proyecto pero trabajando sobre partes diferentes. Cada programador se debe concentrar en su propia área y no necesita comprender todos los detalles de las otras partes (hablamos de esto en la Sección 3.2 donde tratamos la abstracción y la modularización). En efecto, cada programador debe estar capacitado para usar las clases de otros miembros del equipo como si fueran clases de biblioteca, haciendo uso de ellas a través de la información y sin tener la necesidad de conocer cómo funcionan internamente.

Para este trabajo, cada miembro del equipo debe escribir la documentación de la clase en forma similar a la documentación de la biblioteca estándar de Java de modo que permita a otras personas usar la clase sin necesidad de leer el código. Este punto también lo tratamos en este capítulo.

## 5.2

# El sistema Soporte Técnico

Como lo hacemos siempre, exploraremos los temas con un ejemplo. Esta vez usaremos la aplicación *Soporte Técnico* que puede encontrarla en el sitio web o en el CD como un proyecto de nombre *soporte-tecnico1*.

Esta aplicación es un programa que intenta brindar soporte técnico a los clientes de una empresa ficticia de desarrollo de software DodgySoft. Un tiempo atrás, DodgySoft tenía un departamento de soporte técnico en el que los clientes eran atendidos telefónicamente por personal situados en puestos de trabajo que recibían las llamadas en las que los clientes pedían ayuda y consejos para sus problemas técnicos con los productos de DodgySoft. Recientemente, el negocio no anduvo bien y DodgySoft decidió levantar el departamento de soporte técnico para ahorrar dinero. Ahora, quieren desarrollar el sistema Soporte Técnico para dar la impresión de que todavía lo proveen personalmente. Se supone que el sistema imita las respuestas que daría una persona de este departamento. Los clientes se pueden comunicar con el sistema de soporte técnico *on-line*.

### 5.2.1 Explorar el sistema Soporte Técnico

**Ejercicio 5.1** Abra y ejecute el proyecto *soporte-tecnico-completo*. Puede ejecutarlo creando un objeto de la clase *SistemaDeSoporte* e invocar su método *iniciar*. Ingrese algunos de los problemas que podría tener con el software para probar este sistema. Observe cómo se comporta. Cuando quiera terminar ingrese la palabra «*bye*». En esta etapa, no necesita examinar el código. Este proyecto constituye la solución completa que habremos desarrollado al finalizar el capítulo. El propósito de este ejercicio solamente es dar una idea del plan que queremos llevar a cabo.

**Eliza** La idea del proyecto Soporte Técnico está basada en el programa de inteligencia artificial Eliza desarrollado por Joseph Weizenbaum en el Massachusetts Institute of Technology en los años 60. Puede encontrar más información sobre el programa original buscando en la web las palabras «*Eliza*» y «*Weizenbaum*».

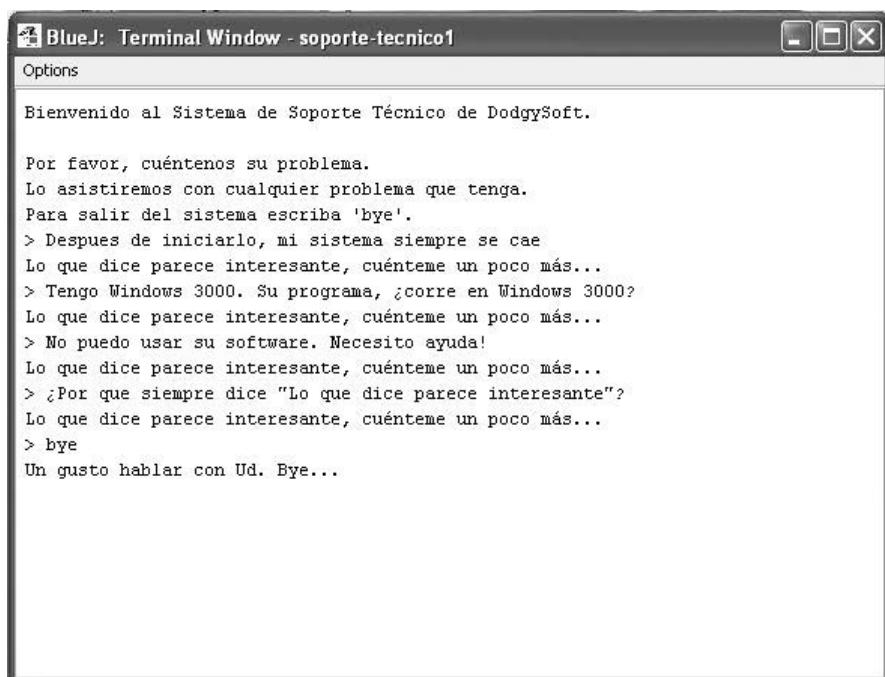
Comenzaremos nuestra exploración más detalladamente usando el proyecto *soporte-tecnico1*. Es una primera y rudimentaria implementación de nuestro sistema; la iremos mejorando a lo largo de este capítulo. De esta manera podremos obtener una mejor comprensión del sistema completo que la que podríamos obtener leyendo la solución del mismo.

En el Ejercicio 5.1 habrá visto que el programa, esencialmente, mantiene un diálogo con el usuario. El usuario puede escribir una pregunta y el sistema responde. Pruebe nuestra versión que es un prototipo del proyecto, *soporte-tecnico1*, haciendo las mismas preguntas que realizó en el Ejercicio 5.1.

En la versión completa, el sistema se las arregla para producir respuestas razonablemente variadas. Algunas de ellas ¡hasta parecen tener sentido! En la versión que vamos a desarrollar, las respuestas son mucho más restringidas (Figura 5.1). Verá rápidamente que la respuesta es siempre la misma: “*Lo que dice parece interesante, cuénteme un poco más...*”.

**Figura 5.1**

Primer diálogo con el Soporte Técnico



```

BlueJ: Terminal Window - soporte-tecnico1
Options

Bienvenido al Sistema de Soporte Técnico de DodgySoft.

Por favor, cuéntenos su problema.
Lo asistiremos con cualquier problema que tenga.
Para salir del sistema escriba 'bye'.
> Despues de iniciararlo, mi sistema siempre se cae
Lo que dice parece interesante, cuénteme un poco más...
> Tengo Windows 3000. Su programa, ¿corre en Windows 3000?
Lo que dice parece interesante, cuénteme un poco más...
> No puedo usar su software. Necesito ayuda!
Lo que dice parece interesante, cuénteme un poco más...
> ¿Por que siempre dice "Lo que dice parece interesante"?
Lo que dice parece interesante, cuénteme un poco más...
> bye
Un gusto hablar con Ud. Bye...

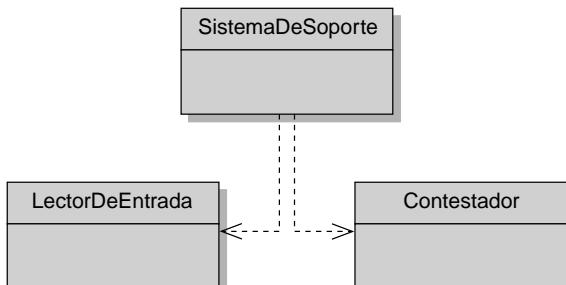
```

De hecho, esta respuesta no es nada interesante y nada convincente si pretendemos tener una persona de soporte técnico sentada del otro lado del diálogo. Trataremos de mejorarlo a la brevedad. Sin embargo, antes de hacerlo, exploraremos más detalladamente qué tenemos hasta ahora.

El diagrama del proyecto muestra tres clases: `SistemaDeSoporte`, `LectorDeEntrada` y `Contestador` (Figura 5.2). `SistemaDeSoporte` es la clase principal que usa la clase `LectorDeEntrada` para tomar alguna entrada desde la terminal y la clase `Contestador` para generar una respuesta.

**Figura 5.2**

Diagrama de clases del sistema Soporte Técnico



Examine un poco más la clase `LectorDeEntrada` creando un objeto de la misma y viendo sus métodos. Verá que sólo tiene un método disponible, denominado `getEntrada` que devuelve una cadena; pruébelo. Este método permite escribir una línea en la terminal de texto y como resultado del método, devuelve siempre lo que se haya escrito. No examinaremos ahora cómo funciona internamente, sólo tenga en cuenta que `LectorDeEntrada` tiene un método `getEntrada` que devuelve una cadena.

Haga lo mismo con la clase `Contestador`. Encontrará que tiene el método `generarRespuesta` que devuelve siempre la cadena “*Lo que dice parece interesante, cuénteme un poco más...*”. Esta cuestión explica lo que hemos visto anteriormente al llevar a cabo un diálogo.

Ahora veamos la clase `SistemaDeSoporte` un poco más de cerca.

### 5.2.2 Lectura de código

El código completo de la clase `SistemaDeSoporte` se muestra en Código 5.1. En Código 5.2 mostramos el código de la clase `Contestador`.

Al ver el Código 5.2 observamos que la clase `Contestador` es trivial: tiene sólo un método y siempre devuelve la misma cadena. Esto es algo que mejoraremos más adelante. Por ahora, nos concentraremos en la clase `SistemaDeSoporte`.

La clase `SistemaDeSoporte` declara dos campos de instancia para contener un objeto `LectorDeEntrada` y un objeto `Contestador` y su constructor crea y asigna estos dos objetos.

#### Código 5.1

La clase  
`SistemaDeSoporte`

```
/*
 * Esta clase implementa un sistema de soporte técnico.
 * Es la
 *   * clase de mayor nivel del proyecto. El sistema de
 *     soporte se
 *       * comunica mediante la terminal de texto con entradas y
 *         salidas
 *       * en ella.
 *       * La clase usa un objeto de clase LectorDeEntrada para
 *         leer las
 *         * entradas del usuario y un objeto de clase Contestador
 *           para
 *             * generar las respuestas.
 *             * Contiene un ciclo que repetidamente lee las entradas y
 *               genera
 *                 * las respuestas hasta que el usuario decide salir.
 *                 *
 *                 * @author      Michael Kölling y David J. Barnes
 *                 * @version    0.1 (2006.03.30)
 *                 */
public class SistemaDeSoporte
{
    private LectorDeEntrada lector;
```

**Código 5.1  
(continuación)**

La clase  
SistemaDeSoporte

```
private Contestador contestador;

/**
 * Crea un sistema de soporte técnico.
 */
public SistemaDeSoporte()
{
    lector = new LectorDeEntrada();
    contestador = new Contestador();
}

/**
 * Inicia el sistema de soporte técnico. Imprimirá
 * un mensaje
 * de bienvenida y establece un diálogo con el
 * usuario hasta
 * que el usuario lo finalice.
 */
public void iniciar()
{
    boolean terminado = false;
    imprimirBienvenida();
    while(!terminado) {
        String entrada = lector.getEntrada();
        if(entrada.startsWith("bye")) {
            terminado = true;
        }
        else {
            String respuesta =
            contestador.generarRespuesta();
            System.out.println(respuesta);
        }
    }
    imprimirDespedida();
}
/**
 * Imprime un mensaje de bienvenida en la pantalla.
 */
private void imprimirBienvenida()
{
    System.out.println(
        "Bienvenido al Sistema de Soporte Técnico
de DodgySoft.");
    System.out.println();
    System.out.println("Por favor, cuéntenos su
problema.");
    System.out.println(
        "Lo asistiremos con cualquier problema que
tenga.");
}
```

**Código 5.1  
(continuación)**

La clase  
SistemaDeSoporte

```
        System.out.println("Para salir del sistema
escriba 'bye'.");
    }
    /**
     * Imprime un mensaje de despedida en la pantalla.
     */
    private void imprimirDespedida()
    {
        System.out.println("Un gusto hablar con Ud.
Bye...");
    }
}
```

**Código 5.2**

La clase  
Contestador

```
/**
 * La clase contestador representa un objeto generador de
respuestas.
 * Se lo usa para generar una respuesta automatizada.
 *
 * @author Michael Kölling y David J. Barnes
 * @version 0.1 (2006.03.30)
 */
public class Contestador
{
    /**
     * Construye un Contestador, no hay nada para hacer.
     */
    public Contestador()
    {
    }
    /**
     * Genera una respuesta.
     * @return Una cadena que se mostrará como una
respuesta
     */
    public String generarRespuesta()
    {
        return "Lo que dice parece interesante,
cuénteme un poco más...";
    }
}
```

Al final, la clase tiene dos métodos de nombre `imprimirBienvenida` e `imprimirDespedida` que simplemente imprimen algún texto en la terminal: un mensaje de bienvenida y un mensaje de despedida respectivamente.

La parte más interesante de este código es el método que está en el medio, `iniciar`, que trataremos con un poco más de detalle.

Al comienzo de este método hay una llamada al método `imprimirBienvenida` y al final del mismo una llamada al método `imprimirDespedida`. Estas dos llamadas imprimen estas secciones de texto en el momento apropiado. El resto de este método consiste en la declaración de una variable booleana y en un *ciclo while*. La estructura es:

```
boolean terminado = false;

while (!terminado) {
    hacer algo
    if (condición de salida) {
        terminado = true;
    }
    else {
        hacer algo más
    }
}
```

Este modelo de código es una variante del *ciclo while* tratado en la Sección 4.7. Usamos la variable `terminado` como una bandera que se evalúa verdadera (`true`) cuando queremos terminar el ciclo (y junto con él, el programa completo). Nos aseguramos de que se haya inicializado en falso (`false`). (Recuerde que el signo de exclamación corresponde al operador lógico *no*.)

La parte principal del ciclo, la parte que se ejecuta repetidamente mientras no termine el ciclo, consiste en tres sentencias, si excluimos la evaluación de la condición de salida:

```
String entrada = lector.getEntrada();
...
String respuesta = contestador.generarRespuesta();
System.out.println(respuesta);
```

Por lo que el ciclo repetidamente:

- lee alguna entrada del usuario
- pide al contestador que genere una respuesta y
- muestra la respuesta en la pantalla.

(¡Habrá notado que la respuesta no depende para nada de la entrada! Esto es algo que tendremos seguramente que mejorar más adelante.)

La última parte para examinar es la evaluación de la condición de salida. La intención es que el programa termine una vez que el usuario escribe la palabra «*bye*». La sección relevante de código que encontramos en la clase dice

```
String entrada = lector.getEntrada();
If (entrada.startsWith("bye")) {
    terminado = true;
}
```

Si comprende estas partes por separado, es una buena idea leer nuevamente el método `iniciar` en el código completo de la clase (Código 5.1) para ver si puede comprenderlo cuando se presenta todo junto.

En el último fragmento de código examinado se utiliza un método de nombre `startsWith(<comienza con>)`. Dado que este método se invoca sobre la variable `entrada` que contiene un objeto `String`, debe ser un método de la clase `String`. Pero, ¿qué hace este método? ¿Cómo podemos buscar información sobre él?

Podemos suponer, simplemente a partir de su nombre, que este método comprueba si la cadena de entrada comienza con la palabra «bye». Podemos verificar si realmente lleva a cabo esta comprobación mediante un experimento. Ejecute el sistema Soporte Técnico nuevamente y escriba «bye bye» o «bye a todos». Verá que ambas versiones provocan la salida del sistema. Observe sin embargo, que al escribir «Bye» o «bye» (comienza con una letra mayúscula o deja un espacio en blanco delante de la palabra), el sistema no reconoce a estas palabras como la palabra «bye». Este hecho podría ser un poco desconcertante para el usuario pero dejaría de serlo si pudiéramos resolver estos problemas, y lo lograremos si conocemos un poco más sobre la clase String.

¿Cómo podemos encontrar más información sobre el método `startsWith` o sobre otros métodos de la clase String?

## 5.3

## Lectura de documentación de clase

La clase String es una de las clases de la biblioteca estándar de Java. Podemos encontrar más detalles sobre ella leyendo la documentación de la biblioteca para la clase String.

Para acceder a la documentación de la clase, seleccione el elemento *Java Class Libraries* del menú *Help* de BlueJ. Se abrirá un navegador mostrando la página principal de la documentación del API de Java (Application Programming Interface)<sup>1</sup>.

El navegador mostrará tres marcos. En el marco superior izquierdo verá una lista de paquetes. Debajo de este marco, aparece un listado de todas las clases de la biblioteca de Java. El marco más grande de la derecha se usa para mostrar los detalles de los paquetes o clases seleccionados.

En la lista de clases de la izquierda busque y seleccione la clase String; luego, el marco de la derecha mostrará la documentación de la clase String (Figura 5.3).

**Ejercicio 5.2** Investigue la documentación de la clase String. Luego busque la documentación de algunas otras clases. ¿Cuál es la estructura de la documentación de clase? ¿Cuáles son las secciones más comunes a todas las descripciones de clases? ¿Cuál es su propósito?

**Ejercicio 5.3** Busque el método `startsWith` en la documentación de la clase String. Describa con sus propias palabras qué es lo que hace este método.

**Ejercicio 5.4** ¿Existe algún método en la clase String que compruebe si una cadena termina con un sufijo determinado? De ser así, ¿cuál es su nombre y cuáles son sus parámetros y su tipo de retorno?

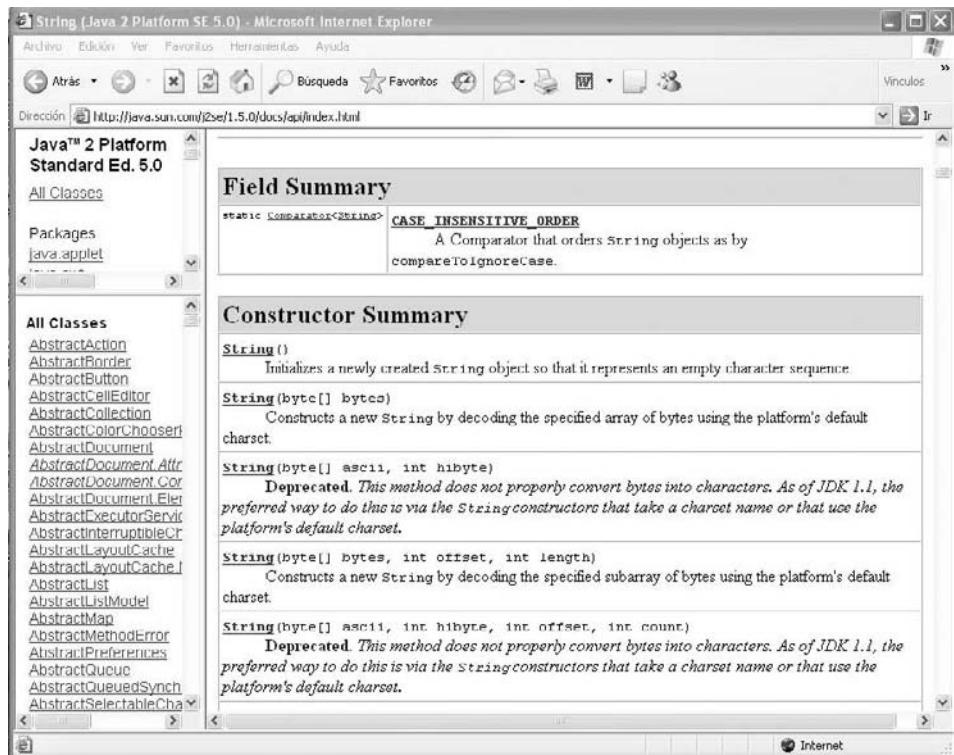
**Ejercicio 5.5** ¿Existe algún método en la clase String que devuelva el número de caracteres de una cadena? De ser así, ¿cuál es su nombre y sus parámetros?

**Ejercicio 5.6** Si encontró métodos para las últimas dos tareas, ¿cómo los encontró? Encontrar los métodos que se buscan, ¿es fácil o complicado? ¿Por qué?

<sup>11</sup> Por defecto, esta función accede a la documentación a través de Internet. No funcionará si su máquina no está conectada a la red. BlueJ puede configurarse para usar una copia local de la documentación de Java (API), que es recomendable ya que acelera el acceso a la documentación y puede funcionar sin una conexión a Internet. Para más detalles vea el Apéndice F.

**Figura 5.3**

La documentación de la biblioteca estándar de Java



### 5.3.1

## Comparar interfaz e implementación

### Concepto

La **interfaz** de una clase describe lo que es capaz de hacer dicha clase y la manera en que se puede usar sin mostrar su implementación.

Habrá visto que la documentación incluye diferentes piezas de información, entre otras:

- el nombre de la clase
- una descripción general del propósito de la clase;
- una lista de los constructores y los métodos de la clase;
- los parámetros y los tipos de retorno de cada constructor y de cada método;
- una descripción del propósito de cada constructor y cada método.

Toda esta información reunida recibe el nombre de *interfaz* de la clase. Observe que la interfaz no muestra el código con que está implementada la clase. Si una clase está bien escrita (es decir, su interfaz está bien redactada) entonces el programador no necesita ver el código fuente para usar dicha clase. La interfaz de la clase proporciona toda la información necesaria. Estamos nuevamente frente a la abstracción en acción.

### Concepto

El código completo que define una clase se denomina la **implementación** de dicha clase.

El código que subyace y que hace que la clase funcione se denomina la *implementación* de la clase. Generalmente, un programador trabaja sobre la implementación de una clase por vez, mientras que utiliza otras clases mediante sus interfaces.

La distinción entre interfaz e implementación es un concepto muy importante y será tratada repetidamente en este capítulo y a lo largo del libro.

Es importante ser capaz de distinguir entre los distintos significados de la palabra *interfaz* en cada contexto en particular.

**Nota:** el término inglés «interface» tiene varios significados en el contexto de programación y de Java. Se lo usa para describir la parte visible y pública de una clase (que es lo que hemos usado hasta ahora) pero también tiene otro significado. A la interfaz de usuario (frecuentemente una interfaz gráfica) también se la conoce como *la interface*; pero también Java tiene una construcción de lenguaje denominada *interface* (que trataremos en el Capítulo 10) que está relacionada con estas ideas pero cuyo significado es diferente del que estamos hablando ahora.

También se utiliza la terminología interfaz referida a métodos individuales. Por ejemplo, la documentación de la clase `String` nos muestra la interfaz del método `length`:

```
public int length()  
    Returns the length of this string. The length is equal  
    to the number of 16-bit Unicode characters in the  
    string.
```

*Returns:*

*the length of the sequence of characters represented by  
this object.*

La interfaz de un método consiste en su *signatura* y un comentario (que se muestra en el ejemplo en letra cursiva). La signatura de un método incluye, en este orden:

- un modificador de acceso que discutiremos más adelante (en este caso, `public`);
- el tipo de retorno del método (en este caso, `int`);
- el nombre del método;
- una lista de parámetros (que en este caso es vacía).

La interfaz de un método proporciona todos los elementos necesarios para saber cómo usarlo.

### 5.3.2

### Usar métodos de clases de biblioteca

#### Concepto

##### Objetos inmutables.

Se dice que un objeto es inmutable si su contenido o su estado no puede ser cambiado una vez que se ha creado.

Los objetos `String` son un ejemplo de objetos inmutables.

Volvamos al sistema de Soporte Técnico. Ahora queremos mejorar un poco el procesamiento de la línea de entrada. Hemos visto con anterioridad que nuestro sistema no es muy tolerante: si escribimos «Bye» o «bye» en lugar de «bye», el sistema no reconoce que se está intentando escribir lo mismo, desde el sentido humano. Queremos cambiar este aspecto para que se ajuste más a la lectura que puede hacer un usuario. Una cosa que tenemos que tener en cuenta es que un objeto `String` no puede ser modificado realmente una vez que está creado, en consecuencia, tenemos que crear un nuevo objeto `String` a partir de la cadena original.

La documentación de la clase `String` nos informa que tiene un método de nombre `trim` que elimina los espacios en blanco al principio y al final de una cadena. Podemos usar este método para solucionar el segundo problema, es decir, el caso en que la cadena «bye» tiene un blanco al comienzo.

---

<sup>2</sup> N. del T. El método `length` devuelve el largo de una cadena; coincide con el número de caracteres que contiene la secuencia de caracteres del objeto `String`.

**Ejercicio 5.7** Busque el método `trim` en la documentación de la clase `String`. Escriba la firma de dicho método. Escriba un ejemplo de invocación a este método sobre una variable de nombre `texto`. ¿Qué informa la documentación sobre los caracteres de control al comienzo de una cadena?

Después de estudiar la interfaz del método `trim` vemos que podemos eliminar los espacios en blanco de una cadena de entrada con una línea de código similar a la siguiente:

```
entrada = entrada.trim();
```

Este código le solicita al objeto almacenado en la variable `entrada` crear una nueva cadena similar a la dada, pero eliminados los espacios en blanco antes y después de la palabra. Luego la nueva cadena se almacena en la variable `entrada` por lo que pierde su viejo contenido, y en consecuencia, después de esta línea de código, `entrada` hace referencia a una cadena sin espacios al inicio y al final.

Ahora podemos insertar esta línea en nuestro código de modo que quede así:

```
String entrada = lector.getEntrada();
entrada = entrada.trim();
if (entrada.startsWith("bye")) {
    terminado = true;
}
else {
    Se omitió el código
}
```

Las primeras dos líneas podrían unirse para formar una sola línea:

```
String entrada = lector.getEntrada().trim();
```

El efecto de esta línea de código es idéntico al de las dos primeras líneas del fragmento de código anterior. El lado derecho de la asignación se puede leer como si hubiera un paréntesis de la siguiente manera:

```
(lector.getEntrada()).trim()
```

La versión que elija usar es sólo cuestión de gusto. La decisión podría hacerse en base a la legibilidad del código: utilice la versión que le resulte más fácil de leer y de comprender. Frecuentemente, los programadores novatos prefieren la versión de dos líneas mientras que los programadores experimentados usan el estilo de una sola línea.

**Ejercicio 5.8** Implemente la mejora que hemos tratado en su versión del proyecto `soporte-tecnico1`. Pruébelo para confirmar si resulta tolerante con espacios adicionales alrededor de la palabra «`bye`».

Hasta ahora, hemos resuelto el problema causado por los espacios sobrantes en la entrada pero todavía no hemos resuelto el problema de las letras mayúsculas. Sin embargo, la investigación más detallada de la clase `String` sugiere una posible solución a este problema, pues describe un método de nombre `toLowerCase` (*pasar a minúsculas*).

**Ejercicio 5.9** Mejore el código de la clase `SoporteDeSistema` del proyecto `soporte-tecnico1` de modo que ignore la capitalización de la entrada usando el método `toLowerCase` de la clase `String`. Recuerde que este método no cambia realmente la cadena sobre la que actúa sino que da como resultado la creación de una nueva cadena con un contenido ligeramente diferente.

### 5.3.3 Comprobar la igualdad de cadenas

Una solución alternativa podría haber sido comprobar si la cadena de entrada *es* realmente la cadena «*bye*» en lugar de ver si *comienza con* esta palabra. Un intento (¡incorrecto!) de escribir este código podría ser el siguiente:

```
if (entrada == "bye") { // ¡no siempre funciona!
    ...
}
```

El problema aquí radica en que es posible que existan varios objetos `String` independientes que representen la misma cadena. Por ejemplo, dos objetos `String` podrían contener ambos los caracteres «*bye*». ¡El operador (`==`) evalúa si ambos operandos hacen referencia al mismo objeto, no si sus valores son iguales! Y esta es una diferencia importante.

En nuestro ejemplo, nos interesa la cuestión de si la variable `entrada` y la constante de cadena «*bye*» representan el mismo valor, no si hacen referencia al mismo objeto, por lo que no resulta correcto usar el operador `==`. El uso de este operador podría retornar un resultado falso aun cuando el contenido de la variable `entrada` fuera «*bye*».

La solución para este problema es usar el método `equals` definido en la clase `String`. Este método comprueba correctamente si dos objetos `String` tienen el mismo contenido. El código correcto es el siguiente:

```
if (entrada.equals("chau")) {
    ...
}
```

Por supuesto que este método puede combinarse con los métodos `trim` y `toLowerCase`.

**Cuidado:** la comparación de dos cadenas mediante el operador `==` puede producir resultados incomprensibles e inesperados. Como regla general, las cadenas siempre se pueden comparar mediante el método `equals` en lugar de hacerlo con el operador `==`.

**Ejercicio 5.10** Busque el método `equals` en la documentación de la clase `String`. ¿Cuál es su tipo de retorno?

**Ejercicio 5.11** Modifique su implementación para usar el método `equals` en lugar del método `startsWith`.

## 5.4

## Agregar comportamiento aleatorio

Hasta ahora hemos hecho una pequeña mejora al proyecto Soporte Técnico pero aún así resulta demasiado rudimentario. Uno de los problemas principales del sistema reside en que continúa ofreciendo la misma respuesta independientemente del ingreso del usuario. Ahora mejoraremos este punto mediante la definición de un conjunto de frases posibles con las cuales responder al usuario. Luego tendremos que hacer que el programa seleccione aleatoriamente una de las frases cada vez que se espera una respuesta. Esta será una extensión de la clase `Contestador` de nuestro proyecto.

Para llevar a cabo esta mejora usaremos un `ArrayList` para almacenar las cadenas que funcionarán como respuestas, generaremos un número entero por azar y usaremos este número aleatorio como índice, para recuperar la respuesta desde la lista de frases. En esta versión, la respuesta del sistema aún no dependerá de la entrada del usuario (implementaremos esta funcionalidad más adelante) pero, por lo menos, las respuestas serán más variadas y el aspecto del programa será un poco mejor.

En primer lugar, debemos investigar cómo podemos generar un número entero por azar.

**Aleatorio y pseudo-aleatorio:** la generación de números por azar mediante una computadora no es en realidad tan fácil como uno podría pensar. Las computadoras operan de una manera bien definida y determinística que se apoya en el hecho de que todo cálculo es predecible y repetible, en consecuencia, existe poco espacio para un comportamiento realmente aleatorio.

Los investigadores, a lo largo del tiempo, han propuesto muchos algoritmos para producir secuencias semejantes a los números aleatorios. Estos números no son típicamente números aleatorios verdaderos, pero siguen reglas muy complicadas. Estos números se conocen como números *pseudo-aleatorios*.

En un lenguaje como Java, afortunadamente, la generación de números pseudo-aleatorios ha sido implementada en una clase de la biblioteca, de modo que, todo lo que tenemos que hacer para obtener un número de este tipo es escribir algunas invocaciones a dicha biblioteca.

Si quiere obtener más información sobre este tema, busque en la web «números pseudo-aleatorios».

### 5.4.1 La clase Random

La biblioteca de clases de Java contiene una clase de nombre `Random` que será de gran ayuda para nuestro proyecto.

**Ejercicio 5.12** Busque la clase `Random` en la documentación de la biblioteca de Java. ¿En qué paquete está? ¿Qué hace esta clase? ¿Cómo se puede construir una instancia? ¿Cómo puede generar números por azar? Tenga en cuenta que probablemente no comprenda todo lo que aparece en la documentación, sólo trate de encontrar lo que necesita saber.

**Ejercicio 5.13** Intente escribir en papel un fragmento de código que genere un número entero aleatorio mediante esta clase.

Para generar un número aleatorio tenemos que:

- crear una instancia de la clase `Random` y
- hacer una llamada a un método de esa instancia para obtener un número.

Al leer la documentación vemos que existen varios métodos de nombre `nextAlgo` para generar valores por azar de varios tipos diferentes. El método que genera un número entero por azar es el de nombre `nextInt`.

El párrafo siguiente ilustra el código necesario para generar y mostrar un número entero por azar:

```
Random generadorDeAzar;  
  
generadorDeAzar = new Random();  
int indice = generadorDeAzar.nextInt();  
System.out.println(indice);
```

Este fragmento de código crea una nueva instancia de la clase `Random` y la almacena en la variable `generadorDeAzar`. Luego, invoca al método `nextInt` de esta variable para obtener un número por azar, almacena el número generado en la variable `indice` y eventualmente lo imprime en pantalla.

**Ejercicio 5.14** Escriba código (en BlueJ) para probar la generación de números aleatorios. Para llevar a cabo esta tarea, cree una nueva clase de nombre `PruebaRandom`. Puede crear esta clase en el proyecto `soporte-tecnico1` o en un nuevo proyecto, esta cuestión no tiene importancia. En la clase `PruebaRandom` implemente dos métodos: `imprimirUnAleatorio` (que imprime un número aleatorio) y otro `imprimirVariosAleatorios (int cantidad)` (que tiene un parámetro que especifica la cantidad de números que se desea generar y luego los imprime).

Su clase debe crear una única instancia de la clase `Random` (en su constructor) y almacenarla en un campo. No debe crear una nueva instancia de `Random` cada vez que desea un nuevo número por azar.

#### 5.4.2 Números aleatorios en un rango limitado

Los números aleatorios que hemos visto hasta ahora fueron generados en el rango total de los números enteros de Java (-2147483648 a 2147483647). Este rango es bueno como para experimentar pero no resulta demasiado útil; es más frecuente que necesitamos números aleatorios dentro de un rango determinado.

La clase `Random` también ofrece un método que soporta esta restricción, su nombre también es `nextInt`, pero tiene un parámetro para especificar el rango de números que queremos usar.

**Ejercicio 5.15** Busque el método `nextInt` en la clase `Random` que permite especificar el rango de los números que se desean generar. ¿Cuáles son los posibles números aleatorios que se generarían si se invoca este método con un parámetro de valor 100?

**Ejercicio 5.16** Escriba un método en su clase `PruebaRandom` de nombre `lanzarDado` que devuelva un número comprendido entre 1 y 6 (inclusive).

**Ejercicio 5.17** Escriba un método de nombre `getRespuesta` que devuelva aleatoriamente una de las siguientes cadenas: «sí», «no» o «quizás».

**Ejercicio 5.18** Extienda su método `getRespuesta` de modo que utilice un `ArrayList` para almacenar un número arbitrario de respuestas y luego devuelva aleatoriamente, sólo una de las respuestas.

Cuando se utiliza un método para generar números por azar en un rango especificado, debe tenerse el cuidado de verificar si los límites se incluyen o no en el del intervalo. El método `nextInt(int n)` de la clase `Random` de la biblioteca de Java especifica que genera números desde 0 (inclusive) hasta `n` (exclusive). Esto quiere decir que el valor 0 está incluido entre los posibles valores de los resultados, mientras que el valor especificado por `n` no está incluido. El máximo número posible que devuelve es `n-1`.

**Ejercicio 5.19** Agregue un método a su clase `PruebaRandom` que tenga un parámetro `max` y genere números por azar en el rango que va desde 1 hasta `max` (inclusive).

**Ejercicio 5.20** Agregue un método a su clase `PruebaRandom` que tenga dos parámetros, `min` y `max`, y genere un número por azar en el rango comprendido entre `min` y `max` (inclusive).

### 5.4.3 Generar respuestas por azar

Ahora veremos una extensión de la clase `Contestador` para seleccionar una respuesta por azar de una lista de frases predefinidas. El Código 5.2 muestra el código de la clase `Contestador` tal como figura en nuestra primera versión.

Ahora agregaremos código a la primera versión para:

- declarar un campo de tipo `Random` para contener al generador de números aleatorios;
- declarar un campo de tipo `ArrayList` para guardar nuestras posibles respuestas;
- crear los objetos `Random` y `ArrayList` en el constructor de `Contestador`;
- llenar la lista de respuestas con algunas frases;
- seleccionar y devolver una frase aleatoriamente, cuando se invoca al método `generarRespuesta`.

El Código 5.3 muestra una versión del código de la clase `Contestador` con estos agregados.

#### Código 5.3

El código de  
Contestador con  
respuestas aleatorias

```
import java.util.ArrayList;
import java.util.Random;
/**
 * La clase contestador representa un objeto generador de
 * respuestas.
 * Se lo usa para generar una respuesta automática por azar
 * seleccionando una frase de una lista predefinida de
 * respuestas.
 *
 * @author Michael Kölling y David J. Barnes
 * @version 0.2 (2006.03.30)
 */
public class Contestador
{
```

**Código 5.3  
(continuación)**

El código de  
Contestador con  
respuestas aleatorias

```
private Random generadorDeAzar;
private ArrayList<String> respuestas;
/**
 * Crea un contestador.
 */
public Contestador()
{
    generadorDeAzar = new Random();
    respuestas = new ArrayList<String>();
    llenarRespuestas();
}
/**
 * Genera una respuesta.
 * @return Una cadena que se podría mostrar
 * como una respuesta
 */
public String generarRespuesta()
{
    // Toma un número aleatorio para el índice de
    // la lista
    // de respuestas por defecto.
    // El número estará entre 0(inclusive) y el
    // tamaño de
    // la lista(exclusive).
    int indice =
generadorDeAzar.nextInt(respuestas.size());
    return respuestas.get(indice);
}

/**
 * Construye una lista de respuestas por defecto
desde donde
 * se tomará una, cuando no sepamos más qué decir.
 */
private void llenarRespuestas()
{
    respuestas.add("Parece complicado. ¿Podría
describir \n" +
                    "el problema más
detalladamente?");
    respuestas.add("Hasta ahora, ningún cliente
informó \n" +
                    "sobre este problema.
\n" +
                    "¿Cuál es la
configuración de su equipo?");
    respuestas.add("Lo que dice parece interesante,
\n" +
```

### Código 5.3 (continuación)

El código de  
Contestador con  
respuestas aleatorias

```

    "cuénteme un poco más...
");
        respuestas.add("Necesito un poco más de
información. \n");
        respuestas.add("¿Verificó si tiene algún
conflicto \n" +
                    "con una dll? \n" );
        respuestas.add("Ese problema está explicado en
el manual. \n" +
                    "¿Leyó el manual? ");
        respuestas.add("Su descripción es un poco
confusa. \n" +
                    "¿Cuenta con algún
experto que lo \n" +
                    "ayude a describir el
problema \n" +
                    "de manera más
precisa?");
        respuestas.add("Eso no es una falla, es una
característica \n" +
                    "del programa. \n" );
        respuestas.add("¿Ha podido elaborar esto?");
    }
}

```

En esta versión hemos colocado el código que rellena la lista de respuestas dentro de un método propio de nombre `rellenarRespuestas` que se invoca en el constructor. Así nos aseguramos de que la lista de respuestas se llenará tan pronto como se cree un objeto `Contestador`, pero el código para construir la lista de respuestas se escribió por separado para que la clase resulte más fácil de leer y de comprender.

El segmento de código más interesante de la clase es el método `generarRespuesta`. Dejando de lado los comentarios dice así:

```

public String generarRespuesta()
{
    int indice =
generadorDeAzar.nextInt(respuestas.size());
    return respuestas.get(indice);
}

```

La primera línea de código de este método hace tres cosas:

- consulta el tamaño de la lista de respuestas invocando su método `size`;
- genera un número aleatorio comprendido entre 0 (inclusive) y el tamaño (exclusive);
- almacena el número aleatorio en la variable local `indice`.

Parece demasiada cantidad de código para una sola línea; también podríamos haber escrito:

```
int tamanioLista = respuestas.size();
int indice = generadorDeAzar.nextInt(tamanioLista);
```

Este código es equivalente al de la primera línea del párrafo de código anterior. La versión que prefiera nuevamente depende de cuál le resulta más fácil de leer.

Es importante tomar nota de que este fragmento de código genera un número aleatorio en el rango 0 a `tamanioLista-1` (incluidos ambos valores). Estos valores encajan perfectamente con los valores legales de los índices del `ArrayList`. Recuerde que el rango de índices de un `ArrayList` de tamaño `tamanioLista` va desde 0 hasta `tamanioLista-1`, por lo que los números por azar calculados se ajustan perfectamente al utilizarlos como índices para acceder aleatoriamente a un elemento de la lista de respuestas.

La última línea de código es:

```
return respuestas.get(indice);
```

Esta línea hace dos cosas:

- Recupera la respuesta de la posición `indice` mediante el método `get`.
- Devuelve la cadena seleccionada como resultado del método mediante la sentencia `return`.

Si no es cuidadoso, su código podría generar un número aleatorio fuera de los valores válidos de los índices del `ArrayList`. En consecuencia, cuando trate de usar ese índice para acceder a un elemento de la lista obtendrá un `IndexOutOfBoundsException`.

#### 5.4.4 Lectura de documentación de clases parametrizadas

Hasta ahora, le hemos pedido que busque la documentación de la clase `String` del paquete `java.lang` y de la clase `Random` del paquete `java.util`. Habrá observado, al realizar estas búsquedas, que algunos nombres de las clases que aparecen en la lista de la documentación tienen un formato ligeramente diferente, tal es el caso de `ArrayList<E>` o de `HashMap<K, V>`. Estas diferencias se deben a que el nombre de la clase está seguido de alguna información extra, encerrada entre los símbolos de menor y de mayor. Las clases similares a éstas se denominan «clases parametrizadas» o «clases genéricas». La información contenida entre los símbolos de menor y de mayor nos dice que, cuando usemos estas clases deberemos suministrar uno o más nombres de tipos entre dichos símbolos, para completar la definición. Ya hemos puesto en práctica esta idea en el Capítulo 4 cuando usamos varios `ArrayList` parametrizados con nombres de tipos tales como `String` y `Lote`:

```
private ArrayList<String> notas;
private ArrayList<Lote> lotes;
```

La documentación del API de Java refleja el hecho de que podemos parametrizar un `ArrayList` con cualquier otra clase que queramos usar como tipo. Por lo tanto, si busca en la lista de métodos de `ArrayList<E>` verá métodos tales como:

```
boolean add(E o)
E get(int index)
```

Estas signaturas nos indican que el tipo de objetos que podemos agregar a un `ArrayList` depende del tipo usado para parametrizarlo y lo mismo ocurre con el tipo de objetos que devuelve su método `get`. En efecto, si creamos un objeto `ArrayList<String>` la documentación nos informa que dicho objeto tendrá los siguientes dos métodos:

```
boolean add(String o)
String get(int index)
```

mientras que si creamos un objeto `ArrayList<Lote>` tendrá estos dos métodos:

```
boolean add(Lote o)
Lote get(int index)
```

Más adelante, en este mismo capítulo, le pediremos que busque la documentación para otros tipos parametrizados.

## 5.5

## Paquetes y la sentencia import

En la parte superior del código todavía hay dos líneas de las que no hemos hablado:

```
import java.util.ArrayList;
import java.util.Random;
```

Hemos encontrado por primera vez a la sentencia `import` en el Capítulo 4. Ahora llegó el momento de verla un poco más de cerca.

Las clases de Java se almacenan en la biblioteca de clases pero no están disponibles automáticamente para su uso, tal como las otras clases del proyecto actual. Para poder disponer de alguna de estas clases, debemos explicitar en nuestro código que queremos usar una clase de la biblioteca. Esta acción se denomina *importación de la clase* y se implementa mediante la sentencia `import`. La sentencia `import` tiene la forma general

```
import nombre-de-clase-calificado;
```

Dado que la biblioteca de Java contiene miles de clases, es necesaria alguna estructura en la organización de la biblioteca para facilitar el trabajo con este enorme número de clases. Java utiliza *paquetes (packages)* para acomodar las clases de la biblioteca en grupos que permanecen juntos. Los paquetes pueden estar anidados, es decir, los paquetes pueden contener otros paquetes.

Ambas clases, `ArrayList` y `Random` están en el paquete `java.util`. Esta información se puede encontrar en la documentación de la clase. El *nombre completo* o *nombre calificado* de una clase es el nombre de su paquete, seguido por un punto y por el nombre de la clase. Por lo que los nombres calificados de las dos clases que usamos aquí son `java.util.ArrayList` y `java.util.Random`.

Java también nos permite importar paquetes completos con sentencias de la forma

```
import nombre-del-paquete.*;
```

Por lo que la siguiente sentencia importaría todas las clases del paquete `java.util`:

```
import java.util.*;
```

La enumeración de todas las clases utilizadas separadamente, tal como aparece en nuestra primera versión, da un poco más de trabajo en términos de escritura pero funciona bien como parte de la documentación. Esta lista claramente indica las clases de

biblioteca que son realmente usadas por nuestras clases. De aquí en adelante, en este libro, tenderemos a usar el estilo del primer ejemplo, es decir, listar todas las clases importadas una por una.

Existe una excepción a esta regla: algunas clases se usan tan frecuentemente que casi todas las clases debieran importarlas. Estas clases se han ubicado en el paquete `java.lang` y este paquete se importa automáticamente dentro de cada clase. La clase `String` es un ejemplo de una clase ubicada en `java.lang`.

**Ejercicio 5.21** Implemente la solución de respuestas aleatorias tratada en esta sección, en su versión del sistema de Soporte Técnico.

**Ejercicio 5.22** ¿Qué ocurre cuando agrega más (o menos) respuestas posibles en la lista de respuestas? La selección por azar de una respuesta, ¿funciona adecuadamente? Justifique su respuesta.

La solución que hemos discutido aquí también está en el CD y en el sitio web bajo el nombre *soporte-tecnico2*. Sin embargo, le recomendamos como siempre, implementar la extensión de la clase por sus propios medios partiendo de la primera versión.

## 5.6

## Usar mapas para las asociaciones

Ahora tenemos una solución para nuestro sistema de soporte técnico que genera respuestas por azar. Esta versión es mejor que la primera pero aún no resulta muy convincente. En particular, la entrada del usuario no tienen ninguna influencia sobre la respuesta, y este es el punto que ahora nos proponemos mejorar.

El plan es que si tenemos un conjunto de palabras que pueden aparecer con cierta frecuencia en las preguntas, podríamos asociar estas palabras con alguna respuesta en particular. Si la entrada del usuario contiene alguna de nuestras palabras conocidas podríamos generar alguna respuesta relacionada con ellas. Este método es todavía muy imperfecto ya que no captura ningún significado de la entrada del usuario, tampoco reconoce un contexto, pero puede resultar sorprendentemente efectivo y además, es un buen próximo paso.

Para llevar a cabo el plan usaremos un `HashMap`. Puede encontrar la documentación de la clase `HashMap` en la documentación de la biblioteca de Java. Un `HashMap` es una especialización de un `Map` que también está documentado en la biblioteca. Verá que necesita leer la documentación de ambas clases para comprender qué es un `HashMap` y cómo funciona.

**Ejercicio 5.23** ¿Qué es un `HashMap`? ¿Cuál es su propósito y cómo se usa? Responda estas preguntas por escrito. Use la documentación de la biblioteca de Java de las clases `HashMap` y `Map` para responder estas preguntas. Tenga en cuenta que encontrará bastante difícil comprender todo ya que la documentación de estas clases no es muy buena. Trataremos los detalles más adelante en este capítulo pero vea qué cosas puede descubrir por su propio medios antes de seguir leyendo.

**Ejercicio 5.24** `HashMap` es una clase parametrizada. Nombre los métodos de esta clase que dependen del tipo usado para parametrizarla. ¿Considera que se podría usar el mismo tipo para sus dos parámetros?

## 5.6.1 Concepto de mapa

### Concepto

Un **mapa** es una colección que almacena pares llave/valor como entradas. Los valores se pueden buscar suministrando la llave.

Un mapa es una colección de pares de objetos llave/valor. Tal como el `ArrayList`, un mapa puede almacenar un número flexible de entradas. Una diferencia entre el `ArrayList` y un `Map` es que, en un `Map` cada entrada no es un único objeto sino un *par* de objetos. Este par está compuesto por un objeto *llave* y un objeto *valor*.

En lugar de buscar las entradas en esta colección mediante un índice entero (como hicimos con el `ArrayList`) usamos el objeto *llave* para buscar el objeto *valor*.

Un ejemplo cotidiano de un mapa es un directorio telefónico. Un directorio telefónico contiene entradas y cada entrada es un par: un nombre y un número de teléfono. Se usa una agenda telefónica para buscar un nombre y obtener un número de teléfono. No usamos un índice para encontrar el teléfono ya que el índice indicaría la posición de la entrada en la agenda y no el número telefónico buscado.

Un mapa puede organizarse de manera tal que resulte fácil buscar en él un valor para una llave. En el caso de la agenda telefónica, la organización está dada por un orden alfabético. Con el almacenamiento de las entradas por orden alfabético según sus llaves, resulta fácil encontrar la llave y buscar el valor correspondiente. La búsqueda inversa, es decir, buscar la llave para un valor dado, por ejemplo, buscar el nombre de un número de teléfono determinado, no resulta tan fácil con un mapa. En consecuencia, los mapas son ideales para una única forma de búsqueda, en la que conocemos la llave a buscar y necesitamos conocer solamente el valor asociado a esta llave.

## 5.6.2 Usar un `HashMap`

Un `HashMap` es una implementación particular de un `Map`. Los métodos más importantes de la clase `HashMap` son `put` y `get`.

El método `put` inserta una entrada en el mapa y el método `get` recupera el valor correspondiente a una llave determinada. El siguiente fragmento de código crea un `HashMap` e inserta tres entradas en él. Cada entrada es un par llave/valor que está compuesto por un nombre y un número de teléfono.

```
HashMap<String, String> agenda = new HashMap<String, String>();
agenda.put("Charles Nguyen", " (531) 9392 4587");
agenda.put("Lisa Jones", " (402) 4536 4674");
agenda.put("William H. Smith", " (998) 5488 0123");
```

Tal como hemos visto anteriormente con `ArrayList`, cuando se declara una variable `HashMap` y se crea un objeto `HashMap`, se debe indicar el tipo de objetos que se almacenarán en el mapa y, adicionalmente, el tipo de objetos que se usará para la llave. En la agenda telefónica usaremos cadenas tanto para las llaves como para los valores, pero estos dos tipos pueden ser diferentes.

El siguiente código busca el número de teléfono de Lisa Jones y lo imprime:

```
String numero = agenda.get("Lisa Jones");
System.out.println(numero);
```

Observe que se pasa la llave (el nombre «Lisa Jones») al método `get` para recuperar el correspondiente valor (el número de teléfono).

Lea nuevamente la documentación de los métodos `get` y `put` de la clase `HashMap` y vea si la explicación coincide con su conocimiento actual.

**Ejercicio 5.25** Cree una clase PruebaMap (ya sea dentro de su proyecto actual o en un nuevo proyecto). Use un `HashMap` para implementar una agenda telefónica de manera similar al ejemplo dado anteriormente. (Recuerde que debe importar la clase `java.util.HashMap`.) En la clase `PruebaMap` implemente dos métodos:

```
public void ingresarNumero(String nombre, String numero)
y
public String buscarNumero(String nombre)
```

Estos métodos deben usar los métodos `get` y `put` de la clase `HashMap` para implementar su funcionalidad.

**Ejercicio 5.26** ¿Qué ocurre cuando agrega una entrada al mapa con una llave que ya existe?

**Ejercicio 5.27** ¿Qué ocurre cuando agrega una entrada en el mapa con un valor que ya existe?

**Ejercicio 5.28** ¿Cómo puede verificar si el mapa contiene una llave determinada? (Aporte un ejemplo en código Java.)

**Ejercicio 5.29** ¿Qué ocurre cuando trata de buscar un valor y ese valor no existe en el mapa?

**Ejercicio 5.30** ¿Cómo puede controlar la cantidad de entradas que contiene el mapa?

### 5.6.3 Usar un mapa en el sistema Soporte Técnico

En el sistema Soporte Técnico podemos hacer un buen uso de un mapa usando palabras conocidas como llaves y las respuestas asociadas como valores. El Código 5.4 muestra un ejemplo en el que se crea un `HashMap` de nombre `mapaDeRespuestas` y se ingresan tres entradas en él. Por ejemplo, la palabra «lento» se asocia con el texto:

*«Me parece que esto tiene que ver con su hardware. Actualizar su procesador podría resolver todos estos problemas. ¿Ha tenido algún inconveniente con nuestro software?»*

Ahora, cuando alguien ingrese una pregunta que contenga la palabra «lento» podremos buscar e imprimir esta respuesta. Observe que la cadena de respuesta en el código ocupa varias líneas pero concatenadas con el operador `+`, de modo que el valor que entra en el `HashMap` es de una sola línea.

#### Código 5.4

Asociación de palabras seleccionadas con posibles respuestas

```
private HashMap mapaDeRespuestas<String, String>;
...
public Contestador()
{
    mapaDeRespuestas = new HashMap<String, String>();
    rellenarMapaDeRespuestas();
```

**Código 5.4  
(continuación)**

Asociación de palabras seleccionadas con posibles respuestas

```

        }
        /**
         * Ingresa todas las palabras llave conocidas y sus
         * respuestas asociadas, en nuestro mapa de
         * respuestas.
         */
        private void llenarMapaDeRespuestas()
        {
            mapaDeRespuestas.put("lento",
                "Me parece que esto tiene
                que ver con su hardware. \n" +
                "Actualizar su procesador
                podría resolver \n" +
                "todos estos problemas. \n"
                +
                "¿Ha tenido algún
                inconveniente con nuestro software?");
            mapaDeRespuestas.put("problema",
                "Bueno, Ud. sabe, todos los
                programas tiene \n" +
                "algún defecto. \n" +
                "Pero nuestros ingenieros
                están trabajando \n" +
                "duro para solucionarlos. \n"
                +
                "¿Puede describir el problema
                más detalladamente? \n");
            mapaDeRespuestas.put("caro",
                "El precio de nuestro
                producto es muy competitivo. \n" +
                "Realmente, ¿Ha visto y
                comparado todas nuestras \n" +
                "características");
        }
    
```

Un primer intento de escribir un método para generar las respuestas podría ser similar al método generarRespuesta que ofrecemos a continuación. En este punto y para simplificar las cosas por el momento, asumimos que el usuario ingresa solamente una palabra, por ejemplo «lento».

```

public String generarRespuesta(String palabra)
{
    String respuesta = mapaDeRespuestas.get(palabra);
    if (respuesta != null) {
        return respuesta;
    }
    else {
        // si llega acá es porque la palabra no
        fue reconocida
    }
}
    
```

```
        // En este caso, tomamos una de nuestras
        respuestas por defecto
        return tomarRespuestaPorDefecto();
    }
}
```

En este fragmento de código buscamos la palabra ingresada por el usuario en nuestro mapa de respuestas. Si encontramos una entrada que contenga la palabra ingresada por el usuario, la usamos para obtener la respuesta asociada. Si no encontramos una entrada para esa palabra, invocamos al método `tomarRespuestaPorDefecto`. Este método puede contener ahora el código de nuestra versión anterior de `generarRespuesta` que genera una respuesta aleatoriamente a partir de la lista de respuestas por defecto (tal como muestra el Código 5.3). La nueva lógica consiste en recuperar una respuesta adecuada si reconocemos la palabra o una respuesta aleatoria de nuestra lista de respuestas por defecto si no reconocemos la palabra ingresada.

**Ejercicio 5.31** Implemente las modificaciones de las que hablamos aquí en su propia versión del sistema de Soporte Técnico. Pruébelo para ver si funciona correctamente.

Este enfoque de asociar palabras llave con respuestas funciona bastante bien siempre y cuando el usuario no ingrese preguntas completas, es decir, funciona bien sólo cuando ingrese una sola palabra. La mejora final para completar la aplicación consiste en dejar que el usuario ingrese nuevamente preguntas completas y luego obtener respuestas que coincidan si reconocemos cualquiera de las palabras que contiene la pregunta.

Esta situación posiciona el problema en reconocer las palabras llave en la oración ingresada por el usuario. En la versión actual, el ingreso del usuario es devuelto por el `LectorDeEntrada` como una única cadena. Ahora queremos modificar este hecho para construir una nueva versión en la que el `LectorDeEntrada` devuelva la entrada del usuario como un conjunto de palabras. Técnicamente, la entrada será un conjunto de cadenas en el que cada cadena del conjunto representa una sola de las palabras ingresadas por el usuario.

Si logramos hacerlo, entonces podemos pasar el conjunto completo de palabras de la entrada del usuario al `Contestador`, que evaluará cada palabra del conjunto para ver si es reconocida y tiene una respuesta asociada.

Para implementar esta mejora en Java, necesitamos saber dos cosas: cómo dividir una única cadena en las varias palabras que contiene y cómo usar conjuntos. Estos son los puntos que trataremos en las próximas dos secciones.

## 5.7

## Usar conjuntos

La biblioteca estándar de Java incluye diferentes variantes de conjuntos, implementados en clases diferentes. La clase que usaremos se denomina `HashSet`.

**Ejercicio 5.32** ¿Cuáles son las similitudes y las diferencias entre un `HashSet` y un `ArrayList`? Utilice las descripciones de `Set`, `HashSet`, `List` y `ArrayList` que están en la documentación de la biblioteca para averiguarlo, dado que un `HashSet` es un caso especial de `Set` y un `ArrayList` es un caso especial de `List`.

Los dos tipos de funcionalidad que necesitamos de un conjunto son: ingresar elementos en él y más tarde, recuperar estos elementos. Afortunadamente, estas tareas no tienen demasiada dificultad para nosotros. Considere el siguiente fragmento de código:

```
import java.util.HashSet;
import java.util.Iterator;
...
HashSet<String> miConjunto = new HashSet<String>();

miConjunto.add("uno");
miConjunto.add("dos");
miConjunto.add("tres");
```

Compare este código con las sentencias que necesitamos para entrar elementos en un `ArrayList`. No hay prácticamente ninguna diferencia, excepto que esta vez creamos un `HashSet` en lugar de un `ArrayList`. Ahora veamos un recorrido por todos los elementos:

```
for(String : miConjunto) {
    Hacer algo con cada elemento
}
```

Nuevamente estas sentencias son las mismas que las que usamos para recorrer un `ArrayList` en el Capítulo 4.

#### Concepto

Un **conjunto** es una colección que almacena cada elemento individual una sola vez como máximo. No mantiene un orden específico.

Brevemente: los diferentes tipos de colecciones de Java se usan de manera muy similar. Una vez que comprendió cómo usar una de ellas, puede usarlas todas. Las diferencias reales residen en el comportamiento de cada colección. Por ejemplo, una lista contiene todos los elementos ingresados en el orden deseado, provee acceso a sus elementos a través de un índice y puede contener el mismo elemento varias veces. Por otro lado, un conjunto no mantiene un orden específico (el iterador puede devolver los elementos en diferente orden del que fueron ingresados) y asegura que cada elemento en el conjunto está una única vez. En un conjunto, el ingresar un elemento por segunda vez simplemente no tiene ningún efecto.

**List, Map y Set** Es tentador asumir que se puede usar un `HashSet` de manera similar a un `HashMap`. En realidad, tal como lo ilustramos, la forma de usar un `HashSet` es más parecida a la forma de usar un `ArrayList`. Cuando tratamos de comprender la forma en que se usan las diferentes clases de colecciones, la segunda parte del nombre es la mejor indicación de los datos que almacenan, y la primera palabra describe la forma en que se almacenan. Generalmente estamos más interesados en el «qué» (la segunda parte) antes que en el «cómo». De modo que un `TreeSet` debiera usarse de manera similar a un `HashSet`, mientras que un `TreeMap` debiera usarse de manera similar a un `HashMap`.

## 5.8

### Dividir cadenas

Ahora que hemos visto cómo usar un conjunto, podemos investigar cómo podemos dividir una cadena de entrada en palabras separadas para almacenarlas en un conjunto de palabras. La solución se muestra en una nueva versión del método `getEntrada` de la clase `LectorDeEntrada`. (Código 5.5)

**Código 5.5**

El método  
getEntrada  
devuelve un conjunto  
de palabras

```
/*
 * Lee una línea de texto desde la entrada estándar
 * (la terminal de
 *   * texto) y la retorna como un conjunto de palabras.
 *
 * @return Un conjunto de cadenas en el que cada
String es una de las
 *           palabras que escribió el usuario.
 */
public HashSet<String> getEntrada()
{
    System.out.print("> ");
    // imprime el prompt
    String linea =
lector.lineaSiguiente().trim().toLowerCase();

    String[] arregloDePalabras = linea.split(" ");
    // agrega las palabras del arreglo en el
    hashset
    HashSet<String> palabras = new HashSet<String>();
    for (String palabra : arregloDePalabras) {
        palabras.add(palabra);
    }

    return palabras;
}
```

En este código, además de usar un `HashSet` también utilizamos el método `split` de la clase `String`, que está definido en la biblioteca estándar de Java.

El método `split` puede dividir una cadena en distintas subcadenas y las devuelve en un arreglo de cadenas. El parámetro del método `split` establece la clase de caracteres de la cadena original que producirá la división en palabras. Hemos determinado que queremos dividir nuestra cadena mediante cada carácter espacio en blanco.

Las restantes líneas de código crean un `HashSet` y copian las palabras desde el arreglo al conjunto, antes de retornar el conjunto<sup>3</sup>.

**Ejercicio 5.33** El método `split` es más poderoso de lo que parece a partir de nuestro ejemplo. ¿Cómo puede definir exactamente cómo se dividirá la cadena? Dé algunos ejemplos.

---

<sup>3</sup> Existe una manera más elegante y breve de hacer lo mismo. Podríamos escribir `HashSet<String> palabras = new HashSet<String>(Arrays.asList(arregloDePalabras));` para reemplazar las cuatro líneas de código. Esta manera usa la clase `Arrays` de la biblioteca estándar y un *método estático* (también conocido como *método de clase*) que aún no hemos tratado en este libro. Si tiene curiosidad por este tema, recurra a la Sección 7.15.1 donde hablamos sobre los *métodos de clase* e intente usar esta versión.

**Ejercicio 5.34** Si quiere dividir una cadena en subcadenas, ya sea mediante cada carácter espacio en blanco o cada carácter de tabulación ¿Cómo podría invocar al método `split`? ¿Cómo se podría hacer si las palabras están separadas mediante el carácter dos puntos (:)?

**Ejercicio 5.35** ¿Cuál es la diferencia de resultados al devolver las palabras en un `HashSet` en comparación con devolverlas en un `ArrayList`?

**Ejercicio 5.36** Si existe más de un espacio en blanco entre dos palabras, por ejemplo, dos o tres espacios ¿qué ocurre?, ¿hay algún problema?

**Ejercicio 5.37** *Desafío.* Lea la nota al pie sobre el método `Arrays.asList`. Busque y lea las secciones de este libro que tratan sobre variables de clase y métodos de clase. Explique con sus propias palabras cómo funcionan.

¿Cuáles son los ejemplos de los otros métodos que proporciona la clase `Arrays`?

Cree una clase de nombre `PruebaOrdenamiento`. Cree en ella un método que acepte como parámetro un arreglo de valores enteros e imprima en la terminal los elementos ordenados (de menor a mayor).

## 5.9

## Terminar el sistema de Soporte Técnico

Para poner en acción las modificaciones que realizamos, tenemos que ajustar las clases `SistemaDeSoporte` y `Contestador` de modo que trabajen correctamente con un conjunto de palabras en lugar de con una sola cadena. El Código 5.6 muestra la nueva versión del método `iniciar` de la clase `SistemaDeSoporte` que no presenta demasiados cambios. Los cambios son:

- La variable `entrada`, que recibe el resultado desde `lector.getEntrada()`, ahora es de tipo `HashSet`.
- El control para finalizar la aplicación se hace mediante el método `contains` de la clase `HashSet` en lugar de hacerlo mediante un método de la clase `String`. (Busque este método en la documentación.)
- La clase `HashSet` debe ser importada usando una sentencia `import` (que aquí no se muestra).

### Código 5.6

Versión final del método `iniciar`

```
public void iniciar()
{
    boolean terminado = false;
    imprimirBienvenida();
    while(!terminado) {
        HashSet<String> entrada =
(lector.getEntrada());
        if(entrada.contains("bye")) {
            terminado = true;
```

### Código 5.6 (continuación)

Versión final del método iniciar

```

        }
        else {
            String respuesta =
            contestador.generarRespuesta(entrada);
            System.out.println(respuesta);
        }
    imprimirDespedida();
}
}

```

Finalmente, tenemos que ampliar el método generarRespuesta de la clase Contestador para que acepte un conjunto de palabras como parámetro. Luego, debe recorrer este conjunto y controlar cada una de las palabras en nuestro mapa de palabras conocidas. Si reconoce alguna de las palabras, retorna inmediatamente la respuesta asociada. Si no puede reconocer ninguna de las palabras, tomaremos como antes, una de las respuestas por defecto. El Código 5.7 muestra la solución.

### Código 5.7

Versión final del método generarRespuesta

```

public String generarRespuesta(HashSet<String> palabras)
{
    Iterator<String> it = palabras.iterator();
    while (it.hasNext()) {
        String palabra = (String) it.next();
        String respuesta =
        mapaDeRespuestas.get(palabra);
        if (respuesta != null) {
            return respuesta;
        }
    }
    // si llega acá es porque la palabra no fue
    // reconocida
    // En este caso, tomamos una de nuestras
    // respuestas por defecto
    return getRespuestaPorDefecto();
}

```

Esta es la última modificación a esta aplicación que tratamos en este capítulo. La solución en el proyecto *soporte-tecnico-completo* contiene todos estos cambios; también contiene más palabras asociadas con respuestas que las que se presentan en este capítulo.

Por supuesto que es posible realizar muchas mejoras a esta aplicación, pero no las discutiremos aquí sino que las sugerimos como ejercicios que quedan en manos del lector, algunos de los cuales son ejercicios desafiantes de programación.

**Ejercicio 5.38** Implemente las modificaciones finales de las que hablamos anteriormente, en su propia versión del programa.

**Ejercicio 5.39** Agregue en su aplicación más pares de palabras y respuestas al mapa. Puede copiar alguna de las que ofrece la solución y agregarlas por sus propios medios.

**Ejercicio 5.40** A veces, dos palabras (o variantes de una palabra) pueden vincularse con la misma respuesta. Trabaje con esta idea vinculando sinónimos o expresiones relacionadas con la misma cadena, de modo que no necesite tener varias entradas en el mapa para la misma respuesta.

**Ejercicio 5.41** Identifique en el ingreso del usuario varias palabras que coincidan con las almacenadas en el mapa y responda con la respuesta que más se ajuste.

**Ejercicio 5.42** Para el caso en que no se reconoció ninguna de las palabras, utilice otras palabras del ingreso del usuario para filtrar mejor la respuesta por defecto: por ejemplo, las palabras «por qué», «cómo», «quien».

## 5.10

## Escribir documentación de clase

Cuando se trabaja sobre proyectos es importante escribir la documentación para sus clases, a medida que se desarrolla el código. Es muy común que los programadores no se tomen el trabajo de documentar seriamente y de manera suficiente sus programas y es más frecuente aún, que más tarde este defecto genere serios problemas.

Si no suministra suficiente documentación será muy difícil que otros programadores logren comprender sus clases (¡O que usted mismo no las comprenda pasado un tiempo!). Típicamente, lo que tiene que hacer en estos casos, es leer la implementación de la clase e imaginar qué hace. Esta manera puede funcionar con proyectos pequeños de estudio, pero crea serios problemas en los proyectos reales.

### Concepto

La **documentación** de una clase debiera ser suficientemente detallada como para que otros programadores puedan usarla sin tener que leer su implementación.

No es poco frecuente que las aplicaciones comerciales contengan cientos de miles de líneas de código agrupadas en varios miles de clases. ¡Imagine si tiene que leer todo este código para comprender cómo funciona una aplicación! Parece que jamás tendría éxito.

Cuando usamos las clases de la biblioteca de Java tales como HashSet o Random, nos hemos apoyado exclusivamente en su documentación, para averiguar cómo usarlas. Nunca hemos mirado la implementación de esas clases. Este camino funcionó porque estas clases están suficientemente bien documentadas (aunque, por cierto, esta documentación podría mejorarse). Nuestra tarea hubiera resultado más complicada si hubiéramos tenido que leer las implementaciones de dichas clases antes de usarlas.

Es típico que en un equipo de desarrollo de software, la implementación de las clases sea compartida entre muchos programadores. Mientras que uno de los programadores es el responsable de implementar la clase SoporteDeSistema de nuestro último ejemplo, otros deben implementar el LectorDeEntrada, de modo que el primer programador tendrá que invocar métodos de las otras clases mientras se dedica a su propia clase.

El mismo argumento que damos para las clases de biblioteca es válido para las clases que escribimos: si podemos usar las clases sin tener que leer y comprender su implementación completa, nuestra tarea se vuelve más fácil. Tal como en las clases de biblio-

teca, queremos ver solamente la interfaz pública de la clase en lugar de su implementación. En consecuencia, es muy importante escribir una buena documentación para nuestras propias clases.

El sistema Java incluye una herramienta denominada javadoc que se puede utilizar para generar la interfaz que describa nuestros archivos fuente. La documentación de la biblioteca estándar que hemos usado, por ejemplo, fue creada a partir de código fuente de sus clases mediante el javadoc.

### 5.10.1 Usar javadoc en BlueJ

El entorno BlueJ utiliza javadoc para posibilitar la creación de la documentación de las clases. La función *Generate Documentation* del menú principal, genera la documentación de todas las clases del proyecto, mientras que la opción *Interface View* del editor muestra un resumen de la documentación de una sola clase. Si le interesa, para encontrar más detalles sobre este tema puede leer el Tutorial de BlueJ al que se accede mediante el menú *Help* de BlueJ.

### 5.10.2 Elementos de la documentación de una clase

La documentación de una clase debe incluir como mínimo:

- el nombre de la clase;
- un comentario que describa el propósito general y las características de la clase;
- un número de versión;
- el nombre del autor (o de los autores);
- la documentación de cada constructor y de cada método.

La documentación de cada constructor y de cada método debe incluir:

- el nombre del método;
- el tipo de retorno;
- los nombres y tipos de los parámetros;
- una descripción del propósito y de la función del método;
- una descripción de cada parámetro;
- una descripción del valor que devuelve.

Además, cada proyecto debiera tener un comentario general, frecuentemente guardado en un archivo de nombre «Leeme» o «ReadMe». En BlueJ, este comentario del proyecto resulta accesible a través del ícono de nota que se muestra en el extremo superior izquierdo del diagrama de clases.

**Ejercicio 5.43** Use la función *Generate Documentation* para generar la documentación de su proyecto Soporte Técnico. Examínela. ¿Es correcta? ¿Es suficiente? ¿Qué partes son útiles y cuáles no? ¿Encuentra errores en la documentación?

Algunos elementos de la documentación tales como los nombres y los parámetros de los métodos pueden extraerse siempre del código. Otras partes, tales como los comentarios

que describen la clase, los métodos y los parámetros, necesitan un poco más de atención ya que pueden ser fácilmente olvidados, estar incompletos o hasta pueden ser incorrectos.

En Java, los comentarios de estilo javadoc se escriben con un símbolo especial de al comienzo:

```
/**  
 * Este es un comentario javadoc  
 */
```

El símbolo de inicio de un comentario debe tener dos asteriscos para que javadoc lo reconozca. Este tipo de comentario, ubicado inmediatamente antes de la declaración de clase es interpretado como un comentario de clase. Si el comentario está ubicado directamente arriba de la signatura de un método, es considerado como un comentario de método.

Los detalles exactos de la manera en que se produce y se da formato a la documentación son diferentes en los distintos lenguajes y entornos de programación, sin embargo, el contenido debiera ser más o menos el mismo.

En Java y mediante javadoc, se dispone de varios símbolos especiales para dar formato a la documentación. Estos símbolos comienzan con el símbolo @ e incluyen:

```
@version  
@autor  
@param  
@return
```

**Ejercicio 5.44** Busque ejemplos de uso de símbolos de javadoc en el código del proyecto de *Soporte Técnico*. ¿Cómo influyen en el formato de la documentación?

**Ejercicio 5.45** Busque y describa otros símbolos de javadoc. Uno de los lugares en que puede buscar es en la documentación en línea de Java distribuido por Sun Microsystems, que contiene un documento denominado *javadoc – The Java Api Documentation Generator* (por ejemplo, en <http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javadoc.html>). En este documento, los símbolos clave se denominan *javadoc tags (etiquetas de javadoc)*.

**Ejercicio 5.46** Documente adecuadamente todas las clases de su versión del proyecto de *Soporte Técnico*.

## 5.11

## Comparar público con privado

Llegó el momento de discutir más detalladamente un aspecto de las clases que ya hemos encontrado numerosas veces pero que aún no hemos tratado lo suficiente: los *modificadores de acceso*.

Los modificadores de acceso son las palabras clave public o private que aparecen al comienzo de las declaraciones de campos y de las signaturas de los métodos. Por ejemplo:

```
// declaración de campo  
private int numeroDeAsientos;  
// métodos  
public void setEdad(int nuevaEdad)
```

```
{
    ...
}
private int calcularPromedio()
{...}
```

Los campos, los constructores y los métodos pueden ser públicos o privados, a pesar de que la mayoría de los campos que hemos visto son privados y la mayoría de los constructores y de los métodos son públicos. Volveremos a ellos a continuación.

### Concepto

Los **modificadores de acceso** definen la visibilidad de un campo, de un constructor o de un método. Los elementos públicos son accesibles dentro de la misma clase o fuera de ella; los elementos privados son accesibles solamente dentro de la misma clase.

Los modificadores de acceso definen la visibilidad de un campo, de un constructor o de un método. Por ejemplo, si un método es público puede ser invocado dentro de la misma clase o desde cualquier otra clase. Por otro lado, los métodos privados solo pueden ser invocados dentro de la clase en que están declarados, no están visibles para las otras clases.

Ahora que ya hemos hablado sobre la diferencia entre interfaz e implementación de una clase (Sección 5.3.1) podemos comprender más fácilmente el propósito de estas palabras clave.

Recuerde: la interfaz de una clase es el conjunto de detalles que necesita ver otro programador que utilice dicha clase. Proporciona información sobre cómo usar la clase. La interfaz incluye las signaturas y los comentarios del constructor y de los métodos. También nos referimos a la interfaz como la *parte pública* de una clase. Su propósito es definir qué es lo que hace la clase.

La implementación es la sección de una clase que define precisamente cómo funciona la clase. Los cuerpos de los métodos que contienen sentencias Java y la mayoría de los campos forman parte de la implementación. También nos referimos a la implementación como la *parte privada* de una clase. El usuario de una clase no necesita conocer su implementación. En realidad, existen buenas razones para *evitar que un usuario conozca* la implementación (o por lo menos, que use ese conocimiento). Este principio se denomina *ocultamiento de la información*.

La palabra clave `public` declara que un elemento de una clase (un campo o un método) forma parte de la interfaz (es decir, es visible públicamente); la palabra clave `private` declara que un elemento es parte de la implementación (es decir, permanece oculto para los accesos externos).

### 5.11.1

### Ocultamiento de la información

### Concepto

El **ocultamiento de la información** es un principio que establece que los detalles internos de implementación de una clase deben permanecer ocultos para las otras clases. Asegura una mejor modularización de la aplicación.

En muchos lenguajes de programación orientados a objetos, el interior de una clase (su implementación) permanece oculta para las otras clases. Hay dos aspectos en este punto: primero, un programador que hace uso de una clase no necesita conocer su interior; segundo, a un usuario *no se le permite conocer* los detalles internos.

El primer principio, *necesidad de conocer*, tiene que ver con la abstracción y la modularización tratada en el Capítulo 3. Si necesitáramos conocer todos los detalles internos de todas las clases que queremos usar, no terminaríamos nunca de implementar sistemas grandes.

El segundo principio, *no se permite conocer*, es diferente. También tiene que ver con la modularización pero en un contexto diferente. El lenguaje de programación no permite el acceso a una sección privada de una clase mediante sentencias en otra clase. Esto asegura que una clase no dependa de cómo está implementada exactamente otra clase.

Este punto es muy importante para el trabajo de mantenimiento. Una tarea muy común de mantenimiento de un programa es la modificación o extensión de la implementación de una clase para mejorarlo o para solucionar defectos. Idealmente, las modificaciones en la implementación de una clase no debieran generar la necesidad de cambiar también las otras clases. Esta característica se conoce como *acoplamiento*: si se cambia una parte de un programa no debiera ser necesario hacer cambios en otras partes del programa, cuestión que se conoce como alto y bajo acoplamiento. El bajo acoplamiento es bueno porque hace que el trabajo de mantenimiento del programador sea mucho más fácil: en lugar de comprender y modificar muchas clases, deberá comprender y modificar sólo una clase. Por ejemplo, si un programador Java hace una mejora de la implementación de la clase `ArrayList`, es esperable que no tengamos la necesidad de modificar nuestro código para usar esta clase y es así porque nuestro código no ha hecho ninguna referencia a la implementación de `ArrayList`.

Por lo que, para ser más precisos, la regla de que a un usuario «no se le debe permitir conocer el interior de una clase» no se refiere al programador de otras clases sino a la clase en sí misma. Generalmente, no es un problema el hecho de que un programador conozca los detalles de implementación, pero una clase no debiera «conocer» (o depender) de los detalles internos de otras clases. El programador de ambas clases podría ser hasta la misma persona pero las clases aún tendrían que permanecer bajamente acopladas.

Las características de acoplamiento y de ocultamiento de la información son muy importantes y las volveremos a tratar en capítulos posteriores.

Por ahora, es importante comprender que la palabra clave `private` refuerza el ocultamiento de la información al impedir el acceso a esta parte de la clase desde otras clases. Esto asegura el bajo acoplamiento y hace que la aplicación resulte más modular y más fácil de mantener.

### 5.11.2 Métodos privados y campos públicos

La mayoría de los métodos que hemos visto hasta ahora fueron públicos y esto asegura que otras clases puedan llamar a estos métodos. Sin embargo, algunas veces hemos usado métodos privados. En la clase `SistemaDeSoporte` del sistema de *Soporte Técnico*, por ejemplo, hemos visto que los métodos `imprimirBienvenida` e `imprimirDespedida` fueron declarados como métodos privados.

La razón de disponer de ambas opciones es que dichos métodos realmente se usan con fines diferentes. Se los utiliza para proveer de operaciones a los usuarios de una clase (métodos públicos) y para dividir una tarea grande en varias tareas más pequeñas y así lograr que la tarea grande sea más fácil de manejar. En el segundo caso, las subtareas no tienen la finalidad de ser invocadas directamente desde el exterior de la clase pero se las ubica como métodos separados con la intencionalidad de lograr que la implementación de una clase sea más fácil de leer. En este caso, tales métodos deben ser privados. Los métodos `imprimirBienvenida` e `imprimirDespedida` son ejemplos de métodos privados con dicha finalidad.

Otra buena razón para tener un método privado es cuando una tarea necesita ser usada (como una subtarea) en varios métodos de una clase. En lugar de escribir el código varias veces, podemos escribirlo una única vez en un solo método privado y luego

invocar este método desde diferentes lugares de la clase. Veremos un ejemplo de este tipo más adelante.

En Java, los campos también pueden ser declarados privados o públicos. Hasta ahora no hemos visto, en los ejemplos, ningún campo que haya sido declarado público y existe una buena justificación. La declaración de los campos como públicos rompe con el principio de ocultamiento de la información. Hace que una clase que depende de esa información sea vulnerable a operaciones incorrectas, si se modifica la implementación. Sin embargo, el lenguaje Java nos permite declarar campos públicos; nosotros consideramos que este es un mal estilo de programación y que no debiéramos hacer uso de esta opción. Algunos otros lenguajes orientados a objetos no admiten campos públicos.

Una razón más para mantener los campos como privados reside en que permiten que un objeto crezca manteniendo el control sobre su estado. Si el acceso a los campos privados se canaliza a través de métodos de acceso y de modificación, entonces un objeto tiene la habilidad de asegurar que el campo nunca se configura con un valor que resulte inconsistente con su estado. Este nivel de integridad no es posible si los campos son públicos.

Abreviando, los campos debieran ser siempre privados.

Java tiene dos niveles más de acceso. Uno se declara mediante la palabra clave `protected` como modificador de acceso y el otro se usa cuando no se declara ningún modificador de acceso. Discutiremos estos puntos más adelante en otros capítulos.

## 5.12

## Aprender sobre las clases a partir de sus interfaces

El proyecto *pelotas* (que está en el CD y en el sitio web) es otro buen proyecto para usar en el estudio de los conceptos tratados en este capítulo. No lo usaremos para introducir ningún concepto nuevo sino para revisar los puntos discutidos anteriormente en un contexto diferente. En consecuencia, esta sección es mayormente una secuencia de ejercicios con algunos comentarios.

El proyecto pelotas tiene tres clases: `PelotasDemo`, `ReboteDePelota` y `Canvas` (Figura 5.4).

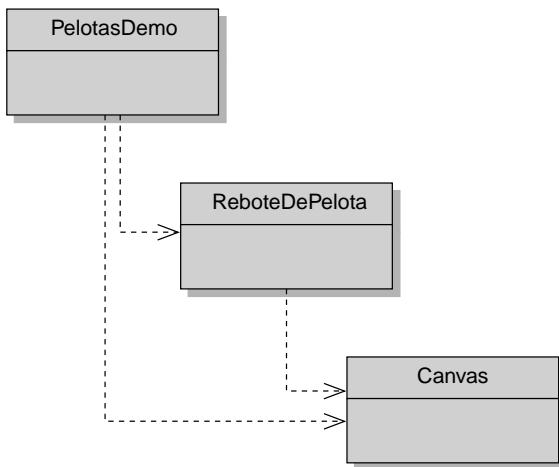
La clase `Canvas` proporciona una ventana en la pantalla que puede usarse para dibujar en ella. Tiene operaciones para dibujar líneas, figuras y texto. Puede usarse un canvas mediante la creación de una instancia y haciéndola visible mediante el método `setVisible`. La clase `Canvas` no requiere ninguna modificación. Lo mejor es, probablemente, tratarla como una clase de biblioteca: abrir el editor y visualizar su interfaz, en donde se muestra la clase a través de la documentación producida por javadoc.

La clase `PelotasDemo` ofrece dos demostraciones cortas que muestran la manera en que se pueden producir salidas gráficas usando el *canvas*. El método `dibujarDemo` es un ejemplo de uso de varias de las operaciones para dibujar y el método `rebotar` muestra una pequeña simulación del rebote de dos pelotas.

La clase `ReboteDePelota` se usa para la demostración de los rebotes e implementa el comportamiento de una pelota que rebota.

**Figura 5.4**

El proyecto  
PelotasDemo



El mejor punto de comienzo para comprender y experimentar con este proyecto es probablemente la clase `PelotasDemo`.

**Ejercicio 5.47** Cree un objeto `PelotasDemo` y ejecute los métodos `dibujarDemo` y `rebotar`. Luego lea el código de `PelotasDemo` y describa detalladamente cómo funcionan estos métodos.

**Ejercicio 5.48** Lea la documentación de la clase `Canvas` y luego responda las siguientes cuestiones por escrito, incluyendo fragmentos de código Java.

¿Cómo crea un `Canvas`? ¿Cómo lo vuelve visible? ¿Cómo dibuja una línea? ¿Cómo puede borrar algo? ¿Cuál es la diferencia entre `dibujar` y `rellenar`? ¿Qué hace el método `espera`?

**Ejercicio 5.49** Experimente las operaciones de la clase `Canvas` realizando algunos cambios en el método `dibujarDemo` de la clase `PelotasDemo`. Dibuje algunas líneas, algunas figuras y algún texto.

**Ejercicio 5.50** Dibuje un marco alrededor del `canvas` dibujando un rectángulo ubicado a 20 píxeles de distancia de los bordes de la ventana. Ponga esta funcionalidad dentro de un método denominado `dibujarMarco` en la clase `PelotasDemo`.

El último ejercicio, dibujar un marco a cierta distancia de los bordes de la ventana, presenta algunas opciones. Primero, podemos resolverlo dibujando cuatro líneas. Alternativamente, podemos dibujar un rectángulo usando el método `dibujar`. La firma de `dibujar` es

```
public void dibujar (Shape figura)
```

El parámetro, especificado como de tipo `Shape`, puede ser un `Rectangle`. En realidad, puede ser cualquier caso especial de figura que esté disponible en la biblioteca Java. Este ejemplo hace uso de la especialización a través de la herencia, una técnica que discutiremos en el Capítulo 8. El método `dibujarDemo` incluye un ejemplo de la

manera en que se puede crear y dibujar un rectángulo. También puede estudiar la interfaz de la clase `Rectangle` en la documentación de la biblioteca de Java.

La segunda cuestión es la forma de determinar el tamaño del rectángulo a dibujar. Por un lado, puede conocer el tamaño del objeto `canvas` en el momento en que se lo crea, que de hecho, es de 600 por 500 píxeles. (Encuentre el lugar del código en el que se especifica este tamaño.) De modo que podemos establecer que necesitamos un rectángulo de 560 píxeles de ancho por 460 píxeles de alto, dibujado a partir de la posición 20,20.

Por otro lado, esta forma no es muy elegante porque no es robusta para las modificaciones. Si más adelante, un programador de mantenimiento decide hacer un canvas de mayor tamaño, el marco resultará incorrecto. El código del método `dibujarMarco` también debe ser cambiado para que funcione como se espera. Sería más elegante usar el método `dibujarMarco` de modo que el marco se adapte automáticamente al tamaño del canvas; así cuando más adelante, el canvas cambie su tamaño, el marco continuará dibujándose correctamente.

Podemos llevar a cabo esta funcionalidad preguntando primeramente al canvas por su tamaño. Al buscar en la interfaz del `Canvas` podemos ver que ofrece un método `getTamaño` que retorna un objeto de tipo `Dimension` (¿de qué se trata?). Necesitamos encontrar información sobre este objeto estudiando la documentación de la biblioteca para esta clase.

**Ejercicio 5.51** Mejore su método `dibujarMarco` de modo que el marco se adapte automáticamente al tamaño del canvas. Para realizarlo, necesita averiguar la manera en que se usa un objeto de clase `Dimension`.

Una vez que haya implementado este ejercicio, puede probarlo manualmente cambiando el tamaño del canvas e invocando nuevamente al método `dibujarMarco`.

A continuación, debemos hacer algo más con el rebote de las pelotas.

**Ejercicio 5.52** Modifique el método `rebotar` para que permita que el usuario seleccione la cantidad de pelotas que estarán rebotando.

Para el último ejercicio, deberá usar una colección para almacenar las pelotas. De esta manera, el método puede tratar con una, tres o 75 pelotas, cualquier número, el que se desee. Las pelotas serán ubicadas inicialmente en una fila en la parte superior del canvas.

¿Qué tipo de colección debería elegir? Hasta ahora hemos visto `ArrayList`, `HashMap` y `HashSet`. Antes de escribir su implementación, intente realizar los siguientes ejercicios.

**Ejercicio 5.53** Entre las colecciones `ArrayList`, `HashMap` y `HashSet`, ¿cuál es la colección más adecuada para almacenar las pelotas en el nuevo método `rebotar`? Justifique por escrito su elección.

**Ejercicio 5.54** Modifique el método `rebotar` para que las pelotas se ubiquen aleatoriamente en cualquier lugar de la mitad superior de la pantalla.

**Ejercicio 5.55** Escriba un nuevo método de nombre `rebotarEnCaja`. Este método dibuja un rectángulo (una caja) en la pantalla y una o más pelotas dentro de la caja. Para las pelotas, no use la clase `ReboteDePelota`, en su

lugar cree una nueva clase `CajaDePelotas` que mueva las pelotas dentro de la caja, rebotando contra las paredes de la caja de modo que siempre permanezcan dentro de ella. La posición inicial y la velocidad de la pelota se determinarán por azar. El método `rebotarEnCaja` debería tener un parámetro que especifique la cantidad de pelotas que habrá dentro de la caja.

**Ejercicio 5.56** Determine aleatoriamente los colores de las pelotas dentro del método `rebotarEnCaja`.

## 5.13

## Variables de clase y constantes

Hasta ahora, no hemos entrado a ver el código de la clase `ReboteDePelota`. Si realmente está interesado en comprender cómo funciona esta animación, puede querer estudiar esta clase. Es una clase razonablemente simple, el único método que resulta un poco más difícil de comprender es el método `mover`, en el que las pelotas cambian su posición a lo largo de su trayectoria.

Dejamos en manos del lector la mayor parte del estudio de este método, excepto un detalle que queremos tratar ahora. Comenzamos con un ejercicio.

**Ejercicio 5.57** En la clase `ReboteDePelota` encontrará una definición de la gravedad (un solo número entero). Aumente o disminuya el valor de la gravedad, compile y ejecute nuevamente el rebote de las pelotas a través de la clase `PelotasDemo`. ¿Observa algún cambio?

El detalle más interesante en esta clase aparece en la línea

```
private static final int GRAVEDAD = 3;
```

Esta es una construcción que no habíamos visto nunca hasta ahora. En realidad, esta línea introduce dos nuevas palabras clave que aparecen juntas: `static` y `final`.

### 5.13.1

#### La palabra clave `static`

##### Concepto

Las clases pueden tener campos: estos campos se conocen como **variables de clase** o variables estáticas. En todo momento, existe exactamente una copia de una variable de clase, independientemente del número de instancias que se hayan creado.

La palabra clave `static` está en la sintaxis de Java para definir variables de clase. Las variables de clase son campos que se almacenan en la misma clase y no en un objeto. Este hecho produce diferencias fundamentales con respecto a las variables de instancia (los campos que hemos tratado hasta ahora). Considere este segmento de código (una parte de la clase `ReboteDePelota`).

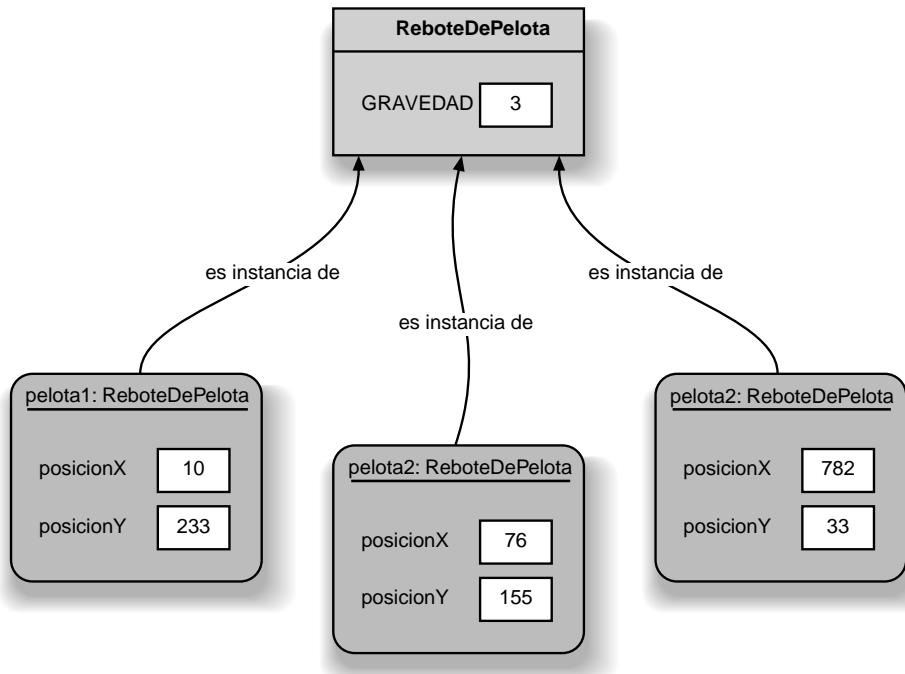
```
public class ReboteDePelota
{
    // Efecto de gravedad
    private static final int GRAVEDAD = 3;

    private int posicionX;
    private int posicionY;
    Se omiten otros campos y métodos
}
```

Ahora, imagine que se crean tres instancias de la clase `ReboteDePelota`. La situación resultante se muestra en la Figura 5.5.

**Figura 5.5**

Variables de instancia  
y una variable de  
clase



Como podemos ver en el diagrama, las variables de instancia (`posicionX` y `posicionY`) se almacenan en cada objeto. Dado que hemos creado tres objetos, tenemos tres copias independientes de estas variables.

Por otro lado, la variable de clase `GRAVEDAD` se almacena en la clase propiamente dicha; en consecuencia, existe siempre sólo una copia de esta variable, independientemente del número de instancias creadas.

El código de la clase puede acceder (leer y asignar) a esta clase de variable de la misma forma en que accede a las variables de instancia. Se puede acceder a la variable de clase desde cualquiera de las instancias de la clase; como resultado, los objetos comparten esta variable.

Las variables de clase se usan frecuentemente en los casos en que un valor debe ser siempre el mismo para todas las instancias de una clase. En lugar de almacenarse una copia con el mismo valor en cada objeto, que sería un desperdicio de espacio y puede ser más difícil de coordinar, puede compartirse un único valor entre todas las instancias.

Java también soporta *métodos de clase* (también conocidos como *métodos estáticos*) que son métodos que pertenecen a una clase. Hablaremos de ellos más adelante.

### 5.13.2 Constantes

Un uso frecuente de la palabra clave `static` es la declaración de *constants*. Las constantes son similares a las variables pero no pueden cambiar su valor durante la ejecución de una aplicación. En Java, las constantes se definen con la palabra clave `final`. Por ejemplo:

```
private final int TOPE = 10;
```

En esta sentencia definimos una constante de nombre TOPE con el valor 10. Observamos que las declaraciones de constantes son similares a las declaraciones de campos pero con dos diferencias:

- deben incluir la palabra clave `final` antes del nombre del tipo y
- deben ser inicializadas con un valor en el momento de su declaración.

Si no se intentara modificar un valor en tiempo de ejecución, es una buena idea declararlo como final. De esta manera se asegura que, más adelante, no cambie accidentalmente su valor. Cualquier intento de cambiar un campo constante dará por resultado un mensaje de error en tiempo de compilación. Por convención, las constantes se escriben frecuentemente con letras mayúsculas; nosotros seguimos esta convención en este libro.

En la práctica, es muy frecuente el caso en que las constantes se relacionen con todas las instancias de una clase. En esta situación declaramos *constantes de clase*. Las constantes de clase son campos de clase constantes. Se declaran usando una combinación de las palabras clave `static` y `final`. Por ejemplo:

```
private static final int TOPE = 10;
```

La definición de GRAVEDAD en nuestro proyecto del rebote de una pelota es otro ejemplo de una constante de clase. Este es el estilo en el que se definen las constantes en la mayoría de los casos; las constantes específicas de una instancia se usan con mucha menos frecuencia.

**Ejercicio 5.58** Escriba declaraciones de constantes para los siguientes casos:

- una variable pública que se usa para medir la tolerancia, con el valor 0.001.
- una variable privada que se usa para indicar una marca, con el valor entero 40.
- una variable pública de tipo carácter que se usa para indicar que se accede al comando de ayuda mediante la letra «a».

**Ejercicio 5.59** Lea el código de la clase EntradaDeLog del proyecto *analizador-weblog* en el Capítulo 4. ¿Cómo se utilizan las constantes en esa clase? ¿Considera que es un buen uso de las constantes?

**Ejercicio 5.60** Suponga que una modificación al proyecto *analizador-weblog* implica que no se necesitan almacenar más los valores de los años en el arreglo `valoresDeDatos` de la clase `EntradaDeLog`. ¿Cuántas clases sería necesario modificar si ahora, el valor del mes se almacena en la posición de índice 0, el valor del día en la posición de índice 1, etc.? ¿Observa el modo en que el uso de constantes para valores especiales simplifica este tipo de proceso?

## 5.14

### Resumen

Es esencial, para un programador competente, trabajar con bibliotecas de clases y con interfaces de clases. En este tópico hay dos aspectos: leer las descripciones de la biblioteca de clase (especialmente de las interfaces de clase) y escribirlas.

Es importante conocer algunas clases esenciales de la biblioteca estándar de Java y ser capaz de encontrar más, cuando se necesiten. En este capítulo hemos presentado algunas de las clases más importantes y hemos discutido la manera de navegar en la documentación de la biblioteca.

También es importante ser capaz de documentar cualquier clase que se escribe en el mismo estilo que las clases de biblioteca, de modo que otros programadores puedan usar estas clases fácilmente sin tener que comprender su implementación. Esta documentación debiera incluir buenos comentarios sobre cada proyecto, cada clase y cada método. El uso de javadoc en programas Java ayudará a crear esta documentación.

Términos introducidos en este capítulo

**intertaz, implementación, mapa, conjunto, javadoc, modificador de acceso, ocultamiento de información, acoplamiento, variable de clase, estático, constante, final**

## Resumen de conceptos

- **biblioteca de Java** La biblioteca de clases estándar de Java contiene muchas clases que son muy útiles. Es importante saber cómo usar la biblioteca.
- **documentación de la biblioteca** La documentación de la biblioteca estándar de Java muestra detalles sobre todas las clases de la biblioteca. El uso de esta documentación es esencial para hacer un buen uso de las clases de la biblioteca.
- **interfaz** La interfaz de una clase describe lo que hace la clase y cómo puede usarse sin mostrar su implementación.
- **implementación** El código completo que define una clase se denomina implementación de dicha clase.
- **inmutable** Se dice que un objeto es inmutable si su contenido o su estado no puede ser modificado una vez que fue creado. Los Strings son ejemplos de objetos inmutables.
- **mapa** Un mapa es una colección que almacena entradas de pares de valores llave/valor. Los valores pueden ser buscados mediante el suministro de una llave.
- **conjunto** Un conjunto es una colección que almacena cada elemento una única vez. No mantiene ningún orden específico.
- **documentación** La documentación de una clase debe ser suficientemente detallada como para que otros programadores puedan usar la clase sin necesidad de leer su implementación.
- **modificadores de acceso** Los modificadores de acceso definen la visibilidad de un campo, un constructor o un método. Los elementos públicos son accesibles dentro de la misma clase y desde otras clases; los elementos privados son accesibles solamente dentro de la misma clase a la que pertenecen.
- **ocultamiento de la información** El ocultamiento de la información es un principio que establece que los detalles internos de la implementación de una clase deben permanecer ocultos para las otras clases. Asegura la mejor modularización de una aplicación.
- **variables de clase, variables estáticas** Las clases pueden tener campos que se conocen como variables de clase o variables estáticas. En todo momento, existe una única copia de una variable de clase, independientemente del número de instancias que se hayan creado.

**Ejercicio 5.61** Hay un rumor circulando por Internet de que George Lucas (el creador de las películas *Viaje a las Estrellas*) usa una fórmula para crear los nombres de los personajes de sus historias (Jar Jar Binks, ObiWan Kenobi, etc.). La fórmula, aparentemente, es la siguiente:

Para el primer nombre del personaje de *Viaje a las Estrellas*:

1. Tome las tres primeras letras de su apellido.
2. Agregue a la sílaba anterior, las dos primeras letras de su primer nombre.

Para el apellido del personaje:

1. Tome las dos primeras letras del apellido de soltera de su madre.
2. Agregue a la sílaba anterior, las tres primeras letras del nombre de la ciudad o del pueblo en que nació.

Y ahora su tarea: cree un proyecto en BlueJ de nombre *viaje-estrellas*. Cree en él una clase de nombre **GeneradorDeNombre**. Esta clase debe tener un método de nombre **generarNombreDePersonaje** que genera un nombre completo para un personaje de la película siguiendo el método descrito anteriormente. Deberá buscar un método de la clase **String** que permita generar subcadenas.

**Ejercicio 5.62** El siguiente fragmento de código intenta imprimir una cadena en letras mayúsculas:

```
public void imprimirMayusculas (String s)
{
    s.toUpperCase();
    System.out.println(s);
}
```

Sin embargo, este código no funciona. Encuentre el motivo por el que no funciona y explíquelo. ¿Cómo debería escribirse para que funcione de manera adecuada?

**Ejercicio 5.63** Asuma que queremos intercambiar el valor de dos variables enteras **a** y **b**. Para llevar a cabo esta tarea escribimos un método

```
public void intercambiar (int i1, int i2)
{
    int temp = i1;
    i1 = i2;
    i2 = temp;
}
```

Luego invocamos este método sobre nuestras variables **a** y **b**:

```
intercambiar (a, b);
```

¿Se intercambian realmente **a** y **b** después de la invocación? Si prueba este código notará que ¡no funciona! ¿Por qué no funciona? Explíquelo detalladamente.

## CAPÍTULO

# 6

## Objetos con buen comportamiento

Principales conceptos que se abordan en este capítulo

- prueba
- prueba de unidad
- depuración
- prueba automatizada

Construcciones Java que se abordan en este capítulo

(En este capítulo no se introduce ninguna construcción nueva de Java.)

### 6.1

## Introducción

Al llegar a este lugar del libro, si ya leyó los capítulos anteriores y realizó los ejercicios que hemos sugerido, seguramente ya escribió un buen número de clases y habrá notado que la clase que escribe raramente es perfecta después del primer intento de escritura del código. Suele ocurrir que la clase no funciona correctamente desde el principio y que es necesario trabajar un poco más para completarla.

Los problemas que se presentan al escribir un programa cambiarán con el tiempo. Los principiantes, generalmente, se topan con errores de sintaxis de Java. Los errores de sintaxis son errores en la estructura del código propiamente dicho; son fáciles de solucionar porque el compilador los señala y muestra algún mensaje de error.

Los programadores más experimentados, que se enfrentan con problemas más complicados, generalmente tienen menos dificultad con la sintaxis del lenguaje y se concentran más en los errores de lógica.

Un error de lógica ocurre cuando el programa compila y se ejecuta sin errores obvios pero da resultados incorrectos. Los problemas de lógica son mucho más severos y difíciles de encontrar que los errores de sintaxis.

La escritura de programas sintácticamente correctos es relativamente fácil de aprender y existen buenas herramientas, como los compiladores, para detectar errores de sintaxis y corregirlos. Por otro lado, la escritura de programas lógicamente correctos es muy difícil para cualquier problema que no sea trivial y la prueba de que un programa es correcto, en general, no puede ser automática; en realidad, es tan difícil que

es bien conocido el hecho de que la mayoría del software que se vende comercialmente contiene un número significativo de fallos.

En consecuencia, es esencial que un ingeniero de software competente aprenda la forma de manejar la exactitud y los caminos para reducir el número de errores en una clase.

En este capítulo discutiremos varias actividades que están relacionadas con mejorar la exactitud de un programa que incluyen la prueba, la depuración y la escritura de código con fines de mantenimiento.

#### Concepto

La **prueba** es la actividad cuyo objetivo es determinar si una pieza de código (un método, una clase o un programa) produce el comportamiento pretendido.

La *prueba* es una actividad dedicada a determinar si un segmento de código contiene errores. No es fácil construir una buena prueba, hay mucho para pensar cuando se prueba un programa.

La *depuración* viene a continuación de la prueba. Si las pruebas demostraron que se presentó un error, usamos técnicas de depuración para encontrar exactamente dónde está ese error y corregirlo. Puede haber una cantidad significativa de trabajo entre saber que existe un error y encontrar su causa y solucionarlo.

Probablemente, el punto más fundamental se centra en escribir código con fines de mantenimiento. Se trata de escribir código de tal manera que, en primer término, se eviten los errores, y si aun así aparecen, puedan ser encontrados lo más fácilmente posible. Esto está fuertemente relacionado con el estilo de código y los comentarios. Idealmente, el código debería ser fácil de comprender de modo que el programador original evite introducir errores y un programador de mantenimiento pueda encontrar fácilmente los posibles errores.

En la práctica, no siempre es tan simple, pero hay grandes diferencias entre el número de errores y el esfuerzo que lleva depurar código bien escrito y código no tan bien escrito.

## 6.2

## Prueba y depuración

La prueba y la depuración son habilidades cruciales en el desarrollo de software. Frequentemente necesitará controlar sus programas para ver si tienen errores y luego, cuando ocurran, localizarlos en el código. Además, también puede llegar a tener la responsabilidad de probar programas escritos por otras personas o bien, modificarlos. En el último caso, la tarea de depuración está más relacionada con el proceso de comprender un poco más el código, pero existen una cantidad de técnicas que se podrían usar para ambas tareas. En las secciones que siguen analizaremos las siguientes técnicas de prueba y depuración:

- pruebas de unidad en BlueJ
- pruebas automatizadas
- seguimiento manual
- sentencias de impresión
- depuradores

Veremos las primeras dos técnicas en el contexto de algunas clases que usted mismo podría haber escrito, y las restantes técnicas de depuración en el contexto de comprender código escrito por otra persona.

## 6.3

## Pruebas de unidad en BlueJ

El término *prueba de unidad* se refiere a la prueba de partes individuales de una aplicación en contraposición con el término *prueba de aplicación* que es la prueba de una aplicación en su totalidad. Las unidades que se prueban pueden ser de tamaños diversos: puede ser un grupo de clases, una sola clase o simplemente un método. Debemos observar que la prueba de unidad puede escribirse mucho antes de que una aplicación esté completa. Puede probarse cualquier método, una vez que esté escrito y compilado.

Dado que BlueJ nos permite interactuar directamente con objetos individuales, ofrece caminos únicos para conducir las pruebas sobre clases y métodos. Uno de los puntos que queremos enfatizar en esta sección es que nunca es demasiado pronto para comenzar la prueba. La experimentación y prueba temprana conlleva varios beneficios. En primer lugar, nos dan una experiencia valiosa con un sistema que hace posible localizar problemas tempranamente para corregirlos, a un costo mucho menor que si se hubieran encontrado en una etapa más avanzada del desarrollo. En segundo término, podemos comenzar por construir una serie de casos de prueba y resultados que pueden usarse una y otra vez a medida que el sistema crece. Cada vez que hacemos un cambio en un sistema, estas pruebas nos permiten controlar que no hayamos introducido errores inadvertidamente en el resto del sistema como resultado de las modificaciones. Para ilustrar esta forma de prueba en BlueJ usaremos el proyecto *agenda-diaria-prototipo* que representa un estado muy incipiente en el desarrollo de un software para implementar un calendario electrónico de escritorio. Una vez que se haya completado este software, este sistema pretende permitir que un individuo registre sus compromisos diariamente en el curso de un año.

Abra el proyecto *agenda-diaria-prototipo*. Ya se han desarrollado tres clases: Cita, Dia y Semana. Como estas clases serán fundamentales para el sistema completo, deseamos probarlas para controlar si funcionan como deben y para analizar si estamos conformes con algunas de las decisiones que hemos tomado en su diseño e implementación. La clase Cita describe objetos pasivos cuyo propósito es registrar el motivo de la cita y su duración con un número entero de horas. Para nuestra discusión sobre las pruebas nos concentraremos en la clase Dia, que se muestra en el Código 6.1. Un objeto de esta clase mantiene el rastro del conjunto de citas que se anotaron en un solo día. Cada día registra su posición única dentro del año, un valor en el rango 1-366. Esta versión contiene las dos simplificaciones siguientes: las citas siempre se realizan entre horas límite y ocupan un número entero de horas; de modo que las citas pueden registrarse a las 9 a.m., a las 10 a.m., etc., hasta las 5 p.m. (o 17 horas en un reloj de 24 horas).

Como parte de esta prueba, hay varios aspectos de la clase Dia que queremos controlar:

- El campo `citas`, ¿tiene suficiente espacio como para contener el número de citas que se requiere?
- El método `mostrarCitas`, ¿imprime correctamente la lista de citas que se anotaron?
- El método `anotarCita`, ¿actualiza correctamente el campo `citas` cuando se anota una nueva cita?
- El método `encontrarEspacio`, ¿devuelve el resultado correcto cuando se le solicita encontrar lugar para una nueva cita?

Encontraremos que todos estos puntos pueden probarse convenientemente usando el banco de objetos de BlueJ. Además, veremos que la naturaleza interactiva de BlueJ posibilita la simplificación de algunas de las pruebas mediante alteraciones controladas de una clase sometida a prueba.

**Código 6.1**

La clase Dia

```
/*
 * Mantiene las citas de un día completo de un
 * calendario.
 *
 * @author David J. Barnes y Michael Kölling
 * @version 2006.03.30
 */
public class Dia
{
    // La primera y última hora del día en que es
    // posible crear una cita.
    public static final int PRIMER_HORA = 9;
    public static final int ULTIMA_HORA = 17;
    // El número de horas posible de un día.
    public static final int MAX_CITAS_POR_DIA =
        ULTIMA_HORA
    - PRIMER_HORA + 1;
    // Un número de día en un año en particular. (1-366)
    private int diaNumero;
    // La lista actual de citas de un dia.
    private Cita[] citas;
    /**
     * Constructor de objetos de clase Dia
     * @param diaNumero El número de este día en el
     año (1-366).
     */
    public Dia(int diaNumero)
    {
        this.diaNumero = diaNumero;
        citas = new Cita[MAX_CITAS_POR_DIA];
    }
    /**
     * Trata de buscar lugar para una nueva cita.
     * @param cita La nueva cita que se ubicará.
     * @return La hora más temprana en que se puede ubicar
     *         la cita. Devuelve -1 si el espacio
     es insuficiente.
     */
    public int buscarEspacio(Cita cita)
    {
        int duracion = cita.getDuracion();
        for(int fila = 0; fila < MAX_CITAS_POR_DIA;
fila++) {
```

**Código 6.1  
(continuación)**

La clase Dia

```
        if(citas[fila] == null) {
            final int hora = PRIMER_HORA + fila;
            // Potencial punto de inicio.
            if(duracion == 1) {
                // Se necesita una sola fila.
                return hora;
            }
            else {
                // ¿Cuántas filas se
                necesitan?
                int cantidad_filas_requeridas =
                duracion - 1;
                for(int filaSiguiente = fila +
                1;

                cantidad_filas_requeridas > 0 &&
                citas[filaSiguiente] == null;

                filaSiguiente++) {
                    cantidad_filas_requeridas--;
                }
                if(cantidad_filas_requeridas ==
                0) {
                    // Se encontró espacio
                    suficiente.
                    return hora;
                }
            }
        }
        // No se dispone de espacio suficiente.
        return -1;
    }
    /**
     * Anota una cita.
     * @param hora La hora en que comienza la cita.
     * @param cita La cita que se hará.
     * @return true si la cita fue exitosa, false en
     caso contrario
     */
    public boolean anotarCita(int hora, Cita cita)
    {
        if(horaValida(hora)) {
            int horaInicio = hora - PRIMER_HORA;
            if(citas[horaInicio] == null) {
                int duracion = cita.getDuracion();
                // Completa todas las filas hasta
                cubrir la
```

**Código 6.1  
(continuación)**

La clase Dia

```
// duración de la cita
for(int i = 0; i < duracion; i++)
{
    citas[horaInicio + i] = cita;
}
return true;
}
else {
    return false;
}
}
else {
    return false;
}
}

/**
 * @param hora A qué hora del día. Debe ser una
hora comprendida
 * entre la PRIMER_HORA y la
ULTIMA_HORA.
 * @return La Cita a la hora dada. Devuelve null
si la hora
 * no es válida o si no hay ninguna
Cita en la hora dada.
 */
public Cita getCita(int hora)
{
    if(horaValida(hora)) {
        return citas[hora - PRIMER_HORA];
    }
    else {
        return null;
    }
}
/**
 * Imprime la lista de las citas del día.
 */
public void mostrarCitas()
{
    System.out.println("==> Dia " + diaNumero + "
===");
    int hora = PRIMER_HORA;
    for(Cita cita : citas) {
        System.out.print(hora + ": ");
        if(cita != null) {

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

**Código 6.1  
(continuación)**

La clase Dia

```

        else {
            System.out.println();
        }
    hora++;
}
/**
 * @return El número de este día en el año (1 -
366).
 */
public int getDiaNumero()
{
    return diaNumero;
}

/**
 * @return true si la hora está comprendida entre
PRIMER_HORA y
*          ULTIMA_HORA, false en caso contrario.
*/
public boolean horaValida(int hora)
{
    return hora >= PRIMER_HORA && hora <=
ULTIMA_HORA;
}
}

```

### 6.3.1 Usar inspectores

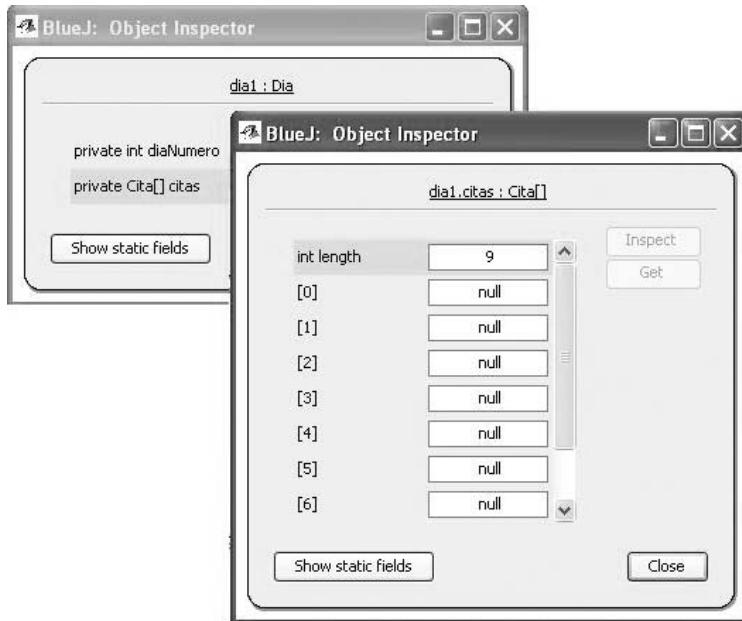
Para preparar la prueba, cree un objeto `Dia` en el banco de objetos y abra su inspector seleccionando la función *Inspect* del menú contextual del objeto. Seleccione el campo `citas` y abra el inspector del arreglo (Figura 6.1). Verifique si el arreglo dispone de espacio suficiente como para contener las citas de un día completo. Deje abierto el inspector del arreglo para asistirlo en las pruebas subsiguientes.

Un componente esencial de la prueba de clases que usan estructuras de datos, es controlar que se comporten adecuadamente tanto cuando las estructuras están vacías como cuando están llenas. Por lo tanto, la primera prueba que se puede llevar a cabo sobre la clase `Dia` es invocar su método `mostrarCitas` antes de que se anoten citas en este día. Este proceso mostrará la lista de cada período del día en el que se pueden anotar citas. Más tarde, controlaremos que este método también funcione correctamente cuando se complete la lista de citas.

Una característica clave de una buena prueba consiste en asegurarse de controlar los *límites* dado que son, con gran frecuencia, los lugares en los que las cosas funcionan mal. Los límites asociados con la clase `Dia` son el inicio y el final del día. De modo que, así como verificamos si podemos anotar citas en el medio del día, será importante controlar si las podemos anotar correctamente tanto en la primera como en la última posición del arreglo `citas`. En vías de conducir las pruebas a través de este

camino cree tres objetos `Cita` en el banco de objetos, cada uno de una hora de duración y luego, trate de hacer los siguientes ejercicios como una prueba inicial del método `anotarCita`.

**Figura 6.1**  
Inspector del arreglo  
`citas`



**Ejercicio 6.1** Use los tres objetos `Cita` para registrar citas a las 9 a.m., a la 1 p.m. y a las 5 p.m., respectivamente. Cuando una cita se anota exitosamente, el método `anotarCita` devuelve el valor `true`. Use el inspector del arreglo para confirmar que cada cita está en la ubicación correcta después de ser registrada.

**Ejercicio 6.2** Invoca al método `mostrarCitas` para confirmar que imprime correctamente la información que mostró el inspector del arreglo.

**Ejercicio 6.3** Ahora controle que no puedan registrarse dos citas en la misma hora. Pruebe anotar una nueva cita a la misma hora en que ya se anotó otra existente. El método deberá retornar el valor `false`, pero también use el inspector del arreglo para confirmar que la nueva cita no haya reemplazado a la cita original.

**Ejercicio 6.4** Una buena prueba para el control de los límites consiste en controlar los valores que están más próximos a los extremos del rango válido de datos, pero fuera de ellos. Controle que el comportamiento es correcto cuando se trata de anotar una cita a las 8 a.m. o a las 6 p.m.

**Ejercicio 6.5** Cree algunas citas más de una hora de duración y complete todas las horas de un solo objeto `Dia` para asegurarse de que esto es posible. Controle que la salida que produce el método `mostrarCitas` resulte correcta cuando el arreglo esté completo.

**Ejercicio 6.6** Controle que no es posible agregar una cita más en un día que ya está completo. ¿Necesita controlar la doble anotación en cada hora del día o alcanza, para estar seguro, con controlar algunas de las posibilidades? Si piensa que es suficiente controlar sólo algunas, ¿qué horas controlaría? *Pista:* el principio de prestar atención especial a los límites ¿es relevante en esta situación? ¿Será suficiente controlar los límites?

**Ejercicio 6.7** ¿Es posible reutilizar un único objeto `Cita` en diferentes horas de un solo día? De ser así, estas pruebas ¿tienen la misma legitimidad que si se usan diferentes objetos? ¿Puede predecir las circunstancias en las que podría querer usar un solo objeto `Cita` en varios lugares de un calendario como un todo?

**Ejercicio 6.8** *Desafío.* Trate de repetir algunas de las pruebas anteriores sobre un nuevo objeto `Dia` usando algunas citas de dos horas de duración. Puede encontrar que estas pruebas modificadas disparan uno o más errores. Trate de corregir estos errores de modo que se puedan registrar correctamente citas de dos horas de duración. Las modificaciones que se realizan en la clase `Dia`, ¿son suficientemente seguras como para asumir que todas las pruebas llevadas a cabo con citas de una hora de duración seguirán funcionando como antes? En la Sección 6.4 trataremos algunas de las características de las pruebas que se realizan cuando se corrige o mejora el software.

A partir de estos ejercicios resulta fácil ver lo valiosos que son los inspectores para dar respuestas inmediatas sobre el estado de un objeto, evitando frecuentemente, la necesidad de agregar sentencias de impresión a una clase cuando se la está probando o depurando.

### 6.3.2

### Pruebas positivas y pruebas negativas

#### Concepto

Una prueba positiva es la prueba de aquellos casos que esperamos que resulten exitosos.

#### Concepto

Una prueba negativa es la prueba de aquellos casos que esperamos que fallen.

En una aplicación, cuando tenemos que decidir qué parte probar, generalmente distinguimos los casos de *pruebas positivas* de los casos de *pruebas negativas*. Una prueba positiva es la prueba de la funcionalidad que esperamos que realmente funcione. Por ejemplo, anotar una cita de una hora de duración en el medio de un día que aún está vacío es una prueba positiva. Cuando probamos con casos positivos nos tenemos que convencer de que el código realmente funciona como esperábamos.

Una *prueba negativa* es la prueba de aquellos casos que esperamos que fallen. Anotar dos citas en una misma hora o registrar una cita fuera de los límites válidos del día son ambos ejemplos de pruebas negativas. Cuando probamos con casos negativos esperamos que el programa maneje este error de cierta manera especificada y controlada.

**Cuidado:** es un error muy común en probadores inexpertos, llevar a cabo sólo pruebas positivas. Las pruebas negativas, es decir, probar que aquello que podría andar mal realmente anda mal y lo hace de una manera bien definida, es crucial para un buen procedimiento de prueba.

**Ejercicio 6.9** ¿Cuáles de las pruebas mencionadas en los ejercicios anteriores son positivas y cuáles negativas? Haga una tabla de cada categoría. ¿Se le ocurren otras pruebas positivas? ¿Y otras negativas?

## 6.4

# Pruebas automatizadas

Una razón por la que se suelen abandonar las pruebas completas de una aplicación es porque insumen mucho tiempo y además, es una actividad relativamente aburrida cuando se la realiza a mano. Esta es una característica que se presenta cuando las pruebas no se deben realizar una sola vez, sino posiblemente varios cientos o miles de veces. Afortunadamente, existen técnicas disponibles que nos permiten automatizar las pruebas repetitivas y así eliminar el trabajo pesado asociado que traen aparejadas. La siguiente sección presenta la automatización de las pruebas en el contexto de una *prueba de regresión*.

### 6.4.1 Prueba de regresión

Sería bueno si pudiéramos asumir que sólo el hecho de corregir los errores mejora la calidad de un programa. Lamentablemente, la experiencia muestra que es demasiado fácil introducir más errores al modificar un software. Cuando se soluciona un error en un lugar determinado se puede, al mismo tiempo, introducir un nuevo error.

Como consecuencia, es deseable ejecutar *pruebas de regresión* cada vez que se realiza una modificación en el software. Las pruebas de regresión consisten en ejecutar nuevamente las pruebas pasadas previamente para asegurarse de que la nueva versión aún las pasa. Probablemente, estas pruebas son mucho más realizables cuando se las puede automatizar de alguna manera. Una de las formas más fáciles de automatizar las pruebas de regresión es escribir un programa que actúa como un *equipo de pruebas* o una *batería de pruebas*. El proyecto *agenda-diaria-prueba* proporciona una ilustración de la manera en que podemos comenzar a construir un equipo de prueba para aquellas pruebas que hemos ejecutado anteriormente sobre el proyecto *agenda-diaria-prototipo*. El Código 6.2 muestra la clase *PruebaUnaHora* de dicho proyecto.

#### Código 6.2

Un equipo básico de prueba para probar las citas de una hora de duración

```
/*
 * Lleva a cabo pruebas de la clase Dia que consisten
 * en anotar citas de una hora de duración.
 *
 * @author David J. Barnes y Michael Kölling
 * @version 2006.03.30
 */
public class PruebaUnaHora
{
    // El objeto Dia que será probado.
    private Dia dia;
    /**
     * Constructor de objetos de la clase PruebaUnaHora
     */
    public PruebaUnaHora()
    {
    }
    /**
     * Prueba la funcionalidad básica tanto al

```

**Código 6.2****(continuación)**

Un equipo básico de prueba para probar las citas de una hora de duración

```
* comienzo, al final como en la mitad del día.  
*/  
public void anotarTresCitas()  
{  
    // Comienza con un objeto Dia nuevo.  
    dia = new Dia(1);  
    // Crea tres citas de una hora de duración.  
    Cita primera = new Cita("Conferencia de Java", 1);  
    Cita segunda = new Cita("Clase de Java", 1);  
    Cita tercera = new Cita("Ver a John", 1);  
  
    // Registrar cada cita en una hora diferente.  
    dia.anotarCita(9, primera);  
    dia.anotarCita(13, segunda);  
    dia.anotarCita(17, tercera);  
  
    dia.mostrarCitas();  
}  
  
/**  
 * Verifica que no esté permitido registrar dos  
citas en una misma hora.  
 */  
public void probarDobleCita()  
{  
    // Inicializa el día con tres citas legítimas.  
    anotarTresCitas();  
    Cita citaMala = new Cita("Error", 1);  
    dia.anotarCita(9, citaMala);  
  
    // Muestra que la citaMala no quedó registrada.  
    dia.mostrarCitas();  
}  
/**  
 * Prueba la funcionalidad básica completando un día  
 * con citas.  
 */  
public void completarElDia()  
{  
    // Comienza con un objeto Dia nuevo.  
    dia = new Dia(1);  
    for(int hora = Dia.PRIMER_HORA; hora <=  
        Dia.ULTIMA_HORA; hora++) {  
        dia.anotarCita(hora, new Cita("Prueba " +  
            hora, 1));  
    }  
  
    dia.mostrarCitas();  
}
```

Cada método de la clase que conforma el equipo de pruebas ha sido escrito para representar una única prueba, es decir, para capturar los pasos que hemos realizado al ejecutar las pruebas manualmente en la Sección 6.3.1. Por lo que el método `anotarTresCitas` está destinado a probar que es posible registrar tres citas legítimas en un objeto `Dia` nuevo y el método `completarElDia` prueba que es posible registrar una cita en cada hora de un día completo. Ambos métodos crean un nuevo objeto `Dia` para asegurar que las pruebas comienzan a partir de su estado inicial. Por otro lado, el método `probarDobleCita` usa el objeto `Dia` creado por el método `anotarTresCitas` porque necesita un objeto en el que ya existan algunas citas anotadas.

Una clase como `PruebaUnaHora` facilita la implementación de pruebas de regresión sobre la clase `Dia`: simplemente tenemos que crear una sola instancia, ejecutar cada uno de sus métodos y verificar los resultados.

**Ejercicio 6.10** Agregue otros métodos, los que le parezcan adecuados, a la clase `PruebaUnaHora` para probar el registro de citas de una hora de duración. Luego ejecute las pruebas de regresión sobre su versión corregida de la clase `Dia`.

**Ejercicio 6.11** Cree una clase `PruebaDosHoras` para construir un conjunto de pruebas para registrar citas de dos horas de duración.

**Ejercicio 6.12** Cree otras clases que le parezcan convenientes, para probar la restante funcionalidad de la clase `Dia`.

**Ejercicio 6.13** En un proyecto complejo, podría ser necesario ejecutar varios cientos o miles de pruebas de regresión para actividades de mantenimiento o de mejoras. ¿Cuán fácil le parece que podría ser controlar los resultados de esas pruebas usando las técnicas que hemos delineado en esta sección? ¿Todavía existe algún elemento manual para los procesos de pruebas de regresión?

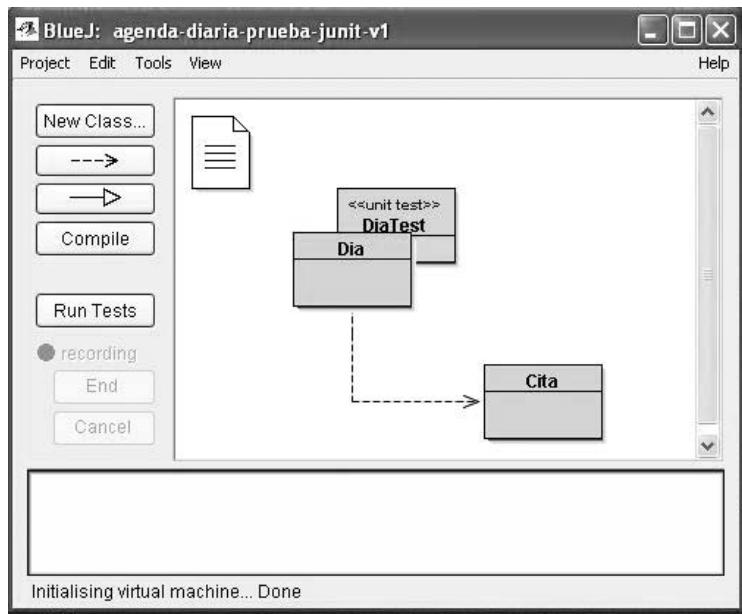
#### 6.4.2 Control automático de los resultados de las pruebas

Las técnicas descritas en la Sección 6.4.1 están, de alguna manera, encaminadas hacia la automatización del proceso de prueba, pero todavía requieren una importante cantidad de intervención humana. Por ejemplo, las listas de citas que se imprimen deben controlarse a mano, por lo que requieren que el controlador conozca cuáles debieran ser los resultados. Las pruebas de regresión automáticas podrían ser más efectivas si pudiéramos construir pruebas que se autocontrolen y que requieran la intervención humana sólo cuando el resultado de una o más de ellas indiquen un posible problema. El proyecto `agenda-diaria-prueba-junit-v1` representa un paso significativo en esta dirección. La Figura 6.2 muestra el diagrama de clases de este proyecto.

Lo primero que resalta en esta figura es que el diagrama incluye un estilo diferente de clase, `DiaTest`, ubicada inmediatamente detrás de la clase `Dia`. La clase `DiaTest` es una clase de prueba y es clasificada por BlueJ como una prueba de unidad; en el ícono de la clase aparece explícitamente el texto `<<unit test>>` y su color es diferente al de las clases ordinarias del diagrama. Lo segundo que se observa son los elementos adicionales que aparecen ubicados debajo del botón `Compile`. Lea el párrafo que se presenta debajo de la siguiente figura para asegurarse de que aparezcan estos

**Figura 6.2**

Un proyecto con una clase de prueba



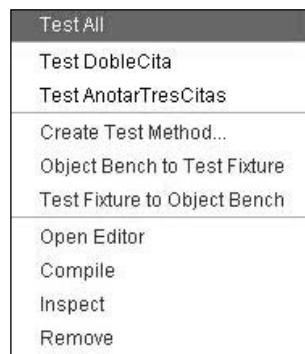
elementos en su proyecto. Una diferencia más se relaciona con el menú que aparece cuando se hace clic sobre la clase de prueba con el botón derecho del ratón (Figura 6.3): en lugar de una lista de constructores, hay tres nuevas secciones en el menú.

Las clases de prueba son una característica de BlueJ y están diseñadas para implementar pruebas de regresión. Se basan en el marco de trabajo para pruebas JUnit creado por Erich Gamma y Kent Beck. Una clase de prueba generalmente está asociada con una clase ordinaria del proyecto. En este caso, *DiaTest* está asociada con la clase *Dia* y decimos que *Dia* es la *clase de referencia* de *DiaTest*.

**JUnit, [www.junit.org](http://www.junit.org)** JUnit es un popular marco de trabajo (*framework*) para implementar en Java pruebas de unidad organizadas y pruebas de regresión. Está disponible independientemente del entorno específico de desarrollo que se use, así como también está integrado a muchos entornos. JUnit fue desarrollado por Erich Gamma y Kent Beck. Puede encontrar el software y gran cantidad de información sobre él en <http://www.junit.org>.

**Figura 6.3**

Menú contextual de una clase de prueba



Abra el proyecto *agenda-diaria-prueba-junit-v1*. Una vez abierto, seleccione las opciones *Tools/Preferences/Miscellaneous* del menú y asegúrese de que la opción *Show unit testing tools* esté activada. Inmediatamente, bajo el botón *Compile* de la ventana principal de BlueJ podrá ver algunos elementos adicionales, incluyendo el botón *Run Tests*. Presione este último botón y aparecerá la ventana que se muestra en la Figura 6.4. Las tildes ubicadas a la izquierda de cada nombre de prueba indican que las pruebas resultaron exitosas. Puede lograr el mismo resultado seleccionando la opción *Test All* del menú contextual asociado a la clase de prueba.

**Figura 6.4**

La ventana de resultados de la prueba



Las clases de prueba, en cierto sentido, son claramente diferentes de las clases ordinarias y si abre el código fuente de la clase *DiaTest* podrá ver que tiene algunas características nuevas. En este punto del libro no vamos a discutir en detalle la manera en que funcionan las clases de prueba, pero vale la pena hacer notar que pese a que el código de la clase *DiaTest* podría haber sido escrito por una persona, en realidad fue *generado automáticamente* por BlueJ. Algunos comentarios fueron agregados más tarde para volver más legible la clase. Una clase de prueba se crea primeramente usando el botón derecho del ratón sobre una potencial clase de referencia y seleccionando la opción *Create Test Class* del menú contextual. Observe que la clase *Dia* ya tiene una clase de prueba por lo que este elemento adicional no aparece en el menú de esta clase, pero en la clase *Cita* aparece este menú pues, actualmente, no tiene una clase de prueba asociada.

El punto clave de una clase de prueba es que contiene código fuente tanto para ejecutar las pruebas sobre una clase de referencia como para controlar si las pruebas resultaron exitosas o no. Por ejemplo, esta es una de las sentencias del método *testDobleCita* que controla si es posible o no registrar una segunda cita a las 9 a.m.:

```
assertEquals(false, dia1.anotarCita(9, cita2));
```

Cuando se ejecuta esta prueba, BlueJ es capaz de mostrar los resultados en la ventana mostrada en la Figura 6.4.

En la próxima sección hablaremos sobre la forma en que BlueJ soporta la automatización de pruebas de regresión para que pueda crear sus propias pruebas automatizadas.

**Ejercicio 6.14** Cree una clase de prueba para la clase `Cita` en el proyecto `agenda-diaria-prueba-junit-v1`.

**Ejercicio 6.15** ¿Qué métodos se crean automáticamente cuando se crea una nueva clase de prueba?

### 6.4.3 Grabar una prueba

Como hemos dicho al comienzo de la Sección 6.4, la automatización de pruebas es deseable porque la creación y recreación manual de pruebas es un proceso que insume mucho tiempo. BlueJ posibilita combinar la efectividad de las pruebas manuales con el poder de las pruebas automatizadas habilitándonos para grabar las pruebas manuales y luego ejecutarlas, con el fin de aplicar pruebas de regresión. La clase `DiaTest` del proyecto `agenda-diaria-prueba-junit-v1` se creó mediante este proceso. Usaremos el proyecto `agenda-diaria-prueba-junit-v2` para ilustrar las facilidades que ofrece BlueJ en cuanto a la grabación de pruebas.

Suponga que queremos probar a fondo el método `buscarEspacio` de la clase `Dia`. Este método trata de encontrar espacio para una cita. Existen varias pruebas que quisiéramos llevar a cabo:

- buscar espacio en un día que está vacío (positiva);
- buscar espacio cuando ya existe por lo menos una cita, pero el día aún no está completo (positiva);
- tratar de encontrar espacio en un día que está totalmente ocupado (negativa);
- tratar de encontrar espacio para citas de dos horas de duración cuando no existen espacios de dos horas consecutivas (negativa).

Describiremos cómo crear la primera de estas pruebas y dejamos al lector el resto de las pruebas a modo de ejercicios.

Abra el proyecto `agenda-diaria-prueba-junit-v2`. Para grabar una prueba se le indica a BlueJ que comience la grabación, a continuación se realiza manualmente la prueba y luego se indica la finalización de la prueba. Se logra el primer paso mediante el menú contextual de la clase de prueba. Esta acción le indica a BlueJ cuál es la clase en la que se quiere almacenar la nueva prueba. Seleccione la opción `Create Test Method...` del menú contextual de la clase `DiaTest`; BlueJ solicita un nombre para el método de prueba; si el nombre no comienza con la palabra «`test`», BlueJ la agregará como un prefijo.

Para esta prueba vamos a controlar que la llamada al método `buscarEspacio` en un día completamente libre devuelva la hora 9:00 a.m. como la primera hora disponible, por lo tanto puede resultar apropiado un nombre tal como `buscarEspacio9`. Una vez que se ingresó un nombre y que se hace clic en `Ok`, aparece un círculo rojo a la izquierda del diagrama de clases y se habilitan los botones `End` y `Cancel`. Se usa el botón `End` para indicar el fin del proceso de creación de la prueba y el botón `Cancel` para abandonar dicho proceso.

Una vez que comenzó la grabación, llevamos a cabo las acciones que deseamos tal como lo haríamos en una prueba manual:

- Crear un objeto `Dia`.
- Crear un objeto `Cita` de una hora de duración.
- Invocar al método `buscarEspacio` sobre el objeto `Dia`.

#### Concepto

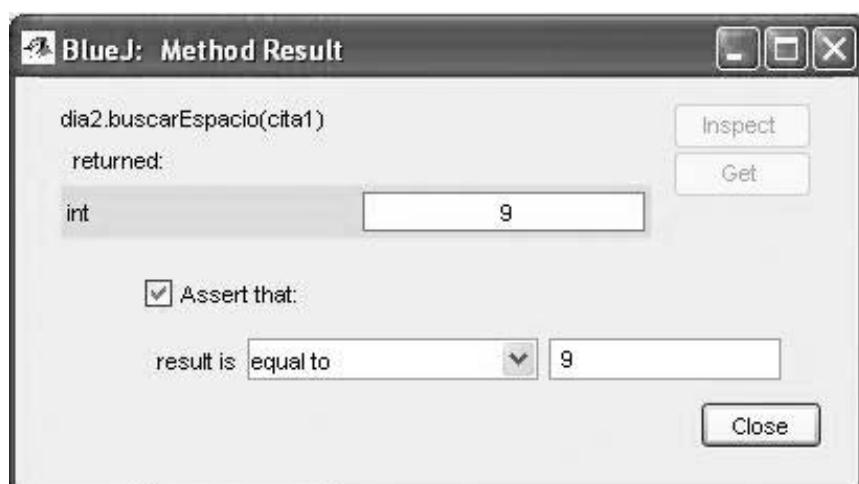
Una aserción es una expresión que establece una condición que esperamos que resulte verdadera. Si la condición es falsa, decimos que falló esta aserción que indica que hay un error en nuestro programa.

Antes de llegar al último paso no habrá ninguna diferencia con respecto a la interacción normal con los objetos. Sin embargo, una vez que se haya invocado al método `buscarEspacio` aparecerá una nueva ventana de diálogo (Figura 6.5). Esta ventana es una versión extendida de la ventana normal de resultados y es una parte crucial del proceso de pruebas automatizadas. Su propósito es permitirnos especificar los resultados que *debiera dar* este método. Esta especificación se denomina *aserción*.

En este caso, esperamos que el método devuelva el valor 9 y queremos incluir un control en nuestra prueba para asegurar que este sea realmente el caso. Ahora debemos asegurarnos de que esté seleccionada la caja de verificación *Assert that*, ingresar el 9 en el diálogo y seleccionar el botón *Close*.

**Figura 6.5**

El diálogo de resultado del método con la facilidad de aserción



Como este es el último paso de la prueba, presionamos el botón *End* para detener la grabación. En este punto, BlueJ agrega código a la clase `DiaTest` para nuestro nuevo método `testBuscarEspacio9`, luego compila la clase y limpia el banco de objetos. El método generado resultante se muestra en el Código 6.3.

**Código 6.3**

Un método de prueba generado automáticamente

```
public void testBuscarEspacio9()
{
    Dia dia1 = new Dia(1);
    Cita cita1 = new Cita("Conferencia de Java", 1);
    assertEquals(9, dia1.buscarEspacio(cita1));
}
```

Como puede verse, el método contiene sentencias que reproducen las acciones realizadas cuando se las estaba grabando: se crean los objetos `Dia` y `Cita` y se invoca el método `buscarEspacio`. La llamada a `assertEquals` controla que el resultado devuelto por `buscarEspacio` coincida con el valor 9 esperado. Se proporcionan los siguientes ejercicios para que pueda probar este proceso por sus propios medios. Se incluye un ejemplo para mostrar lo que ocurre en el caso en que el valor actual no coincide con el valor esperado.

**Ejercicio 6.16** Use el proyecto `agenda-diaria-prueba-junit-v2`; cree un método en la clase `DiaTest` para controlar que el método `buscarEspacio` devuelve el valor 10 para una cita de una hora de duración si un día ya tiene registrada una única cita a las 9 a.m. En esencia, necesita llevar a cabo pasos similares a los que se usaron para crear el método `testBuscarEspacio9`, pero use el método `anotarCita` para la primera cita y el método `buscarEspacio` para la segunda cita. Necesitará especificar aserciones para los resultados de ambas llamadas.

**Ejercicio 6.17** Cree una prueba para controlar que `buscarEspacio` retorne el valor -1 si se intenta buscar lugar para una cita en un día que ya está completo.

**Ejercicio 6.18** Cree una clase de prueba cuya clase de referencia sea `Cita`. Grabe dos métodos de prueba distintos para controlar que los campos `descripcion` y `duracion` de un objeto `Cita` se inicializan correctamente después de su creación.

**Ejercicio 6.19** Cree la siguiente prueba negativa de la clase `DiaTest`. Cree un objeto `Dia`, un objeto `Cita` de una hora y un objeto `Cita` de dos horas. Registre la primera cita a las 10 a.m. y luego trate de registrar la segunda cita a las 9 a.m. Dado que puede fallar la invocación al método `anotarCita`, el valor que se debe ingresar en la aserción es `false`. Ejecute la prueba. ¿Qué muestra la ventana de resultados de la prueba?

#### 6.4.4

#### Objetos de prueba

##### Concepto

A *fixture* es un conjunto de objetos con un estado definido que sirve como base para las pruebas de unidades.

Cuando se construye un conjunto de métodos de prueba, es común que se deban crear objetos similares para cada prueba. Por ejemplo, para cada prueba de la clase `Dia` hay que crear por lo menos un objeto `Dia` y uno o más objetos `Cita`. Un grupo de objetos que se usa en una o más pruebas se conoce como un *fixture*. En el menú asociado con la clase de prueba existen dos opciones que nos habilitan para trabajar con *fixtures* en BlueJ: *Object Bench to Test Fixture* y *Test Fixture to Object Bench*. Usando el proyecto `agenda-diaria-prueba-junit-v2`, cree en el banco de objetos un objeto `Dia` y un objeto `Cita` y luego seleccione la opción *Object Bench to Test Fixture* de la clase `DiaTest`. Los objetos desaparecerán del banco de objetos y si examina el código de la clase `DiaTest` verá que su método `setUp` tiene un código similar al Código 6.4, en donde `dia1` y `cita1` han sido definidos como campos.

La importancia del método `setUp` radica en que se invoca automáticamente, inmediatamente antes de la llamada a cada método de prueba. Esto quiere decir que los métodos de prueba individuales no necesitan más crear sus propias versiones de grupos de

objetos. Por lo tanto, podemos editar métodos tales como `testDobleCita` y eliminar sus primeras dos sentencias:

```
Dia dia1 = new Dia(1);
Cita cita1 = new Cita("Conferencia de Java", 1);
```

ya que las restantes sentencias de dicho método usarán los objetos del *fixture*.

#### Código 6.4

Creación de un *fixture*

```
/**
 * Establece el fixture para la prueba.
 * Se invoca antes de la ejecución de cada método.
 */
protected void setup()
{
    Dia dia1 = new Dia(1);
    Cita cita1 = new Cita("Conferencia de Java", 1);
}
```

Una vez que tenemos un fixture asociado a una clase de prueba, también se simplifica la grabación de más pruebas porque cada vez que se cree un nuevo método de prueba, los objetos del fixture aparecerán automáticamente en el banco de objetos.

En cualquier momento se pueden agregar más objetos al fixture; una de las formas más fáciles de hacerlo es seleccionar *Test Fixture to Object Bench*, agregar más objetos al banco de objetos de la manera habitual y luego seleccionar *Object Bench to Test Fixture*. Por supuesto que también podríamos simplemente editar el método `setUp` y agregar más campos directamente en la clase de prueba.

La automatización de pruebas es un concepto poderoso porque hace más probable que las pruebas se escriban en primer lugar y más probable que se ejecuten y reejecuten a medida que el programa se desarrolle. Podría formarse el hábito de comenzar por escribir pruebas de unidad tempranamente en el desarrollo de un proyecto y mantenerlas actualizadas a medida que el proyecto avance. En el Capítulo 12 volveremos al tema de las aserciones en el contexto del manejo de errores.

**Ejercicio 6.20** Agregue otras pruebas automatizadas en la clase `DiaTest` del proyecto *agenda-diaria-prueba-junit-v2* hasta que encuentre que adquirió confianza razonable en la correcta operación de las clases. Use tanto pruebas positivas como negativas. Si descubre algún error, asegúrese de grabar las pruebas para resguardarlas contra recurrencias de estos errores en versiones posteriores.

En la próxima sección veremos programas escritos desde la perspectiva más amplia de un proyecto llevado adelante por varias personas.

## 6.5

## Modularización e interfaces

En el Capítulo 3 hemos presentado el concepto de modularización en el contexto de un proyecto que implementa un reloj digital. Resaltamos que la modularización es cru-

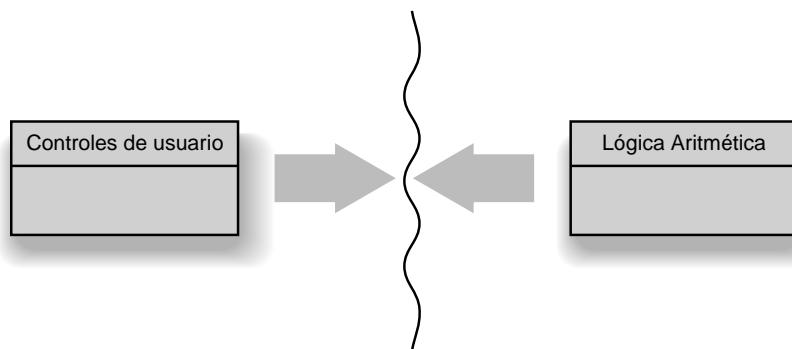
cial en cualquier proyecto en el que diferentes personas implementen los varios componentes del mismo. Sin embargo, no alcanza sólo con dividir una tarea en varias clases, además, debe haber directivas claras para las diferentes implementaciones que indiquen qué deben hacer y cómo encajan todos los componentes en la aplicación final. Sin estas directivas, el resultado final probablemente sería equivalente a intentar pasar un taco cuadrado por un orificio redondo.

Cuando varios componentes de un software colaboran para completar una misma tarea decimos que la *interfaz* entre ellos debe ser clara y bien definida. Por interfaz entendemos aquellas partes de una clase que se conocen y que se utilizan en otras clases, y este fue justamente el significado que le hemos dado a las interfaces en el Capítulo 5.

Por ejemplo, consideremos un proyecto de desarrollo de software para operar una calculadora aritmética. Una manera de dividir este proyecto es en dos grandes piezas: una parte responsable de permitir a los usuarios el ingreso de los cálculos y la otra para implementar la lógica de los cálculos. La Figura 6.6 pretende ilustrar el hecho de que cada módulo hace uso del otro, por lo que se debe haber hecho algo para definir la interfaz entre ellas.

**Figura 6.6**

Diferentes módulos de una calculadora

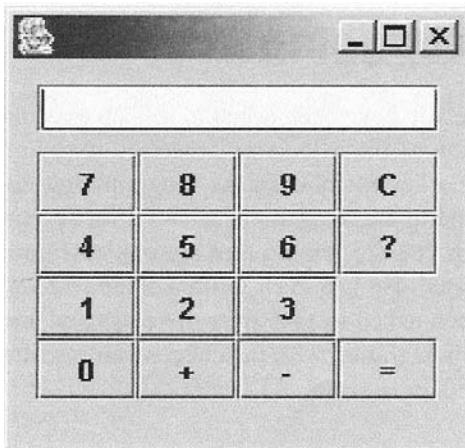


Cuando dos módulos se desarrollen simultáneamente, con frecuencia será necesario definir la interfaz antes de comenzar a trabajar sobre la implementación de cada uno. Esto puede hacerse, generalmente, mediante las firmas de los métodos porque proporcionan suficiente información de una clase sobre cómo interactuar con otra sin necesidad de saber cómo están implementados dichos métodos. Este es un concepto importante. Tratamos, tanto como sea posible, de separar las interfaces de las clases de los detalles de implementación. (Ya hemos discutido algunas ventajas de este punto en el Capítulo 5.) En el proyecto de la calculadora existen diferentes maneras que podemos elegir para implementar los controles de usuario: como una pieza de software puro con una vista gráfica de botones para presionar (Figura 6.7) o como una pieza de hardware a la manera de un dispositivo portátil. La implementación del componente que maneja la lógica aritmética no se verá afectada por las diferencias citadas.

En las próximas secciones exploraremos la implementación de un software simple de calculadora basado en dos clases: `MotorDeCalculadora` e `InterfazDeUsuario`. La interfaz que definimos entre ellas se muestra en el Código 6.5.

**Figura 6.7**

La interfaz de usuario de un software para una calculadora

**Código 6.5**

La interfaz de la unidad lógico aritmética

```
// Devuelve el valor que se mostrará
public int getValorEnVisor();
// Se llama cuando se presiona un botón de dígito
public void numeroPresionado(int numero);
// Se llama cuando se presiona el operador más
public void mas();
// Se llama cuando se presiona el operador menos
public void menos();
// Se llama para completar un cálculo
public void igual();
// Se llama para reinicializar la calculadora
public void limpiar();
```

La clase `MotorDeCalculadora` proporcionará la implementación de esta interfaz. La interfaz representa una especie simple de contrato entre la clase `MotorDeCalculadora` y otras partes del programa que la usarán. La interfaz describe un conjunto mínimo de métodos que serán implementados en el componente lógico y para cada método están completamente definidos su tipo de retorno y sus parámetros. Observe que la interfaz no brinda detalles sobre lo que hará su implementación internamente cuando se notifique que se presionó el operador más, por ejemplo; estos detalles quedan en manos de sus implementadores. Además, la implementación de la clase podría contener otros métodos que no aparecen en este listado.

En las secciones siguientes trataremos la implementación de esta interfaz para ilustrar varias técnicas de lectura de código y de depuración.

**6.6****Un escenario de depuración**

Imagine que se le pide unirse a un equipo de proyecto que ya está armado y que está trabajando en la implementación de la calculadora descrita en las secciones anteriores. Fue designado porque un miembro clave del equipo de programación, Hacker T. Largebrain, ha sido promocionado para dirigir otro proyecto. Antes de irse, Hacker ase-

guró al equipo al que usted se suma, que su implementación de la interfaz lógica estaba terminada y totalmente probada. También escribió algunos programas de prueba para verificar que este fuera el caso. Usted fue contratado para revisar la clase y simplemente asegurar que está comentada apropiadamente antes de integrarla con las clases escritas por otros miembros del equipo.

Usted decide que la mejor manera de comprender el programa de Hacker, antes de documentarlo, es examinar su código y el comportamiento de sus objetos.

## 6.7

## Comentarios y estilo

Abra el proyecto *calculadora-motor* para ver las clases que contiene. En esta etapa del desarrollo, la clase *MotorDeCalculadoraProbador* toma el lugar de la interfaz de usuario. Ilustra otro aspecto positivo de la definición de interfaces entre módulos: facilita el desarrollo de simulacros de otros módulos con el fin de probar uno.

Si lee el texto de la clase *MotorDeCalculadora* encontrará que su autor puso especial atención en el buen estilo de algunas áreas:

- La clase ha sido comentada con un comentario multilínea en la parte superior indicando el propósito de la misma. También incluyó anotaciones indicando el autor y el número de versión.
- Cada método de la interfaz tiene un comentario que indica su propósito, sus parámetros y su tipo de retorno. Ciertamente, estos comentarios facilitarán la generación de documentación para la interfaz, tal como lo discutimos en el Capítulo 5.
- El esquema de la clase es consistente, con cantidades adecuadas de espacios en blanco para la indentación que usa para distinguir los niveles de los bloques anidados y las estructuras de control.
- Las variables tienen nombres significativos y los nombres de los métodos han sido bien elegidos.

Pese a que estas convenciones parecen insumir demasiado tiempo durante la implementación, pueden redundar en un beneficio enorme para ayudar a otro a comprender el código (tal como tenemos que hacer en este escenario) o en ayudar a recordar qué hace una clase si dejamos de trabajar un tiempo en ella.

También notamos otro detalle que parece menos prometedor: Hacker no usó una clase de prueba especializada para capturar sus pruebas sino que escribió su propia clase de prueba. Dado que sabemos que BlueJ permite implementar pruebas de unidad, nos preguntamos por qué Hacker no utilizó esta facilidad.

Esta decisión no necesariamente tiene que ser mala. Las clases de prueba escritas a mano pueden ser buenas pero nos generan una pequeña sospecha. ¿Hacker sabía realmente lo que estaba haciendo? Volveremos sobre este punto más adelante.

Tal vez, ¡las habilidades de Hacker son tan grandes como él cree y no tengamos demasiado que hacer para que la clase quede lista para integrarla con las otras! Trate de hacer los siguientes ejercicios para ver si es éste el caso.

**Ejercicio 6.21** Asegúrese de que las clases del proyecto estén compiladas y luego cree en BlueJ un objeto *MotorDeCalculadoraProbador*. Invoque el método `testAll`. ¿Qué se imprime en la ventana terminal? ¿Cree lo que dice la última línea?

**Ejercicio 6.22** Usando el objeto que creó en el ejercicio anterior, invoque el método `testMas`. ¿Qué resultado da? ¿Es el mismo resultado que se imprimió cuando invocó a `testAll`? Invoque una vez más al método `testMas`. ¿Qué resultado da ahora? ¿Debiera dar siempre la misma respuesta? Si es así, ¿cuál debiera ser la respuesta? Lea el código del método `testMas` para verificar sus respuestas.

**Ejercicio 6.23** Repita los ejercicios anteriores con el método `testMenos`. ¿Da siempre el mismo resultado?

Los experimentos realizados a través de estos últimos ejercicios le deben haber alertado sobre el hecho de que no parece estar todo bien en la clase `MotorDeCalculadora`. Probablemente contenga errores, pero, ¿cuáles son y cómo encontrarlos? En las secciones que siguen consideraremos diferentes maneras mediante las que podemos tratar de localizar el lugar en la clase donde ocurren los errores.

## 6.8

### Seguimiento manual

#### Concepto

Un seguimiento manual o prueba de escritorio es la actividad en la que trabajamos sobre un segmento de código línea por línea mientras se observan los cambios de estado y otros comportamientos de la aplicación.

Los seguimientos manuales son técnicas poco usadas, quizás porque son de bajo nivel de depuración y de prueba, sin embargo, no debemos caer en el error de pensar que no son útiles. Un seguimiento manual involucra la impresión del código de las clases que se están tratando de comprender o depurar y que se reciben rápidamente de la computadora. Es demasiado fácil perder gran cantidad de tiempo sentado frente a una pantalla sin hacer ningún progreso frente a un problema de programación. Reubicar y concentrar el esfuerzo generalmente libera la mente para atacar el problema en una dirección completamente diferente. Hemos encontrado muchas veces que salir a almorzar o conducir desde la oficina son momentos en que se nos ocurren ideas que de otra manera hubiéramos tenido que pasar horas trabajando con el teclado.

Un seguimiento manual involucra tanto la lectura de clases como el seguimiento del control del flujo entre las clases y los objetos. Ayuda a la comprensión tanto de las maneras en que interactúan los objetos unos con otros como de la forma en que se comportan internamente. En efecto, un seguimiento manual (también denominado *prueba de escritorio*) es una simulación en papel y lápiz de lo que ocurre dentro de la computadora cuando se ejecuta un programa. En la práctica, es la mejor forma de concentrarse en una porción pequeña de la aplicación, tal como un grupo lógico de acciones o la llamada a un método.

#### 6.8.1

#### Un seguimiento de alto nivel

Ilustraremos la técnica del seguimiento manual con el proyecto *calculadora-motor*. Le resultará de utilidad imprimir el código de las clases `MotorDeCalculadora` y `MotorDeCalculadoraProbador` para seguir los pasos de esta técnica.

Comenzaremos por examinar el método `testMas` de la clase `MotorDeCalculadoraProbador` ya que contiene un grupo lógico de acciones que nos ayudarán a comprender cómo funcionan juntos varios de los métodos de la clase `MotorDeCalculadora` para completar los cálculos de una calculadora.

A medida que atravesamos este camino, tomaremos nota en papel y lápiz de las preguntas que surgen en nuestra mente.

1. En este primer paso, no queremos entrar en mucho detalle, simplemente queremos ver cómo el método `testMas` usa un objeto motor, sin explorar los detalles internos del mismo. Desde el principio de la experimentación nos pareció que hay algunos errores que queremos localizar, pero no sabemos si los errores están en el probador o en el motor. Por lo tanto, nuestro primer paso será controlar que el probador esté usando el motor adecuadamente.
2. Vemos que la primera sentencia de `testMas` asume que el campo motor hace referencia a un objeto válido:

```
motor.limpiar();
```

Podemos verificar si es así controlando el constructor del probador. Es un error común que los campos de los objetos no hayan sido inicializados adecuadamente, ya sea en su declaración o en un constructor. Si intentamos usar un campo sin un objeto asociado, el error más probable que ocurra es el error en tiempo de ejecución `NullPointerException`.

3. La primera llamada a `limpiar` se presenta como un intento de poner el motor de la calculadora en un estado inicial válido, listo para recibir las instrucciones para llevar a cabo un cálculo. Todo esto parece razonable, sería equivalente a presionar el botón de limpiar de una calculadora real. En este punto, no nos fijamos en la clase del motor que hace exactamente el método `limpiar`. Esto tendrá que esperar hasta que hayamos adquirido un cierto nivel de confianza en que las acciones del probador son razonables. En cambio, tomamos nota simplemente de que `limpiar` pone al motor en un estado inicial válido tal como se espera.
4. La siguiente sentencia en `testMas` representa el ingreso de un dígito mediante el método `numeroPresionado`:

```
motor.numeroPresionado(3);
```

Esta línea también es razonable ya que el primer paso para realizar un cálculo es ingresar el primer operando. Una vez más no vemos qué hace el motor con el número, sólo asumimos que lo almacena en algún lugar para usarlo más tarde en el cálculo.

5. La siguiente sentencia invoca a `mas`, por lo que ahora sabemos que el valor del operando de la izquierda es 3. Podríamos tomar nota de este hecho sobre el impreso o tildar esta afirmación en uno de los comentarios de `testMas`. De manera similar podríamos anotar o confirmar que la operación que se está ejecutando es la suma. Esto parece algo trivial pero es muy fácil que los comentarios de una clase se desvien del código que se supone que documentan; de modo que controlar los comentarios mientras leemos el código nos evitan que nos olvidemos de ellos más tarde.
6. A continuación, se ingresa otro dígito como el operando de la derecha mediante una nueva llamada a `numeroPresionado`.
7. La realización de la suma se pide mediante el método `igual`. Podemos tomar nota, de la misma manera en que se hizo en `testMas`, de que el método `igual` pareciera que no devuelve el resultado del cálculo, en contra de lo que esperábamos. Esto es algo más que podríamos controlar cuando veamos la clase `MotorDeCalculadora`.

8. La última sentencia del método `testMas` obtiene el valor que aparecería en el visor de la calculadora:

```
return motor.getValorEnVisor();
```

9. Presumiblemente, este es el resultado de la suma pero no podemos estar seguros sin ver los detalles de `MotorDeCalculadora`. Nuevamente tomamos nota de controlar que este sea realmente el caso.

Mediante nuestro examen completo de `testMas` hemos ganado un grado razonable de confianza en que usa el motor adecuadamente, es decir, simula una secuencia reconocible de teclas presionadas para realizar un cálculo sencillo. Podríamos recalcar que el método no es particularmente ambicioso, ambos operandos son números de un solo dígito y se usa un solo operador. Sin embargo, esto no es inusual al probar métodos porque es importante probar la funcionalidad más básica antes de probar combinaciones más complejas. Aunque es útil observar que se podrían haber agregado algunas pruebas más complejas en el probador.

**Ejercicio 6.24** Realice un seguimiento del método `testMenos`. ¿Surgen otras preguntas sobre cosas que probablemente quiera controlar cuando vea el detalle de `MotorDeCalculadora`?

Antes de entrar a ver la clase `MotorDeCalculadora`, es valioso realizar un seguimiento del método `testAll` para ver cómo usa los métodos `testMas` y `testMenos` que hemos visto.

1. El método `testAll` es una secuencia lineal de sentencias de impresión.
2. Contiene una llamada a cada uno de los métodos `testMas` y `testMenos` y se imprimen los valores que estos devuelven para que el usuario los vea. Podríamos observar que no hay ninguna sentencia que le indique al usuario cuál debe ser el resultado, lo que dificulta la confirmación de si los resultados son correctos.
3. Vemos que la última sentencia establece audazmente que:

Pasaron todas las pruebas.

¡Pero el método no contiene pruebas para establecer la verdad de esta afirmación! Debe haber medios apropiados de establecer ambas cosas: cuáles deben ser los valores de los resultados y si han sido calculados correctamente o no. Esto es algo que debemos remediar tan pronto como podamos regresar al código de esta clase.

En esta etapa, no debemos distraernos del objetivo final y realizar cambios que no están dirigidos directamente por los errores que estamos buscando. Si hacemos esta clase de cambios podríamos caer fácilmente en el enmascaramiento de los errores. Uno de los requerimientos cruciales de la depuración exitosa es ser capaz de disparar fácilmente el error que estamos buscando y reproducirlo, por este camino es mucho más fácil de evaluar el efecto de un intento de corrección.

Luego de haber controlado la clase de prueba, estamos en condiciones de examinar el código de la clase `MotorDeCalculadora`. Podemos hacerlo armados de una secuencia razonable de llamadas a métodos para explorar y un conjunto de preguntas, ambos obtenidos a partir del seguimiento manual del método `testMas`.

## 6.8.2 Controlar el estado mediante el seguimiento

El estilo del objeto `MotorDeCalculadora` es muy diferente del estilo de su probador. El motor es un objeto completamente pasivo. No inicia ninguna actividad por sí mismo sino que simplemente responde a invocaciones externas de métodos. Este es el estilo de comportamiento típico de un servidor. Con frecuencia, los objetos servidores descansan fuertemente sobre su propio estado para determinar cómo deben responder a las llamadas de métodos. Esto es particularmente cierto en el motor de la calculadora. Por lo que, una parte importante al conducir el seguimiento es estar seguro de que siempre disponemos de una representación exacta de su estado. Una forma de hacer esto en papel y lápiz es construyendo una tabla de los campos del objeto y sus valores (Figura 6.8). Se puede agregar una nueva línea para llevar el registro de los valores que surgen durante la ejecución, después de cada llamada a método.

**Figura 6.8**

Tabulación informal  
del estado de un  
objeto

Método llamado	valorEnVisor	operandoIzquierdo	operadorAnterior
<code>estado inicial</code>	0	0	‘ ‘
<code>limpiar</code>	0	0	‘ ‘
<code>numeroPresionado(3)</code>	3	0	‘ ‘

Esta técnica hace que resulte muy fácil volver atrás si aparece algo que anda mal. También es posible comparar los estados después de dos invocaciones al mismo método.

1. Cuando comenzamos el seguimiento de `MotorDeCalculadora`, documentamos el estado inicial del motor tal como se hizo en la primer fila de valores en la Figura 6.8. Todos sus campos se inicializan en el constructor. Tal como observamos cuando hicimos el seguimiento del probador, es importante la inicialización del objeto y podríamos tomar nota aquí de controlar que la inicialización por defecto sea suficiente; particularmente, el valor por defecto de `operadorAnterior` podría no representar un operador significativo. Además, esto nos hace pensar si realmente es importante tener un operador previo antes del primer operador real en la calculadora. Al anotar estas cuestiones no necesariamente tenemos que descubrir las respuestas en forma directa pero nos proporcionan sugerencias a medida que obtenemos más información sobre la clase.
2. El siguiente paso consiste en ver cómo cambia el estado del motor una llamada a `limpiar`. Tal como se muestra en la segunda fila de datos de la tabla de la Figura 6.8, el estado permanece sin cambios en este punto porque el `valorEnVisor` todavía está en cero. Pero podemos anotar otra pregunta: ¿por qué este método establece solamente el valor de un campo? Si se supone que este método pretende implementar una forma de reinicializar la calculadora, ¿por qué no limpia todos los campos?
3. Luego se investiga una llamada a `numeroPresionado` con el parámetro actual 3. El método multiplica el `valorEnVisor` existente por 10 y luego le suma el nuevo dígito. Esta acción modela correctamente el efecto de agregar un nuevo dígito a la derecha de un número existente. Descansa en el hecho de que `valorEnVisor` tenga un valor inicial cero cuando se ingresa el primer dígito de un nuevo número, y nuestra investigación del método `limpiar` nos dio la certeza de que es así. Por lo que en este método todo parece estar bien.

4. Continuando el orden de las llamadas en el método `testMas`, vemos ahora el método `mas`. Su primera sentencia invoca al método `aplicarOperadorPrevio`. Aquí tenemos que decidir si continuamos ignorando las invocaciones anidadas de métodos o si hacemos un corte y vemos qué hace. Dando una mirada rápida al método `aplicar` vemos que es muy corto; además, claramente va a alterar el estado del motor y no podremos seguir documentando los cambios de estado a menos que lo estudiemos. Por lo que ciertamente decidimos continuar la llamada anidada. Es importante recordar de dónde venimos, de modo que podríamos hacer una marca en la lista de métodos indicando que estamos dentro del método `mas` antes de continuar con el método `aplicar`. Si para seguir la llamada de un método anidado tenemos que entrar en más llamadas anidadas, necesitaremos usar algo más que una simple marca que nos ayude a encontrar nuevamente el camino de regreso al llamador. En este caso, es mejor marcar los puntos de llamadas con valores numéricos ascendentes, reutilizando los valores previos como valores de retorno de las llamadas.
5. El método `aplicarOperadorPrevio` nos da bastante idea sobre cómo se usa el campo `operadorPrevio`. Aparece también la respuesta a una de las preguntas que nos hicimos anteriormente: si era correcto el tener un espacio en blanco como valor inicial en el operador previo. El método controla explícitamente si el `operadorPrevio` contiene un '+' o un '-' antes de aplicarlo. Por lo que ningún otro valor dará por resultado que se aplique a una operación incorrecta. Al final de este método, el valor de `operandoIzquierdo` tendrá que estar cambiado, por lo que podemos anotar su nuevo valor en la tabla de estado.
6. Volviendo al método `mas`, los dos campos restantes tienen establecidos sus valores, por lo que la siguiente fila de la tabla de estado contendrá los siguientes valores:

mas	0	3	‘+’
-----	---	---	-----

El seguimiento del motor se puede continuar de manera similar, documentando los cambios de estado, obteniendo una mejor comprensión sobre su comportamiento interno y surgiendo preguntas a lo largo del proceso. Los siguientes ejercicios podrán ayudarlo a completar el seguimiento.

**Ejercicio 6.25** Complete la tabla de estado basada en la siguiente subsecuencia de llamadas dentro del método `testMas`:

```
numeroPresionado(4);
igual();
```

**Ejercicio 6.26** Cuando realizó el seguimiento del método `igual` ¿percibió las mismas inseguridades que encontramos en `aplicarOperadorPrevio` sobre el valor por defecto del campo `operadorPrevio`?

**Ejercicio 6.27** Realice el seguimiento de una llamada al método `limpiar` inmediatamente después de la llamada al método `igual` al final de su tabla de estado y registre el nuevo estado. El motor, ¿está en el mismo estado que antes de la llamada a `limpiar`? Si no es así, ¿qué impacto piensa que podría tener en cualquier subsecuencia de cálculos?

**Ejercicio 6.28** A la luz del seguimiento, ¿qué cambios cree que debieran hacerse en la clase `MotorDeCalculadora`? Realice estos cambios sobre una

versión de la clase en papel y luego realice nuevamente el seguimiento. No necesita hacer el seguimiento completo de la clase `MotorDeCalculadora-Probador`, sólo repetir las acciones de su método `testAll`.

**Ejercicio 6.29** Trate de realizar el seguimiento de la siguiente secuencia de llamadas en su versión corregida del motor:

```
limpiar();
numeroPresionado(9);
mas();
numeroPresionado(1);
menos();
numeroPresionado(4);
igual();
```

¿Cuál debe ser el resultado? ¿Se comporta correctamente el motor y deja la respuesta correcta en `valorEnVisor`?

### 6.8.3 Seguimiento verbal

Otra manera de usar la técnica de seguimiento para encontrar errores en un programa es tratar de explicar a otra persona lo que hace una clase o un método. Esta forma funciona de dos maneras completamente diferentes:

- La persona a la que le explica el código podría encontrar el error por usted.
- Encontrará con frecuencia que el simple hecho de tratar de poner en palabras lo que debiera hacer una pieza de código es suficiente para activar en su mente una comprensión del por qué no lo hace.

El último efecto es tan común que podemos explicar una pieza de código a alguien que no está para nada familiarizado con ella, no con la expectativa de que encuentre los errores, ¡pero esto ocurrirá!

## 6.9

## Sentencias de impresión

Probablemente, la técnica más común usada para comprender y depurar un programa, aun por programadores experimentados, es agregar en los métodos sentencias de impresión temporalmente. Las sentencias de impresión son populares porque existen en la mayoría de los lenguajes, están disponibles para todos y son muy fáciles de agregar mediante un editor. El software o el lenguaje no necesita características adicionales para usarlas. Cuando se ejecuta un programa, estas sentencias de impresión adicionales proveen al usuario de información tal como:

- qué métodos se han invocado;
- los valores de los parámetros;
- el orden en que se han invocado los métodos;
- los valores de las variables locales y de los campos en lugares estratégicos.

El Código 6.6 muestra un ejemplo de cómo quedaría el método `numeroPresionado` con el agregado de sentencias de impresión. Esta información es particularmente útil

para proporcionar una imagen de la manera en que cambia el estado de un objeto cuando se invocan métodos de modificación. En apoyo a esta técnica, es valioso incluir métodos de depuración que muestren el valor actual de todos los campos de un objeto. El Código 6.7 muestra el método `informarEstado` para la clase `MotorDeCalculadora`.

### Código 6.6

Un método con sentencias de impresión con fines de depuración

```
/**  
 * El número que se presionó  
 */  
public void numeroPresionado(int numero)  
{  
    System.out.println("Se invocó a numeroPresionado con: " +  
        numero);  
    valorEnVisor = valorEnVisor * 10 + numero;  
    System.out.println("El valorEnVisor es: " + valorEnVisor +  
        " al final de  
numeroPresionado");  
}
```

### Código 6.7

Un método para informar estado

```
/** Imprime los valores de los campos de este objeto.  
 * @param donde Lugar donde ocurre este estado del  
objeto  
 */  
public void informarEstado (String donde)  
{  
    System.out.println("valorEnVisor: " + valorEnVisor +  
        " operandoIzquierdo: " +  
operandoIzquierdo +  
        " operadorPrevio: " + operadorPrevio +  
        " en " + donde);  
}
```

Si cada método de `MotorDeCalculadora` contiene una sentencia de impresión al comienzo y una invocación a `informarEstado` al final, la Figura 6.9 muestra la salida que podría generar una invocación al método `testMas` de la clase probadora. (Esta salida fue generada a partir de una versión del motor de la calculadora que se encuentra en el proyecto *calculadora-motor-impresion*.) Salidas como estas nos permiten hacernos una idea de cómo se controla el flujo entre los diferentes métodos. Por ejemplo, podemos ver a partir del orden en que se informan los valores del estado, que una llamada al método `mas` contiene una llamada anidada al método `aplicarOperadorprevio`.

Las sentencias de impresión pueden ser muy efectivas para ayudarnos a comprender los programas o para ubicar errores, pero existen algunas desventajas:

- Generalmente, no es muy práctico agregar sentencias de impresión a cada método de una clase. Por lo que sólo son completamente efectivas si se agregan en los métodos correctos.

**Figura 6.9**

Salida de la depuración de una llamada al método `testMas`

```
se invocó el método limpiar
valorEnVisor: 0 operandoIzquierdo: 0 operadorPrevio: al
final de limpiar
se invocó el método numeroPresionado con : 3
valorEnVisor: 3 operandoIzquierdo: 0 operadorPrevio: al
final de numeroPr...
se invocó el método mas
se invocó el método aplicarOperadorPrevio
valorEnVisor: 3 operandoIzquierdo: 3 operadorPrevio: al
final de aplicarO...
valorEnVisor: 0 operandoIzquierdo: 3 operadorPrevio: + al
final de mas
se invocó el método numeroPresionado con : 4
valorEnVisor: 4 operandoIzquierdo: 3 operadorPrevio: + al
final de numeroP...
se invocó el método igual
valorEnVisor: 7 operandoIzquierdo: 0 operadorPrevio: + al
final de igual
```

- Agregar demasiadas sentencias de impresión puede llevarnos a perder de vista información. En una cantidad muy grande de información de salida es muy difícil identificar lo que necesitamos ver. En particular, las sentencias de impresión dentro de los ciclos traen aparejados estos problemas.
- Una vez que cumplieron con su propósito, puede resultar tedioso eliminarlas.
- Existe también la posibilidad de que habiéndolas eliminado, resulten nuevamente necesarias. ¡Puede ser muy frustrante tener que agregarlas nuevamente!

**Ejercicio 6.30** Abra el proyecto *calculadora-motor-impresion* y complete las sentencias de impresión adicionales para cada método y para el constructor.

**Ejercicio 6.31** Cree un objeto *MotorDeCalculadoraProbador* en el proyecto y ejecute el método `testAll`. ¿Resulta de ayuda esta salida para identificar dónde están los problemas?

**Ejercicio 6.32** La salida producida por las sentencias de impresión agregadas en la clase *MotorDeCalculadora*, ¿le resulta poca, demasiada o adecuada? Si le parece que es poca o demasiada, agregue más sentencias de impresión o elimine algunas hasta que la salida tenga un nivel adecuado de información.

**Ejercicio 6.33** ¿Cuáles son las respectivas ventajas y desventajas de usar seguimiento manual o sentencias de impresión para la depuración? Fundamente su respuesta.

### 6.9.1 Activar o desactivar la información de depuración

Si una clase todavía se encontraba en desarrollo cuando se le agregaron sentencias de impresión, generalmente no queremos ver esta salida cada vez que se use la clase. Es mejor que podamos encontrar una manera de activar la impresión o desactivarla, según necesitemos. La forma más común de llevar esto a cabo es agregar un campo lógico

(boolean) a la clase y luego hacer que la impresión dependa del valor de este campo. El Código 6.8 ilustra esta idea.

### Código 6.8

Controlar si se imprime o no la información de depuración

```
/***
 * Se presionó un botón de número
 */
public void numeroPresionado (int numero)
{
    if (depuracion) {
        System.out.println("se invocó numeroPresionado con: " +
numero);
    }
    valorEnVisor = valorEnVisor * 10 + numero;
    if (depuracion){
        informarEstado();
    }
}
```

Una variante más económica de este tema consiste en reemplazar las llamadas directas a sentencias de impresión por invocaciones a los métodos de impresión agregados a la clase<sup>1</sup>. El método de impresión sólo imprimirá si el campo `depuracion` es verdadero (`true`). Por lo tanto, las llamadas al método de impresión no necesitarán ser resguardadas por una sentencia `if`. El Código 6.9 ilustra esta aproximación. Observe que esta versión asume que `informarEstado` controla el campo `depuracion` o también, que invoca al nuevo método `imprimirDepuracion`.

### Código 6.9

Un método para imprimir selectivamente la información de depuración

```
/***
 * Se presionó un botón de número.
 */
public void numeroPresionado (int numero)
{
    imprimirDepuracion(" se invocó numeroPresionado con " +
numero);
    valorEnVisor = valorEnVisor * 10 + numero;
    informarEstado(),
}
/** Solamente imprime la información de depuración cuando
el campo
 * depuracion es true
 * @param info La información de depuración
 */
```

---

<sup>1</sup> En realidad, podríamos mover este método a una clase de depuración especializada, pero queremos mantener las cosas simples en esta discusión.

### Código 6.9 (continuación)

Un método para imprimir selectivamente la información de depuración

```
public void imprimirDepuracion(String info)
{
    if (depuracion) {
        System.out.println(info);
    }
}
```

## 6.10

### Elegir una estrategia de prueba

Hemos visto que existen varias estrategias de prueba diferentes: seguimiento manual y verbal, uso de sentencias de impresión (ya sean temporales o permanentes con activadores), pruebas interactivas mediante el banco de objetos, escribir nuestras propias clases de prueba o usar una clase de prueba de unidad dedicada.

En la práctica, podríamos usar estrategias diferentes en momentos diferentes. Los seguimientos, las sentencias de impresión y las pruebas interactivas son útiles para la prueba inicial de clases recién escritas o para investigar cómo funciona un segmento de un programa. Su ventaja es que estas técnicas son rápidas y fáciles de usar, funcionan bien en cualquier lenguaje de programación y son independientes del entorno (excepto las pruebas interactivas). La principal desventaja es que estas pruebas no se pueden repetir fácilmente más adelante para realizar pruebas de regresión.

El uso de clases de pruebas de unidad tiene la ventaja, una vez que se las construyó, de que las pruebas se pueden ejecutar cualquier número de veces.

Por lo que el camino que eligió Hacker de escribir su propia clase de prueba fue un paso en la dirección correcta pero, por supuesto, tuvo grietas. Ahora sabemos que su problema fue que, pese a que su clase contiene llamadas a métodos razonables para la prueba, no incluyó ninguna aserción sobre el resultado de los métodos, y esto hizo que no detectara el fallo de la prueba.

También sabemos, por supuesto, que podría haber sido mejor y más fácil usar una clase de prueba de unidad dedicada.

**Ejercicio 6.34** Abra el primer proyecto *calculadora-motor* y agregue una forma de prueba mejor que reemplace la clase de prueba que hizo Hacker, asociada con la clase *MotorDeCalculadora*. Agregue pruebas similares a las que usó Hacker (y cualquier otra que encuentre útil) e incluya aserciones correctas.

## 6.11

### Depuradores

En el Capítulo 3 presentamos el uso de un depurador para comprender cómo opera una aplicación existente y cómo interactúan sus objetos. De manera muy similar, podemos usar el depurador para seguir el rastro de los errores.

El depurador es esencialmente una herramienta de software que proporciona apoyo para realizar un seguimiento de un segmento de código. Típicamente fijamos puntos de interrupción en las sentencias en donde queremos comenzar nuestro seguimiento y luego usamos las funciones *Step* y *Step Into* para llevarlo a cabo.

Una de las ventajas es que el depurador automáticamente tiene el cuidado de mantener el trazo del estado de cada objeto y al hacer esto, es más rápido y produce menos errores que cuando hacemos lo mismo manualmente. Una desventaja de los depuradores es que no mantienen registro permanente de los cambios de estado por lo que resulta difícil volver atrás y controlar el estado en que estaba unas cuantas sentencias antes.

Un depurador, típicamente, ofrece información sobre la *secuencia de llamadas* (o pila de llamadas o *stack*) en cada momento. La secuencia de llamadas muestra el nombre del método que contiene la sentencia actual, y el nombre del método desde donde fue llamado, y el nombre del método que fue llamado, etc. Por lo que, la secuencia de llamadas contiene un registro de todos los métodos activos y aún no terminados, de manera similar a la que hemos hecho manualmente durante nuestro seguimiento escribiendo marcas próximas a las sentencias de invocación de métodos.

En BlueJ, la secuencia de llamadas se muestra en la parte izquierda de la ventana del depurador. Cada nombre de método en dicha secuencia puede ser seleccionado para inspeccionar los valores actuales de las variables locales de dicho método.

**Ejercicio 6.35** Desafío. En la práctica probablemente encontrará que el intento de Hacker T. LargeBrain de programar la clase *MotorDeCalculadora* está lleno de errores que serán trabajosos de corregir. En su lugar, escriba su propia versión de la clase. El proyecto *calculadora-gui* contiene clases que proporcionan el entorno gráfico (GUI) que se muestra en la Figura 6.7. Asegúrese de documentar su clase y de crear un conjunto de pruebas para su implementación de modo que ¡su experiencia con el código de Hacker no sea repetida por sus sucesores!

## 6.12

### Poner en práctica las técnicas

En este capítulo hemos descrito varias técnicas que pueden usarse tanto para comprender un programa nuevo como para probar errores en otro. El proyecto *ladrillos* le ofrece una oportunidad de probar dichas técnicas en un nuevo escenario. El proyecto contiene parte de una aplicación para una compañía productora de ladrillos. Los ladrillos se envían a los clientes en palletes (pilas de ladrillos). La clase *Pallete* provee métodos que calculan el ancho y el alto de un pallete individual, de acuerdo con el número de ladrillos que tiene.

**Ejercicio 6.36** Abra el proyecto *ladrillos*. Pruébelo. Existen por lo menos cuatro errores en este proyecto. Vea si puede encontrarlos y corregirlos. ¿Qué técnicas usó para encontrar los errores? ¿Qué técnicas fueron las más útiles?

## 6.13

### Resumen

Cuando escribimos programas, debemos anticipar que contendrán errores lógicos. Por lo tanto, es esencial considerar los procesos de prueba y de depuración, ambas, como actividades normales durante todo el proceso de desarrollo del software. BlueJ es particularmente bueno en el apoyo de pruebas interactivas de unidades tanto de métodos como de clases. También hemos visto algunas técnicas básicas para automatizar los

procesos de prueba y realizar depuraciones sencillas. Sin embargo, nunca eliminamos por completo los errores. En el Capítulo 7 veremos algunas maneras en las que podemos reducir las oportunidades de introducir errores cuando escribimos programas orientados a objetos.

Términos introducidos en este capítulo

**error de sintaxis, error de lógica, prueba, depuración, prueba de unidad, prueba positiva, prueba negativa, prueba de regresión, seguimiento manual, secuencia de llamadas**

## Resumen de conceptos

- **prueba** La prueba es la actividad de descubrir si una pieza de código (un método, una clase o un programa) produce el comportamiento pretendido.
- **depuración** La depuración es el intento de apuntar con precisión y corregir el código de un error.
- **prueba positiva** Una prueba positiva es la prueba de los casos que se espera que resulten exitosos.
- **prueba negativa** Una prueba negativa es la prueba de los casos en que se espera que falle.
- **aserción** Una aserción es una expresión que establece una condición que esperamos que resulte verdadera. Si la condición es falsa, decimos que falló la aserción. Esto indica un error en nuestro programa.
- **fixture** Un fixture es un conjunto de objetos en un estado definido que sirven como una base para las pruebas de unidad.
- **seguimiento** Un seguimiento es la actividad de trabajar a través de un segmento de código línea por línea, mientras se observan cambios de estado y otros comportamientos de la aplicación.



## CAPÍTULO

# 7

## Diseñar clases

Principales conceptos que se abordan en este capítulo

- diseño dirigido por responsabilidades
- acoplamiento
- cohesión
- refactorización

Construcciones Java que se abordan en este capítulo

`static` (para métodos), `Math`, tipos enumerados

En este capítulo veremos algunos de los factores que influyen en el diseño de una clase. ¿Qué hace que un diseño sea bueno o malo? A corto plazo, la escritura de buenas clases puede tomar más tiempo que la escritura de clases malas, pero a largo plazo, el esfuerzo adicional para escribir clases de buena calidad, generalmente se verá justificado. Existen algunos principios que podemos seguir y que nos ayudan a escribir clases de buena calidad. En particular, el enfoque que presentamos se basa en que el diseño de clases debe estar dirigido por responsabilidades y que esas clases deben encapsular sus datos.

Este capítulo, como muchos de los anteriores, está estructurado alrededor de un proyecto. El proyecto puede ser estudiado mientras se va leyendo y siguiendo nuestro argumento o puede estudiarse con mayor profundidad haciendo los ejercicios en paralelo, mientras se recorre el capítulo.

El proyecto de trabajo se divide en tres partes. En la primera parte, discutimos sobre los cambios de código necesarios y desarrollamos y mostramos las soluciones completas de los ejercicios. La solución de esta parte está disponible en un proyecto que acompaña este libro. La segunda parte sugiere más cambios y extensiones y discutimos las posibles soluciones en un nivel alto (el nivel de diseño de clase) pero dejamos a los lectores el trabajo de bajo nivel (el código) y la implementación completa.

La tercera parte sugiere aún más mejoras bajo la modalidad de ejercicios. En este caso, no aportamos soluciones y en los ejercicios se aplica el material tratado a lo largo del capítulo.

Implementar todas las partes da por resultado un buen proyecto de programación de varias semanas. También puede llevarse a cabo como un proyecto grupal.

## 7.1

# Introducción

Es posible implementar una aplicación y lograr que realice su tarea mediante un diseño de clases mal logrado. El hecho de ejecutar una aplicación terminada en general, no indica si está bien estructurada internamente.

El problema surge, típicamente, cuando un programador de mantenimiento quiere hacer algunos cambios en una aplicación existente. Si por ejemplo, el programador intenta solucionar un fallo o quiere agregar funcionalidad a un programa, una tarea que debiera ser fácil y obvia con un buen diseño de clases, podría resultar muy difícil de manejar e insumir una gran cantidad de trabajo si las clases están mal diseñadas.

En las aplicaciones grandes, este efecto ya ocurre durante la implementación original. Si la implementación comienza con una mala estructura, su finalización puede volverse muy compleja y puede que no se termine de completar el programa, o que contenga fallos o que su construcción tome más tiempo de lo necesario. En la realidad, las compañías frecuentemente mantienen, extienden y venden una aplicación durante varios años. No es poco frecuente que una implementación de software que podemos comprar hoy en un comercio haya comenzado 10 años atrás. En esta situación, la compañía de software no se puede arriesgar a tener un código mal estructurado.

Dado que los malos efectos del diseño de clases de mala calidad se vuelven más obvios cuando se trata de adaptar o extender una aplicación, esto es exactamente lo que haremos. En este capítulo usaremos un ejemplo denominado *word-of-zuul*, que es un juego simple de aventuras, basado en texto e implementado rudimentariamente. En su estado original, el juego no es muy ambicioso por un motivo, está incompleto. Al final del capítulo, como siempre, estará en posición de ejercitarse su imaginación y diseñar e implementar su propio juego y crearlo realmente interesante y divertido.

**word-of-zuul** Nuestro juego *word-of-zuul* está modelado en base al juego original de aventuras que fue desarrollado en los inicios de la década del 70 por Will Crowther y expandido por Don Woods. Al juego original también se le conoce bajo el nombre *Colossal Cave Adventure*. Fue un juego maravillosamente imaginativo y sofisticado para su época, que consistía en encontrar el camino a través de un complejo sistema de cuevas, ubicar el tesoro escondido, usar palabras secretas y otros misterios, todo en función del esfuerzo de obtener el máximo puntaje. Puede leer más sobre este juego en lugares como <http://jerz.setonhill.edu/if/canon/Adventure.html> y en <http://www.rickadams.org/adventure/> o haciendo una búsqueda en Internet con las palabras «*Colossal Cave Adventure*».

Mientras trabajamos para extender la aplicación original, tendremos la oportunidad de discutir algunos aspectos del diseño de clases existente. Veremos que la implementación del juego con que comenzamos tiene ejemplos de decisiones de diseño mal tomadas y también veremos cómo impactan estas decisiones en nuestras tareas y cómo podemos corregirlas.

En los ejemplos de este libro encontrará dos versiones del proyecto zuul: *zuul-malo* y *zuul-mejorado*. Ambas versiones implementan la misma funcionalidad pero difieren un poco en la estructura de clases, uno de los proyectos representa un diseño de mala calidad

y el otro, es un diseño mejorado. El hecho de que podamos implementar la misma funcionalidad en ambos casos, de una manera buena y de una mala, ilustra la cuestión de que el diseño de mala calidad no es, generalmente, consecuencia de tener un problema difícil para resolver. La mala calidad del diseño tiene más que ver con las decisiones que se toman cuando se resuelve un problema en especial. No podemos usar el argumento de que no había otra manera de resolver el problema como una excusa para un diseño de mala calidad.

Por lo tanto, usaremos el proyecto como ejemplo de un diseño de mala calidad de modo que podamos explorar los motivos por los que está mal y mejorarlo. La versión mejorada es una implementación de los cambios que se discuten en este libro.

**Ejercicio 7.1** Abra el proyecto *zuul-malo*. (Este proyecto es «malo» porque su implementación contiene malas decisiones de diseño y ¡no queremos que quede ninguna duda de que no debe usarse este proyecto como ejemplo de práctica de buena programación!) Ejecute y explore la aplicación. El comentario del proyecto aporta alguna información sobre cómo ejecutarlo.

Mientras explora la aplicación, responda las siguientes preguntas:

- ¿Qué hace la aplicación?
- ¿Qué comandos acepta el juego?
- ¿Qué hace cada comando?
- ¿Cuántas habitaciones hay en el escenario?
- Dibuje un mapa de las habitaciones existentes.

**Ejercicio 7.2** Después de conocer qué hace la aplicación, trate de encontrar qué hace cada clase individualmente. Escriba en papel el propósito de cada clase. Para hacer esto, necesita ver el código fuente. Tenga en cuenta que no necesita ni tiene por qué comprender todo el código, se suele alcanzar con la lectura de los comentarios del código y de los encabezados de los métodos.

## 7.2

## Ejemplo del juego *world-of-zuul*

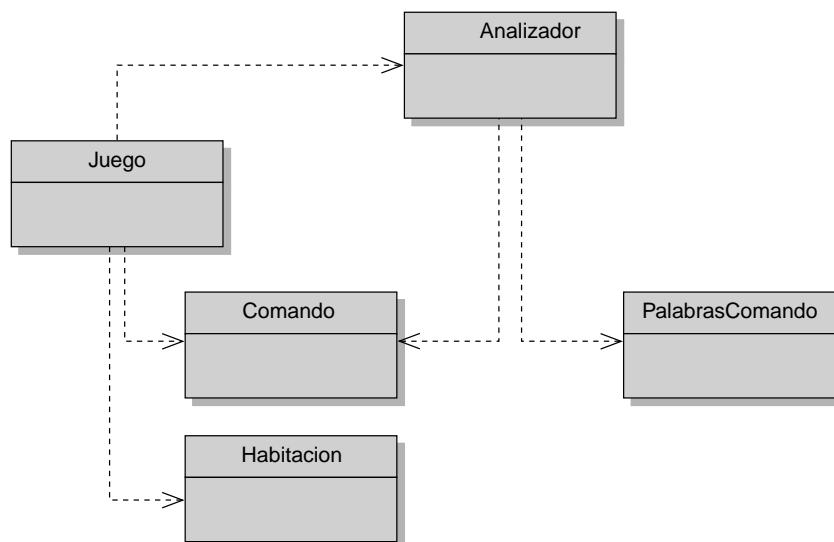
A partir del Ejercicio 7.2 habrá notado que el juego *zuul* no es muy aventurero, en realidad, es bastante aburrido en su estado actual, pero nos proporciona una buena base para diseñar e implementar nuestro propio juego que esperamos sea más interesante.

Comenzamos por analizar las clases que ya están en nuestra primera versión y tratar de descubrir qué hacen. El diagrama de clases se muestra en la Figura 7.1.

El proyecto presenta cinco clases que son: *Analizador*, *PalabrasComando*, *Comando*, *Habitacion* y *Juego*. Una investigación del código muestra, afortunadamente, que estas clases están bien documentadas y podemos tener una idea global de lo que hacen con sólo leer los comentarios de las clases en la parte superior de cada una de ellas. (Este punto también sirve para ilustrar que el mal diseño está vinculado con algo más profundo que la manera en que aparecen las clases o lo bien documentadas que estén.) Nuestra comprensión del juego se apoyará en la lectura del código para ver qué métodos tiene cada clase y qué parecen hacer. Resumimos aquí el propósito de cada clase:

**Figura 7.1**

Diagrama de clases  
de Zuul



- **PalabrasComando** Esta clase define todos los comandos válidos del juego mediante un arreglo de cadenas que contiene las palabras que se usarán como comandos.
- **Analizador** El analizador lee líneas de entrada desde la terminal y trata de interpretarlas como comandos. Crea objetos de clase **Comando** que representan el comando que ingresó el usuario.
- **Comando** Un objeto **Comando** representa un comando ingresado por el usuario. Tiene métodos que nos permiten controlar fácilmente si el comando es válido y tomar la primera y la segunda palabras del comando como cadenas independientes.
- **Habitacion** Un objeto **Habitacion** representa una ubicación en el juego. Las habitaciones deben tener salidas que conducen a otras habitaciones.
- **Juego** La clase **Juego** es la clase principal del programa. Establece el inicio del juego e ingresa en un ciclo de lectura y ejecución de comandos. También contiene el código que implementa cada comando de usuario.

**Ejercicio 7.3** Diseñe su propio escenario sin usar la computadora. No piense en la implementación, en las clases o en la programación en general, sólo piense en inventar un juego interesante. Este diseño podría concretarse en grupo.

El juego puede ser cualquiera que se base en la estructura de un jugador moviéndose a través de diferentes ubicaciones. Acá hay algunos ejemplos:

- Usted es un glóbulo blanco viajando por el cuerpo en busca de ataques de virus...
- Usted está perdido en un centro comercial y debe encontrar la salida...
- Usted es un topo en su madriguera y no puede recordar dónde almacenó su reserva de alimento antes de que llegue el invierno...
- Usted es un aventurero que busca un calabozo lleno de monstruos y otros personajes...
- Usted es del escuadrón antibombas y debe encontrar y desactivar una bomba antes de que explote...

Asegúrese de que su juego tenga un objetivo, de modo que tenga un final y que el jugador pueda «ganar». Pruebe pensar en distintas cosas que hagan que el juego se vuelva interesante: trampas, elementos mágicos, personajes que lo ayudarán sólo si los alimenta, límites de tiempo, cualquier cosa que se le ocurra. Deje fluir su imaginación.

En esta etapa no se preocupe sobre cómo implementará estas cosas.

## 7.3

# Introducción al acoplamiento y a la cohesión

Si tenemos que justificar nuestra afirmación de que algunos diseños son mejores que otros, necesitamos definir algunos términos que nos permitan discutir los puntos que consideramos importantes en el diseño de clases. Dos términos son centrales cuando hablamos sobre la calidad de un diseño de clases: *acoplamiento* y *cohesión*.

### Concepto

El término **acoplamiento** describe la interconectividad de las clases. Nos esforzamos por lograr acoplamiento débil en un sistema, es decir, un sistema en el que cada clase es altamente independiente y se comunica con otras clases mediante una pequeña interfaz bien definida.

### Concepto

El término **cohesión** describe cuánto se ajusta una unidad de código a una tarea lógica o a una entidad. En un sistema altamente cohesivo cada unidad de código (método, clase o módulo) es responsable de una tarea bien definida o de una entidad. Un diseño de clases de buena calidad exhibe un alto grado de cohesión.

El término *acoplamiento* se refiere a la interconectividad de las clases. Ya hemos discutido en capítulos anteriores que apuntamos a diseñar nuestras aplicaciones como un conjunto de clases cooperativas que se comunican mediante sus interfaces bien definidas. El grado de acoplamiento indica cuán fuertemente están conectadas estas clases. Nos esforzamos por lograr un grado bajo de acoplamiento o *acoplamiento débil*.

El grado de acoplamiento determina el grado de dificultad de realizar modificaciones en una aplicación. En una estructura de clases fuertemente acopladas, un cambio en una clase hace necesario también cambiar otras varias clases. Este hecho es el que tratamos de evitar porque el efecto de hacer un pequeño cambio puede rápidamente propagarse a la aplicación completa. Además, encontrar todos los lugares en que resulta necesario hacer los cambios y realmente llevar a cabo estos cambios puede ser difícil y demorar demasiado tiempo.

Por otro lado, en un sistema débilmente acoplado, podemos con frecuencia modificar una clase sin tener que realizar cambios en ninguna otra y la aplicación continúa funcionando. Discutiremos ejemplos particulares de acoplamiento fuerte y débil en este capítulo.

El término *cohesión* se relaciona con el número y la diversidad de tareas de las que es responsable una sola unidad de la aplicación. La cohesión es relevante para unidades formadas por una sola clase y para métodos individuales.<sup>1</sup>

Idealmente, una unidad de código debiera ser responsable de una tarea cohesiva, es decir, una tarea que pueda ser vista como una unidad lógica. Un método debiera implementar una operación lógica y una clase debiera representar un tipo de entidad. La razón principal que subyace al principio de cohesión es la reusabilidad: si un método de una clase es responsable de una única cosa bien definida es más probable que pueda ser usado nuevamente en un contexto diferente. Una ventaja complementaria, consecuencia de este principio, es que, cuando se requiere un cambio de un aspecto de una aplicación, probablemente encontraremos todas las piezas de código relevantes ubicadas en la misma unidad.

<sup>1</sup> Algunas veces usamos el término *módulo* (o *paquete* en Java) para referirnos a unidades de varias clases. La cohesión también es relevante en este nivel.

Discutiremos la influencia de la cohesión en la calidad del diseño de clases mediante los ejemplos que siguen.

**Ejercicio 7.4** Dibuje en papel el mapa del juego que inventó en el Ejercicio 7.3. Abra el proyecto *zuul-malo* y grábelo con un nombre diferente (por ejemplo, con el nombre *zuul*). Este proyecto es el que usará para realizar las mejoras y las modificaciones a lo largo de este capítulo. Puede dejar de lado el sufijo *malo* ya que muy pronto dejará de serlo (es lo que esperamos).

Como un primer paso, modifique el método `crearHabitaciones` de la clase *Juego* para crear las habitaciones y las salidas que inventó para su propio juego. ¡Pruébelo!

## 7.4

### Duplicación de código

#### Concepto

La **duplicación de código**, es decir, tener el mismo segmento de código en una aplicación más de una vez, es una señal de mal diseño y debe ser evitada.

#### Código 7.1

Secciones seleccionadas de la clase *Juego* (mal diseñada)

```
public class Juego
{
    // ... se omitió parte de código
    private void crearHabitaciones()
    {
        Habitacion exterior, teatro, bar, laboratorio,
        oficina;

        // crea las habitaciones
        exterior = new Habitacion(
            "el exterior de la entrada
        principal a la universidad");
        teatro = new Habitacion("en el anfiteatro");
        bar = new Habitacion("en el bar del campus");
        laboratorio = new Habitacion("en el laboratorio
        de computación");
        oficina = new Habitacion(
            "en la oficina del director
        de computación");

        // inicializa las salidas de las habitaciones
        exterior.establecerSalidas(null, teatro,
        laboratorio, bar);
    }
}
```

**Código 7.1  
(continuación)**

Secciones seleccionadas de la clase Juego (mal diseñada)

```
teatro.establecerSalidas(null, null, null, exterior);
bar.establecerSalidas(null, exterior, null, null);
laboratorio.establecerSalidas(exterior, oficina,
null, null);
oficina.establecerSalidas(null, null, null,
laboratorio);
habitacionActual = exterior; // el juego arranca desde afuera
}
// ... se omitió parte del código ...

/**
 * Imprime el mensaje de apertura para el jugador.
 */
private void imprimirBienvenida()
{
    System.out.println();
    System.out.println("Bienvenido a World of Zuul!");
    System.out.println(
        "Zuul es un nuevo e increíblemente
aburrido juego de aventuras.");
    System.out.println("Escriba 'ayuda' cuando la
necesite.");
    System.out.println();
    System.out.println("Usted está en " +
habitacionActual.getDescripcion());
    System.out.print("Salidas: ");
    if(habitacionActual.salidaNorte != null)
        System.out.print("norte ");
    if(habitacionActual.salidaEste != null)
        System.out.print("este ");
    if(habitacionActual.salidaSur != null)
        System.out.print("sur ");
    if(habitacionActual.salidaOeste != null)
        System.out.print("oeste ");
    System.out.println();
}
// ... se omitió parte del código ...

/**
 * Tratar de ir en otra dirección. Si existe una
salida, entra en la
 * nueva habitación, en caso contrario imprime un
mensaje de error.
 */
private void irAHabitacion(Comando comando)
{
    if(!comando.tieneSegundaPalabra()) {
        // Si no hay segunda palabra no sabemos
a dónde ir...
```

**Código 7.1  
(continuación)**

Secciones

seleccionadas de la clase Juego (mal diseñada)

```
        System.out.println("¿A dónde quiere ir?");
        return;
    }
    String direccion = comando.getSegundaPalabra();
    // Tratar de salir de la habitación actual.
    Habitacion siguienteHabitacion = null;
    if(direccion.equals("norte")){
        siguienteHabitacion =
            habitacionActual.salidaNorte;
    }
    if(direccion.equals("este")){
        siguienteHabitacion =
            habitacionActual.salidaEste;
    }
    if(direccion.equals("sur")){
        siguienteHabitacion =
            habitacionActual.salidaSur;
    }
    if(direccion.equals("oeste")){
        siguienteHabitacion =
            habitacionActual.salidaOeste;
    }
    if (siguienteHabitacion == null){
        System.out.println("¡No hay ninguna
puerta!");
    }
    else {
        habitacionActual = siguienteHabitacion;
        System.out.println("Usted está en " +
            habitacionActual.getDescripcion());
        System.out.print("Salidas: ");
        if(habitacionActual.salidaNorte != null){
            System.out.print("norte ");
        }
        if(habitacionActual.salidaEste != null){
            System.out.print("este ");
        }
        if(habitacionActual.salidaSur != null){
            System.out.print("sur ");
        }
        if(habitacionActual.salidaOeste != null){
            System.out.print("oeste ");
        }
        System.out.println();
    }
}
// ... se omitió parte del código ...
}
```

Ambos métodos, `imprimirBienvenida` e `irAHabitacion` contienen las siguientes líneas de código:

```
System.out.println("Usted está en " +
habitacionActual.getDescripcion());
System.out.print("Salidas: ");
if(habitacionActual.salidaNorte != null){
    System.out.print("norte ");
}
if(habitacionActual.salidaEste != null){
    System.out.print("este ");
}
if(habitacionActual.salidaSur != null){
    System.out.print("sur ");
}
if(habitacionActual.salidaOeste != null){
    System.out.print("oeste ");
}
System.out.println();
```

Generalmente, la duplicación de código es un síntoma de mala cohesión. El problema aquí radica en el hecho de que cada uno de los dos métodos en cuestión hace dos cosas: `imprimirBienvenida` imprime el mensaje de bienvenida e imprime la información sobre la ubicación actual, mientras que `irAHabitacion` modifica la ubicación actual y luego imprime información sobre la ubicación (nueva) actual.

Ambos métodos imprimen información sobre la ubicación actual pero ninguno puede llamar al otro porque cada uno de ellos, además hace otras cosas. Esto es un mal diseño.

Un diseño mejor usaría un método separado, más cohesivo, cuya única tarea sea imprimir la información sobre la ubicación actual (Código 7.2). Luego, ambos métodos, `imprimirBienvenida` e `irAHabitacion` podrían hacer llamadas a este nuevo método cuando necesiten imprimir esta información. De esta manera, se evita escribir dos veces el mismo código y cuando necesitemos hacer una modificación, lo haremos una sola vez.

### Código 7.2

`imprimirInformacionDeUbicacion` como un método separado

```
private void imprimirInformacionDeUbicacion()
{
    System.out.println("Usted está en " +
habitacionActual.getDescripcion());
    System.out.print("Salidas: ");
    if(habitacionActual.salidaNorte != null){
        System.out.print("norte ");
    }
    if(habitacionActual.salidaEste != null){
        System.out.print("este ");
    }
    if(habitacionActual.salidaSur != null){
        System.out.print("sur ");
    }
    if(habitacionActual.salidaOeste != null){
        System.out.print("oeste ");
    }
    System.out.println();
}
```

**Ejercicio 7.5** Implemente y use el método `imprimirInformacionDeUbicacion` en su proyecto, tal como lo discutimos en esta sección. Pruebe sus cambios.

## 7.5

# Hacer extensiones

El proyecto *zuul-malo* funciona, podemos ejecutarlo y realiza correctamente todo lo que tiene intención de hacer; sin embargo, en algunos aspectos está mal diseñado. Un buen diseño alternativo realizaría las tareas de la misma manera, pero con sólo ejecutar el programa no notaríamos ninguna diferencia.

Sin embargo, una vez que tratemos de realizar modificaciones al proyecto, notaremos diferencias significativas en la cantidad de trabajo que requiere hacer cambios en un código mal diseñado, en comparación con cambios en una aplicación bien diseñada. Investigaremos este tema haciendo algunos cambios en el proyecto. Mientras tanto, discutiremos ejemplos de diseños de mala calidad cuando los encontraremos en el código existente, y mejoraremos el diseño de clases antes de implementar nuestras extensiones.

### 7.5.1 La tarea

La primer tarea que intentaremos llevar a cabo será agregar una nueva dirección de movimiento. Actualmente, un jugador puede moverse en cuatro direcciones: *norte*, *este*, *sur* y *oeste*. Queremos permitir construcciones de varios niveles (como sótanos, bodegas, calabozos, o cualquier cosa que desee agregar más adelante en su juego) y agregar como posibles direcciones *arriba* y *abajo*. Por ejemplo, un jugador podría escribir "ir abajo" para desplazarse hacia un sótano.

### 7.5.2 Encontrar el código relevante

Una inspección a las clases dadas nos muestra que por lo menos dos clases están involucradas en este cambio: *Habitacion* y *Juego*.

*Habitacion* es la clase que almacena (además de otras cosas) las salidas de cada una de las habitaciones y, tal como vemos en el Código 7.1, en la clase *Juego* se usa la información de la salida de la habitación actual para imprimir o mostrar la información sobre las salidas y moverse de un lugar a otro.

La clase *Habitacion* es bastante breve. Su código se muestra en Código 7.3. Al leer el código podemos ver que las salidas se mencionan en dos lugares diferentes: se listan como campos en la parte superior de la clase y se asignan en el método `establecerSalidas`. Para agregar dos direcciones nuevas necesitaremos agregar dos nuevas salidas en estos dos lugares (`salidaArriba` y `salidaAbajo`).

Da un poco más de trabajo encontrar todos los lugares relevantes en la clase *Juego*. El código es un poco más largo (aquí no se muestra completo) y encontrar todos los lugares relevantes requiere más paciencia y cuidado.

La lectura del código que se muestra en Código 7.1 nos permite ver que la clase *Juego* hace uso intenso de la información sobre las salidas de una habitación. El objeto *Juego* contiene una referencia a una habitación mediante la variable *habitacionActual* y accede frecuentemente a la información de las salidas de esta habitación:

- En el método `crearHabitaciones` se definen las salidas.
- En el método `imprimirBienvenida`, se imprimen las salidas de la habitación actual para que el jugador sepa dónde ir cuando comience el juego.
- En el método `irAHabitacion` se usan las salidas para encontrar la siguiente habitación. Luego se las usa nuevamente para imprimir las salidas de la habitación siguiente a la que ya hemos ingresado.

Si ahora queremos agregar dos direcciones de salida nuevas, tendremos que agregar las opciones *arriba* y *abajo* en todos estos lugares. De cualquier manera, lea la siguiente sección antes de hacerlo.

### Código 7.3

Código de la clase  
Habitacion (mal  
diseñada)

```
public class Habitacion
{
    public String descripcion;
    public Habitacion salidaNorte;
    public Habitacion salidaSur;
    public Habitacion salidaEste;
    public Habitacion salidaOeste;
    /**
     * Crea una habitación descrita por "descripcion".
     * Inicialmente,
     * la habitación no tiene salidas. "descripcion" es
     algo así como
     * "una cocina" o "un patio".
     */
    public Habitacion(String descripcion)
    {
        this.descripcion = descripcion;
    }
    /**
     * Define las salidas de esta habitación. Cada
     dirección conduce a
     * otra habitación o bien es null (es decir, no
     hay salida).
     */
    public void establecerSalidas(Habitacion norte,
        Habitacion este,
        Habitacion sur, Habitacion oeste)
    {
        if(norte != null){
            salidaNorte = norte;
        }
        if(este != null){
            salidaEste = este;
        }
        if(sur != null){
            salidaSur = sur;
        }
    }
}
```

**Código 7.3  
(continuación)**

Código de la clase  
Habitacion (mal  
diseñada)

```

        if(oeste != null){
            salidaOeste = oeste;
        }
    }
    /**
     * Devuelve la descripción de la habitación (una de
     las que se
     * definieron en el constructor).
     */
    public String getDescripcion()
    {
        return descripcion;
    }
}

```

## 7.6

## Acoplamiento

El hecho de que existan tantas habitaciones en las que se enumeran todas sus salidas es un síntoma de un diseño de clases pobre. En la clase Habitacion, cuando se declaran las variables para las salidas necesitamos listar una variable para cada una de las salidas; en el método establecerSalidas existe una sentencia condicional por cada salida; en el método irAHabitacion hay una sentencia condicional para cada salida; en el método imprimirInformacionDeUbicacion existe una sentencia condicional para cada salida, y así sucesivamente. Esta decisión de diseño ahora nos genera bastante trabajo: cuando agregamos nuevas salidas necesitamos encontrar todos estos lugares y agregar dos nuevos casos. ¡Imagine el efecto que tendría si hubiéramos decidido usar direcciones tales como noroeste, suroeste, etc.!

Para mejorar la situación, en lugar de usar variables independientes para almacenar las salidas, decidimos usar un `HashMap`. Con esta decisión, estaremos capacitados para escribir código que pueda cubrir cualquier número de salidas y que no requiera de tantas modificaciones. El `HashMap` contendrá una correspondencia entre un nombre de dirección (por ejemplo, «norte») y la habitación a la que se llega mediante dicha dirección (un objeto `Habitacion`). Por lo tanto, cada entrada tiene una cadena como clave y un objeto `Habitacion` como valor.

Este es un cambio en la manera en que una habitación almacena internamente la información sobre las habitaciones vecinas. Teóricamente, este es un cambio que debiera afectar solamente a la *implementación* de la clase `Habitacion` (*cómo* se almacena la información de las salidas), pero no a su *interfaz* (*qué* almacenan las habitaciones).

Idealmente, cuando sólo se cambia la implementación de una clase, las restantes clases no debieran verse afectadas por el cambio. Este sería un caso de acoplamiento débil.

En nuestro ejemplo, este ideal no funciona. Si eliminamos las variables para las salidas de la clase `Habitacion` y las reemplazamos por un `HashMap`, la clase `Juego` no compilará más. Esta clase hace numerosas referencias a las variables de salidas de las habitaciones, que podrían causar errores.

Vemos que tenemos aquí un caso de acoplamiento alto. En función de limpiar esta situación, desacoplaremos estas clases antes de introducir el Hashmap.

### 7.6.1 Usar encapsulamiento para reducir el acoplamiento

Uno de los principales problemas de este ejemplo es el uso de campos públicos. Todos los campos de la clase Habitacion para las salidas han sido declarados como públicos. Claramente, el programador de esta clase no siguió los lineamientos que hemos establecido anteriormente en este libro («¡Nunca usar campos públicos!»). ¡Ya vemos el resultado! En este ejemplo, la clase Juego puede acceder directamente a estos campos (y hace un uso extensivo de este hecho). Al hacer públicos estos campos, la clase Habitacion ha expuesto en su interfaz no sólo el hecho de que tiene salidas sino también cómo se almacena exactamente la información de cada salida. Esto rompe uno de los principios fundamentales del diseño de clases de buena calidad: el *encapsulamiento*.

#### Concepto

El **encapsulamiento** apropiado en las clases reduce el acoplamiento y por lo tanto, lleva a un mejor diseño.

Una pauta para el encapsulamiento (ocultar la información de la implementación) sugiere que solamente la información sobre lo *que* puede hacer una clase debe estar visible desde el exterior, pero no *cómo* lo hace. Esto tiene una gran ventaja: si ninguna otra clase conoce cómo está almacenada nuestra información entonces podemos cambiar fácilmente la forma de almacenarla sin romper otras clases.

Podemos reforzar esta separación del *qué* y del *cómo* declarando los campos como privados y usando un método de acceso para acceder a ellos. Se muestra el primer paso de nuestra clase Habitacion modificada en el Código 7.4.

#### Código 7.4

Usar un método de acceso para disminuir el acoplamiento

```
public class Habitacion
{
    private String descripcion;
    private Habitacion salidaNorte;
    private Habitacion salidaSur;
    private Habitacion salidaEste;
    private Habitacion salidaOeste;
    // se omiten los métodos existentes que no se modifican
    public Habitacion getSalida (String direccion)
    {
        if(direccion.equals("norte")){
            return (salidaNorte);
        }
        if(direccion.equals("este")){
            return (salidaEste);
        }
        if(direccion.equals("sur")){
            return (salidaSur);
        }
        if(direccion.equals("oeste")){
            return (salidaOeste);
        }
    }
}
```

Una vez que se ha hecho esta modificación en la clase Habitacion necesitamos cambiar también la clase Juego. En cualquier lugar en donde se acceda a una variable de salida, ahora usaremos el método de acceso. Por ejemplo, en lugar de escribir:

```
siguienteHabitacion = habitacionActual.salidaEste;
```

ahora escribimos

```
siguienteHabitacion = habitacionActual.getSalida("este");
```

Esto también hace que una sección de la clase Juego resulte mucho más simple. En el método `irAHabitacion`, el reemplazo aquí sugerido dará por resultado el siguiente fragmento de código:

```
Habitacion siguienteHabitacion = null;
if(direccion.equals("norte")){
    siguienteHabitacion = habitacionActual.getSalida("norte");
}
if(direccion.equals("este")){
    siguienteHabitacion = habitacionActual.getSalida("este");
}
if(direccion.equals("sur")){
    siguienteHabitacion = habitacionActual.getSalida("sur");
}
if(direccion.equals("oeste")){
    siguienteHabitacion = habitacionActual.getSalida("oeste");
}
```

Este segmento de código completo ahora puede reemplazarse por:

```
Habitacion siguienteHabitacion =
habitacionActual.getSalida(direccion);
```

**Ejercicio 7.6** Realice las modificaciones que hemos descrito para las clases Habitacion y Juego.

**Ejercicio 7.7** Realice una modificación similar en el método `imprimirInformacionDeUbicacion` de la clase Juego de modo que los detalles de las salidas se preparen en la clase Habitacion en lugar de prepararse en la clase Juego. Defina un método en Habitacion con la siguiente firma:

```
/**
 * Devuelve la descripción de las salidas de la habitación,
 * por ejemplo, "Salidas: norte oeste".
 * @return La descripción de las salidas disponibles.
 */
public String getStringDeSalidas( )
```

Hasta ahora, no hemos modificado la representación de las salidas en la clase Habitacion, sólo hemos limpiado la interfaz. El *cambio* en la clase Juego es mínimo, en lugar de acceder a un campo público usamos una llamada a un método, pero la *ganancia* es enorme. Ahora podemos modificar la forma de almacenar las salidas de la habitación en la clase Juego sin necesidad de preocuparnos por romper cualquier otra clase. La representación interna en Habitacion ahora está completamente desacoplada de su interfaz y el diseño está en la forma que tendría que haber estado inicialmente, ahora resulta fácil reemplazar los campos independientes para las salidas por un HashMap. El código modificado se muestra en Código 7.5.

**Código 7.5**

Código fuente de la clase Habitacion

```
import java.util.HashMap;
// se omitió el comentario de clase
class Habitacion
{
    private String descripcion;
    private HashMap<String, Habitacion> salidas;

    /**
     * Crea un lugar descrito por "descripcion".
     Inicialmente,
     * el lugar no tiene salidas. "descripcion" es algo
     así como
     * "una cocina" o "un patio".
     */
    public Habitacion(String descripcion)
    {
        this.descripcion = descripcion;
        salidas = new HashMap<String, Habitacion>();
    }
    /**
     * Define las salidas de esta habitación. Cada
     dirección conduce a
     * otra habitación o bien es null (es decir, no
     hay salida).
     */
    public void establecerSalidas(Habitacion norte,
Habitacion este,
        Habitacion sur, Habitacion oeste)
    {
        if(norte != null)
            salidas.put("norte", norte);
        if(este != null)
            salidas.put("este", este);
        if(sur != null)
            salidas.put("sur", sur);
        if(oeste != null)
            salidas.put("oeste", oeste);
    }
    /**
     * Devuelve la habitación a la que se llega si
     vamos desde esta
     * habitación en dirección "direccion". Si no existe
     ninguna habitación
     * en esta dirección, devuelve null.
     */
    public Habitacion getSalida(String direccion)
    {
        return salidas.get(direccion);
    }
}
```

**Código 7.5  
(continuación)**

Código fuente de la clase Habitacion

```

    /**
     * Devuelve la descripción de la habitación (una
     de las que se
     * definieron en el constructor).
     */
    public String getDescripcion()
    {
        return descripcion;
    }
}

```

Merece la pena enfatizar que podemos hacer esta modificación sin tener que controlar si se produce alguna ruptura en algún otro lugar. Dado que sólo hemos cambiado los aspectos internos de la clase Habitacion, que por definición, no pueden ser usados en otras clases, esta modificación no impacta sobre otras clases. La interfaz permanece sin cambios.

Un resultado que deriva de este cambio es que nuestra clase Habitacion ahora es aún más corta. En lugar de listar cuatro variables independientes, solamente tenemos una; además, el método getSalida está considerablemente simplificado.

Recordemos que el objetivo original que nos llevó a esta serie de modificaciones fue que resulte más fácil agregar dos posibles nuevas salidas en las direcciones arriba y abajo. Ahora se ha vuelto muchísimo más fácil. Dado que usamos un HasMap para almacenar las salidas, agregar estas dos direcciones adicionales se podrá hacer sin modificar nada. Podemos también obtener la información sobre la salida mediante el método getSalida sin ningún problema.

El único lugar que tiene conocimiento sobre las cuatro salidas existentes (norte, este, sur y oeste) que está aún codificado en el fuente es el método establecerSalidas. Esta es la última parte que necesita ser mejorada. En este momento, la signatura del método es

```
public void establecerSalidas(Habitacion norte, Habitacion
este, Habitacion sur, Habitacion oeste)
```

Este método forma parte de la interfaz de la clase Habitacion de modo que cualquier cambio que hagamos en él inevitablemente afectará a algunas otras clases en virtud del acoplamiento. Es importante notar que jamás podemos desacoplar completamente las clases en una aplicación, de lo contrario, no podrían interactuar entre ellos objetos de diferentes clases. Más bien tratamos de mantener un grado de acoplamiento tan bajo como sea posible. Si, de todos modos, tenemos que hacer un cambio en el método establecerSalidas para acomodar las direcciones adicionales, nuestra solución preferida es reemplazar el método completo por este otro método:

```

    /**
     * Define una salida para esta habitación.
     * @param direccion La dirección de la salida.
     * @return vecina La habitación que se encuentra en la
     dirección dada.
    */

```

```
public void establecerSalida(String direccion, Habitacion
    vecina)
{
    salidas.put(direccion, vecina);
}
```

Ahora, se pueden establecer las salidas de esta habitación de a una por vez y se puede usar cualquier dirección de salida. En la clase Juego, el cambio que resulta de modificar la interfaz de Habitacion es el siguiente. En lugar de escribir

```
laboratorio.establecerSalidas(exterior, oficina, null, null);  
ahora escribimos
```

```
laboratorio.establecerSalida("norte", exterior);  
laboratorio.establecerSalida("este", oficina);
```

Hemos eliminado completamente la restricción de que Habitacion sólo pueda almacenar cuatro salidas. La clase Habitacion ahora está lista para almacenar las direcciones *arriba* y *abajo*, así como también cualquier otra dirección que se nos ocurra (noroeste, sureste, etc.).

**Ejercicio 7.8** Implemente los cambios descritos en esta sección en su propio proyecto *zuul*.

## 7.7

# Diseño dirigido por responsabilidades

### Concepto

**Diseño dirigido por responsabilidades** es el proceso de diseñar clases asignando responsabilidades bien definidas a cada una. Este proceso puede usarse para determinar las clases que deben implementar una parte de cierta función de una aplicación.

Hemos visto en la sección anterior que el uso apropiado del encapsulamiento reduce el acoplamiento y puede reducir significativamente la cantidad de trabajo necesaria para realizar modificaciones en una aplicación. Sin embargo, el encapsulamiento no es el único factor que influye en el grado de acoplamiento, otro aspecto se conoce como *diseño dirigido por responsabilidades*.

El diseño dirigido por responsabilidades expresa la idea de que cada clase será responsable de manejar sus propios datos. Con frecuencia, cuando necesitamos agregar nueva funcionalidad a una aplicación, necesitamos preguntarnos en qué clases debemos agregar un método para implementar esta nueva función. ¿Qué clase será responsable de la tarea? La respuesta es que la clase que es responsable de almacenar algunos datos también será responsable de manipularlos.

Un buen diseño dirigido por responsabilidades influye en el grado de acoplamiento y por consiguiente, también influye en la facilidad con que una aplicación puede ser modificada o extendida. Como es habitual, discutiremos este tema con más detalles mediante nuestro ejemplo.

### 7.7.1

#### Responsabilidades y acoplamiento

Las modificaciones de la clase Habitacion que hemos discutido en la Sección 7.6.1 hacen que ahora sea mucho más fácil agregar en la clase Juego nuevas direcciones para los movimientos arriba y abajo. Investigaremos esta cuestión con un ejemplo. Supongamos que queremos agregar una nueva habitación debajo de la oficina (el sótano). Todo lo que tenemos que hacer para lograr esto es realizar algunos pequeños

cambios en el método `crearHabitaciones` para crear la habitación y hacer dos llamadas para establecer sus salidas:

```
private void crearHabitaciones()
{
    Habitacion exterior, teatro, bar, laboratorio, oficina,
    sotano;
    ...
    sotano = new Habitacion("en el sótano");
    ...
    oficina.establecerSalida("abajo", sotano);
    sotano.establecerSalida("arriba", oficina);
}
```

Esta modificación funcionará sin problemas debido a la nueva interfaz de la clase `Habitacion`. Este cambio ahora es muy fácil y confirma que el diseño es de mejor calidad.

Se puede ver una evidencia más de esto si comparamos la versión original del método `imprimirInformacionDeUbicacion` que se muestra en Código 7.2 con el método `getStringDeSalidas` que se muestra en Código 7.6 y representa una solución al Ejercicio 7.7.

#### Código 7.6

El método

`getStringDeSalidas`  
de `Habitacion`

```
/**
 * Devuelve una cadena que describe las salidas de la
 * habitación,
 * por ejemplo "Salidas: norte oeste".
 * @return Una descripción de las salidas disponibles.
 */
public String getStringDeSalidas()
{
    String stringDeSalidas = "Salidas: ";
    if (salidaNorte != null)
        stringDeSalidas += "norte ";
    if (salidaEste != null)
        stringDeSalidas += "este ";
    if (salidaSur != null)
        stringDeSalidas += "sur ";
    if (salidaOeste != null)
        stringDeSalidas += "oeste";
    return stringDeSalidas;
}
```

Dado que la información sobre las salidas ahora se almacena solamente en la habitación propiamente dicha, la habitación es responsable de aportar esa información. La habitación puede realizar esta tarea mucho mejor que cualquier otro objeto ya que tiene todo el conocimiento sobre la estructura del almacenamiento interno de los datos de las salidas. Ahora, dentro de la clase `Habitacion` podemos partir de saber que las salidas están almacenadas en un `HashMap` y recorrerlo para describir todas las salidas de cada habitación.

En consecuencia, reemplazamos la versión del método `getStringDeSalidas` que se muestra en Código 7.6 por la versión que aparece en Código 7.7. Este método busca en el `HashMap` todos los nombres de las salidas (las llaves del `HashMap` son los nombres de las salidas) y los concatena para obtener una sola cadena, que finalmente es la que retorna como resultado. (Para este trabajo necesitamos importar las clases `Set` e `Iterator`.)

### Código 7.7

Una versión revisada  
de  
`getStringDeSalidas`

```
/** Devuelve una cadena que describe las salidas de la
habitación,
 * por ejemplo "Salidas: norte oeste".
 * @return La descripción de las salidas disponibles.
 */
public String getStringDeSalidas()
{
    String stringADevolver = "Salidas: ";
    Set<String> llaves = salidas.keySet();
    for (String salida : llaves)
        stringADevolver += " " + salida;
    return stringADevolver;
}
```

**Ejercicio 7.9** Busque el método `keySet` en la documentación del `HashMap`. ¿Qué función cumple este método?

**Ejercicio 7.10** Explique detalladamente y por escrito el funcionamiento del método `getStringDeSalidas` que se muestra en Código 7.7.

Nuestro objetivo de reducir el acoplamiento demanda que, tanto como sea posible, los cambios en la clase `Habitacion` no requieran cambios en la clase `Juego`. Aún podemos mejorar este punto.

Actualmente, y de acuerdo con el código, la clase `Juego` aún sabe que la información que queremos sobre una habitación consiste en una cadena para la descripción y en una cadena para todas las salidas posibles:

```
System.out.println("Ud. está " +
habitacionActual.getDescripcion());
System.out.println(habitacionActual.getStringDeSalidas());
```

¿Qué pasa si agregamos otros elementos en las habitaciones de nuestro juego, como por ejemplo, monstruos o más jugadores?

Cuando describimos lo que vemos, la lista de elementos como monstruos y otros jugadores debiera incluirse en la descripción de la habitación. En estos casos, necesitaríamos no sólo modificar la clase `Habitacion` para agregar estos elementos sino también realizar los cambios correspondientes en el segmento de código anterior que imprime la descripción.

Esto es nuevamente una consecuencia de la regla del diseño dirigido por responsabilidades: dado que la clase `Habitacion` contiene información sobre una habitación,

también debe encargarse de generar una descripción de cada habitación. Podemos mejorar este punto agregando a la clase Habitacion el siguiente método:

```
/**
 * Devuelve una larga descripción de esta habitación, en la
 * forma:
 *           Ud. está en la cocina.
 *           Salidas: norte, oeste
 *           * @return La descripción de la habitación que
 * incluye sus salidas.
 */
public String getDescripcionLarga()
{
    return "Ud. está " + descripcion + ".\n" +
getStringDeSalidas();
}
```

Luego, en la clase Juego escribimos

```
System.out.println(habitacionActual.getDescripcionLarga());
```

La «descripción larga» de una habitación ahora incluye la cadena de descripción, información sobre las salidas y podría, en el futuro, incluir cualquier otra cosa que haya que decir sobre una habitación. Cuando realicemos estas futuras extensiones tendremos que hacer cambios solamente en una única clase: en la clase Habitacion.

**Ejercicio 7.11** Implemente los cambios descritos en esta sección en su propio proyecto *zuul*.

**Ejercicio 7.12** Dibuje un diagrama de objetos con todos los objetos de su juego, en la forma en que se encuentran exactamente cuando se inicia el juego.

**Ejercicio 7.13** ¿Qué se modifica en el diagrama de objetos cuando se ejecuta el comando **ir**?

## 7.8

## Localización de cambios

Otro aspecto de los principios de desacoplamiento y de responsabilidades se refiere a la *localización de los cambios*. Apuntamos a crear un diseño de clases que facilite las modificaciones posteriores mediante la ubicación de los efectos de un cambio determinado.

Idealmente, debe cambiarse una única clase para realizar una modificación. Algunas veces, es necesario cambiar varias clases, pero apuntamos a que el cambio afecte a la menor cantidad de clases posible. Además, los cambios que requieran las otras clases debieran ser obvios, fáciles de detectar y fáciles de llevar adelante.

En los proyectos grandes, logramos este objetivo siguiendo las reglas de diseño de buena calidad tales como usar diseño dirigido por responsabilidades y apuntar a un bajo acoplamiento y a una alta cohesión. Además, como siempre, debemos tener en mente la modificación y la extensión cuando creamos nuestras aplicaciones. Es importante anticipar que un aspecto de nuestro programa podría cambiar en vías de que resulte más sencillo implementar este cambio.

### Concepto

Uno de los principales objetivos de un diseño de clases de buena calidad es la **localización de los cambios**: las modificaciones en una clase debieran tener efectos mínimos sobre las otras clases.

## 7.9

## Acoplamiento implícito

Hemos visto que el uso de campos públicos es una práctica que probablemente crea un gran acoplamiento entre las clases. Con este denso acoplamiento, puede ser necesario hacer cambios en más de una clase para algo que podría ser una simple modificación. Por lo tanto, los campos públicos deben evitarse. Sin embargo, existe aún una forma peor de acoplamiento: el *acoplamiento implícito*.

El acoplamiento implícito es una situación en la que una clase depende de la información interna de otra pero esta dependencia no es inmediatamente obvia. El denso acoplamiento en el caso de los campos públicos no era bueno, pero por lo menos era obvio. Si cambiamos los campos públicos en una clase y nos olvidamos de otra, la aplicación no compilará más y el compilador indicará el problema. En los casos de acoplamiento implícito, el omitir un cambio necesario puede no ser detectado.

Podemos ver el problema que surge si tratamos de agregar más palabras para usar como comandos del juego.

Supongamos que queremos agregar el comando *ver* al conjunto de comandos válidos. El propósito de *ver* es simplemente mostrar nuevamente la descripción de la habitación y las salidas posibles («examinamos la habitación para *ver* qué hay»). Este comando podría ser útil si hemos ingresado una secuencia de comandos en una habitación y la descripción ha quedado fuera del alcance de la vista, y no podemos recordar dónde están las salidas de la habitación actual.

Podemos introducir una nueva palabra comando agregándola simplemente al arreglo de palabras conocidas, es decir, en el arreglo `comandosValidos` de la clase `PalabrasComando`:

```
// un arreglo constante que contiene todas las palabras
// comando válidas
private static final String comandosValidos[] = {
    "ir", "salir", "ayuda", "ver"
};
```

De paso, esto muestra un ejemplo de buena cohesión: en lugar de definir las palabras comando en el analizador, que podría haber sido una posibilidad obvia, el autor creó una clase independiente sólo para definir las palabras que se usan como comandos. Esto hace que ahora nos resulte muy fácil buscar el lugar en que están definidas las palabras comando y también es fácil agregar una nueva. El autor obviamente tuvo en mente que se podrían agregar comandos más adelante y creó una estructura que hace que resulte muy fácil agregarlos.

Ya podemos probarlo. Sin embargo, después de hacer esta modificación, cuando ejecutamos el juego y escribimos el comando *ver*, no ocurre nada. Esto contrasta con el comportamiento de una palabra comando desconocida: si escribimos cualquier palabra desconocida vemos la respuesta

No sé qué significa...

Por lo tanto, el hecho de que no veamos esta respuesta indica que la palabra fue reconocida, pero no ocurre nada porque aún no hemos implementado una acción para este comando.

Podemos solucionar este problema agregando un método para el comando *ver* en la clase Juego:

```
private void ver()
{
    System.out.println(habitacionActual.getDescripcionLarga());
```

Después de agregar este método, sólo necesitamos agregar un caso más para el comando *ver* en el método *procesarComando* que invocará al método *ver* cuando este comando sea reconocido:

```
if (palabraComando.equals("ayuda")) {
    imprimirAyuda();
}
else if (palabraComando.equals("ir")) {
    irAHabitacion(commando);
}
else if (palabraComando.equals("ver")) {
    ver();
}
else if (palabraComando.equals("salir")) {
    quiereSalir = salir(commando);
}
```

Pruebe este código y verá que funciona.

**Ejercicio 7.14** Agregue el comando *ver* en su propia versión del juego *zuul*.

**Ejercicio 7.15** Agregue otro comando a su juego. Para empezar, puede elegir algo simple tal como un comando *comer* que, cuando se ejecute, imprima «*Ya ha comido y no tiene más hambre*». Más adelante, podremos mejorarlo de modo que, por ejemplo, tenga hambre y necesite encontrar comida.

El acoplamiento entre las clases Juego, Analizador y PalabrasComando parece ser bueno, resultó fácil realizar esta extensión y rápidamente lo tenemos funcionando.

El problema que mencionamos antes, acoplamiento implícito, se torna evidente cuando usamos el comando *ayuda*. La salida en pantalla es

Está perdido. Está solo. Vagabundea  
por la universidad.

Sus palabras comando son:  
ir salir ayuda

Ahora observamos un pequeño problema: el texto de la ayuda está incompleto, el nuevo comando *ver* no está en la lista.

Este problema parece fácil de solucionar: podemos editar el texto de la cadena de ayuda en el método *imprimirAyuda* de Juego. Esto se hace rápidamente y no parece ser un gran problema pero, suponga que no hubiéramos notado este error ahora. ¿Pensó en este problema antes de que lo mencionáramos?

Este es un problema fundamental porque cada vez que se agregue un comando, el texto de la ayuda necesita ser cambiado y es muy fácil olvidarse de hacer este cambio. El programa compila y ejecuta y todo parece estar bien. Un programador de manteni-

miento podría bien creer que el trabajo está terminado y liberar un programa que ahora contiene un fallo.

Este es un ejemplo de acoplamiento implícito. Cuando los comandos cambian, el texto de ayuda debe ser modificado (acoplamiento) pero nada en el programa fuente indica claramente esta dependencia (por lo que es implícita).

Un buen diseño de clases evitará esta forma de acoplamiento siguiendo la regla de diseño dirigido por responsabilidades: dado que la clase PalabrasComando es responsable de las palabras que usan como comandos del juego, también debe ser responsable de imprimirlas. Por lo tanto, agregamos el siguiente método en la clase PalabrasComando:

```
/**  
 * Imprime todos los comandos válidos en System.out.  
 */  
public void mostrarTodos()  
{  
    for (String comando : comandosValidos) {  
        System.out.print(comando + " ");  
    }  
    System.out.println();  
}
```

La idea aquí es que el método imprimirAyuda de la clase Juego, en lugar de imprimir un texto fijo con las palabras comando, invoque a un método que le solicita a la clase PalabrasComando que imprima todas sus palabras comando. Hacer esto asegura que las palabras comando correctas siempre serán impresas y al agregar un nuevo comando, también se agregará en el texto de ayuda sin hacer ningún otro cambio.

El único problema que resta es que el objeto Juego no contiene una referencia al objeto PalabrasComando. Puede ver en el diagrama de clases (Figura 7.1) que no hay ninguna flecha desde Juego hacia PalabrasComando y esto indica que la clase Juego aún no conoce la existencia de la clase PalabrasComando. En cambio, el juego justamente tiene un analizador y el analizador hace referencia a las palabras comando.

Ahora podríamos agregar un método en el analizador, que maneja el objeto PalabrasComando para el objeto Juego, de modo que puedan comunicarse. Sin embargo, esto podría incrementar el grado de acoplamiento en nuestra aplicación: Juego dependería de PalabrasComando, cosa que actualmente no ocurre. Podríamos ver el efecto en el diagrama de clases: Juego tendría una flecha hacia PalabrasComando.

De hecho, las flechas en el diagrama son un buen primer indicador del grado de intensidad del acoplamiento de un programa: cuanto más flechas, más acoplamiento. Como una aproximación a un buen diseño de clases podemos apuntar a crear diagramas con pocas flechas.

Por lo tanto, el hecho de que Juego no tuviera una referencia a PalabrasComando ¡es bueno! No debemos cambiar esto. Desde el punto de vista de Juego, el que exista la clase PalabrasComando es un detalle de implementación del analizador. El analizador devuelve comandos y si usa un objeto PalabrasComando para lograr este objetivo o alguna otra cosa, se deja por completo en manos de la implementación del analizador.

Se desprende que un mejor diseño permitiría que Juego hable con el Analizador, quien en su debido turno puede hablar con PalabrasComando. Podemos implementar

esta idea agregando el siguiente código en el método `imprimirAyuda` dentro de `Juego`:

```
System.out.println("Las palabras comando son: ");
analizador.mostrarComandos();
```

Luego, todo lo que falta es el método `mostrarComandos` del Analizador que delega esta tarea a la clase `PalabrasComando`. Aquí está el método completo (en la clase `Analizador`):

```
/**
 * Imprimir una lista de palabras comando válidas
 */
public void mostrarComandos()
{
    comandos.mostrarTodos(),
}
```

**Ejercicio 7.16** Implemente la versión mejorada para imprimir las palabras comando, tal como se describió en esta sección.

**Ejercicio 7.17** Si ahora agregara un nuevo comando, ¿necesitaría todavía cambiar la clase `Juego`? ¿Por qué?

La implementación completa de todos los cambios discutidos hasta ahora, en este capítulo, está disponible en los ejemplos de código en un proyecto de nombre `zuul-mejorado`. Si ha realizado todos los ejercicios, puede ignorar este proyecto y continuar usando el propio. Si no ha resuelto los ejercicios pero quiere hacer los siguientes ejercicios de este capítulo como un proyecto de programación, puede usar como punto de partida el proyecto `zuul-mejorado`.

## 7.10

## Pensar en futuro

El diseño que ahora tenemos implementado contiene importantes mejoras con respecto a la versión original, sin embargo, todavía es posible mejorarlo más.

Una característica de un buen diseñador de software es la habilidad de pensar en el futuro. ¿Qué podría cambiar? ¿Qué podemos asumir con seguridad que permanecerá sin cambios durante la vida del programa?

La presunción que hemos codificado fuertemente en nuestras clases es que este juego se manejará mediante entradas y salidas de texto en la terminal. Pero, ¿siempre será así?

Más adelante, una extensión interesante del juego podría ser agregarle una interfaz gráfica de usuario con menús, botones e imágenes. En este caso, podríamos no querer más imprimir la información en la terminal de texto. Podríamos seguir manteniendo palabras comando y mostrarlas cuando un jugador ingrese un comando de ayuda. De ser así, podríamos mostrar la información en un campo de texto en la ventana del juego en lugar de usar `System.out.println`.

Encapsular toda la información de la interfaz de usuario en una sola clase o en un conjunto de clases claramente definido forma parte de un buen diseño. En nuestra solución y a partir de la Sección 7.9, por ejemplo, el método `mostrarTodos` de la clase `PalabrasComando` no sigue esta regla de diseño. Sería mejor definir que la clase `Pa-`

labrasComando sea la responsable de *generar* (¡pero no *imprimir!*) la lista de palabras comando, pero que la clase Juego decida cómo se presenta esta información al usuario.

Podemos lograr este objetivo fácilmente modificando el método `mostrarTodos` de manera tal que devuelva una cadena que contenga todas las palabras comando en lugar de imprimirlas directamente. (Cuando hagamos esta modificación, probablemente podríamos renombrar este método como `getListaDeComandos`.) Luego, esta cadena puede imprimirse en el método `imprimirAyuda` de la clase Juego.

Observe que este cambio no nos reporta ninguna ganancia ahora, pero a partir de esta mejora en el diseño se podrían obtener beneficios en el futuro.

**Ejercicio 7.18** Implemente los cambios sugeridos. Asegúrese de que su programa continúe funcionando como lo hacía antes de estas modificaciones.

**Ejercicio 7.19** Busque información sobre el patrón *model-view-controller*.

Puede realizar una búsqueda por la web o bien usar cualquier otra fuente. ¿Cómo se relaciona con el tópico discutido aquí? ¿Qué sugiere? ¿Cómo podría aplicarse a este proyecto? (Investigue solamente la aplicación de este patrón en este proyecto. Su efectiva implementación podría ser un ejercicio avanzado de desafío.)

## 7.11

## Cohesión

Ya hemos presentado la idea de cohesión en la Sección 7.3: una unidad de código siempre debe ser responsable de una y sólo una tarea. Ahora investigaremos el principio de cohesión con mayor profundidad y analizaremos algunos otros ejemplos.

El principio de cohesión puede aplicarse a clases y a métodos: las clases deben mostrar un alto grado de cohesión y lo mismo ocurre con los métodos.

### 7.11.1

#### Cohesión de métodos

##### Concepto

**Método cohesivo:** un método cohesivo es responsable de una y sólo una tarea bien definida.

##### Código 7.8

Dos métodos con un buen grado de cohesión

```
 /**
 * Rutina principal para jugar. Ciclo que se ejecuta
 * hasta que se termine
 * de jugar.
 */
public void jugar()
{
    imprimirBienvenida();
```

**Código 7.8  
(continuación)**

Dos métodos con un buen grado de cohesión

```

    // Entra en el ciclo principal. Acá leemos
    // repetidamente los comandos
    // y se los ejecuta hasta que termine el juego.

    boolean terminado = false;
    while (! terminado) {
        Comando comando = analizador.getComando();
        terminado = procesarComando(comando);
    }
    System.out.println("Gracias por jugar. Adiós.");
}

/**
 * Imprime el mensaje de apertura para el jugador.
 */
private void imprimirBienvenida()
{
    System.out.println();
    System.out.println("Bienvenido a World of Zuul!");
    System.out.println("World of Zuul es un nuevo e
increíblemente
                                aburrido juego de
aventuras.");
    System.out.println("Escriba 'ayuda' cuando la
necesite.");
    System.out.println();

    System.out.println(habitacionActual.getDescripcionLarga());
}

```

Desde un punto de vista funcional, podríamos haber escrito las sentencias del método `imprimirBienvenida` directamente dentro del método `jugar` y así lográbamos el mismo resultado sin tener que definir otro método y realizar una llamada a este método. De paso, se puede decir lo mismo para el método `procesarComando` que también es invocado dentro del método `jugar`: este código también podría haber sido escrito directamente dentro del método `jugar`.

Sin embargo, es más fácil de comprender lo que hace un segmento de código y realizar modificaciones si se usan métodos breves y cohesivos. En la estructura de métodos que hemos elegido, todos los métodos son razonablemente cortos y fáciles de comprender y sus nombres indican sus propósitos bastante claramente. Estas características representan una ayuda valiosa para un programador de mantenimiento.

### 7.11.2 Cohesión de clases

La regla de cohesión de clases establece que cada clase debe representar una única entidad bien definida en el dominio del problema.

**Concepto**

**Clase cohesiva:** una clase cohesiva representa una única entidad bien definida.

Como un ejemplo de cohesión de clases ahora discutiremos otra extensión del proyecto *zuul*. Queremos agregar elementos al juego. Cada habitación puede contener un elemento y cada elemento tiene una descripción y un peso. El peso de un elemento podría usarse más adelante para determinar si puede ser tomado o no.

Una aproximación sencilla podría ser el agregado de dos campos en la clase *Habitacion*: *descripcionDeElemento* y *pesoDeElemento*. Esta idea podría funcionar. Ahora quisiéramos especificar los detalles de cada elemento de cada habitación e imprimir estos detalles cuando se ingresa en una habitación.

Sin embargo, este abordaje no presenta un buen grado de cohesión: la clase *Habitacion* describe tanto una habitación como un elemento de la misma, lo que también sugiere que un elemento está ligado a una habitación en particular, que podría no ser el caso.

Un mejor diseño crearía una clase separada para modelar los elementos, probablemente de nombre *Elemento*. Esta clase podría tener campos para la descripción y el peso, y una habitación podría contener simplemente una referencia a un objeto *elemento*.

**Ejercicio 7.20** Extienda su proyecto de aventuras o el proyecto *zuul-mejorado*, de modo que una habitación pueda contener un solo elemento. Los elementos tienen una descripción y un peso. Cuando se crean las habitaciones y se establecen sus salidas, también deberían crearse los elementos para este juego. Cuando un jugador entre en una habitación, debería mostrarse la información sobre el elemento presente en ella.

**Ejercicio 7.21** ¿Cómo podría generarse la información del elemento presente en una habitación? ¿Qué clase debería generar la cadena de descripción del elemento? ¿Qué clase debería imprimirla? Explique por escrito sus razonamientos. Si al responder este ejercicio siente que debería modificar su implementación, pues ¡adelante! y realice estos cambios.

El beneficio real de separar en el diseño las habitaciones de los elementos puede verse si cambiamos un poco la especificación: en una futura variante de nuestro juego, queremos permitir que cada habitación no tenga sólo un elemento sino un número ilimitado de elementos. En el diseño que usa una clase separada *Elemento* es fácil implementar este cambio: podemos crear múltiples objetos *Elemento* y almacenarlos en una colección de elementos en la habitación.

Con la primera aproximación sencilla, este cambio sería casi imposible de implementar.

**Ejercicio 7.22** Modifique el proyecto de modo que una habitación pueda tener cualquier número de elementos. Para lograrlo, use una colección. Asegúrese de que la habitación tenga un método *agregarElemento* que ubique un elemento en ella. Asegúrese de que todos los elementos se muestren cuando un jugador entra en una habitación.

### 7.11.3 Cohesión para la legibilidad

Hay varias maneras en que un diseño se ve beneficiado por la alta cohesión. Las dos más importantes son la *legibilidad* y la *reusabilidad*.

El ejemplo discutido en la Sección 7.11.1, la cohesión del método `imprimirBienvenida`, es claramente un ejemplo en el que al aumentar la cohesión, una clase se vuelve más legible y por lo tanto, más fácil de comprender y mantener.

El ejemplo de cohesión de clases en la Sección 7.11.2 también tiene un componente de legibilidad. Si existe una clase separada `Elemento`, un programador de mantenimiento fácilmente reconocerá por dónde comenzar a leer el código si necesita realizar un cambio en las características de un elemento. La cohesión de clases también incrementa la legibilidad de un programa.

#### 7.11.4 Cohesión para la reusabilidad

La segunda gran ventaja de la cohesión es el alto potencial para la reutilización.

El ejemplo de cohesión de clases de la Sección 7.11.2 también muestra un ejemplo de reusabilidad: al crear una clase separada `Elemento` podemos crear múltiples elementos y por lo tanto, usar el mismo código para más de un elemento.

La reusabilidad es otro aspecto importante de los métodos cohesivos. Considere un método en la clase `Habitacion` con la siguiente firma:

```
public Habitacion dejarHabitacion(String direccion)
```

Este método podría devolver la habitación ubicada en la dirección dada (por lo que podría usarse como la nueva `habitacionActual`) y también imprimir la descripción de la nueva habitación a la que se entra.

La inclusión de este método parece ser un diseño posible y realmente se le puede hacer funcionar. Sin embargo, en nuestra versión tenemos estas tareas en dos métodos separados:

```
public Habitacion establecerSalida(String direccion)  
public String getDescripcionLarga()
```

El primer método es responsable de devolver la siguiente habitación, mientras que el segundo genera la descripción de la misma.

La ventaja de este diseño es que las tareas separadas pueden ser reutilizadas más fácilmente. Por ejemplo, el método `getDescripcionLarga` se usa no sólo en el método `irAHabitacion` sino también en `imprimirBienvenida` y en la implementación del comando `ver`. Todo esto sólo es posible porque existe un alto grado de cohesión. Esta reutilización de código no sería posible si se hubiera diseñado el método `dejarHabitacion`.

**Ejercicio 7.23** Implemente el comando `volver`. Este comando no tiene una segunda palabra. Al escribir el comando `volver` el jugador regresa a la última habitación en que estaba.

**Ejercicio 7.24** Pruebe adecuadamente su nuevo comando. ¡No se olvide de realizar una prueba negativa! ¿Qué hace su programa si un jugador escribe una segunda palabra después del comando `volver`? ¿Tiene un comportamiento sensible a una segunda palabra? ¿Existen otros casos de pruebas negativas?

**Ejercicio 7.25** ¿Qué hace su programa si escribe dos veces `volver`? ¿Es adecuado este comportamiento?

**Ejercicio 7.26** Desafío. Implemente el comando *volver* de modo que al usarlo repetidamente haga que se retroceda varias habitaciones; en realidad, si se usa con la frecuencia necesaria, permite recorrer todo el camino desde el principio del juego. Para hacerlo, use un *Stack*. (Necesitará buscar información sobre las pilas (*stacks*). Busque en la documentación de la biblioteca de Java.)

## 7.12

# Refactorización

### Concepto

La **refactorización** es la actividad de reestructurar un diseño existente para mantener un buen diseño de clases cuando se modifica o se extiende una aplicación.

Cuando diseñamos aplicaciones, debemos tratar de pensar hacia adelante, anticipar los posibles cambios que podrían ser deseables en el futuro y crear clases altamente cohesivas y débilmente acopladas y métodos que faciliten las modificaciones. Este es un noble objetivo, pero resulta claro que no siempre podemos anticipar todas las futuras adaptaciones y que no es factible preparar un diseño que contemple todas las posibles extensiones que pensamos.

Este es el motivo por el que resulta importante la *refactorización*.

La refactorización es la actividad de reestructurar las clases y los métodos existentes con el fin de adaptarlos a los cambios de funcionalidad y de requerimientos. Es frecuente que, durante el tiempo de vida de una aplicación, se le vaya agregando funcionalidad. Un efecto común que se produce de manera colateral es el lento crecimiento de la longitud de los métodos y de las clases.

Para un programador de mantenimiento, es tentador agregar código adicional en las clases y métodos existentes. Sin embargo, el agregado de código en reiteradas ocasiones suele tener como consecuencia la disminución del grado de cohesión. Es muy probable que si se agrega más y más código a un método o a una clase, llegue un momento en el que representará más de una tarea claramente definida o más de una entidad.

La refactorización consiste justamente en repensar y rediseñar las estructuras de las clases y de los métodos. El efecto más común es que las clases se abran en dos o que los métodos se dividan en dos o más métodos; la refactorización también incluye la unión de clases o de métodos que da por resultado una sola clase o un solo método, pero este caso es menos frecuente.

### 7.12.1

#### Refactorización y prueba

Antes de proponer un ejemplo de refactorización, necesitamos reflexionar sobre el hecho de que, cuando pensamos en refactorizar un programa, generalmente nos estamos proponiendo realizar cambios potencialmente grandes en algo que ya funciona. Cuando algo se modifica existe la posibilidad de que se introduzcan errores, por lo tanto, es importante proceder cautelosamente, y antes de llevar a cabo la refactorización debemos asegurarnos de que exista un conjunto de pruebas para la versión actual del programa. Si las pruebas no existen, es prioritario crear algunas pruebas que se adecuen para implementar pruebas regresivas sobre la versión rediseñada. La refactorización debe comenzar sólo cuando existen las pruebas. Idealmente, la refactorización debe seguir dos pasos:

- El primer paso es repensar el diseño de modo que mantenga la misma funcionalidad que la versión original. En otras palabras, reestructuramos el código para

mejorar su calidad, no para cambiar o aumentar su funcionalidad. Una vez que este paso está completo, se deben ejecutar las pruebas regresivas para asegurarse de que no se hayan introducido errores no deseados.

- El segundo paso se puede dar, únicamente, una vez que se ha restablecido la funcionalidad básica en la versión refactorizada. En ese momento estamos en una posición segura como para mejorar el programa. Una vez que se ha finalizado con la refactorización, por supuesto que será necesario ejecutar las pruebas en la nueva versión.

La implementación de varios cambios al mismo tiempo (repensar y agregar nuevas características) hace que se vuelva más difícil ubicar la fuente de los problemas, cuando estos ocurren.

**Ejercicio 7.27** ¿Qué tipos de pruebas para la funcionalidad básica se podrían establecer en la versión actual del juego?

**Ejercicio 7.28** ¿Cómo podrían automatizarse las pruebas en un programa que toma datos interactivamente? ¿Es posible armar alguna especie de guión? Por ejemplo, ¿podrían almacenarse los ingresos del usuario en un archivo en lugar de ser interactivos? ¿Qué clases necesitarían modificaciones para que esto sea posible?

### 7.12.2 Un ejemplo de refactorización

A modo de ejemplo de refactorización, continuaremos con la extensión del juego que consiste en agregar nuevos elementos en las habitaciones. En la Sección 7.11.2 comenzamos con el agregado de elementos, y sugerimos una estructura tal que las habitaciones puedan contener cualquier número de elementos. Una extensión lógica de esta modificación sería que un jugador pueda recoger los elementos y trasladarlos por las distintas habitaciones. Esta es una especificación informal de nuestro próximo objetivo:

- El jugador puede tomar los elementos de la habitación actual.
- El jugador puede tomar cualquier número de elementos, pero sólo hasta un peso máximo.
- Algunos elementos no pueden ser tomados.
- El jugador puede dejar los elementos en la habitación actual.

Para llevar a cabo estos objetivos podemos hacer lo siguiente:

- Si aún no lo hemos hecho, agregamos al proyecto la clase `Elemento`. Como se discutió anteriormente, un elemento tiene una descripción (una cadena) y un peso (un entero).
- También debemos agregar un campo `nombre` en la clase `Elemento` que nos permitirá hacer referencia al elemento con un nombre más corto que su descripción. Por ejemplo, si en la habitación actual hay un libro, los valores del campo de este elemento podrían ser:

*nombre: libro*

*descripción: un libro viejo, lleno de polvo y con tapas de cuero gris*

*peso: 1200*

Cuando el jugador entra en una habitación, podemos imprimir la descripción del elemento para informarle lo que hay en ella; pero si pensamos en los comandos, será más fácil usar el nombre del elemento que su descripción. Por ejemplo, el jugador podría escribir sólo *tomar libro* para recoger el libro.

- Podemos asegurarnos de que algunos elementos no puedan seleccionarse simplemente haciéndolos muy pesados (más peso del que un jugador puede resistir). ¿O deberíamos tener otro campo lógico como por ejemplo, `puedeSerSeleccionado`? ¿Qué diseño considera que es mejor? ¿Tiene alguna importancia? Trate de responder estas cuestiones pensando en las futuras modificaciones que se podrían hacer al juego.
- Agregamos los comandos *tomar* y *dejar* para recoger y soltar los elementos. Ambos comandos tienen el nombre del elemento como segunda palabra.
- En algún lugar tenemos que agregar un campo (alguna forma de colección) para almacenar los elementos que actualmente fueron recogidos por el jugador. También tenemos que agregar un campo para el máximo peso que un jugador puede cargar, de modo que podamos verificarlo cada vez que el jugador trate de tomar un nuevo elemento. ¿Dónde debiera estar este campo? Una vez más, para tomar la decisión ayuda el hecho de pensar sobre las futuras extensiones.

La última tarea es sobre la que discutiremos ahora con más detalle en vías de ilustrar el proceso de refactorización.

La primer pregunta que nos hacemos cuando pensamos sobre la manera de permitir que los jugadores puedan cargar elementos es: ¿dónde debemos agregar los campos para los elementos cargados por el jugador y para el máximo peso? Una rápida mirada a las clases existentes muestra que la clase `Juego` es el único lugar en el que encajan estos campos. No pueden almacenarse en las clases `Habitacion`, `Elemento` o `Comando` ya que existen varias instancias de estas clases y no siempre son accesibles; tampoco tiene sentido agregarlos en las clases `Analizador` o `PalabrasComando`.

Lo que refuerza la decisión de ubicar estos cambios en la clase `Juego` es el hecho de que ya almacena la habitación actual (la información sobre dónde está el jugador en cada momento), de modo que agregar los elementos actuales (información sobre lo que el jugador tiene) parece encajar con esto bastante bien.

Este abordaje puede funcionar, sin embargo no es una solución que esté bien diseñada. La clase `Juego` ya es bastante grande, y es un buen argumento el tener en cuenta que ya contiene demasiado tal como está; agregar más cosas en ella no mejorará el diseño.

Nos preguntamos nuevamente a qué clase o a qué objeto debe pertenecer esta información. Pensando cuidadosamente sobre el tipo de información que estamos agregando (elementos recogidos, peso máximo) nos damos cuenta de que se trata de ¡información sobre un jugador! La consecuencia lógica es que creemos una clase `Jugador`, siguiendo los principios del diseño dirigido por responsabilidades. Luego, podemos agregar estos campos a la clase `Jugador` y crear un objeto `jugador` al comienzo del juego para almacenar los datos.

El campo `habitacionActual` que ya existe en la clase `Juego` también almacena información sobre el jugador: la ubicación actual del jugador. En consecuencia, también deberíamos mover este campo a la clase `Jugador`.

En este momento, al analizar la situación, es obvio que este diseño encaja mejor con el principio de diseño dirigido por responsabilidades. ¿Quién debe ser responsable de almacenar información sobre el jugador? Por supuesto, la clase Jugador.

En la versión original teníamos una sola parte de la información del jugador, la habitación actual. El hecho de que debiéramos haber tenido una clase Jugador desde el principio del diseño del juego es motivo de discusión, existen argumentos en pro y en contra. El juego hubiera estado mejor diseñado, de modo que la respuesta es afirmativa, sería mejor que hubiera existido esta clase. Pero podría considerarse como un exceso el tener una clase con un solo campo y con métodos que no hacen nada importante.

Algunas veces, hay zonas grises como ésta, en donde cualquiera de las decisiones es defendible, pero luego de agregar nuevos campos, la situación se aclara. Ahora tenemos un argumento fuerte para que exista una clase Jugador: almacenará los campos y tendrá métodos tales como `dejarElemento` y `tomarElemento` (que pueden incluir el control del peso y podrían devolver falso si no se puede cargar el elemento).

Lo que hemos hecho cuando introducimos la clase Jugador y movimos el campo `habitacionActual` desde la clase Juego hacia la clase Jugador es refactorización. Hemos reestructurado la forma en que representamos los datos para lograr un mejor diseño ante requerimientos de cambio.

Los programadores que no están tan bien entrenados como nosotros (o que son cómodos) podrían dejar el campo `habitacionActual` en el lugar en que estaba, viendo que el programa funciona igual que antes y que hacer este cambio no parece ser muy necesario. Habrían dado por terminado el trabajo con un diseño de clases un poco desordenado.

El efecto que puede tener este cambio puede verse mejor si pensamos un poco más adelante. Supongamos que queremos extender el juego para permitir varios jugadores.

Con nuestro nuevo y buen diseño, este cambio es muy fácil y rápido. Ya tenemos una clase Jugador (el Juego contiene un objeto Jugador) y es fácil crear varios objetos Jugador y almacenarlos en Juego como una colección de jugadores en lugar de almacenar un solo jugador. Cada objeto jugador podría contener su propia habitación actual, sus elementos y su peso máximo. Diferentes jugadores podrían tener también diferentes pesos máximos, abriendo el concepto amplio de tener jugadores con capacidades bastante diferentes, sus capacidades de acarrear elementos serían justamente una entre muchas posibilidades.

El programador cómodo, que deja el campo `habitacionActual` en la clase Juego, tendrá serios problemas a la hora de extender el juego para varios jugadores. Dado que el juego tiene una sola habitación actual, no pueden almacenarse fácilmente las ubicaciones actuales de varios jugadores. Generalmente, el mal diseño se nos vuelve en contra para crear más trabajo para nuestro futuro.

Una buena refactorización es tanto una manera de pensar como un conjunto de habilidades técnicas. Mientras realizamos cambios y extensiones en las aplicaciones, regularmente nos debemos preguntar si el diseño original aún representa la mejor solución. A medida que cambia la funcionalidad, también cambian los argumentos a favor o en contra sobre ciertos diseños. Lo que fue un buen diseño para una aplicación simple podría dejar de serlo cuando se agregan algunas extensiones.

Reconocer estos cambios y realizar efectivamente estas modificaciones de refactorización en el código, generalmente ahorra una gran cantidad de tiempo y de esfuerzo al final. Cuanto antes limpiemos nuestro diseño, más trabajo ahorraremos.

Debemos estar preparados para *refactorizar* métodos (convertir una secuencia de sentencias del cuerpo de un método existente en un método nuevo e independiente) y clases (tomar partes de una clase y crear una nueva clase a partir de ella). Considerar regularmente la refactorización mantiene nuestro diseño de clases limpio y finalmente, nos ahorra trabajo. Por supuesto que uno de los resultados que se puede llegar a obtener de esta refactorización y que nos puede hacer más difícil la vida ocurre cuando no probamos la versión refactorizada contra la versión original. Siempre que nos embarquemos en una tarea de refactorización mayor, es esencial asegurarnos de que existen de antemano baterías adecuadas de pruebas, y que se mantienen actualizadas a través del proceso de refactorización. Tenga presente estos puntos cuando intente hacer los siguientes ejercicios.

**Ejercicio 7.29** Refactorice su proyecto para introducir la clase *Jugador*. Un objeto *jugador* deberá almacenar como mínimo la habitación actual del jugador, pero podría almacenar también el nombre del jugador y alguna otra información.

**Ejercicio 7.30** Implemente una extensión que permita que un jugador tome un solo elemento. Esto incluye implementar dos nuevos comandos: *tomar* y *dejar*.

**Ejercicio 7.31** Extienda su implementación para permitir que un jugador cargue cualquier número de elementos.

**Ejercicio 7.32** Agregue una restricción que permita al jugador tomar elementos pero sólo hasta un peso máximo especificado. El peso máximo que un jugador puede cargar es un atributo del jugador.

**Ejercicio 7.33** Implemente un comando *elementos* que imprima todos los elementos que actualmente se han cargado y su peso total.

**Ejercicio 7.34** Agregue el elemento *galleta mágica* en una habitación. Agregue el comando *comer galleta*. Si un jugador encuentra y come la galleta mágica, aumenta el peso que puede cargar. (Podría modificar un poco esta idea para que encaje mejor con su propio escenario del juego.)

## 7.13

## Refactorización para independizarse del idioma

Una característica del juego *zuul* que aún no hemos comentado es que la interfaz de usuario está estrechamente ligada a comandos u órdenes escritos en español. Este aspecto está incluido tanto en la clase *PalabrasComando*, donde se almacena la lista de comandos válidos, como en la clase *Juego*, donde el método *procesarComando* compara explícitamente cada palabra comando con un conjunto de palabras escritas en español. Si deseamos cambiar la interfaz con el fin de permitir que los usuarios utilicen el juego en diferentes idiomas, deberíamos encontrar todos los lugares del código en donde se usan las palabras comando y cambiarlas. Este es un ejemplo de una forma de acoplamiento implícito, que hemos discutido en la Sección 7.9.

Si queremos que el programa sea independiente del idioma, la situación ideal sería que el texto real de las palabras comando se almacene en un único lugar del código y que en todas las restantes partes se haga referencia a los comandos de manera independiente del idioma. Una característica del lenguaje de programación que torna posible esta solución está dada por los *tipos enumerados* o *enumeraciones*. Exploraremos esta característica de Java mediante los proyectos *zuul-con-enumerações*.

### 7.13.1 Tipos enumerados

El Código 7.9 muestra una definición de tipo enumerado en Java, de nombre *PalabraComando*.

#### Código 7.9

Un tipo enumerado para las palabras comando

```
/*
 * Representación para todas las palabras comando válidas
 * del juego.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 2006.03.30
 */
public enum PalabraComando
{
    // Un valor para cada palabra comando, más una para
    // los comandos no
    // reconocidos.
    IR, SALIR, AYUDA, DESCONOCIDA;
}
```

En su forma más simple, una definición de un tipo enumerado consiste en una envoltura exterior que utiliza la palabra `enum` en lugar de la palabra `class`, y un cuerpo que es simplemente una lista de nombres de variables que denotan el conjunto de valores que pertenece a este tipo. Por convención, los nombres de estas variables se escriben en mayúsculas. Nunca creamos objetos de un tipo enumerado. En efecto, cada nombre dentro de la definición del tipo representa una única instancia de un tipo enumerado que ya se ha creado para usarla. Nos referimos a estas instancias de la siguiente manera: `PalabraComando.IR`, `PalabraComando.SALIR`, etc. Aunque la sintaxis que se usa es similar, es importante evitar pensar en estos valores como si fueran las constantes de clase numéricas que discutimos en la Sección 5.13. A pesar de la simplicidad de su definición, los valores del tipo enumerado son objetos propiamente dichos, por lo tanto, no son iguales que los enteros.

¿Cómo podemos usar el tipo `PalabraComando` para avanzar un paso en desacoplar la lógica del juego *zuul* de un idioma natural en particular? Una de las primeras mejoras que podemos hacer es en la siguiente serie de pruebas del método `procesarComando` de la clase `Juego`:

```
if (palabraComando.equals("ayuda")) {
    imprimirAyuda();
}
```

```

else if (palabraComando.equals("ir")) {
    irAHabitacion(comando);
}
else if (palabracomando.equals("salir")) {
    quiereSalir = salir(comando);
}

```

Si `palabraComando` se declara de tipo `PalabraComando` en lugar de tipo `String`, entonces estas líneas se pueden rescribir así:

```

if (palabraComando == PalabraComando.AYUDA) {
    imprimirAyuda();
}
else if (palabraComando == PalabraComando.IR) {
    irAHabitacion(comando);
}
else if (palabracomando == PalabraComando.SALIR)) {
    quiereSalir = salir(comando);
}

```

Ahora, sólo nos falta hacer los arreglos para los comandos que escribirá el usuario de modo que se correspondan con los respectivos valores de `PalabraComando`. Abra el proyecto `zuul-con-enumeraciones` para ver la manera en que lo hemos resuelto. El cambio más significativo se puede encontrar en la clase `PalabrasComando`, en donde, en lugar de usar un arreglo de cadenas para definir los comandos válidos, ahora usamos una correspondencia entre cadenas y objetos `PalabraComando`:

```

public PalabrasComando()
{
    comandosValidos = new HashMap<String, PalabraComando>;
    comandosValidos.put("ir", PalabraComando.IR);
    comandosValidos.put("ayuda", PalabraComando.AYUDA);
    comandosValidos.put("salir", PalabraComando.SALIR);
}

```

El comando escrito por un usuario ahora puede ser fácilmente convertido a su correspondiente valor de tipo enumerado.

**Ejercicio 7.35** Revise el código del proyecto `zuul-con-enumeraciones-v1` para ver la manera en que se usa el tipo `PalabraComando`. Las clases `Comando`, `PalabrasComando`, `Juego` y `Analizador` han sido adaptadas a partir de la versión `zull-mejorado` para acomodarse a este cambio. Verifique que el programa aún funciona como es esperable.

**Ejercicio 7.36** Agregue al juego un comando `ver`, según lo descrito en la Sección 7.9.

**Ejercicio 7.37** Traduzca el juego para que use diferentes palabras comando en lugar de `ir` y `salir` para los comandos `IR` y `SALIR`. Podrían ser palabras provenientes de un idioma real o palabras inventadas. ¿Sólo tiene que editar la clase `PalabrasComando` para funcione esta modificación?

**Ejercicio 7.38** Elija un comando diferente en lugar de `ayuda` y verifique que funcione correctamente. Después de realizar los cambios, ¿qué observa en el mensaje de bienvenida que se imprime cuando comienza el juego?

**Ejercicio 7.39** En un nuevo proyecto, defina su propio tipo enumerado de nombre Posicion con los valores SUPERIOR, MEDIO, INFERIOR.

### 7.13.2 Más desacoplamiento de la interfaz de comandos

El tipo PalabraComando nos permitió llevar a cabo un desacople importante entre el idioma de la interfaz del usuario y la lógica del juego, y es casi totalmente posible traducir los comandos a cualquier otro idioma con sólo editar la clase PalabrasComando. (En alguna etapa, también querremos traducir las descripciones de las habitaciones y otras cadenas de salida, probablemente leyéndolas de un archivo, pero dejaremos esto para más adelante.) Hay un poco más de desacoplamiento de las palabras comando que quisiéramos llevar a cabo. Actualmente, cuando se introduce un nuevo comando en el juego debemos agregar un nuevo valor a PalabraComando y una asociación entre ese valor y el texto para el usuario, en la clase PalabrasComando. Sería útil si pudiéramos hacer que el tipo PalabraComando definiera su propio contenido. En efecto, queremos mover el texto que se asocia a cada comando desde la clase PalabrasComando a la definición del tipo PalabraComando.

Java permite que las definiciones de los tipos enumerados contengan mucho más que una lista de valores de tipos. No exploraremos esta característica en detalle pero daremos una idea de lo que es posible hacer. El Código 7.10 muestra el tipo PalabraComando reforzado que parece muy similar a una definición común de clase. Este código se puede encontrar en el proyecto *zuul-con-enumerations-v2*.

#### Código 7.10

Asociación de cadenas de comandos con valores de un tipo enumerado

```
 /**
 * Representación para todas las palabras comando válidas
 * del juego
 * junto con una cadena en un idioma en particular.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 2006.03.30
 */
public enum PalabraComando
{
    // Un valor para cada palabra comando junto con la
    // cadena
    // correspondiente a la interfaz de usuario.
    IR("ir"), SALIR("salir"), AYUDA("ayuda"), DESCONOCIDA("?");
    // La cadena comando
    private String cadenaComando;
    /**
     * Inicializar con la palabra comando correspondiente.
     * @param cadenaComando La cadena comando.
     */
    PalabraComando(String cadenaComando)
    {
        this.cadenaComando = cadenaComando;
    }
}
```

**Código 7.10  
(continuación)**

Asociación de cadenas de comandos con valores de un tipo enumerado

```

    /**
     * @return La palabra comando como una cadena.
     */
    public String toString()
    {
        return cadenaComando;
    }
}

```

Los puntos principales a tener en cuenta en esta nueva versión de PalabraComando son:

- Cada valor está seguido por un parámetro; en este caso el texto del comando asociado con ese valor.
- La definición del tipo incluye un constructor que no tiene la palabra `public` en su encabezado. Los constructores de los tipos enumerados nunca son públicos porque no podemos crear instancias de ellos. El parámetro asociado a cada valor se pasa mediante el parámetro del constructor.
- La definición del tipo incluye un campo, `cadenaComando`. El constructor almacena la cadena comando en este campo.
- El método `toString` se utiliza para devolver el texto asociado con un valor en particular.

Con el texto de los comandos almacenado en el tipo `PalabraComando`, la clase `PalabrasComando` del proyecto `zuul-con-enumeraciones-v2` utiliza una manera diferente para crear la correspondencia entre el texto y los valores enumerados:

```

comandosValidos = new HashMap<String, PalabraComando>();
for (PalabraComando comando : PalabraComando.values()) {
    comandosValidos.put(comando.toString(), comando);
}
}

```

Cada tipo enumerado define un método `values` que devuelve un arreglo que contiene todos los valores del tipo. El código anterior recorre el arreglo e invoca al método `toString` para obtener la cadena de comando asociada con cada valor.

**Ejercicio 7.40** Agregue su propio comando `ver` al proyecto `zuul-con-enumeraciones-v2`. ¿Sólo necesita modificar el tipo `PalabraComando`?

**Ejercicio 7.41** Modifique la palabra asociada con el comando `ayuda` en `PalabraComando`. Este cambio, ¿se ve automáticamente reflejado en el texto de bienvenida cuando se inicia el juego? Dé una mirada al método `imprimirBienvenida` de la clase `Juego` para ver la forma en que se resolvió el problema.

## 7.14

## Pautas de diseño

Una advertencia que se hace frecuentemente a los programadores novatos para escribir buenos programas orientados a objetos es: «*No pongan demasiadas cosas en un solo método*» o «*No pongan todo en una sola clase*». Ambas sugerencias tienen su mérito

pero frecuentemente conllevan a preguntas sobre su longitud: «*¿Qué largo debe tener un método?*» o «*De qué tamaño debe ser una clase?*».

Después de la discusión realizada en este capítulo, estas preguntas pueden responderse en términos de cohesión y de acoplamiento. Un método es demasiado largo si hace más de una tarea lógica. Una clase es demasiado compleja si representa más de una entidad lógica.

Notará que estas respuestas no aportan reglas claras que especifiquen exactamente qué hacer. Los términos tales como *una tarea lógica* aún son de interpretación abierta y diferentes programadores tomarán decisiones diferentes en varias situaciones.

Estas son *pautas* (no reglas fijas). El tener estas pautas en mente puede mejorar significativamente su diseño de clases y permitirle resolver problemas más complejos y escribir programas mejores y más interesantes.

*Es importante comprender que estos ejercicios son sugerencias, no especificaciones fijas. Este juego tiene muchas formas posibles de ser extendido y se estimula al lector a que invente sus propias extensiones. No necesita hacer todos estos ejercicios para crear un juego interesante, podría querer hacer más ejercicios o bien, otros diferentes. Aquí presentamos algunas sugerencias para que pueda comenzar.*

**Ejercicio 7.42** Agregue en su juego alguna manera de limitar el tiempo. El jugador no completa cierta tarea en el tiempo especificado, pierde. Un tiempo límite puede implementarse fácilmente contando el número de movimientos o el número de comandos ingresados. No necesita usar el tiempo real.

**Ejercicio 7.43** Implemente una puerta trampa en algún lugar (o alguna otra clase de puerta que pueda ser atravesada sólo de una única manera).

**Ejercicio 7.44** Agregue un *disparador* al juego. Un disparador es un dispositivo que puede ser *cargado* y *disparado*. Cuando carga el disparador, se memoriza la habitación actual; cuando dispara el disparador, se transporta inmediatamente al jugador a la habitación en la que fue cargado. El disparador podría ser un equipamiento estándar o un elemento que el jugador pueda encontrar. Por supuesto que necesita comandos para cargar y disparar el disparador.

**Ejercicio 7.45** Agregue puertas bloqueadas en su juego. El jugador necesita encontrar (o bien obtener) una llave para abrir la puerta.

**Ejercicio 7.46** Agregue una habitación transportadora. Cuando el jugador entre en esta habitación, será transportado aleatoriamente a una de las otras habitaciones. *Nota:* no es trivial lograr un buen diseño para esta tarea. Puede ser interesante para esta tarea discutir alternativas de diseño con otros estudiantes. (Discutimos alternativas de diseño para esta tarea al final del Capítulo 9. El lector aventurero o avanzado puede saltar a esta parte y dar una leída.)

**Ejercicio 7.47 Desafío.** En el método `procesarComando` en Juego hay una secuencia de sentencias que despachan comandos cuando se reconoce una

palabra comando. Este no es un diseño muy bueno dado que cada vez que agregamos un comando tenemos que agregar un caso en la sentencia if. ¿Puede mejorar este diseño? Diseñe las clases de modo que el manejo de los comandos sea más modular y puedan agregarse más comandos más fácilmente. Implementelo y pruébelo.

**Ejercicio 7.48** Agregue personajes al juego. Los personajes son similares a los elementos pero pueden hablar. Ellos dicen algún texto cuando se les encuentra por primera vez y pueden darle alguna ayuda si se le da el elemento correcto.

**Ejercicio 7.49** Agregue personajes que se mueven. Son como los personajes anteriores pero cada vez que el jugador escribe un comando, estos personajes se pueden mover a una habitación adyacente.

## 7.15

# Ejecutar un programa fuera de BlueJ

Cuando nuestro juego esté terminado, podríamos querer pasárselo a otras personas para que jueguen con él. Para pasar el juego, sería bueno que la gente pudiera jugar sin necesidad de iniciarla dentro del entorno BlueJ. Para ser capaces de hacer esto necesitamos una cosa más: los *métodos de clase* que en Java se conocen también como *métodos estáticos*.

### 7.15.1 Métodos de clase

Hasta ahora, todos los métodos que hemos visto han sido *métodos de instancia*: se invocan sobre una instancia de una clase. Lo que distingue a los métodos de clase de los métodos de instancia es que los métodos de clase pueden ser invocados sin tener una instancia, alcanza con tener la clase.

En la Sección 5.13 hablamos sobre variables de clase. Los métodos de clase están relacionados conceptualmente y usan una sintaxis relacionada con las variables de clase (la palabra clave en Java es `static`). Así como las variables de clase pertenecen a la clase antes que a una instancia, lo mismo ocurre con los métodos de clase.

Un método de clase se define agregando la palabra clave `static` antes del nombre del tipo en la firma del método:

```
public static int getNumeroDeDiasDeEsteMes()
{
    ...
}
```

Estos métodos pueden ser invocados utilizando la notación usual de punto, especificando el nombre de la clase en la que está definido seguido del punto y luego del nombre del método. Si, por ejemplo, el método anterior está declarado en una clase de nombre `Calendario`, la siguiente sentencia lo invoca:

```
int dias = Calendario.getNumeroDeDiasDeEsteMes();
```

Observe que antes del punto se usa el nombre de la clase, no se ha creado ningún objeto.

**Ejercicio 7.50** Lea la documentación de la clase `Math` del paquete `java.lang`. Esta clase contiene varios métodos estáticos. Busque el método que calcula el máximo entre dos números enteros. ¿Cuál es su signatura?

**Ejercicio 7.51** ¿Por qué piensa que los métodos de la clase `Math` son estáticos? ¿Podrían escribirse como métodos de instancia?

**Ejercicio 7.52** Escriba una clase de prueba que tenga un método para comprobar cuánto tiempo insume el contar desde 1 hasta 100 en un ciclo. Como ayuda para la medición del tiempo, puede usar el método `currentTimeMillis` de la clase `System`.

### 7.15.2 El método main

Si queremos iniciar una aplicación Java fuera del entorno BlueJ necesitamos usar un método de clase. En BlueJ, típicamente creamos un objeto e invocamos uno de sus métodos, pero fuera de este entorno una aplicación comienza sin que exista ningún objeto. Las clases son las únicas cosas que tenemos inicialmente, por lo que el primer método que será invocado debe ser un método de clase.

La definición de Java para iniciar aplicaciones es bastante simple: el usuario especifica la clase que será iniciada y el sistema Java luego invocará un método denominado `main` ubicado dentro de dicha clase. Este método debe tener una firma específica. Si no existe tal método en esa clase se informa un error. En el Apéndice E se describen los detalles de este método y los comandos necesarios para iniciar el sistema Java fuera del entorno BlueJ.

**Ejercicio 7.53** Encuentre los detalles del método `main` y agregue un método como este en su clase `Juego`. El método debiera crear un objeto `Juego` e invocar su método `jugar`. Pruebe el método `main` invocándolo desde BlueJ. Los métodos de clase pueden ser invocados en BlueJ desde el menú contextual de la clase.

**Ejercicio 7.54** Ejecute su juego fuera del entorno BlueJ.

### 7.15.3 Limitaciones de los métodos de clase

Dado que los métodos de clase están asociados con una clase antes que con una instancia, tienen dos limitaciones importantes. La primera limitación es que un método de clase no podrá acceder a ningún campo de instancia definido en la clase. Esto es lógico ya que los campos de instancia están asociados con objetos individuales. En cambio, los métodos de clase tienen el acceso restringido a las variables de clase de sus propias clases. La segunda limitación es como la primera: un método de clase no puede invocar a un método de instancia de la clase. Un método de clase no puede llamar a un método de instancia de la clase. Un método de clase sólo puede invocar a otros métodos de clase definidos en su propia clase.

Encontrará que hacemos muy poco uso de los métodos de clase en los ejemplos de este libro.

## 7.16

## Resumen

En este capítulo hemos discutido lo que frecuentemente se denominan *aspectos no funcionales* de una aplicación. Aquí, la cuestión no es tanto obtener un programa para realizar una cierta tarea sino hacerla con un buen diseño de clases.

Un buen diseño de clases puede marcar la diferencia cuando una aplicación necesita ser corregida, modificada o extendida. También nos permite reutilizar las partes de la aplicación en otros contextos (por ejemplo, para otros proyectos) por lo que brinda beneficios a posteriori.

Hay dos conceptos clave bajo los cuales se evalúan los diseños de clases: acoplamiento y cohesión. El acoplamiento se refiere a las interconexiones de las clases, la cohesión a la modularización en unidades apropiadas. Un buen diseño exhibe bajo acoplamiento y alta cohesión.

Un camino para lograr una buena estructura es seguir un proceso de diseño dirigido por responsabilidades. Cada vez que agregamos una función a la aplicación tratamos de identificar qué clase será la responsable para esta parte de la tarea.

Cuando se extiende un programa, usamos la refactorización para adaptar el diseño en base a los requerimientos de los cambios y asegurar que las clases y los métodos resulten cohesivos y bajamente acoplados.

Términos introducidos en este capítulo

**duplicación de código, acoplamiento, cohesión, encapsulamiento, diseño dirigido por responsabilidades, acoplamiento implícito, refactorización, método de clase**

### Resumen de conceptos

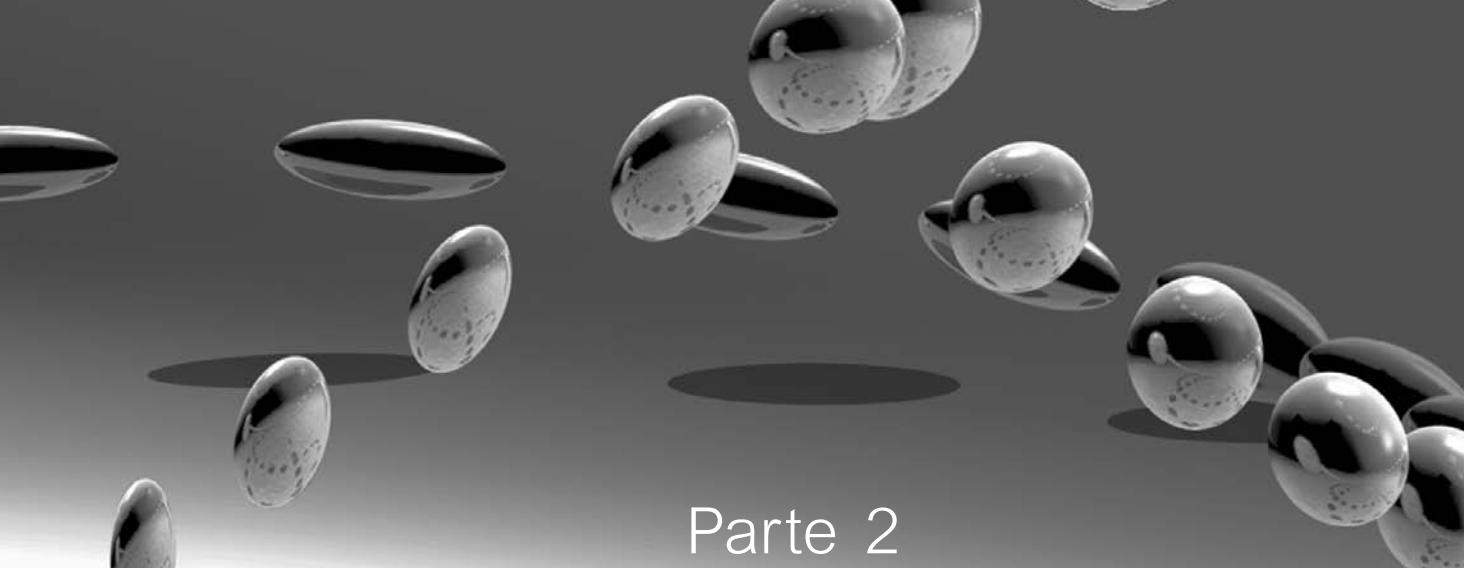
- **acoplamiento** El término acoplamiento describe las interconexiones de las clases. Fomentamos el bajo acoplamiento de un sistema, es decir, un sistema en donde cada clase es bastante independiente y se comunica con otras clases mediante una interfaz pequeña y bien definida.
- **cohesión** La expresión cohesión describe la exactitud con que una unidad de código encaja con una tarea lógica o con una entidad. En un sistema altamente cohesivo cada unidad de código (método, clase o módulo) es responsable de una tarea o entidad bien definida. Un buen diseño de clases exhibe un alto grado de cohesión.
- **duplicación de código** La duplicación de código (tener el mismo segmento de código en una aplicación más de una vez) es una señal de mal diseño. Debe evitarse.
- **Encapsulamiento** El encapsulamiento apropiado de las clases reduce el acoplamiento y conduce a un mejor diseño.
- **diseño dirigido por responsabilidades** Es el proceso de diseñar clases asignando a cada clase responsabilidades bien definidas. Este proceso puede usarse para determinar las clases que implementarán cada parte de una función de una aplicación.

- **localizar cambios** Uno de los principales objetivos de un buen diseño de clases es la localización de los cambios: el hacer cambios en una clase debe tener efectos mínimos en las otras clases.
- **método cohesivo** Un método cohesivo es responsable de una y sólo una tarea bien definida.
- **clase cohesiva** Una clase cohesiva representa una entidad bien definida.
- **refactorización** La refactorización es la actividad de reestructurar un diseño existente para mantener un buen diseño de clases cuando la aplicación se modifica o se extiende.

**Ejercicio 7.55** Sin usar el entorno BlueJ, edite su proyecto Soporte Técnico del Capítulo 5 de modo que pueda ejecutarse fuera de BlueJ. Luego ejecútelo mediante una línea de comando.

**Ejercicio 7.56** ¿Puede invocar un método estático desde un método de instancia? ¿Puede invocar un método de instancia desde un método estático? ¿Puede invocar un método estático desde un método estático? Responda estas preguntas, luego cree una prueba para controlar sus respuestas y verificarlas. Explique en detalle sus respuestas y sus observaciones.

**Ejercicio 7.57** ¿Puede una clase contar cuántas instancias han sido creadas de dicha clase? ¿Qué se necesita para hacer esto? Escriba algún fragmento de código que ilustre lo que necesita para hacerlo. Asuma que quiere un método estático de nombre `numeroDeInstancias` que devuelva el número de instancias que se han creado.



Parte 2  
**Estructuras**  
de las aplicaciones



## CAPÍTULO

# 8

# Mejorar la estructura mediante herencia

Principales conceptos que se abordan en este capítulo

- herencia
- sustitución
- subtipo
- variables polimórficas

Construcciones Java que se abordan en este capítulo

`extends`, `super` (en constructores), enmascaramiento, `Object`, autoboxing, clases «envoltorio»

En este capítulo presentamos algunas construcciones adicionales de programación orientadas a objetos que nos ayudan a mejorar la estructura general de nuestras aplicaciones. Los conceptos principales que usaremos para diseñar programas mejor estructurados son *herencia* y *polimorfismo*.

Ambos conceptos son centrales en orientación a objetos y aparecen de distintas formas en cada tema que abordemos de aquí en adelante. Sin embargo, no sólo los siguientes capítulos descansan fuertemente sobre estos conceptos, sino que muchas de las construcciones y técnicas tratadas en los capítulos anteriores están influenciadas por aspectos de la herencia y del polimorfismo, por lo que revisaremos algunas cuestiones introducidas tempranamente y así comprenderemos mejor las interconexiones entre las diferentes partes del lenguaje Java.

La herencia es una potente construcción que puede usarse para crear soluciones de problemas de diferente naturaleza. Como siempre, discutiremos los aspectos importantes de este concepto mediante un ejemplo. En este ejemplo, sólo introducimos algunos de los problemas que están relacionados con el uso de estructuras de herencia; discutiremos los usos y las ventajas de la herencia y del polimorfismo a medida que avancemos en el capítulo.

El ejemplo que utilizaremos para presentar estas nuevas estructuras se denomina *DoME*.

## 8.1

### El ejemplo DoME

El acrónimo DoME surge a partir de los términos *Database of Multimedia Entertainment* (*Base de Datos de Entretenimientos Multimediales*). El nombre completo es dema-

siado grande para un programa tan simple como el que vamos a desarrollar. (Pero, cuidado porque el marketing es la mitad del juego; dentro de un tiempo, un nombre impactante podría ayudarnos a enriquecernos mediante la venta de muchas copias de nuestro programa, ¿no es cierto?)

En esencia, DoME es una aplicación que nos permite almacenar información sobre discos compactos de música (en CD) y de películas (en DVD). La idea es crear un catálogo de todos los CD y DVD que tenemos, o todos los que hemos visto o escuchado.

La funcionalidad que queremos que brinde DoME incluye como mínimo lo siguiente:

- Debe permitirnos ingresar información sobre los CD y los DVD.
- Debe almacenar esta información de manera permanente, de tal modo que pueda ser usada más adelante.
- Debe brindar una función de búsqueda que nos permita por ejemplo, encontrar todos los CD de un cierto intérprete que hay en la base, o todos los DVD de determinado director. (Nota: por razones de simplicidad, asumimos aquí que sólo tenemos DVD de películas, de modo que al almacenar un DVD sabemos que queremos almacenar información sobre las películas.)
- Debe permitirnos imprimir listados como por ejemplo: listado de todos los DVD que hay en la base o un listado de todos los CD de música.
- Debe permitirnos eliminar información.

Los detalles que queremos almacenar de cada CD son:

- el título del álbum;
- el intérprete (el nombre de la banda o del cantante);
- el número de temas que tiene el CD;
- el tiempo de duración del CD;
- una bandera que indique si tenemos una copia de este CD («lo tengo») y
- un comentario (un texto arbitrario).

Los detalles que queremos almacenar de cada DVD son:

- el título del DVD
- el nombre del director
- el tiempo de duración (definimos este tiempo como la duración de la película principal)
- una bandera que indique si tenemos una copia de este DVD («lo tengo») y
- un comentario (un texto arbitrario).

### 8.1.1 Las clases y los objetos de DoME

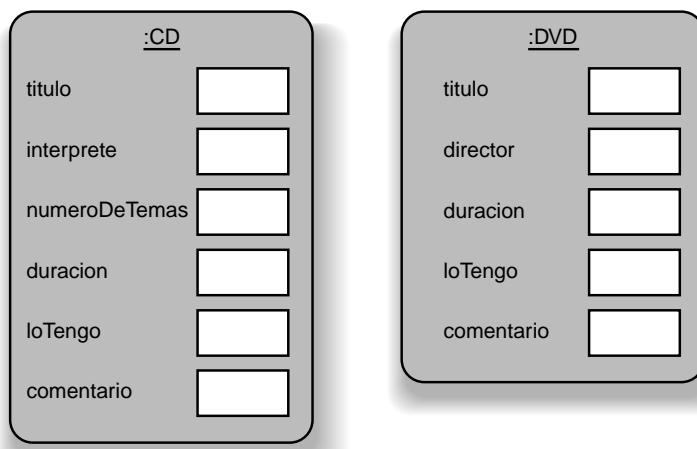
Para implementar esta aplicación, primero tenemos que decidir qué clases usaremos para modelar este problema. En este caso, algunas de estas clases son fáciles de identificar. En el momento de decidir, es muy claro que debemos tener una clase CD para representar a los objetos CD y una clase DVD que represente a los objetos DVD.

Por lo tanto, los objetos de estas clases deben encapsular todos los datos que queremos almacenar sobre ellos (Figura 8.1).

Algunos de estos datos, probablemente, también deberán tener métodos de acceso y métodos de modificación (Figura 8.2)<sup>1</sup>. Para nuestros fines no es importante, por ahora, decidir los detalles exactos de todos los métodos pero podemos hacernos una primera impresión del diseño de esta aplicación. En esta figura hemos definido métodos de acceso y métodos de modificación para aquellos campos que pueden cambiar su contenido a lo largo del tiempo (la bandera que indica si tenemos una copia y el comentario) y asumimos por ahora, que los otros campos se inicializan en el constructor. También hemos agregado un método de nombre `imprimir` que imprimirá los detalles de un objeto CD o de un objeto DVD.

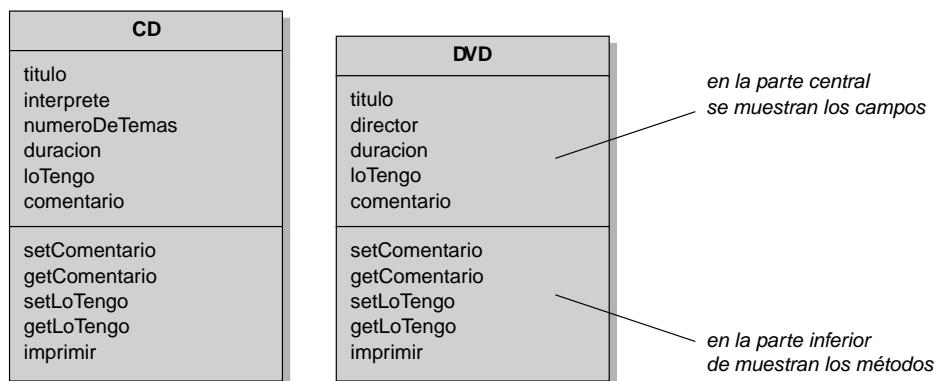
**Figura 8.1**

Campos en los objetos CD y DVD



**Figura 8.2**

Métodos de las clases CD y DVD



<sup>1</sup> El estilo de notación de los diagramas de clases que se usa en este libro y en BlueJ es un subconjunto de una notación más amplia denominada UML. Pese a que no usamos toda la notación UML (ni de lejos) intentamos usar notación UML para aquellas cosas que debemos mostrar. El estilo UML define cómo se muestran los campos y los métodos en un diagrama de clases. La clase está dividida en tres partes que muestra (en este orden y desde arriba) el nombre de la clase, los campos y los métodos.

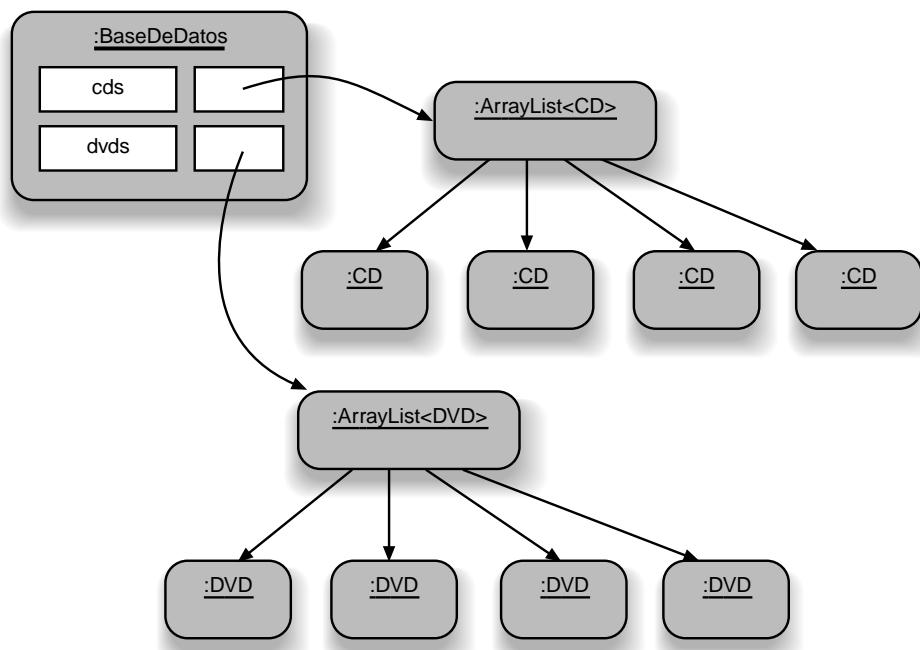
Una vez que hemos definido las clases CD y DVD podemos crear tantos objetos CD y tantos objetos DVD como necesitemos; un objeto por cada CD o cada DVD que queramos almacenar. Aparte de esto, necesitamos otro objeto: un objeto base de datos que pueda contener una colección de CD y una colección de DVD.

El objeto base de datos puede contener dos colecciones de objetos (por ejemplo, una de tipo `ArrayList<CD>` y otra de tipo `ArrayList<DVD>`). Luego, una de estas colecciones puede contener todos los CD y la otra todos los DVD. En la Figura 8.3 se muestra un diagrama de objetos que responde a este modelo.

El diagrama de clases correspondiente al proyecto tal como aparece en BlueJ, se muestra en la Figura 8.4. Observe que BlueJ presenta un diagrama un poco simplificado: no se muestran las clases de la biblioteca estándar de Java (en este caso, la clase `ArrayList`), el diagrama se concentra en las clases definidas por el usuario. BlueJ tampoco muestra los nombres de los campos y de los métodos en el diagrama de clases.

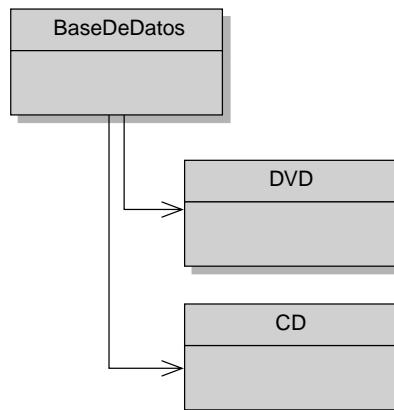
**Figura 8.3**

Objetos en la aplicación DoME



**Figura 8.4**

Diagrama de clases de DoME en BlueJ



En la práctica, para implementar una aplicación DoME completa, deberíamos tener algunas otras clases más para manejar cosas tales como grabar los datos en un archivo y brindar una interfaz de usuario. Estas partes no son muy relevantes en la presente discusión, de modo que, por ahora, saltearemos los detalles sobre estas cuestiones (volveremos sobre ellas más adelante) y nos concentraremos en discutir con más detalle las clases principales aquí mencionadas.

### 8.1.2 Código fuente de DoME

Hasta ahora, el diseño de estas tres clases (CD, DVD y BaseDeDatos) ha sido muy sencillo y claro. La traducción de estas ideas a código Java es igual de fácil. En Código 8.1 se muestra el código fuente de la clase CD que define los campos apropiados, inicializa en su constructor todos los datos que se espera que no cambien a lo largo del tiempo y provee métodos de acceso y de modificación para la bandera `loTengo` y para el comentario; también implementa el método `imprimir` para escribir algunos detalles en la terminal de texto.

Tenga en cuenta que, en este momento, no intentamos de ninguna manera hacer la implementación completa de la clase sino que el código que presentamos sirve para ofrecer una idea de la forma en que quedaría una clase de esta naturaleza. Usaremos esta clase como base para nuestra siguiente discusión sobre herencia.

#### Código 8.1

Código de la clase CD

```
/*
 * La clase CD representa un objeto CD. Se almacena
información
 * sobre el CD que puede ser consultada.
 *
 * @author Michael Kölking and David J. Barnes
 * @version 2006.03.30
 */
public class CD
{
    private String titulo;
    private String interprete;
    private int numeroDeTemas;
    private int duracion;
    private boolean loTengo;
    private String comentario;
    /**
     * Inicializa el CD.
     * @param elTitulo El título del CD.
     * @param elInterprete El intérprete del CD.
     * @param temas El número de temas del CD.
     * @param tiempo El tiempo que dura el CD.
     */
    public CD(String elTitulo, String elInterprete, int
temas, int tiempo)
    {
```

**Código 8.1  
(continuación)**

Código de la clase CD

```
        titulo = elTitulo;
        interprete = elInterprete;
        numeroDeTemas = temas;
        duracion = tiempo;
        loTengo = false;
        comentario = "";
    }
    /**
     * Asigna un comentario para este CD.
     * @comentario El comentario que se ingresará.
     */
    public void setComentario(String comentario)
    {
        this.comentario = comentario;
    }
    /**
     * @return El comentario de este CD.
     */
    public String getComentario()
    {
        return comentario;
    }
    /**
     * Asigna el valor a la bandera que indica si
     * tenemos este CD.
     * @parametro mePertenece true si tenemos el CD,
     * false en caso contrario.
     */
    public void setLoTengo(boolean mePertenece)
    {
        loTengo = mePertenece;
    }
    /**
     * @return true si tenemos una copia de este CD.
     */
    public boolean getLoTengo()
    {
        return loTengo;
    }
    /**
     * Imprime en la terminal de texto los detalles de
     * este CD.
     */
    public void imprimir()
    {
        System.out.print("CD: " + titulo + " (" +
duracion + " minutos)");
        if(loTengo) {
            System.out.println("*");
        }
    }
}
```

### Código 8.1 (continuación)

Código de la clase CD

```

        } else {
            System.out.println();
        }
        System.out.println("      " + interprete);
        System.out.println("      temas: " +
numeroDeTemas);
        System.out.println("      " + comentario);
    }
}

```

Ahora, comparemos el código de la clase CD con el código de la clase DVD que se muestra en Código 8.2. Observando ambas clases, rápidamente notamos que son muy similares. Esto no es sorprendente ya que su propósito es similar: ambas se usan para almacenar información sobre un elemento multimedial (y los elementos tienen ciertas similitudes); difieren solamente en sus detalles: en algunos de sus campos y en el cuerpo del método imprimir.

### Código 8.2

Código de la clase DVD

```

/**
 * La clase DVD representa un objeto DVD. Se almacena
información
 * sobre el DVD que puede ser consultada.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 2006.03.30
 */
public class DVD
{
    private String titulo;
    private String director;
    private int duracion;
    private boolean loTengo;
    private String comentario;
    /**
     * Constructor de objetos de la clase DVD
     * @param elTitulo el título del DVD.
     * @param elDirector El director del DVD.
     * @param tiempo El tiempo de duración del DVD.
     */
    public DVD(String elTitulo, String elDirector, int
tiempo)
    {
        titulo = elTitulo;
        director = elDirector;
        duracion = tiempo;
        loTengo = false;
        comentario = "";
    }
}

```

**Código 8.2  
(continuación)**Código de la clase  
DVD

```
    }
    /**
     * Asigna un comentario para este DVD.
     * @param comentario El comentario que se ingresará.
     */
    public void setComentario(String comentario)
    {
        this.comentario = comentario;
    }
    /**
     * @return El comentario de este DVD.
     */
    public String getComentario()
    {
        return comentario;
    }
    /**
     * Asigna el valor a la bandera que indica si
     * tenemos este DVD.
     * @param mePertenece true si tenemos el DVD,
     * false en caso contrario.
     */
    public void setLoTengo(boolean mePertenece)
    {
        loTengo = mePertenece;
    }
    /**
     * @return true si tenemos una copia de este DVD.
     */
    public boolean getLoTengo()
    {
        return loTengo;
    }
    /**
     * Imprime en la terminal de texto los detalles de
     * este DVD.
     */
    public void imprimir()
    {
        System.out.print("DVD: " + titulo + " (" +
duracion + " minutos)");
        if(loTengo) {
            System.out.println("*");
        } else {
            System.out.println();
        }
        System.out.println("      " + director);
        System.out.println("      " + comentario);
    }
}
```

A continuación, examinamos el código de la clase `BaseDeDatos` (Código 8.3) que también es muy simple: define dos listas (cada una basada en la clase `ArrayList`) para mantener la colección de CD y la colección de DVD. En el constructor, estas listas se crean vacías. La clase ofrece dos métodos para agregar elementos: uno para agregar CD y otro para agregar DVD. El último método de nombre `listar` imprime en la terminal de texto un listado de todos los CD y DVD.

**Código 8.3**

Código de la clase  
`BaseDeDatos`

```
import java.util.ArrayList;
/**
 * La clase BaseDeDatos proporciona facilidades para
almacenar objetos
 * CD y DVD. Se puede imprimir en la terminal de texto,
un listado de todos
 * los CD y DVD.
 *
 * Esta versión no graba los datos en el disco y no
provee ninguna función
 * de búsqueda.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 2006.03.30
 */
public class BaseDeDatos
{
    private ArrayList<CD> cds;
    private ArrayList<DVD> dvds;
    /**
     * Construye una BaseDeDatos vacía.
     */
    public BaseDeDatos()
    {
        cds = new ArrayList<CD>();
        dvds = new ArrayList<DVD>();
    }
    /**
     * Agrega un CD a la base.
     * @param elCD El CD que se agregará a la base de
datos.
     */
    public void agregarCD(CD elCD)
    {
        cds.add(elCD);
    }
    /**
     * Agrega un DVD a la base.
     * @param elDVD El DVD que se agregará a la base
de datos.
     */
}
```

**Código 8.3  
(continuación)**  
Código de la clase  
BaseDeDatos

```

        public void agregarDVD(DVD elDVD)
    {
        dvds.add(elDVD);
    }
    /**
     * Imprime en la terminal de texto un listado de
    todos los CD y DVD
     * que actualmente están almacenados en la base.
    */
    public void listar()
    {
        // imprime la lista de CD
        for(CD cd : cds ) {
            cd.imprimir();
            System.out.println(); // línea vacía
entre los elementos
        }
        // imprime la lista de DVD
        for(DVD dvd : dvds) {
            dvd.imprimir();
            System.out.println(); // línea vacía
entre los elementos
        }
    }
}

```

Tenga en cuenta que este código no implica que la aplicación esté completa: aún no tiene interfaz de usuario (de modo que no se podrá usar fuera del entorno BlueJ) y los datos que se ingresen no se almacenarán en un archivo, por lo que todos los datos que se ingresen se perderán cada vez que finalice la aplicación. Las funciones para ingresar y editar datos, así como para buscar datos y mostrarlos, tampoco son lo suficientemente flexibles como quisiéramos que lo fueran en un programa real.

Sin embargo, en nuestro contexto, todo esto no es importante ya que más adelante podemos trabajar para mejorar esta aplicación. Lo importante es que la estructura básica está hecha y funciona, y esto nos alcanza para discutir los problemas de este diseño y sus posibles mejoras.

**Ejercicio 8.1** Abra el proyecto *dome-v1* que contiene exactamente las clases que hemos discutido aquí. Cree algunos objetos CD y algunos objetos DVD. Cree un objeto BaseDeDatos. Agregue los CD y los DVD en la base y luego imprima un listado del contenido de la base.

**Ejercicio 8.2** Pruebe lo siguiente: cree un objeto CD; ingréselo en la base de datos; imprima un listado del contenido de la base. Verá que no hay ningún comentario asociado a cada elemento: ingrese un comentario para el objeto CD en el banco de objetos (el mismo que ingresó en la base). Cuando imprima

nuevamente el contenido de la base, el CD ¿tendrá un comentario asociado? Pruébelo y explique el comportamiento que observa.

### 8.1.3 Discusión de la aplicación DoME

Aunque nuestra aplicación aún no está completa, hemos llevado a cabo la parte más importante: hemos definido el centro de la aplicación, es decir, la estructura de datos que almacena la información esencial.

Hasta el momento, el diseño ha sido sumamente fácil y ahora podemos avanzar y diseñar el resto que aún falta, pero antes de hacerlo, discutiremos la calidad de la solución lograda.

Existen varios problemas fundamentales en nuestra solución actual; la más obvia es la *duplicación de código*.

Hemos observado que las clases CD y DVD son muy similares, en realidad, la mayoría del código de ambas clases es idéntico con muy pocas diferencias. Ya hemos mencionado los problemas asociados a la duplicación de código en el Capítulo 7. Además del hecho de que tenemos que escribir dos veces cada cosa (o copiar y pegar, y luego arreglar todas las diferencias), frecuentemente se presentan problemas asociados al mantenimiento del código duplicado. Si se deben realizar varios cambios, tendrían que hacerse dos veces. Por ejemplo, si se modifica el tipo del campo `duracion` para que sea un `float` en lugar de un `int` (para poder manejar fracciones de tiempo), este cambio debe hacerse una vez en la clase CD y otra vez en la clase DVD. Además, asociado al mantenimiento del código duplicado, siempre está presente el peligro de introducir errores, ya que el programador de mantenimiento podría no darse cuenta de que se necesita un cambio idéntico en la segunda ubicación (o en la tercera).

Hay otro lugar en el que tenemos duplicación de código: en la clase `BaseDeDatos`. Podemos ver en ella que cada cosa se hace dos veces, una vez para los CD y otra para los DVD. La clase define dos variables para las listas, luego crea dos objetos lista, define dos métodos «agregar» y tiene dos bloques casi idénticos de código en el método `listar` para imprimir ambas listas.

Los problemas que traen aparejados esta duplicación de código se aclaran si analizamos lo qué tendríamos que hacer para agregar otro tipo de elemento multimedial en este programa. Imagine que queremos almacenar información no sólo sobre DVD y CD sino también sobre libros. Los libros se parecen bastante a los elementos antes mencionados, de modo que sería fácil modificar nuestra aplicación para incluir libros. Podríamos introducir otra clase, `Libro`, y escribir, esencialmente, una tercera versión del código que ya está en las clases CD y DVD. Luego tenemos que trabajar en la clase `BaseDeDatos` y agregar otra variable para la lista de libros, otro objeto lista, otro método «agregar» y otro ciclo en el método `listar`.

Y tendríamos que hacer lo mismo para un cuarto tipo de elemento multimedial. Cuanto más repitamos este proceso, más se incrementarán los problemas de duplicación de código y más difícil será realizar cambios más adelante. Cuando nos sentimos incómodos con una situación como ésta, frecuentemente es un buen indicador de que hay una alternativa mejor de abordaje. Para este caso en particular, los lenguajes orientados a objetos proveen una característica distintiva que tiene un gran impacto en programas

que involucran conjuntos de clases similares. En las siguientes secciones introduciremos esta característica que se denomina *herencia*.

## 8.2

### Usar herencia

#### Concepto

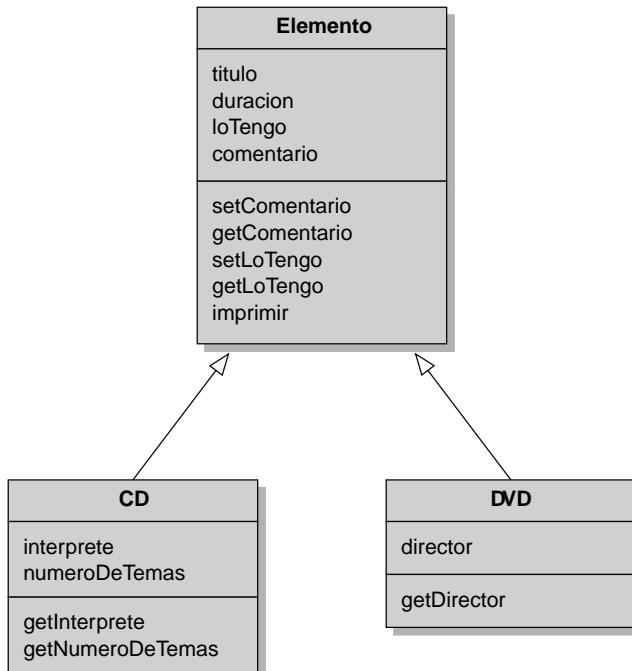
La **herencia** nos permite definir una clase como una extensión de otra.

La herencia es un mecanismo que nos ofrece una solución a nuestro problema de duplicación de código. La idea es simple: en lugar de definir las clases CD y DVD completamente independientes, definimos primero una clase que contiene todas las cosas que tienen en común ambas clases. Podemos llamar a esta clase Elemento y luego declarar que un CD es un Elemento y que un DVD es un Elemento. Finalmente, podemos agregar en la clase CD aquellos detalles adicionales necesarios para un CD y los necesarios para un DVD en la clase DVD. La característica esencial de esta técnica es que necesitamos describir las características comunes sólo una vez.

La Figura 8.5 muestra un diagrama de clases para esta nueva estructura que hemos descrito. El diagrama muestra la clase Elemento en la parte superior; esta clase define todos los campos y métodos que son comunes a todos los elementos (CD y DVD). Debajo de la clase Elemento, aparecen las clases CD y DVD que contienen sólo aquellos campos y métodos que son únicos para cada clase en particular. Aquí hemos agregado tres métodos: `getInterprete` y `getNumeroDeTemas` en la clase CD y `getDirector` en la clase DVD, para ilustrar el hecho de que las clases CD y DVD pueden definir sus propios métodos.

**Figura 8.5**

Las clases CD y DVD se heredan a partir de Elemento



Esta nueva característica de la programación orientada a objetos requiere algunos nuevos términos. En una situación tal como esta, decimos que la clase CD *deriva de* la clase Elemento. La clase DVD también deriva de Elemento. Cuando hablamos de pro-

gramas en Java, también se usa la expresión «la clase CD *extiende* a la clase Elemento» pues Java utiliza la palabra clave «*extends*» para definir la relación de herencia (veremos esto en breve).

La flecha en el diagrama de clases (dibujada generalmente con la punta sin rellenar) representa la relación de herencia.

#### Concepto

Una **superclase** es una clase que es extendida por otra clase.

La clase Elemento (la clase a partir de la que se derivan o heredan las otras) se denomina *clase padre*, *clase base* o *superclase*. Nos referimos a las clases heredadas (en este ejemplo, CD y DVD) como *clases derivadas*, *clases hijos* o *subclases*. En este libro usaremos los términos «superclase» y «subclase» para referirnos a las clases involucradas en una relación de herencia.

Algunas veces, la herencia también se denomina relación «es un». La razón de esta nomenclatura radica en que la subclase es una especialización de la superclase. Podemos decir que «un CD *es un* elemento» y que «un DVD *es un* elemento».

El propósito de usar herencia ahora resulta bastante obvio. Las instancias de la clase CD tendrán todos los campos que están definidos en la clase CD y todos los de la clase Elemento. (CD hereda los campos de la clase Elemento.) Las instancias de DVD tendrán todos los campos definidos en las clases DVD y Elemento. Por lo tanto, logramos tener lo mismo que teníamos antes, con la diferencia de que ahora necesitamos definir los campos *título*, *duración*, *loTengo* y *comentario* sólo una vez (pero podemos usarlos en dos lugares diferentes).

Lo mismo ocurre con los métodos: las instancias de las subclases tienen todos los métodos definidos en ambas, la superclase y la subclase. En general, podemos decir: dado que un CD es un elemento, un objeto CD tiene todas las cosas que tiene un elemento y otras más. Y dado que un DVD también es un elemento, tiene todas las cosas de un elemento y otras más.

Por lo tanto, la herencia nos permite crear dos clases que son bastante similares evitando la necesidad de escribir dos veces la parte que es idéntica. La herencia tiene otras ventajas más que discutiremos a continuación, sin embargo, primero daremos otra mirada más general a las jerarquías de herencia.

## 8.3

### Jerarquías de herencia

La herencia puede usarse en forma mucho más general que el ejemplo que mostramos anteriormente. Se pueden heredar más de dos subclases a partir de la misma superclase y una subclase puede convertirse en la superclase de otras subclases. En consecuencia, las clases forman una *jerarquía de herencia*.

#### Concepto

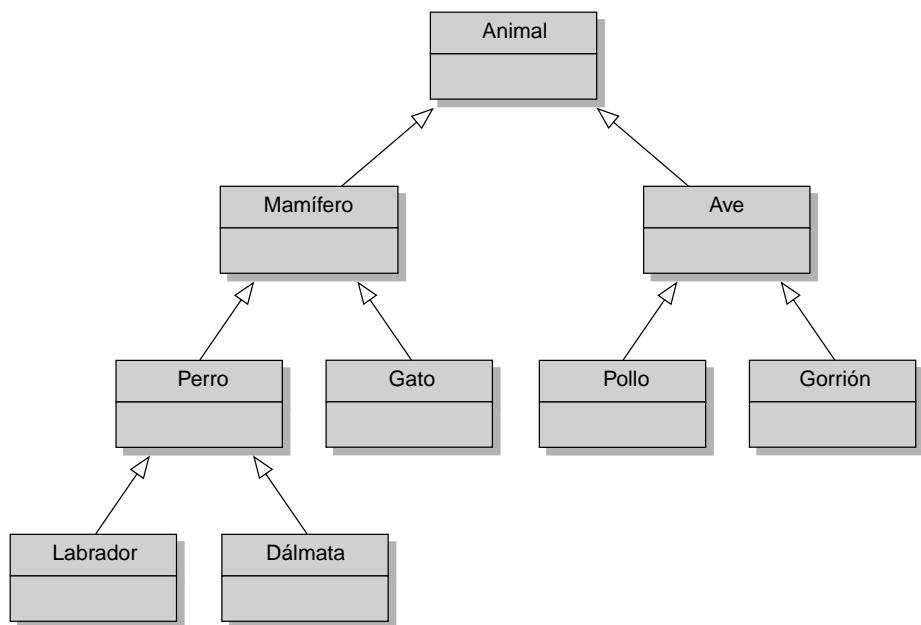
Las clases que están vinculadas mediante una relación de herencia forman una **jerarquía de herencia**.

Probablemente, el ejemplo más conocido de una jerarquía de herencia es la clasificación de las especies que usan los biólogos. En la Figura 8.6 se muestra una pequeña parte de esta clasificación: podemos ver que un dálmatas es un perro, que a su vez es un mamífero y que también es un animal.

Sabemos algunas cosas sobre los labradores, por ejemplo, que son seres vivos, pueden ladrar, y comen carne. Si miramos un poco más de cerca, vemos que sabemos algunas de estas cosas no porque son labradores sino porque son perros, mamíferos o animales. Una instancia de la clase Labrador (un labrador real) tiene todas las características de un labrador, de un perro, de un mamífero y de un animal, porque un labrador es un perro, que a su vez es un mamífero, y así sucesivamente.

**Figura 8.6**

Ejemplo de una jerarquía de herencia



El principio es simple. La herencia es una técnica de abstracción que nos permite categorizar las clases de objetos bajo cierto criterio y nos ayuda a especificar las características de estas clases.

**Ejercicio 8.3** Dibuje una jerarquía de herencia de las personas que hay en su lugar de estudio o de trabajo. Por ejemplo, si usted es un estudiante universitario, probablemente su universidad tienen estudiantes (de primer año, de segundo año, etc.), profesores, tutores, empleados administrativos, etc.

## 8.4

## Herencia en Java

Antes de discutir más detalles de la herencia, veremos cómo se expresa en el lenguaje Java. Aquí presentamos un segmento de código de la clase **Elemento**:

```

public class Elemento
{
    private String titulo;
    private int duracion;
    private boolean loTengo;
    private String comentario;
    // se omitieron constructores y métodos
}
  
```

Hasta ahora, esta clase no tiene nada especial: comienza con una definición normal de clase y declara los campos de la manera habitual. A continuación, examinamos el código de la clase **CD**:

```

public class CD extends Elemento
{
    private String interprete;
  
```

```

private int numeroDeTemas;
// se omitieron constructores y métodos
}

```

En este código hay dos puntos importantes para resaltar. En primer lugar, la palabra clave `extends` define la relación de herencia. La frase «`extends Elemento`» especifica que esta clase es una subclase de la clase `Elemento`.

En segundo término, la clase `CD` define sólo aquellos campos que son únicos para los objetos `CD` (`interprete` y `numeroDeTemas`). Los campos de `Elemento` se heredan y no necesitan ser listados en este código. No obstante, los objetos de la clase `CD` tendrán los campos `titulo`, `duracion` y así sucesivamente.

A continuación, demos un vistazo al código de la clase `DVD`:

```

public class DVD extends Elemento
{
    private String director;
    // se omitieron constructores y métodos
}

```

Esta clase sigue el mismo modelo que la clase `CD`: usa la palabra clave `extends` para definirse como una subclase de `Elemento` y define sus propios campos adicionales.

#### 8.4.1 Herencia y derechos de acceso

Para los objetos de las otras clases, los objetos `DVD` o `CD` aparecen como todos los otros tipos de objetos. En consecuencia, los miembros definidos como públicos, ya sea en la superclase o en la subclase, serán accesibles para los objetos de otras clases, pero los miembros definidos como privados serán inaccesibles. En realidad, la regla de privacidad también se aplica entre una subclase y su superclase: una subclase no puede acceder a los miembros privados de su superclase. Se concluye que si un método de una subclase necesita acceder o modificar campos privados de su superclase, entonces la superclase necesitará ofrecer los métodos de acceso y/o métodos de modificación apropiados. Una subclase puede invocar a cualquier método público de su superclase como si fuera propio, no se necesita ninguna variable.

Esta cuestión de los derechos de acceso es uno de los temas que discutiremos más adelante en el Capítulo 9 cuando presentemos el modificador de acceso `protected`.

**Ejercicio 8.4** Abra el proyecto `dome-v2`. Este proyecto contiene una versión de la aplicación DoME, redactada usando herencia tal como lo hemos descrito anteriormente. Observe que el diagrama de clases muestra la relación de herencia. Abra el código fuente de la clase `DVD` y elimine la frase «`extends Elemento`». Cierre el editor. ¿Qué cambios observa en el diagrama de clases? Agregue nuevamente la frase «`extends Elemento`».

**Ejercicio 8.5** Cree un objeto `CD`. Invoque alguno de sus métodos. ¿Puede invocar los métodos heredados (por ejemplo, `setComentario`)? ¿Qué observa sobre los métodos heredados?

### 8.4.2 Herencia e inicialización

Cuando creamos un objeto, el constructor de dicho objeto tiene el cuidado de inicializar todos los campos con algún estado razonable. Tenemos que ver más de cerca cómo se hace esto en las clases que se heredan a partir de otras clases.

Cuando creamos un objeto CD, pasamos varios parámetros al constructor de CD: el título, el nombre del intérprete, el número de temas y el tiempo de duración. Algunos de estos parámetros contienen valores para los campos definidos en la clase Elemento y otros valores para los campos definidos en la clase CD. Todos estos campos deben ser correctamente inicializados y el Código 8.4 muestra los segmentos de código que se usan para llevar a cabo esta inicialización en Java.

**Código 8.4**

Inicialización de campos de una subclase y de una superclase

```
public class Elemento
{
    private String titulo;
    private int duracion;
    private boolean loTengo;
    private String comentario;
    /**
     * Inicializa los campos del elemento.
     * @param elTitulo el título de este elemento.
     * @param tiempo La duración de este elemento.
     */
    public Elemento(String elTitulo, int tiempo)
    {
        titulo = elTitulo;
        duracion = tiempo;
        loTengo = false;
        comentario = "";
    }
    // Se omitieron métodos
}
public class CD extends Elemento
{
    private String interprete;
    private int numeroDeTemas;
    /**
     * Constructor de objetos de la clase CD
     * @param elTitulo El título del CD.
     * @param elInterprete El intérprete del CD.
     * @param temas El número de temas del CD.
     * @param tiempo La duración del CD.
     */
    public CD(String elTitulo, String elInterprete, int
temas, int tiempo)
    {
        super(elTitulo, tiempo);
        interprete = elInterprete;
```

#### Código 8.4 (continuación)

Inicialización de campos de una subclase y de una superclase

```
        numeroDeTemas = temas;
    }
    // Se omitieron métodos
}
```

Se pueden hacer varias observaciones con respecto a estas clases. En primer lugar, la clase `Elemento` tiene un constructor aun cuando no tenemos intención de crear, de manera directa, una instancia de la clase `Elemento`.<sup>2</sup> Este constructor recibe los parámetros necesarios para inicializar los campos de `Elemento` y contiene el código para llevar a cabo esta inicialización.

En segundo lugar, el constructor `CD` recibe los parámetros necesarios para inicializar tanto los campos de `Elemento` como los de `CD`. La clase `Elemento` contiene la siguiente línea de código:

```
super(elTitulo, tiempo);
```

La palabra clave `super` es, en realidad, una llamada al constructor de la superclase. El efecto de esta llamada es que se ejecuta el constructor de `Elemento`, formando parte de la ejecución del constructor del `CD`. Cuando creamos un `CD`, se invoca al constructor de `CD`, quien en su primer sentencia lo convierte en una llamada al constructor de `Elemento`. El constructor de `Elemento` inicializa sus campos y luego retorna al constructor de `CD` que inicializa los restantes campos definidos en la clase `CD`. Para que esta operación funcione, los parámetros necesarios para la inicialización de los campos del elemento se pasan al constructor de la superclase como parámetros en la llamada a `super`.

#### Concepto

**Constructor de superclase.** El constructor de una subclase debe tener siempre como primera sentencia una invocación al constructor de su superclase. Si el código no incluye esta llamada, Java intentará insertarla automáticamente.

En Java, un constructor de una subclase siempre debe invocar en su primer sentencia al *constructor de la superclase*. Si no se escribe una llamada al constructor de una superclase, el compilador de Java insertará automáticamente una llamada a la superclase, para asegurar que los campos de la superclase se inicialicen adecuadamente. La inserción automática de la llamada a la superclase sólo funciona si la superclase tiene un constructor sin parámetros (ya que el compilador no puede adivinar qué parámetros deben pasarse); en el caso contrario, Java informa un error.

En general, es una buena idea la de incluir siempre en los constructores llamadas explícitas a la superclase, aun cuando sea una llamada que el compilador puede generar automáticamente. Consideramos que esta inclusión forma parte de un buen estilo de programación, ya que evita la posibilidad de una mala interpretación y de confusión en el caso de que un lector no esté advertido de la generación automática de código.

**Ejercicio 8.6** Establezca un punto de interrupción en la primer línea del constructor de la clase `CD` y luego cree un objeto `CD`. Cuando aparezca la ventana del depurador, use el botón *Step Into* para entrar en el código. Observe los campos de instancia y su inicialización. Describa sus observaciones.

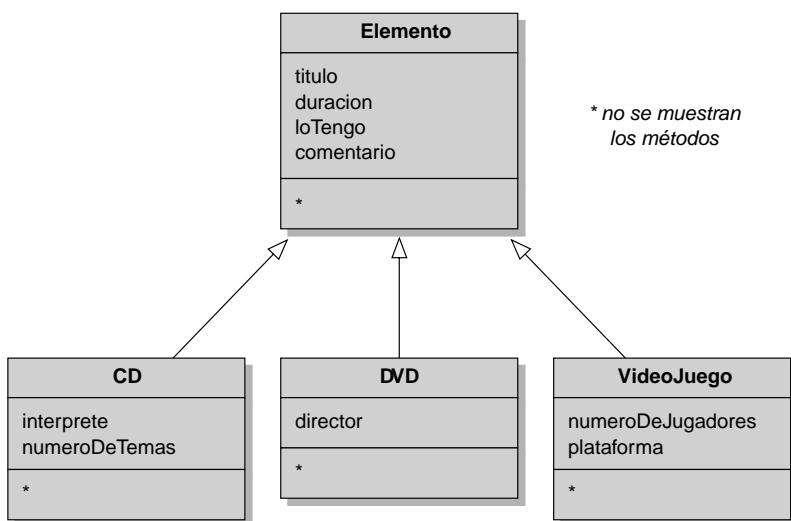
<sup>2</sup> En realidad, por el momento no existe nada que nos prevenga de crear un objeto `Elemento`, pese a que no fue nuestra intención cuando diseñamos estas clases. En el Capítulo 10 veremos algunas técnicas que nos permiten asegurarnos de que no se creen directamente objetos `Elemento` sino sólo objetos `CD` o `DVD`.

## 8.5

# DoME: agregar otros tipos de elementos

Ahora que tenemos armada nuestra jerarquía de herencia para el proyecto DoME de tal manera que los elementos comunes están ubicados en la clase `Elemento`, es mucho más fácil agregar otros tipos de elementos. Por ejemplo, si queremos agregar en nuestra base de datos información sobre juegos de video, podemos definir una nueva subclase de `Elemento` de nombre `JuegoDeVideo` (Figura 8.7). Dado que `JuegoDeVideo` es una subclase de `Elemento`, automáticamente hereda todos los campos y métodos definidos en `Elemento`; por lo tanto, los objetos `JuegoDeVideo` ya tienen un título, una bandera para indicar si lo tenemos, un comentario y un tiempo de duración. (Por supuesto que el tiempo que dura un juego puede variar, pero podríamos utilizar este campo para almacenar el tiempo promedio de un juego.) Luego podemos concentrarnos en agregar atributos que son específicos de los juegos tales como número máximo de jugadores o la plataforma sobre la que corren.

**Figura 8.7**  
Elementos de DoME  
con la clase  
`JuegoDeVideo`



### Concepto

La herencia nos permite **reutilizar** en un nuevo contexto clases que fueron escritas previamente.

Este es un ejemplo de cómo la herencia nos permite reutilizar el trabajo existente. Podemos reutilizar el código que hemos escrito para los DVD y los CD (en la clase `Elemento`) ya que también sirve para la clase `JuegoDeVideo`. La capacidad de reutilizar componentes existentes de software es uno de los grandes beneficios que obtenemos a partir de la facilidad de la herencia. Discutiremos este tema con más detalle más adelante.

El efecto de la reutilización es que se necesita una cantidad menor de código nuevo cuando introducimos elementos adicionales. Dado que los nuevos tipos de elementos pueden ser definidos como subclases de `Elemento`, sólo se debe agregar el código que realmente es diferente del de la clase `Elemento`.

Ahora, imagine que también queremos almacenar juegos de mesa en nuestra base de datos. (Después de todo, esta es una «base de datos de entretenimientos multimediales» y los juegos de mesa son entretenimientos, sólo que usan tecnología de menor nivel...)

Lo primero que se nos ocurre es agregar una cuarta subclase debajo de la clase `Elemento`, sin embargo, a veces es útil analizar las relaciones más cuidadosamente. Tanto los juegos de video como los juegos de mesa tienen un atributo en común: «el máximo número de jugadores». Sería mejor si no definiéramos este campo dos veces: una en

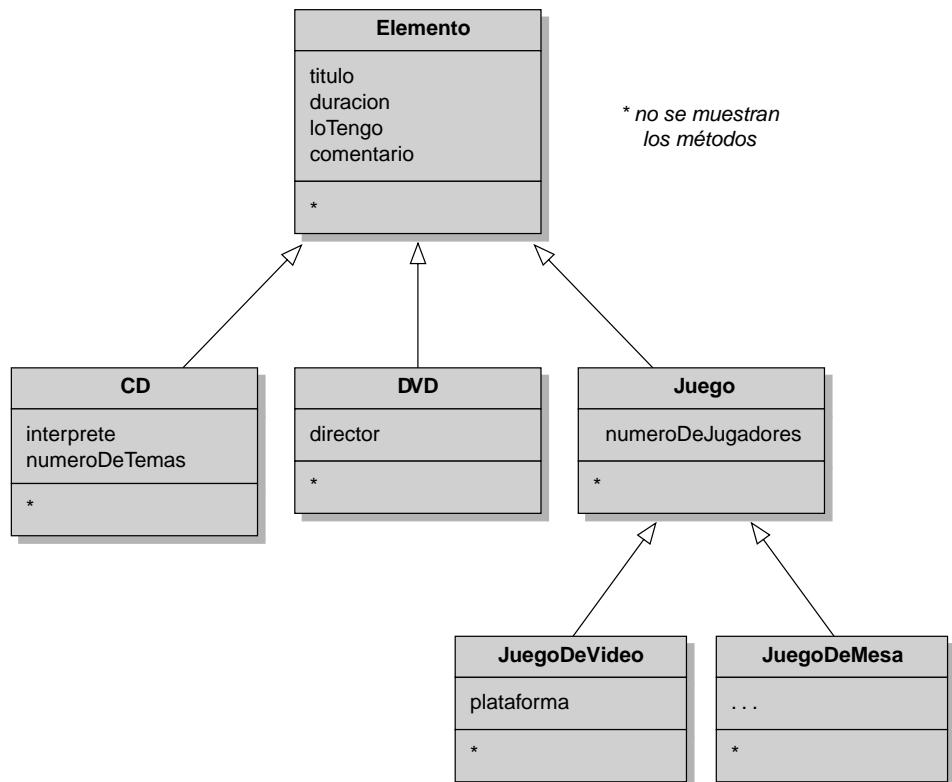
la clase `JuegoDeVideo` y otra en la clase `JuegoDeMesa`. Este sería otro ejemplo de duplicación de código: tendríamos que duplicar el campo y los métodos de acceso y de modificación asociados a este campo. Por lo tanto, la primera idea podría ser que `JuegoDeMesa` sea una subclase de `JuegoDeVideo`; de esta manera, heredaría el campo `numeroDeJugadores` y los métodos que lo acompañan, y evitariamos tener que escribirlos dos veces. Pero hay un problema: también se heredaría el campo que almacena la plataforma sobre la que se ejecutan los juegos y este atributo no tiene sentido en un juego de mesa.

La solución es refactorizar la jerarquía de clases. Podemos introducir una nueva superclase para todos los juegos (de nombre `Juego`) que sea una subclase de `Elemento` (Figura 8.8). De esta manera, toda la información relacionada con los juegos en general (tal como el número de jugadores) puede definirse en la clase `Juego` mientras que la información específica puede moverse a la subclase adecuada. Los objetos de la clase `JuegoDeMesa` ahora tienen todos los campos y métodos de las clases `Elemento`, `Juego` y `JuegoDeMesa`.

Las clases que no se piensan usar para crear instancias, pero cuyo propósito es exclusivamente servir como superclases de otras clases (tal como `Elemento` y `Juego`) se denominan *clases abstractas*. Investigaremos este tema con más detalle en el Capítulo 10.

**Figura 8.8**

Agregado de más tipos de elementos multimediales a DoME



**Ejercicio 8.7** Abra el proyecto `dome-v2`. Agregue al proyecto una clase para los juegos de vídeo. Cree algunos objetos juegos de vídeo y pruebe que todos los métodos funcionan como es de esperar.

## 8.6

# Ventajas de la herencia (hasta ahora)

En la aplicación DoME tuvimos la oportunidad de ver varias ventajas del uso de la herencia. Antes de que exploremos otros aspectos de la herencia resumimos las ventajas generales que hemos encontrado hasta ahora:

- **Evita la duplicación de código** El uso de la herencia evita la necesidad de escribir copias de código idénticas o muy similares dos veces (o con frecuencia, aún más veces).
- **Se reutiliza código** El código que ya existe puede ser reutilizado. Si ya existe una clase similar a la que necesitamos, a veces podemos crear una subclase a partir de esa clase existente y reutilizar un poco de su código en lugar de implementar todo nuevamente.
- **Facilita el mantenimiento** El mantenimiento de la aplicación se facilita pues la relación entre las clases está claramente expresada. Un cambio en un campo o en un método compartido entre diferentes tipos de subclases se realiza una sola vez.
- **Facilita la extensibilidad** En cierta manera, el uso de la herencia hace mucho más fácil la extensión de una aplicación.

**Ejercicio 8.8** Ordene estos elementos en una jerarquía de herencia: manzana, helado, pan, fruta, comida, cereal, naranja, postre, mouse de chocolate, baguette.

**Ejercicio 8.9** ¿En qué relación de herencia podrían estar un *mouse* y un *touch pad*? (Aquí estamos hablando de dispositivos de entrada de computadoras, no de pequeños mamíferos.)

**Ejercicio 8.10** Algunas veces las cosas son más difíciles de lo que parecen ser a primera vista. Considere esto: ¿qué tipo de relación de herencia tienen un *rectángulo* y un *cuadrado*? ¿Cuáles son los argumentos? Fundamente.

## 8.7

# Subtipos

La única cosa que todavía no hemos investigado tiene que ver con la manera en que se modifica el código de la clase *BaseDeDatos* cuando modificamos nuestro proyecto mediante herencia. El Código 8.5 muestra el código completo de la clase *BaseDeDatos*. Podemos comparar este código con el código original de la clase que se muestra en Código 8.3.

### Código 8.5

Código fuente de la clase *BaseDeDatos* (segunda versión)

```
import java.util.ArrayList;
/**
 * La clase BaseDeDatos ofrece facilidades para almacenar
 * objetos
 * que son entretenimientos. Se puede imprimir en la
 * terminal un listado
 * de todos los elementos de entretenimiento.
 *
 * Esta versión no graba los datos en el disco y no brinda
 * ninguna función de búsqueda.
```

### Código 8.5 (continuación)

Código fuente de la clase BaseDeDatos (segunda versión)

```

/*
 * @author Michael Kölking and David J. Barnes
 * @version 2006.03.30
 */
public class BaseDeDatos
{
    private ArrayList<Elemento> elementos;
    /**
     * Construye una BaseDeDatos vacía.
     */
    public BaseDeDatos()
    {
        elementos = new ArrayList<Elemento>();
    }
    /**
     * Agrega un elemento en la base.
     */
    public void agregarElemento(Elemento elElemento)
    {
        elementos.add(elElemento);
    }
    /**
     * Imprime una lista en la terminal de texto de todos
     * los elementos almacenados actualmente.
     */
    public void listar()
    {
        for(Elemento elemento : elementos )
        {
            elemento.imprimir();
        }
        System.out.println(); //una línea vacía entre
        elementos
    }
}

```

Como podemos ver, el código se volvió significativamente más corto y simple debido a nuestro cambio relacionado con el uso de herencia. Mientras que en la primera versión (Código 8.3) cada cosa debía hacerse dos veces, ahora existe una sola vez: tenemos sólo una colección, sólo un método para agregar elementos y un solo ciclo en el método listar.

La razón por la que pudimos acortar el código es que en la nueva versión podemos usar el tipo Elemento en aquellos lugares en que usábamos previamente DVD y CD. Investigamos esta cuestión tomando como ejemplo el método agregarElemento.

En nuestra primera versión, teníamos dos métodos para agregar elementos a la base de datos con las siguientes signaturas:

```

public void agregarCD (CD elCD)
public void agregarDVD (DVD elDVD)

```

**Concepto**

**Subtipo.** Por analogía con la jerarquía de clases, los tipos forman una jerarquía de tipos. El tipo que se define mediante la definición de una subclase es un subtipo del tipo de su superclase.

En nuestra nueva versión tenemos un solo método que sirve para el mismo propósito:

```
public void agregarElemento(Elemento elElemento)
```

Los parámetros de la versión original estaban definidos con los tipos DVD y CD que aseguraban que se pasen objetos DVD y CD a estos métodos, ya que los tipos de los parámetros actuales deben coincidir con los tipos de los parámetros formales. Hasta ahora, hemos interpretado el requerimiento de que los tipos de los parámetros «deben coincidir» como equivalente a decir que «deben ser del mismo tipo»: por ejemplo, que el nombre del tipo de un parámetro actual debe ser el mismo que el nombre del tipo del correspondiente parámetro formal. En realidad, esta es sólo una parte de la verdad porque los objetos de las subclases pueden usarse en cualquier lugar que se requiera el tipo de su superclase.

### 8.7.1 Subclases y subtipos

Anteriormente, hemos hablado de que las clases definen tipos. Un objeto que se crea a partir de la clase DVD es de tipo DVD. También hemos dicho que las clases pueden tener subclases, por lo tanto, los tipos definidos por las clases pueden tener subtipos. En nuestro ejemplo, el tipo DVD es un subtipo del tipo Elemento.

### 8.7.2 Subtipos y asignación

**Concepto**

**Variables y subtipos.** Las variables pueden contener objetos del tipo declarado o de cualquier subtipo del tipo declarado.

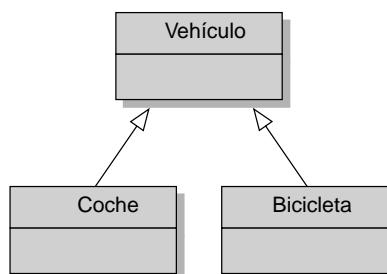
Cuando queremos asignar un objeto a una variable, el tipo del objeto debe coincidir con el tipo de la variable. Por ejemplo:

```
coche miCoche = new Coche();
```

es una asignación válida porque se asigna un objeto de tipo coche a una variable declarada para contener objetos de tipo Coche. Ahora que conocemos la herencia debemos establecer la regla de tipos de manera más completa: una variable puede contener objetos del tipo declarado o de cualquier subtipo del tipo declarado.

**Figura 8.9**

Una jerarquía de herencia



Imagine que tenemos una clase Vehículo con dos subclases, coche y Bicicleta (Figura 8.9). En este caso la regla de tipos admite que las siguientes sentencias son todas legales:

```
Vehiculo v1 = new Vehiculo();
Vehiculo v2 = new Coche();
Vehiculo v3 = new Bicicleta();
```

El tipo de una variable declara qué es lo que puede almacenar. La declaración de una variable de tipo `Vehiculo` determina que esta variable puede contener vehículos. Pero como un coche es un vehículo, es perfectamente legal almacenar un coche en una variable que está pensada para almacenar vehículos. (Puede pensar en la variable como si fuera un garaje: si alguien le dice que puede estacionar un vehículo en un garaje, puede pensar que también es correcto estacionar un coche o una bicicleta en el garaje.)

### Concepto

**Sustitución.** Se pueden usar objetos de subtipos en cualquier lugar en el que se espera un objeto de un supertipo. Esto se conoce como sustitución.

Este principio se conoce como *sustitución*. En los lenguajes orientados a objetos podemos sustituir por un objeto de una subclase en el lugar donde se espera un objeto de una superclase porque el objeto de la subclase es un caso especial de la superclase. Por ejemplo, si alguien nos pide una lapicera podemos responder al pedido perfectamente ofreciendo una lapicera fuente o una lapicera a bolilla. Ambos objetos, lapicera fuente y lapicera a bolilla son subclases de lapicera, de modo que resulta correcto ofrecer cualquiera de ellas cuando se espera una lapicera.

Sin embargo, no está permitido hacer esto de otra manera:

```
Coche a1 = new Vehiculo();      // ¡Es un error!
```

Esta sentencia intenta almacenar un objeto `Vehiculo` en un objeto `Coche`. Java no permitirá esta asignación e informará un error cuando trate de compilar esta sentencia. La variable está declarada para permitir el almacenamiento de coches. Un vehículo, por otro lado, puede o no ser un coche, no sabemos. Por lo tanto, la sentencia es incorrecta y no está permitida.

De manera similar

```
Coche a2 = new Bicicleta();      // ¡Es un error!
```

Esta sentencia también es ilegal. Una bicicleta no es un coche (o más formalmente, el tipo `Bicicleta` no es un subtipo de `Coche`) por lo que la sentencia no está permitida.

**Ejercicio 8.11** Suponga que tiene estas cuatro clases: `Persona`, `Profesor`, `Estudiante` y `EstudianteDeDoctorado`. Tanto `Profesor` como `Estudiante` son subclases de `Persona` y `EstudianteDeDoctorado` es una subclase de `Estudiante`.

¿Cuáles de las siguientes asignaciones son legales y por qué?

```
Persona p1 = new Estudiante();
Persona p2 = new EstudianteDeDoctorado();
EstudianteDeDoctorado ed1 = new Estudiante();
Profesor t1 = new Persona();
Estudiante e1 = new EstudianteDeDoctorado();
e1 = p1;
e1 = p2;
p1 = e1;
t1 = e1;
e1 = ed1;
ed1 = e1;
```

**Ejercicio 8.12** Verifique sus respuestas a las preguntas anteriores creando las clases mencionadas en ese ejercicio y pruébelas en BlueJ.

### 8.7.3 Subtipos y pasaje de parámetros

El pasaje de parámetros (es decir, asignar un parámetro actual a un parámetro formal) se comporta exactamente de la misma manera que la asignación ordinaria a una variable. Este es el motivo por el que podemos pasar un objeto de tipo DVD al método que tiene un parámetro de tipo Elemento. Tenemos la siguiente definición del método `agregarElemento` en la clase `BaseDeDatos`:

```
public void agregarElemento(Elemento elElemento)
{
    ...
}
```

Ahora podemos usar este método para agregar objetos DVD y CD en la base de datos:

```
BaseDeDatos bd = new BaseDeDatos();
DVD dvd = new DVD(...);
CD cd = new CD(...);

bd.agregarElemento(dvd);
bd.agregarElemento(cd);
```

Debido a las reglas de los subtipos, sólo necesitamos un método (con un único tipo de parámetro) para agregar tanto objetos DVD como CD.

Discutiremos con más detalle la cuestión de los subtipos en el siguiente capítulo.

### 8.7.4 Variables polimórficas

En Java, las variables que contienen objetos son variables *polimórficas*. El término «polimórfico» (literalmente: muchas formas) se refiere al hecho de que una misma variable puede contener objetos de diferentes tipos (del tipo declarado o de cualquier subtipo del tipo declarado). El polimorfismo aparece en los lenguajes orientados a objetos en numerosos contextos, las variables polimórficas constituyen justamente un primer ejemplo. Discutiremos otras representaciones del polimorfismo más detalladamente en el próximo capítulo.

Por ahora, sólo observamos la manera en que el uso de una variable polimórfica nos ayuda a simplificar nuestro método `listar`. El cuerpo de este método es

```
for(Elemento elemento : elementos)
{
    elemento.imprimir();
    System.out.println(); //una línea vacía entre
elementos
}
```

En este método recorremos la lista de elementos (contenida en un `ArrayList` mediante la variable `elementos`), tomamos cada elemento de la lista y luego invocamos su método `imprimir`. Observe que los elementos que tomamos de la lista son de tipo CD o DVD pero no son de tipo Elemento. Sin embargo, podemos asignarlos a la variable `elemento` (declarada de tipo Elemento) porque son variables polimórficas. La variable `elemento` es capaz de contener tanto objetos CD como objetos DVD porque estos son subtipos de Elemento.

Por lo tanto, el uso de herencia en este ejemplo ha eliminado la necesidad de escribir dos ciclos en el método `listar`. La herencia evita la duplicación de código no sólo en las clases servidoras sino también en las clases clientes de aquellas.

**Nota:** al hacer los ejercicios debe haberse dado cuenta de que el método `imprimir` tiene un problema: no imprime todos los detalles. La solución de este problema requiere un poco más de explicación y lo haremos en el próximo capítulo.

**Ejercicio 8.13** ¿Qué debe cambiar en la clase `BaseDeDatos` cuando se agrega otra subclase de elemento, por ejemplo, una clase `JuegoDeVideo`? ¿Por qué?

### 8.7.5 Enmascaramiento de tipos

Algunas veces, la regla de que no puede asignarse un supertipo a un subtipo es más restrictiva de lo necesario. Si sabemos que la variable de un cierto supertipo contiene un objeto de un subtipo, podría realmente permitirse la asignación. Por ejemplo:

```
Vehiculo v;
Coche a = new Coche();
v = a;           // es correcta
a = v;           // es un error
```

Las sentencias anteriores no compilarán: obtendremos un error de compilación en la última línea porque no está permitida la asignación de una variable `Vehiculo` en una variable `Coche`. Sin embargo, si recorremos estas sentencias secuencialmente, sabemos que esta asignación podría realmente permitirse. Podemos ver que la variable `v` en realidad contiene un objeto de tipo `Coche`, de modo que su asignación a la variable `a` debiera ser correcta. El compilador no es tan inteligente, traduce el código línea por línea, de modo que analiza la última línea aislada de las restantes, sin saber que es lo que realmente se almacena en la variable `v`. Este problema se denomina *pérdida de tipo*. El tipo del objeto `v` realmente es un `Coche`, pero el compilador no lo sabe.

Podemos resolver este problema diciendo explícitamente al sistema, que la variable `v` contiene un objeto `Coche`, y lo hacemos utilizando el operador de enmascaramiento de tipos:

```
a = (Coche) v;    // correcto
```

El operador de enmascaramiento consiste en el nombre de un tipo (en este caso, `Coche`) escrito entre paréntesis, que precede a una variable o a una expresión. Al usar esta operación, el compilador creerá que el objeto es un `Coche` y no informará ningún error. Sin embargo, en tiempo de ejecución, el sistema Java verificará si realmente es un `Coche`. Si fuimos cuidadosos, todo estará bien; si el objeto almacenado en `v` es de otro tipo, el sistema indicará un error en tiempo de ejecución (denominado `ClassCastException`) y el programa se detendrá<sup>3</sup>.

---

<sup>3</sup> Las excepciones se discuten detalladamente en el Capítulo 12.

Ahora considere este fragmento de código en el que `Bicicleta` también es una subclase de `Vehiculo`:

```
Vehiculo v;
Coche a;
Bicicleta b;
a = new Coche();
v = a;           // correcta
b = (Bicicleta) a;      // error en tiempo de compilación!
b = (Bicicleta) v;      // error en tiempo de ejecución!
```

Las últimas dos asignaciones fallarán. El intento de asignar la variable `a` en la variable `b` (aun estando enmascarada) dará por resultado un error en tiempo de compilación. El compilador se dará cuenta de que `Coche` y `Bicicleta` no constituyen una relación subtipo/supertipo, y por este motivo la variable `a` nunca puede contener un objeto `Bicicleta`; esta asignación no funcionará nunca.

El intento de asignar la variable `v` en `b` (con el enmascaramiento) será aceptado en tiempo de compilación pero fallará en tiempo de ejecución. `Vehiculo` es una superclase de `Bicicleta` y por lo tanto, `v` puede potencialmente contener un objeto `Bicicleta`. Pero en tiempo de ejecución, ocurrirá que el objeto `v` no es una `Bicicleta` sino un coche y el programa terminará prematuramente.

El enmascaramiento debiera evitarse siempre que sea posible, porque puede llevar a errores en tiempo de ejecución y esto es algo que claramente no queremos. El compilador no puede ayudarnos a asegurar la corrección de este caso.

En la práctica, raramente se necesita del enmascaramiento en un programa orientado a objetos bien estructurado. En la mayoría de los casos, cuando se use un enmascaramiento en el código, debiera reestructurarse el código para evitar el enmascaramiento, y se terminará con un programa mejor diseñado. Generalmente, se resuelve el problema de la presencia de un enmascaramiento reemplazándolo por un método polimórfico (en el próximo capítulo hablaremos más de este tema).

## 8.8

### La clase Object

#### Concepto

Todas aquellas clases que no tienen una superclase explícita tienen como su superclase a la clase `Object`.

Todas las clases tienen una superclase. Hasta ahora, nos puede haber parecido que la mayoría de las clases con que hemos trabajado no tienen una superclase, excepto clases como `DVD` y `CD` que extienden otra clase. En realidad, mientras que podemos declarar una superclase explícita para una clase dada, todas las clases que no tienen una declaración explícita de superclase derivan implícitamente de una clase de nombre `Object`.

`Object` es una clase de la biblioteca estándar de Java que sirve como superclase para todos los objetos. Escribir una declaración de clase como la siguiente

```
public class Person
{
    .
}
```

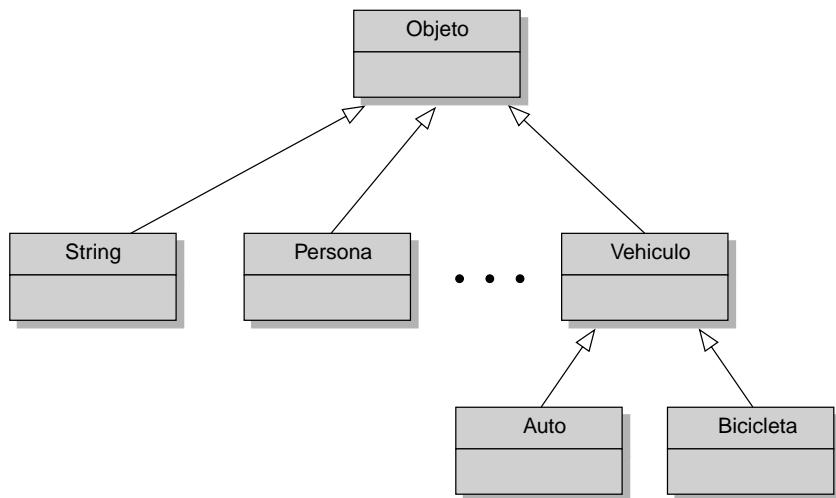
es equivalente a escribir

```
public class Person extends Object
{
    .
}
```

El compilador de Java inserta automáticamente la superclase `Object` en todas las clases que no tengan una declaración explícita `extends` por lo que jamás es necesario hacer esto manualmente. Cada clase simple (con la única excepción de la clase `Object` en sí misma) deriva de `Object`, ya sea directa o indirectamente. La Figura 8.10 muestra algunas clases elegidas aleatoriamente para ilustrar este punto.

**Figura 8.10**

Todas las clases derivan de `Object`



El que todos los objetos tengan una superclase en común tiene dos propósitos. Primero, podemos declarar variables polimórficas de tipo `Object` que pueden contener cualquier objeto. Esta característica de declarar variables que puedan contener cualquier tipo de objeto con frecuencia resulta de poca utilidad, pero existen algunas situaciones en las que puede resultar de ayuda. En segundo lugar, la clase `Object` puede definir algunos métodos que están automáticamente disponibles para cada objeto existente. El segundo punto se pone interesante un poco más adelante, y discutiremos sobre este asunto con más detalles en el próximo capítulo.

## 8.9

## Autoboxing y clases «envoltorio»

Hemos visto que, con una parametrización adecuada, las colecciones pueden almacenar objetos de cualquier tipo; pero queda un problema, Java tiene algunos tipos que no son objetos.

Como sabemos, los tipos primitivos tales como `int`, `boolean` y `char` están separados de los tipos objeto. Sus valores no son instancias de clases y no derivan de la clase `Object`. Debido a esto, no son subtipos de `Object` y normalmente, no es posible ubicarlos dentro de una colección.

Este es un inconveniente pues existen situaciones en las que quisiéramos crear, por ejemplo, una lista de enteros (`int`) o un conjunto de caracteres (`char`). ¿Qué podemos hacer?

La solución de Java para este problema son las *clases envoltorio*. En Java, cada tipo simple o primitivo tiene su correspondiente clase envoltorio que representa el mismo

tipo pero que, en realidad, es un tipo objeto. Por ejemplo, la clase envoltorio para el tipo simple `int` es la clase de nombre `Integer`. En el Apéndice B se ofrece una lista completa de los tipos primitivos y sus correspondientes clases envoltorio.

La siguiente sentencia envuelve explícitamente el valor de la variable `ix` de tipo primitivo `int`, en un objeto `Integer`:

```
Integer ienvuelto = new Integer(ix);
```

y ahora `ienvuelto` puede almacenarse fácilmente por ejemplo, en una colección de tipo `ArrayList<Integer>`. Sin embargo, el almacenamiento de valores primitivos en un objeto colección se lleva a cabo aún más fácilmente mediante una característica del compilador conocida como *autoboxing*.

En cualquier lugar en el que se use un valor de un tipo primitivo en un contexto que requiere un tipo objeto, el compilador automáticamente envuelve al valor de tipo primitivo en un objeto con el envoltorio adecuado. Esto quiere decir que los valores de tipos primitivos se pueden agregar directamente a una colección:

```
private ArrayList<Integer> listaDeMarcas;  
...  
public void almacenarMarcaEnLista(int marca)  
{  
    listaDeMarcas(marca);  
}
```

La operación inversa, *unboxing*, también se lleva a cabo automáticamente, de modo que el acceso a un elemento de una colección podría ser:

```
int primerMarca = listaDeMarcas.remove(0);
```

El proceso de autoboxing se aplica en cualquier lugar en el que se pase como parámetro un tipo primitivo a un método que espera un tipo envoltorio, y cuando un valor primitivo se almacena en una variable de su correspondiente tipo envoltorio. De manera similar, el proceso de unboxing se aplica cuando un valor de tipo envoltorio se pasa como parámetro a un método que espera un valor de tipo primitivo, y cuando se almacena en una variable de tipo primitivo.

## 8.10

## La jerarquía colección

La biblioteca de Java utiliza herencia extensivamente en la definición de las clases de colecciones. Por ejemplo, la clase `ArrayList` deriva de la clase de nombre `AbstractList` que a su vez deriva de la clase `AbstractCollection`. No discutiremos esta jerarquía en este libro ya que está descrita detalladamente en varios lugares fácilmente accesibles. Una buena descripción se puede encontrar en la página web de Sun Microsystems en <http://java.sun.com/docs/books/tutorial/collections/index.html>.

Tenga en cuenta que algunos detalles de esta jerarquía requieren comprender las *interfaces* de Java de las que hablaremos en el Capítulo 10.

**Ejercicio 8.14** Utilice la documentación de la biblioteca de clases estándar de Java para encontrar información sobre la jerarquía de herencia de las clases de colecciones. Dibuje un diagrama que muestre la jerarquía.

## 8.11

## Resumen

Este capítulo presenta un primer vistazo a la herencia. Todas las clases de Java se ubican en una jerarquía de herencia. Cada clase puede tener una declaración explícita de una superclase o deriva implícitamente de la clase `Object`.

Las subclases generalmente representan especializaciones de las superclases. Por este motivo, a la relación de herencia también se la reconoce como una relación «*es-un*» («un coche *es-un* vehículo»).

Las subclases heredan todos los campos y métodos de una superclase. Los objetos de las subclases tienen todos los campos y métodos declarados en su propia clase como así también aquellos provenientes de todas las superclases. Las relaciones de herencia se pueden usar para evitar la duplicación de código, para reutilizar código y para hacer que una aplicación resulte más mantenible y más extendible.

Las subclases también forman subtipos que conducen a variables polimórficas. Los objetos de subtipos pueden ser sustituidos por objetos de supertipo y las variables permiten contener objetos que son instancias de subtipos de su tipo declarado.

La herencia permite que las estructuras de diseño de clases sean más fáciles de mantener y más flexibles. Este capítulo contiene solamente una introducción al uso de la herencia con el objetivo de mejorar las estructuras de los programas; más usos de herencia y sus beneficios serán discutidos en los siguientes capítulos.

Términos introducidos en este capítulo

**herencia, superclase(padre), subclase(hijo), es-un, jerarquía de herencia, clase abstracta, subtipo, sustitución, variable polimórfica, pérdida de tipo, enmascaramiento, autoboxing, clases envoltorio**

### Resumen de conceptos

- **herencia** La herencia nos permite definir una clase como extensión de otra.
- **superclase** Una superclase es una clase que es extendida por otra clase.
- **subclase** Una subclase es una clase que extiende (deriva de) otra clase. Hereda todos los campos y métodos de su superclase.
- **jerarquía de herencia** Las clases que están vinculadas mediante una relación de herencia forman una jerarquía de herencia.
- **constructores de superclase** El constructor de una subclase siempre debe invocar al constructor de la superclase en su primera sentencia. Si el código fuente no incluye esta llamada, Java intentará insertarla automáticamente.
- **reutilización** La herencia nos permite reutilizar en un nuevo contexto clases escritas previamente.

- **subtipos** Por analogía con la jerarquía de clase, los tipos forman una jerarquía de tipos. El tipo definido mediante una definición de subclase es un subtipo del tipo de su superclase.
- **variables y subtipos** Las variables pueden contener objetos de su tipo declarado o de cualquier subtipo de su tipo declarado.
- **sustitución** Los objetos subtipo pueden usarse cada vez que se espera un supertipo. Esto se conoce como sustitución.
- **Object** Todas las clases que no tienen una superclase explícita tienen a **Object** como su superclase.
- **Autoboxing** El proceso de autoboxing se lleva a cabo automáticamente cuando se usa un valor de un tipo primitivo en un contexto que requiere un tipo objeto.

**Ejercicio 8.15** Retome el proyecto *curso-de-laboratorio* del Capítulo 1. Agregue instructores al proyecto (cada curso de laboratorio puede tener muchos estudiantes y un solo instructor). Use herencia para evitar la duplicación de código entre los estudiantes y los instructores (ambos tienen un nombre, detalles de contacto, etc.).

**Ejercicio 8.16** Dibuje una jerarquía de herencia que represente las partes de una computadora (procesador, memoria, disco rígido, compactera, impresora, escáner, teclado, ratón, etc.).

**Ejercicio 8.17** Observe el siguiente código. Se tienen cuatro clases (O, X, T y M) y una variable de cada una de ellas.

```
O o;  
X x;  
T t;  
M m;
```

Las siguientes asignaciones son todas legales (asuma que todas compilan).

```
m = t;  
m = x;  
o = t;
```

Las siguientes asignaciones son todas ilegales (provocan error en la compilación):

```
o = m;  
o = x;  
x = o;
```

¿Qué puede decir sobre la relación entre estas clases?

**Ejercicio 8.18** Dibuje una jerarquía de herencia de **AbstractList** y todas sus subclases (directas o indirectas) tal como se encuentran definidas en la biblioteca estándar de Java.

## CAPÍTULO

# 9

## Algo más sobre herencia

Principales conceptos que se abordan en este capítulo

- método polimórfico
- tipo estático y tipo dinámico
- sobrescritura
- método de búsqueda dinámica

Construcciones Java que se abordan en este capítulo

`super` (en métodos), `toString`, `protected`

En el último capítulo hemos introducido los principales conceptos de la herencia mediante el ejemplo DoME. Si bien hemos visto a través de este ejemplo los fundamentos básicos de la herencia, todavía existen numerosos detalles importantes que aún no hemos investigado. La herencia es central para comprender y usar lenguajes orientados a objetos, y para poder progresar a partir de aquí, es necesario comprenderla con cierto nivel de detalle.

En este capítulo continuaremos usando el ejemplo DoME para explorar las cuestiones más importantes que nos restan ver sobre herencia y polimorfismo.

### 9.1

### El problema: el método imprimir de DoME

Cuando experimentó con los ejemplos de DoME en el Capítulo 8, probablemente notó que la segunda versión, la que usa herencia, tiene un problema: el método `imprimir` no muestra todos los datos de los elementos.

Veamos un ejemplo. Asuma que creamos un objeto CD y un objeto DVD con los siguientes datos:

```
CD: Frank Sinatra: A Swingin' Affair
    16 temas
    64 minutos
    Lo tengo: Sí
    Comentario: es mi álbum favorito de Sinatra
```

```
DVD: O Brother, Where Art Thou?
```

```

Directores: Joel y Ethan Coen
106 minutos
Lo tengo: No
Comentario: ¡La mejor película de los hermanos Coen!

```

Si entramos estos objetos en la base de datos y luego invocamos la primera versión del método listar de la base (el que no usa herencia) se imprime:

```

CD: A Swingin' Affair (64 minutos) *
    Frank Sinatra
    temas: 16
    es mi álbum favorito de Sinatra

```

```

DVD: O Brother, Where Art Thou? (106 minutos)
    Joel y Ethan Coen
    ¡La mejor película de los hermanos Coen!

```

Aquí aparece toda la información y podemos cambiar la implementación del método imprimir para que imprima en cualquier formato que queramos.

Comparamos esta impresión con el resultado de la segunda versión de DoME (con herencia) que imprime solamente

```

título: A Swingin' Affair (64 minutos) *
    es mi álbum favorito de Sinatra

```

```

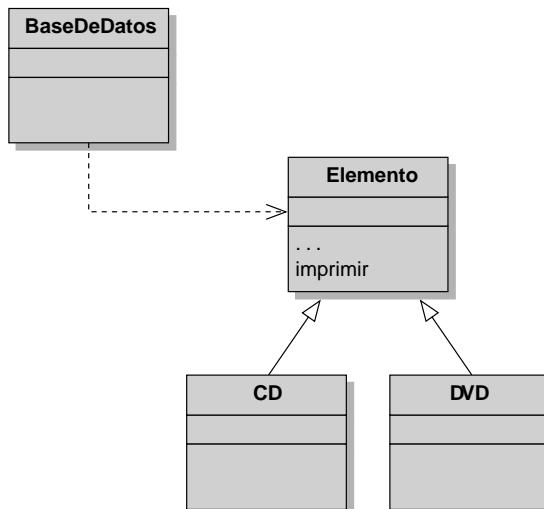
título: O Brother, Where Art Thou? (106 minutos)
    ¡La mejor película de los hermanos Coen!

```

Vemos en este caso que falta la información sobre el intérprete del CD y el número de temas que contiene, así como también falta el director de la película en DVD. El motivo de esto es muy simple: en esta versión, el método imprimir está implementado en la clase Elemento, no en las clases DVD o CD (Figura 9.1). En los métodos de Elemento sólo están disponibles los campos declarados en la clase Elemento. Si tratamos de acceder al campo intérprete del CD desde el método imprimir de Elemento, se informará un error. Este hecho ilustra el importante principio de que la herencia tiene una sola vía: CD hereda los campos de Elemento pero Elemento continúa sin conocer nada sobre los campos de sus subclases.

**Figura 9.1**

Impresión, versión 1:  
el método imprimir  
en la superclase



## 9.2

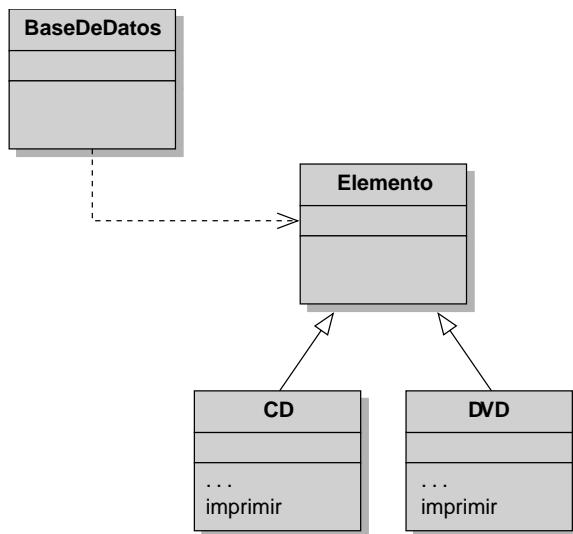
## Tipo estático y tipo dinámico

El intento de resolver el problema de desarrollar un método `imprimir` completo y polimórfico nos conduce a la discusión sobre *tipos estáticos* y *tipos dinámicos* y sobre *despacho de métodos*. Pero, comencemos desde el principio.

Un primer intento de solución del problema de la impresión podría consistir en mover el método `imprimir` a las subclases (Figura 9.2). De esta manera, y dado que el método ahora pertenecería a las clases `CD` y `DVD`, podríamos acceder a los campos específicos de los objetos `CD` y `DVD`; también tendríamos acceso a los campos heredados mediante una llamada a sus métodos de acceso definidos en la clase `Elemento`. Esta modificación nos posibilitaría imprimir nuevamente el conjunto completo de la información. Pruebe este camino completando el Ejercicio 9.1.

**Figura 9.2**

Impresión, versión 2:  
el método `imprimir`  
en las subclases



**Ejercicio 9.1** Abra su última versión del proyecto DoME (si aún no tiene su propia versión, puede usar el proyecto *dome-v2*). Elimine el método `imprimir` de la clase `Elemento` y muévalo a las clases `DVD` y `CD`. Compile. ¿Qué observa?

Cuando tratamos de mover el método `imprimir` desde la clase `Elemento` a las subclases notamos que tenemos algunos problemas: el proyecto no compila más. Hay dos cuestiones importantes:

- Tenemos errores en las clases `CD` y `DVD` porque no podemos acceder a los campos de la superclase.
- Tenemos un error en la clase `BaseDeDatos` porque no puede encontrar el método `imprimir`.

El motivo del primer tipo de error es que los campos de `Elemento` son de acceso privado y por lo tanto, son inaccesibles desde cualquier otra clase, incluyendo las subclases. Dado que no queremos romper el encapsulamiento convirtiendo estos campos en públicos, el camino más fácil para resolver esta cuestión es definir métodos de

acceso público para ellos. Sin embargo, en la Sección 9.8 introduciremos un tipo de acceso designado específicamente para soportar relaciones superclase-subclase.

El motivo del segundo tipo de error requiere una explicación más detallada que se explora en la siguiente sección.

### 9.2.1 Invocar a `imprimir` desde `BaseDeDatos`

Primeramente investigamos el problema de llamar al método `imprimir` desde `BaseDeDatos`. Las líneas de código relevantes de la clase `BaseDeDatos` son

```
for (Elemento elemento : elementos) {
    elemento.imprimir();
    System.out.println();
}
```

La sentencia `for` accede a cada elemento de la colección; la primera sentencia del cuerpo del ciclo trata de invocar al método `imprimir` sobre el elemento. El compilador nos informa que no puede encontrar un método `imprimir` para el elemento.

Por un lado, esto parece lógico: `Elemento` no tienen más un método `imprimir` (véase Figura 9.2) pero por otro lado es molesto. Sabemos que cada objeto elemento de la colección es, de hecho, un objeto `DVD` o un objeto `CD` y ambos tienen métodos `imprimir`. Esto quiere decir que la llamada `elemento.imprimir()` debiera funcionar puesto que, ya sea el elemento un `CD` o un `DVD`, sabemos que cuenta con un método `imprimir`.

Para comprender más detalladamente esta cuestión necesitamos ver más de cerca los tipos. Consideremos la siguiente sentencia:

```
Coche a1 = new Coche();
```

Decimos que el tipo de `a1` es `Coche`. Antes de que encontráramos la herencia, no había ninguna necesidad de distinguir si mediante la expresión «tipo de `a1`» queríamos decir «el tipo de la variable `a1`» o «el tipo del objeto almacenado en `a1`». Esta diferenciación no tenía importancia porque el tipo de la variable y el tipo del objeto almacenado eran siempre iguales.

Ahora que conocemos la existencia de los subtipos necesitamos ser más precisos. Consideremos la siguiente sentencia:

```
Vehiculo v1 = new Coche();
```

¿Cuál es el tipo de `v1`? Esto depende precisamente de qué queremos decir con «tipo de `v1`». El tipo de la variable `v1` es `Vehiculo`; el tipo del objeto almacenado en `v1` es `Coche`. A través del subtipoado y de las reglas de sustitución ahora tenemos situaciones en las que el tipo de la variable y el tipo del objeto almacenado no son exactamente los mismos.

Introducimos algo de terminología para que nos sea más fácil hablar sobre este tema:

- Denominamos *tipo estático* al tipo declarado de una variable porque la variable se declara en el código fuente, la representación estática del programa.
- Denominamos *tipo dinámico* al tipo del objeto almacenado en una variable porque depende de su asignación en tiempo de ejecución, el comportamiento dinámico del programa.

#### Concepto

El **tipo estático** de una variable `v` es el tipo declarado en el código fuente en la sentencia de declaración de la variable.

**Concepto**

El **tipo dinámico** de una variable *v* es el tipo del objeto que está almacenado actualmente en la variable *v*.

Por lo que, volviendo a la sentencia anterior ahora podemos establecer más precisamente que: el tipo estático de *v1* es *Vehiculo* y el tipo dinámico de *v1* es *Coche*.

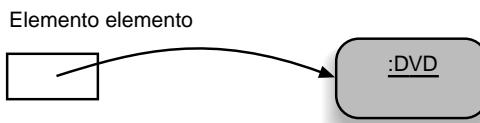
Ahora podemos parafrasear nuestra discusión sobre la llamada al método *imprimir* del elemento en la clase *BaseDeDatos*. En el momento de la llamada

```
elemento.print();
```

el tipo estático de la variable *elemento* es *Elemento* mientras que su tipo dinámico puede ser tanto *CD* como *DVD* (Figura 9.3). No sabemos exactamente cuál es su tipo ya que asumimos que hemos ingresado tanto objetos *CD* como objetos *DVD* en nuestra base de datos.

**Figura 9.3**

Variable de tipo  
Elemento que  
contiene un objeto de  
tipo DVD



El compilador informa un error porque cuando controla los tipos usa el tipo estático. El tipo dinámico se conoce, frecuentemente, sólo en tiempo de ejecución por lo que el compilador no tiene otra opción más que usar el tipo estático cuando quiere hacer alguna verificación de tipos en tiempo de compilación. El tipo estático de *elemento* es *Elemento* y *Elemento* no posee un método *imprimir*. El comportamiento del compilador es razonable porque, en realidad, no tiene ninguna garantía de que todas las subclases de *Elemento* definirán un método *imprimir* y esto, en la práctica, es imposible de controlar.

En otras palabras, para que esta llamada funcione, la clase *Elemento* debe tener un método *imprimir*, de modo que volvemos al punto de partida de nuestro problema original sin haber hecho ningún progreso.

**Ejercicio 9.2** En su proyecto DoME agregue nuevamente un método *imprimir* en la clase *Elemento*. Por ahora, escriba en el cuerpo del método una sola sentencia que imprima sólo el título. Luego modifique el método *imprimir* en las clases *CD* y *DVD* de tal manera que la versión en la clase *CD* imprima solamente el intérprete y la de la clase *DVD* sólo imprima el director. Esta modificación elimina los otros errores encontrados con anterioridad (volveremos a ellos enseñada).

Ahora debiera tener una situación similar a la correspondiente a la Figura 9.4, con métodos *imprimir* en las tres clases. Compile su proyecto. (Si aparecen algunos errores, elimínelos. Este diseño debiera funcionar.)

Antes de ejecutar, prediga cuál de los métodos *imprimir* será invocado cuando se ejecuta el método *listar* de la clase *BaseDeDatos*.

Pruébelo. Ingrese un *CD* y un *DVD* en la base e invoque el método *listar* de *BaseDeDatos*. ¿Qué métodos *imprimir* se ejecutaron? ¿Fue correcta su predicción? Trate de explicar sus observaciones.

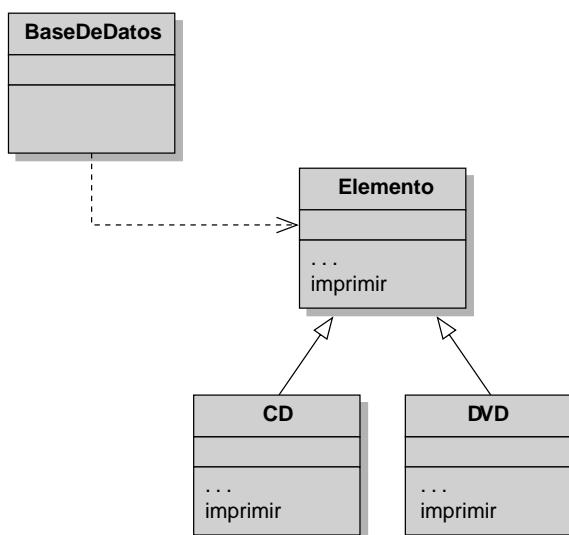
**9.3****Sobrescribir**

El siguiente diseño que discutiremos es uno en el que tanto la superclase como las subclases tienen un método `imprimir` (Figura 9.4). La signatura de todos los métodos `imprimir` es exactamente la misma.

El Código 9.1 muestra los detalles relevantes del código de las tres clases. La clase `Elemento` tiene un método `imprimir` que imprime todos los campos que están declarados en `Elemento` (aquellos que son comunes a los CD y a los DVD) y las subclases `CD` y `DVD` imprimen los campos específicos de los objetos `CD` y `DVD` respectivamente.

**Figura 9.4**

Impresión, versión 3.  
El método `imprimir` en las subclases y en la superclase

**Código 9.1**

Código de los métodos `imprimir` de las tres clases

```

public class Elemento
{
    . .
    public void imprimir()
    {
        System.out.print(titulo + " (" + duracion + "
minutos)");
        if (loTengo) {
            System.out.println("*");
        } else {
            System.out.println();
        }
        System.out.println("     " + comentario);
    }
}
public class CD extends Elemento
{
    . .
}
  
```

### Código 9.1 (continuación)

Código de los métodos `imprimir` de las tres clases

```
public void imprimir()
{
    System.out.println("     " + interprete);
    System.out.println("     temas: " + numeroDeTemas);
}
}
public class DVD extends Elemento
{
    .
    .
    public void imprimir()
    {
        System.out.println("     director: " + director);
    }
}
```

#### Concepto

**Sobrescritura.** Una subclase puede sobrescribir la implementación de un método. Para hacerlo, la subclase declara un método con la misma firma que la superclase pero con un cuerpo diferente. El método sobrescrito tiene precedencia cuando se invoca sobre objetos de la subclase.

Este diseño funciona un poco mejor: compila y puede ser ejecutado (aunque todavía no está perfecto). Proporcionamos una implementación de este diseño mediante el proyecto *dome-v3*. (Si resolvió el Ejercicio 9.2 entonces ya cuenta con una implementación similar a este diseño en su propia versión.)

La técnica que usamos acá se denomina *sobrescritura* (algunas veces también se hace referencia a esta técnica como *redefinición*). La sobrescritura es una situación en la que un método está definido en una superclase (en este ejemplo, el método `imprimir` de la clase `Elemento`) y un método, con exactamente la misma firma, está definido en la subclase.

En esta situación, los objetos de la subclase tienen dos métodos con el mismo nombre y la misma firma: uno heredado de la superclase y el otro propio de la subclase. ¿Cuál de estos dos se ejecutará cuando se invoque este método?

## 9.4

## Búsqueda dinámica del método

Un detalle sorprendente es lo que se imprime exactamente, una vez que ejecutamos el método `listar` de la base de datos. Si creamos nuevamente los objetos descritos en la Sección 9.1, la salida del método `listar` en nuestra nueva versión del programa es

```
Frank Sinatra
temas: 16
```

```
director: Joel y Ethan Coen
```

Podemos ver a partir de esta salida que se ejecutaron los métodos `imprimir` de `CD` y de `DVD` pero no se ejecutó el método de `Elemento`.

Esto puede parecer un poco extraño al principio. Nuestra investigación en la Sección 9.2 ha mostrado que el compilador insistió en que el método `imprimir` esté en la clase `Elemento`, no le alcanzaba con que los métodos estuvieran en las subclases. Este experimento ahora nos muestra que el método de la clase `Elemento` no se ejecuta para nada, pero sí se ejecutan los métodos de las subclases. Brevemente:

- El control de tipos que realiza el compilador es sobre el tipo estático, pero en tiempo de ejecución los métodos que se ejecutan son los que corresponden al tipo dinámico.

Esta es una afirmación bastante importante pero, para comprenderla mejor, veamos con más detalle cómo se invocan los métodos. Este procedimiento se conoce como *búsqueda de método*, *ligadura de método* o *despacho de método*. En este libro, nosotros usamos la terminología «búsqueda de método».

Comenzamos con un escenario sencillo de búsqueda de método. Suponga que tenemos un objeto de clase DVD almacenado en una variable v1 declarada de tipo DVD (Figura 9.5). La clase DVD tiene un método `imprimir` y no tiene declarada ninguna superclase. Esta es una situación muy simple que no involucra herencia ni polimorfismo. Luego, ejecutamos la sentencia

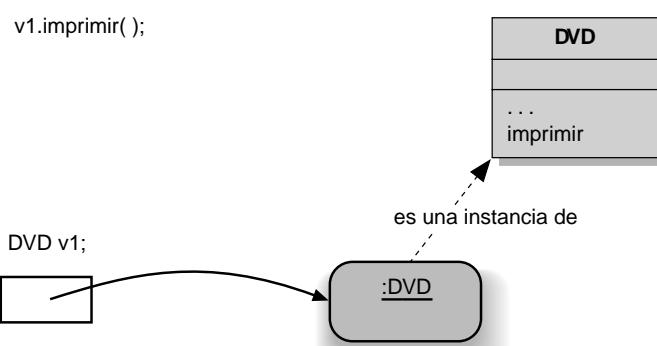
```
v1.imprimir();
```

Cuando se ejecute esta sentencia, se invoca al método `imprimir` en los siguientes pasos:

1. Se accede a la variable v1.
2. Se encuentra el objeto almacenado en esa variable (siguiendo la referencia).
3. Se encuentra la clase del objeto (siguiendo la referencia «es instancia de»).
4. Se encuentra la implementación del método `imprimir` en la clase y se ejecuta.

**Figura 9.5**

Búsqueda de un método con un único objeto



Todo esto es muy claro y no resulta sorprendente.

A continuación, vemos la búsqueda de un método cuando hay herencia. El escenario es similar al anterior, pero esta vez la clase DVD tiene una superclase, Elemento, y el método `imprimir` está definido sólo en la superclase (Figura 9.6).

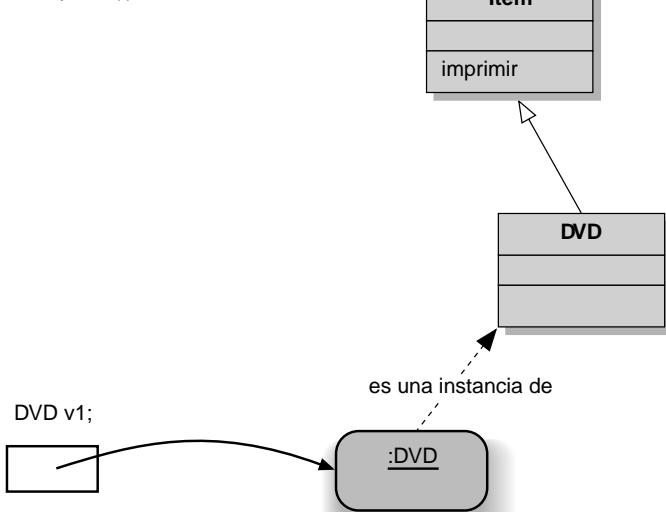
Ejecutamos la misma sentencia. La invocación al método comienza de manera similar: se ejecutan nuevamente los pasos 1 al 3 del escenario anterior pero luego continúa de manera diferente:

4. No se encuentra ningún método `imprimir` en la clase DVD.
5. Dado que no se encontró ningún método que coincida, se busca en la superclase un método que coincida. Si no se encuentra ningún método en la superclase, se busca en la siguiente superclase (si es que existe). Esta búsqueda continúa hacia arriba por toda la jerarquía de

**Figura 9.6**

Búsqueda de un método cuando hay herencia

v1.imprimir( );



herencia de la clase `Object` hasta que se encuentre definitivamente un método. Tenga en cuenta que, en tiempo de ejecución, debe encontrarse definitivamente un método que coincida, de lo contrario la clase no habría compilado.

6. En nuestro ejemplo, el método `imprimir` es encontrado en la clase `Elemento` y es el que será ejecutado.

Este escenario ilustra la manera en que los objetos heredan los métodos. Cualquier método que se encuentre en la superclase puede ser invocado sobre un objeto de la subclase y será correctamente encontrado y ejecutado.

Ahora llegamos al escenario más interesante: la búsqueda de métodos con una variable polimórfica y un método sobrescrito (Figura 9.7). El escenario nuevamente es similar al anterior pero existen dos cambios:

- El tipo declarado de la variable `v1` ahora es `Elemento`, no `DVD`.
- El método `imprimir` está definido en la clase `Elemento` y redefinido (o sobrescrito) en la clase `DVD`.

Este escenario es el más importante para comprender el comportamiento de nuestra aplicación DoME y para encontrar una solución a nuestro problema con el método `imprimir`.

Los pasos que se siguen para la ejecución del método son exactamente los mismos pasos 1 al 4 del primer escenario. Léalos nuevamente.

Es importante hacer algunas observaciones:

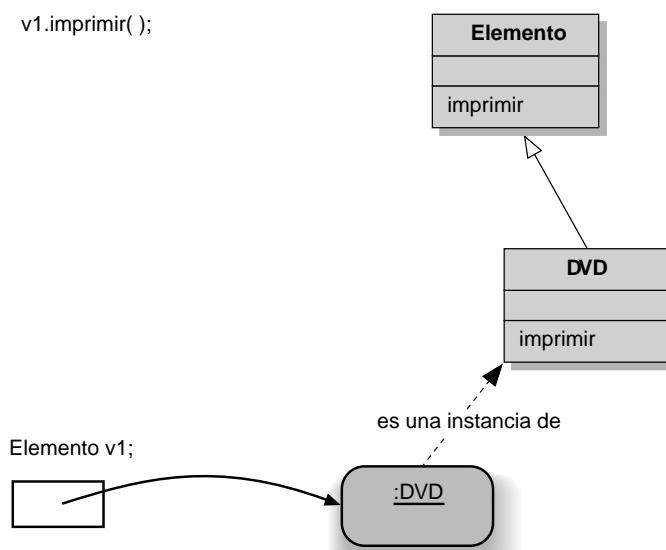
- No se usa ninguna regla especial para la búsqueda del método en los casos en los que el tipo dinámico no sea igual al tipo estático. El comportamiento que observamos es un resultado de las reglas generales.
- El método que se encuentra primero y que se ejecuta está determinado por el tipo dinámico, no por el tipo estático. En otras palabras, el hecho de que el tipo declarado de la variable `v1` ahora es `Elemento` no tiene ningún efecto. La instancia con la que estamos trabajando es de la clase `DVD`, y esto es todo lo que cuenta.

- Los métodos sobrescritos en las subclases tienen precedencia sobre los métodos de las superclases. Dado que la búsqueda de método comienza en la clase dinámica de la instancia (al final de la jerarquía de herencia) la última redefinición de un método es la que se encuentra primero y la que se ejecuta.
- Cuando un método está sobrescrito, sólo se ejecuta la última versión (la más baja en la jerarquía de herencia). Las versiones del mismo método en cualquier superclase no se ejecutan automáticamente.

Esto explica el comportamiento que observamos en nuestro proyecto DoME. Los métodos `imprimir` de las subclases (CD y DVD) sólo se ejecutan cuando se imprimen los elementos, produciendo listados incompletos. En la siguiente sección discutiremos sobre cómo podemos solucionar este inconveniente.

**Figura 9.7**

Búsqueda de un método con polimorfismo y sobrescritura



## 9.5

### Llamada a `super` en métodos

Ahora que conocemos detalladamente cómo se ejecutan los métodos sobrescritos podemos comprender la solución al problema de la impresión. Es fácil ver que lo que queremos lograr es que, para cada llamada al método `imprimir` de, digamos un objeto CD, se ejecuten para el mismo objeto tanto el método `imprimir` de la clase `Elemento` como el de la clase `CD`. De esta manera se imprimirán todos los detalles. (Se discute una solución diferente más adelante en este capítulo.)

De hecho, esta solución es muy fácil de llevar a cabo: podemos simplemente usar el constructor de la superclase que ya hemos encontrado en el contexto de los constructores en el Capítulo 8. El Código 9.2 ilustra esta idea con el método `imprimir` de la clase `CD`.

Cuando ahora se invoque al método `imprimir` sobre un objeto `CD`, inicialmente se invocará al método `imprimir` de la clase `CD`. En su primera sentencia, este método se convertirá en una invocación al método `imprimir` de la superclase que imprime la información general del elemento. Cuando el control regrese del método de la superclase, las restantes sentencias del método de la subclase imprimirán los campos distintivos de la clase `CD`.

**Código 9.2**

Redefinición del método con una llamada a *super*

```
public void imprimir()
{
    super.imprimir();
    System.out.println("      " + interprete);
    System.out.println("temas: " + numeroDeTemas);
}
```

Hay tres detalles importantes para resaltar:

- Al contrario que las llamadas a *super* en los constructores, el nombre del método de la superclase está explícitamente establecido. Una llamada a *super* en un método siempre tiene la forma

*super.nombre-del-método ( parámetros )*

La lista de parámetros por supuesto que puede quedar vacía.

- Nuevamente, y en contra de la regla de las llamadas a *super* en los constructores, la llamada a *super* en los métodos puede ocurrir en cualquier lugar dentro de dicho método. No tiene por qué ocurrir en su primer sentencia.
- Al contrario que en las llamadas a *super* en los constructores, no se genera automáticamente ninguna llamada a *super* y tampoco se requiere ninguna llamada a *super*, es completamente opcional. De modo que el comportamiento por defecto presenta el efecto de un método de una subclase ocultando completamente (sobrescribiendo) la versión de la superclase del mismo método.

**Ejercicio 9.3** Modifique su última versión del proyecto DoME para incluir una llamada a *super* en el método *imprimir*. Pruébelo. ¿Se comporta como era de esperar? ¿Encuentra algún problema con esta solución?

**Ejercicio 9.4** Cambie el formato de la salida de modo que se imprima la cadena: «CD:» o «DVD:» (dependiendo del tipo del elemento) que preceda a los detalles de cada uno.

**9.6****Método polimórfico**

Lo que hemos discutido en las secciones anteriores (9.2 a 9.5) son justamente otras formas de polimorfismo; es lo que se conoce como *despacho de método polimórfico* (o abreviadamente, *método polimórfico*).

Recuerde que una variable polimórfica es aquella que puede almacenar objetos de diversos tipos (cada variable objeto en Java es potencialmente polimórfica). De manera similar, las llamadas a métodos en Java son polimórficas dado que ellas pueden invocar diferentes métodos en diferentes momentos. Por ejemplo, la sentencia

*elemento.imprimir();*

puede invocar al método *imprimir* de CD en un momento dado y al método *imprimir* de DVD en otro momento, dependiendo del tipo dinámico de la variable *elemento*.

**Concepto****Método polimórfico.**

Las llamadas a métodos en Java son polimórficas. El mismo método puede invocar en diferentes momentos diferentes métodos dependiendo del tipo dinámico de la variable usada para hacer la invocación.

## 9.7

## Métodos de Object: `toString`

En el Capítulo 8 hemos mencionado que la superclase universal `Object` implementa algunos métodos que luego forman parte de todos los objetos. El más interesante de estos métodos es `toString` que presentamos aquí. Si está interesado en más detalles sobre este tema puede buscar la interfaz de `Object` en la documentación de la biblioteca estándar de Java.

**Ejercicio 9.5** Busque `toString` en la documentación de la biblioteca de Java. ¿Cuáles son sus parámetros? ¿Cuál es su tipo de retorno?

El propósito del método `toString` es crear una cadena de representación de un objeto. Esto es útil para cualquier objeto que pueda ser representado textualmente en la interfaz de usuario pero también es de ayuda para todos los otros objetos; por ejemplo, los objetos pueden ser fácilmente impresos con fines de depuración de un programa.

La implementación por defecto de `toString` de la clase `Object` no puede aportar una gran cantidad de detalle. Por ejemplo, si llamamos a `toString` sobre un objeto `DVD`, recibimos una cadena similar a esta:

DVD@acdd1

### Concepto

Cada objeto en Java tiene un método `toString` que puede usarse para devolver un `String` de su representación. Típicamente, para que resulte útil, un objeto debe sobrescribir este método.

El valor de retorno muestra simplemente el nombre de la clase del objeto y un número mágico<sup>1</sup>.

**Ejercicio 9.6** Puede probar este asunto fácilmente. Cree un objeto de clase `DVD` en su proyecto y luego invoque al método `toString` a partir del submenú `Object` del menú contextual del objeto.

Para que este método resulte más útil debemos sobrescribirlo en nuestras propias clases. Podemos, por ejemplo, definir el método `imprimir` de `Elemento` en términos de una llamada a su método `toString`. En este caso, el método `toString` no imprimiría los detalles que deseamos pero crearía una cadena con el texto. El Código 9.3 muestra el código modificado.

### Código 9.3

Método `toString` para `Elemento` y para `CD`

```
public class Elemento
{
    .
    .
    public String toString()
    {
        String linea1 = titulo + " (" + duracion + " "
minutos)";
        if (loTengo) {
            return linea1 + "*\n" + " " + comentario +
"\n";
        }
        else {
    
```

<sup>1</sup> El número es, en realidad, la dirección de memoria en donde el objeto está almacenado. Esta información no es muy útil excepto para establecer su identidad. Si este número es el mismo en dos llamadas, estamos viendo el mismo objeto. Si es diferente, tenemos dos objetos distintos.

**Código 9.3  
(continuación)**

Método `toString`  
para Elemento y  
para CD

```

        return linea1 + "\n" + "    " + comentario + "\n";
    }
}
public void imprimir()
{
    System.out.println(toString());
}
}
public class CD extends Elemento
{
    .
    .
    public String toString()
    {
        return super.toString() + "    " + interprete +
            "\n      temas: " + numeroDeTemas + "\n";
    }
    public void imprimir()
    {
        System.out.println(toString());
    }
}
}

```

Finalmente, podríamos planear la eliminación completa de los métodos `imprimir` de estas clases. Un gran beneficio que se obtiene justamente al definir un método `toString` es que no mandamos en las clases `Elemento` exactamente lo que se hizo con el texto de la descripción. En la versión original, el texto siempre se imprime en la terminal. Ahora, cualquier cliente (por ejemplo, la clase `BaseDeDatos`) es libre de hacer lo que quiera con este texto: puede mostrar el texto en el área de texto de una interfaz gráfica de usuario, grabarlo en un archivo, enviarlo por una red o, como antes, imprimirlo en la terminal.

La sentencia que se usa en el cliente para imprimir el elemento podría ser similar a la siguiente:

```
System.out.println(elemento.toString());
```

En realidad, los métodos `System.out.print` y `System.out.println` son especiales con respecto a esto: si el parámetro de uno de estos métodos no es un objeto `String`, el método invoca automáticamente al método `toString` de dicho objeto. Por lo tanto, no necesitamos escribir la llamada explícitamente y en cambio, podríamos escribir

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

Ahora consideremos la versión modificada del método `listar` en la clase `BaseDeDatos` que se muestra en Código 9.4. En esta versión hemos eliminado la llamada a `toString`. ¿Compilará y se ejecutará correctamente?

De hecho, el método funciona como esperábamos. Si comprende el porqué, entonces ¡ya comprende bastante bien la mayoría de los conceptos que hemos presentado en este capítulo y en el anterior! Aquí damos una explicación detallada de la sentencia `print` dentro del ciclo `for`.

**Código 9.4**

Nueva versión del método listar de BaseDeDatos

```
public class BaseDeDatos
{
    // se omitieron los campos, los constructores y los restantes métodos
    /**
     * Imprime una lista en la terminal de texto de todos
     * los CD y
     *         * DVD actualmente almacenados.
     */
    public void listar()
    {
        for(Elemento elemento : elementos ) {
            System.out.println(elemento);
        }
    }
}
```

- El *ciclo for-each* recorre todos los elementos y los ubica en una variable con el tipo estático Elemento. El tipo dinámico es tanto CD como DVD.
- Dado que este objeto se imprime mediante System.out y no es una cadena, se invoca automáticamente su método `toString`.
- La invocación a este método es válida porque la clase Elemento (*¡el tipo estático!*) posee un método `toString`. (Recuerde: el control de tipos se realiza con el tipo estático. Esta llamada no sería permitida si la clase Elemento no tiene un método `toString`.) No obstante, el método `toString` de la clase Object garantiza que este método esté disponible siempre para cualquier clase.
- La salida aparece adecuadamente con todos los detalles necesarios porque cada tipo dinámico posible (CD y DVD) sobrescribe el método `toString` y la búsqueda dinámica del método asegura que se ejecute el método redefinido.

Generalmente, el método `toString` resulta muy útil a los fines de la depuración. Con frecuencia, es muy conveniente que los objetos puedan imprimirse fácilmente en un formato que tenga sentido. La mayoría de las clases de la biblioteca de Java sobrescriben a `toString` (por ejemplo, todas las colecciones pueden imprimirse como esta) y con frecuencia, es una buena idea también sobrescribir este método en nuestras clases.

**9.8****Acceso protegido****Concepto**

La declaración de un campo o un método como **protegido** (`protected`) permite su acceso directo desde las subclases (directas o indirectas).

En el Capítulo 8 vimos que las reglas sobre la visibilidad privada y pública de los miembros de una clase se aplican entre una subclase y su superclase, al igual que entre clases de diferentes jerarquías de la herencia. Esto puede ser algo restrictivo porque la naturaleza de la relación entre una superclase y sus subclases es claramente más estrecha que con otras clases. Por este motivo, los lenguajes orientados a objetos frecuentemente definen un nivel de acceso que está entre medias de la restricción completa del acceso privado y la total disponibilidad del acceso público. En Java este nivel de acceso se denomina *acceso protegido* y es provisto por la palabra clave `protected` como alternativa entre `public` y `private`. El Código 9.5 muestra el ejemplo de un método de acceso protegido que podríamos agregar a la clase Elemento.

**Código 9.5**

Ejemplo de un método protegido

```
protected String getTitulo()
{
    return titulo();
}
```

El acceso protegido permite acceder a los campos o a los métodos dentro de una misma clase y desde todas las subclases, pero no desde otras clases. El método `getTitulo` que se muestra en Código 9.5 puede invocarse desde la clase `Elemento` o desde cualquier subclase, pero desde otras clases.

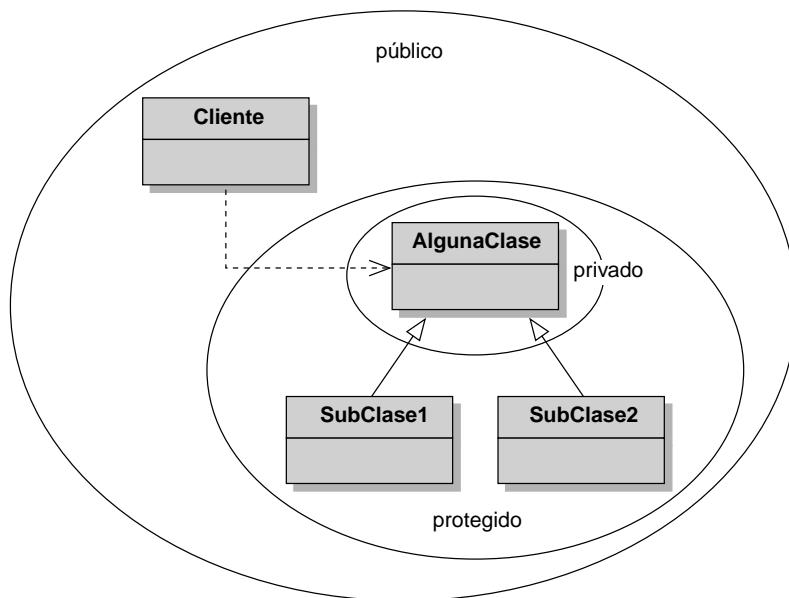
La Figura 9.8 ilustra este punto. Las áreas circulares del diagrama muestran el grupo de clases que pueden acceder a los miembros de la clase `AlgunaClase`.

Mientras que el acceso protegido puede aplicarse a cualquier miembro de una clase, generalmente se reserva para los métodos y los constructores; no es frecuente aplicarlo en los campos porque debilitaría el encapsulamiento. Siempre que sea posible, los campos modificables de las superclases deberían permanecer privados. Sin embargo, existen casos válidos ocasionales en los que es deseable el acceso directo desde una subclase. La herencia representa una forma mucho más cerrada de acoplamiento que una relación normal de cliente.

La herencia vincula las clases de manera muy cercana y la modificación de la superclase puede romper fácilmente la subclase. Este punto debiera tenerse en consideración cuando se diseñan las clases y sus relaciones.

**Figura 9.8**

Niveles de acceso:  
privado, protegido y  
público



**Ejercicio 9.7** La versión de `imprimir` que se muestra en el Código 9.2 produce la salida que se muestra en la Figura 9.9. Reordene las sentencias del método en su versión del proyecto DoME de modo que imprima los detalles tal como se muestran en la Figura 9.10.

**Ejercicio 9.8** El tener que usar una invocación a una superclase en el método `imprimir` es un poco restrictivo en cuanto a la manera en que damos formato a la salida porque depende de la manera en que la superclase da formato a sus campos. Realice todos los cambios necesarios en la clase `Elemento` y en el método `imprimir` de la clase `CD` de modo que produzca la salida que se muestra en la Figura 9.11. Cualquier cambio que realice en la clase `Elemento` estará visible sólo para sus subclases. *Pista:* para realizar esta tarea podría usar campos protegidos.

**Figura 9.9**

Possible salida de `imprimir`: una llamada a la superclase al comienzo de `imprimir` (las zonas sombreadas se imprimen mediante métodos de la superclase)

```
A Swingin' Affair (64 minutos)*
  es mi álbum favorito de Sinatra

Frank Sinatra
temas: 16
```

**Figura 9.10**

Salida alternativa de `imprimir` (las zonas sombreadas se imprimen mediante el método de la superclase)

```
CD: Frank Sinatra: A Swingin' Affair *
  (64 minutos)
  es mi álbum favorito de Sinatra
  temas: 16
```

**Figura 9.11**

Salida de `imprimir` combinando detalles de la subclase y de la superclase (las zonas sombreadas representan los detalles de la superclase)

```
CD: Frank Sinatra: A Swingin' Affair *
  temas: 16, (64 minutos)
  es mi álbum favorito de Sinatra
```

## 9.9

### Otro ejemplo de herencia con sobrescritura

Para discutir otro ejemplo de uso de herencia similar al que trabajamos, volvemos al proyecto del Capítulo 7: el proyecto `zuul`. En el juego `world-of-zuul` se usa un conjunto de objetos `Habitacion` para crear el escenario de un juego sencillo. Uno de los ejercicios al final del capítulo sugería que implemente una habitación transportadora (una habitación que conduzca hacia una ubicación aleatoria del juego cuando se trate de entrar o salir de ella). Vamos a revisitar este proyecto pues su solución puede benefi-

ciarse mucho con la herencia. Si no recuerda bien este proyecto, puede dar una leída rápida al Capítulo 7 o buscar su propio proyecto *zuul*.

No hay una única solución para esta tarea, sino que pueden existir varias soluciones diferentes posibles que pueden llevarse a cabo y que funcionen. Sin embargo, algunas soluciones son mejores que otras: pueden ser más elegantes, más fáciles de leer, más fáciles de mantener y de extender.

Asumimos que queremos implementar esta tarea de modo tal que el jugador sea transportado automáticamente a una habitación por azar cuando trate de salir de la habitación mágica transportadora. La solución más directa que viene primero a la mente de muchas personas es modificar la clase *Juego* que implementa los comandos de los jugadores; uno de estos comandos es «ir», implementado mediante el método *irAHabitacion*. En este método usamos la siguiente sentencia como la sección central de código:

```
habitacionSiguiente = habitacionActual.getSalida(direccion);
```

Esta sentencia nos lleva desde la habitación actual hacia la habitación vecina en la dirección que queremos. Para agregar un transporte mágico podríamos modificar esta sentencia de manera similar a la siguiente:

```
if (habitacionActual.getNombre().equals("Habitación transportadora"))
{
    habitacionSiguiente = getHabitacionPorAzar();
}
else {
    siguienteHabitacion = habitacionActual.getSalida(direccion);
}
```

La idea es simple: sólo controlamos si estamos en la habitación transportadora. Si es así, encontramos la siguiente habitación tomando una por azar (por supuesto que tenemos que implementar el método *getHabitacionPorAzar* de alguna manera), de lo contrario, sólo hacemos lo mismo que antes.

Esta solución funciona, pero tiene varios inconvenientes. El primero es que es una mala idea usar cadenas de texto tal como el nombre de la habitación para identificarla. Imagine que alguien quisiera traducir su juego a otro idioma (por ejemplo, al alemán), debería cambiar los nombres de las habitaciones («Habitación de transporte» se convertiría en «Transporterraum») y de pronto, ¡el juego no funcionaría más! Este es un caso claro de problema de mantenimiento.

La segunda solución que es un poco mejor podría ser el uso de una variable de instancia en lugar del nombre de la habitación para identificar la habitación transportadora. Sería algo similar a esto:

```
if (habitacionActual = habitacionTransportadora) {
    siguienteHabitacion = getHabitacionPorAzar();
}
else {
    siguienteHabitacion = habitacionActual.getSalida(direccion);
}
```

Esta vez asumimos que tenemos una variable de instancia *habitacionTransportadora* de clase *Habitacion* en la que se almacena una referencia a nuestra habitación transportadora. Ahora la verificación es independiente del nombre de la habitación y da por resultado una solución un poco mejor que la anterior.

Aunque todavía existe un caso mucho mejor. Podemos comprender las limitaciones de esta solución cuando pensamos en otro cambio relacionado con el mantenimiento. Imaginemos que queremos agregar dos habitaciones transportadoras más, de modo que nuestro juego tenga tres ubicaciones transportadoras diferentes.

Un aspecto muy bueno de nuestro diseño fue que pudimos planificar modificaciones en un solo lugar y el resto del juego quedó completamente independiente. Por ejemplo, pudimos cambiar fácilmente el esquema de las habitaciones y todo siguió funcionando (¡Un buen puntaje para el mantenimiento!). Aunque, con nuestra solución actual, se rompe esta independencia. Si agregamos dos nuevas habitaciones transportadoras tendremos que agregar dos variables de instancia o un arreglo (para almacenar las referencias a dichas habitaciones) y tenemos que modificar nuestro método `irAHabitacion` para agregar un control para estas habitaciones. En términos de facilidad de modificación, hemos retrocedido.

Por lo tanto, la pregunta es: ¿podemos encontrar una solución que no requiera un cambio en la implementación del comando cada vez que agregamos una nueva habitación transportadora? Esta es nuestra próxima idea.

Podemos agregar un método `esHabitacionTransportadora` en la clase `Habitacion`. De esta manera, el objeto `Juego` no tiene que recordar todas las habitaciones transportadoras que hay ya que las habitaciones lo hacen por sí mismas. Cuando se crean las habitaciones, podrían obtener una bandera lógica que indique si es una habitación transportadora. Entonces, el método `irAHabitacion` podría usar el siguiente segmento de código:

```
if(habitacionActual.esHabitacionTransportadora()) {
    siguienteHabitacion = getHabitacionPorAzar();
}
else {
    siguienteHabitacion = habitacionActual.getSalida(direccion);
}
```

Ahora podemos agregar tantas habitaciones transportadoras como queramos pues no se necesitan hacer más cambios en la clase `Juego`. Sin embargo, la clase `Habitacion` tiene un campo adicional cuyo valor realmente sólo es necesario para indicar la naturaleza de una o dos de las instancias. Los casos especiales de código como éste son típicos indicadores de debilidad del diseño de clases. Esta aproximación tampoco resulta buena para la escalabilidad pues si decidíramos introducir más tipos de habitaciones especiales, cada uno requeriría su propio campo bandera y un método de acceso.

Mediante la herencia podemos hacer un mejor diseño e implementar una solución que sea más flexible que esta.

Podemos implementar una clase `HabitacionTransportadora` como una subclase de la clase `Habitacion`. En esta nueva clase sobrescribimos el método `getSalida` y cambiamos su implementación de modo que devuelva una habitación por azar:

```
public class HabitacionTransportadora extends Habitacion
{
    /**
     * Devuelve una habitación por azar, independiente del
     * parámetro
     * dirección
     */
    public Habitacion getSalida (String direccion)
```

```

    {
        return encontrarHabitacionPorAzar();
    }
    /*
     * Elige una habitación por azar
     */
    private Habitacion encontrarHabitacionPorAzar()
    {
        ... // se omitió la implementación
    }
}

```

La elegancia de esta solución reside en el hecho de que ¡no es necesario ningún cambio en la clase Juego ni en la clase Habitacion! Podemos simplemente agregar esta clase al juego existente y el método `irAHabitacion` continuará funcionando tal como lo hace. Sólo el agregar la creación de una clase `HabitacionTransportadora` al plan de modificación resulta prácticamente suficiente para que funcione. Observe también, que la nueva clase no necesita una bandera para indicar su naturaleza especial, su tipo y su comportamiento distintivo aportan esta información.

Dado que `HabitacionTransportadora` es una subclase de `Habitacion`, puede usarse en cualquier lugar donde se espere un objeto `Habitacion`. Por lo tanto, puede ser usada como una habitación vecina de cualquier otra habitación o puede ser considerada por el objeto `Juego` como la habitación actual.

Lo que hemos dejado de lado, por supuesto, es la implementación del método `encontrarHabitacionPorAzar`. En realidad, probablemente sea mejor implementarlo en una clase separada (por ejemplo, `ProducirHabitacionPorAzar`) que en la misma clase `HabitacionTransportadora`. Dejamos este tema abierto a la discusión como un ejercicio para el lector.

**Ejercicio 9.9** Implemente una habitación transportadora mediante herencia, en su versión del proyecto `zuul`.

**Ejercicio 9.10** Discuta cómo podría usarse la herencia en el proyecto `zuul` para implementar una clase jugador y una clase monstruo.

**Ejercicio 9.11** ¿Podría (o debiera) usarse herencia para crear una relación de herencia (superclase, subclase o clase hermano) entre un personaje del juego y un elemento?

## 9.10

## Resumen

Cuando tratamos con clases, con subclases y con variables polimórficas tenemos que distinguir entre el tipo estático y el tipo dinámico de una variable. El tipo estático es el tipo declarado mientras que el tipo dinámico es el tipo del objeto almacenado actualmente en la variable.

El compilador realiza el control de tipos usando el tipo estático mientras que en tiempo de ejecución, la búsqueda de métodos usa el tipo dinámico. Esto nos permite crear estructuras muy flexibles mediante la sobrescritura de métodos. Aun cuando se usa una variable de supertipo para hacer una llamada a un método, la sobrescritura nos per-

mite asegurar que se invoquen los métodos especializados de cada subtipo en particular. Esto asegura que los objetos de diferentes clases puedan reaccionar de manera diferente a la misma llamada de un método.

Cuando se implementa sobrescritura de métodos se puede usar la palabra clave `super` para invocar la versión del método de la superclase. Si los campos o los métodos se declaran con el modificador de acceso `protected`, las subclases tienen permitido el acceso a ellos pero las restantes clases no.

Términos introducidos en este capítulo

**tipo estático, tipo dinámico, sobrescritura, redefinición, búsqueda de método, despacho de método, método polimórfico, protegido**

## Resumen de conceptos

- **tipo estático** El tipo estático de una variable `v` es el tipo declarado en el código fuente en la sentencia de declaración de la variable.
- **tipo dinámico** El tipo dinámico de una variable `v` es el tipo del objeto que está actualmente almacenado en `v`.
- **sobrescritura** Una subclase puede sobrescribir la implementación de un método. Para hacerlo, la subclase declara un método con la misma firma que la superclase, pero con un cuerpo diferente. El método sobrescrito tiene precedencia en las llamadas a métodos sobre objetos de la subclase.
- **método polimórfico** Las llamadas a métodos en Java son polimórficas. La misma llamada a un método en diferentes momentos puede invocar diferentes métodos, dependiendo del tipo dinámico de la variable usada para hacer la invocación.
- **toString** Cada objeto en Java tiene un método `toString` que puede usarse para devolver una representación String del mismo. Típicamente, para que sea útil, una clase debe sobrescribir este método.
- **protected** La declaración de un campo o de un método como `protected` permite el acceso directo al mismo desde las subclases (directas o indirectas).

**Ejercicio 9.12** En las siguientes líneas de código:

```
Dispositivo disp = new Impresora();  
disp.getNombre();
```

`Impresora` es una subclase de `Dispositivo`. ¿Cuál de estas dos clases debe tener la definición del método `getNombre` en su código para que compile?

**Ejercicio 9.13** En la misma situación que la del ejercicio anterior, si ambas clases tienen una implementación del método `getNombre`, ¿cuál de ellos se ejecutará?

**Ejercicio 9.14** Suponga que escribe una clase `Estudiante` que no tiene una superclase declarada y no escribe un método `toString`. Considere las siguientes líneas de código:

```
Estudiante est = new Estudiante();
String s = est.toString();
```

¿Compilarán estas líneas de código? ¿Qué ocurrirá exactamente cuando intente ejecutar estas líneas?

**Ejercicio 9.15** En la misma situación que el ejercicio anterior (clase `Estudiante` sin método `toString`), ¿compilarán las siguientes líneas? ¿Por qué?

```
Estudiante est = new Estudiante();
System.out.println(est);
```

**Ejercicio 9.16** Suponga que la clase `Estudiante` sobrescribe `toString` de modo tal que devuelve el nombre del estudiante. Ahora se tiene una lista de estudiantes. ¿Compilarán las siguientes líneas de código? De no ser así, ¿por qué no? ¿qué imprimirán? Explique detalladamente lo que ocurre.

```
for(Object est : miLista) {
    System.out.println(est);
}
```

**Ejercicio 9.17** Escriba algunas líneas de código que den por resultado una situación en la que una variable `x` tenga tipo estático `T` y tipo dinámico `D`.



# CAPÍTULO **10**

## Más técnicas de abstracción

Principales conceptos que se abordan en este capítulo

■ clases abstractas

■ interfaces

Construcciones Java que se abordan en este capítulo

`abstract, implements interface, instanceof`

En este capítulo examinamos otras técnicas relacionadas con la herencia, que se pueden usar para perfeccionar las estructuras de clases y mejorar la mantenibilidad y la extensibilidad. Estas técnicas introducen un mejor método de representación de las abstracciones en los programas orientados a objetos.

En los dos capítulos anteriores hemos discutido los aspectos más importantes de la herencia en el diseño de una aplicación, pero hasta ahora se han ignorado varios problemas y usos más avanzados. Completaremos el cuadro de la herencia mediante un nuevo ejemplo.

El próximo proyecto que usaremos en este capítulo es una simulación; lo usamos para discutir nuevamente sobre herencia y para ver que nos enfrentamos con algunos problemas nuevos. Apuntando a resolver estos problemas, se presentan las clases abstractas y las interfaces.

### **10.1**

### **Simulaciones**

Las computadoras se usan frecuentemente para ejecutar simulaciones de un sistema real. Esto incluye simulaciones del tráfico de una ciudad, el informe meteorológico, la simulación de explosiones nucleares, el análisis del stock, simulaciones ambientales y muchas otras más. De hecho, varias de las computadoras más potentes del mundo se usan para ejecutar algún tipo de simulación.

Al crear una simulación por computadora tratamos de modelar el comportamiento de un subconjunto del mundo real en un modelo de software. Cada simulación es, necesariamente, una simplificación del objeto real. La decisión sobre los detalles que se

dejan de lado y los que se incluyen en el modelo es, generalmente, una tarea desafiante. Cuanto más detallada es una simulación, resulta más seguro que los resultados del comportamiento pronosticado se ajusten más a los del sistema real, pero el incremento en el nivel de detalle aumenta los requerimientos: se necesitan computadoras más poderosas y más tiempo de programación. Un ejemplo muy conocido es el pronóstico meteorológico: los modelos climáticos para la simulación de las condiciones del tiempo han sido mejorados en las últimas décadas aumentando la cantidad de detalles que se incluyen. Como resultado, el pronóstico meteorológico ha mejorado significativamente su nivel de certeza (pero están lejos de ser perfectos, como todos bien sabemos por experiencia); muchas de estas mejoras han sido posibles debido a los avances en la tecnología de la computación.

El beneficio de las simulaciones es que podemos llevar a cabo experimentos que no podríamos hacer en un sistema real, ya sea porque no tenemos el control sobre lo real (por ejemplo, el clima) o porque es demasiado costoso, peligroso o irreversible en caso de desastre. Podemos usar una simulación para investigar el comportamiento del sistema bajo ciertas circunstancias o para investigar cuestiones del estilo «que pasaría si...».

Las simulaciones del medio ambiente, por ejemplo, podrían utilizarse para intentar predecir los efectos de la actividad humana en su hábitat natural. Considere el caso de un parque nacional que contenga especies en extinción y el propósito de construir una autopista que lo atraviese completamente separándolo en dos partes. Los partidarios de construir la autopista postularían que el hecho de dividir el parque en dos mitades no afectará a los animales que viven en él pero los ambientalistas proclamarían lo contrario. ¿Cómo podemos decir cuál será el efecto probable sin construir la autopista?

La cuestión que subyace a la pregunta es: si es significativo para la supervivencia de las especies el hecho de tener una zona de hábitat conectada o si les resulta bueno disponer de dos áreas desconectadas (con el mismo tamaño total). Antes de construir primero la autopista y luego observar qué ocurre, podríamos simular el efecto con el fin de tomar una decisión bien informada<sup>1</sup>. (Dicho sea de paso, en este caso particular, este asunto importa: el tamaño de un parque natural tiene un impacto significativo sobre su utilidad como hábitat para los animales.)

Nuestro ejemplo principal en este capítulo describe una simulación ambiental que será necesariamente más simple que el escenario que hemos descrito porque lo usamos, principalmente, para ilustrar nuevas características del diseño orientado a objetos y su implementación. Por lo tanto, no tendrá el potencial para simular con certeza varios aspectos de la naturaleza, pero algunas cosas son bastante interesantes; en particular, demostrará la estructura de las simulaciones típicas.

## 10.2

### La simulación de zorros y conejos

El escenario de simulación que hemos seleccionado para trabajar en este capítulo consiste en la evolución de poblaciones de zorros y de conejos dentro de un campo cerrado, que justamente es un caso particular de lo que se conoce como el *modelo de simulación*.

<sup>1</sup> Una cuestión pendiente en todos los casos de este tipo es, por supuesto, la calidad de la simulación. Uno puede «probar» sobre cualquier cosa con una simulación demasiado simplificada o con una simulación mal diseñada. Es esencial obtener la verdad de la simulación mediante experimentos controlados.

ción predador-presa. Esta simulación se usa frecuentemente para modelar las variaciones de los tamaños de población de especies predadoras que se alimentan a base de especies presa. Entre tales especies existe un delicado balance. Una población grande de presas puede proveer potencialmente de gran cantidad de alimento a una población pequeña de predadores. Sin embargo, un número excesivamente grande de predadores podría terminar con todas las presas y quedarse sin nada para comer. Los tamaños de las poblaciones podrían también estar influenciadas por el tamaño y la naturaleza del ambiente. Por ejemplo, un medio ambiente pequeño y cerrado podría conducir a una superpoblación de modo tal que resulte muy fácil para los predadores localizar a sus presas, o un ambiente contaminado podría reducir el número de presas y en este caso, una población modesta de predadores podría tomar prevenciones para sobrevivir. Dado que, con frecuencia, los predadores son en sí mismos presas de otras especies, la pérdida de una parte de la cadena alimenticia puede tener efectos dramáticos en la supervivencia de las otras partes.

Tal como lo hemos hecho en capítulos anteriores, comenzaremos con una versión de una aplicación que funciona perfectamente bien desde el punto de vista del usuario, pero cuya vista interna no es tan buena cuando se la juzga mediante los principios de un buen diseño orientado a objetos y de la implementación. Usaremos esta versión base para desarrollar varias versiones mejoradas que progresivamente introducen nuevas técnicas de abstracción.

Un problema en particular de la versión base que deseamos resaltar es que no hace un buen uso de las técnicas de herencia presentadas en el Capítulo 8. Sin embargo, comenzaremos por examinar el mecanismo de la simulación sin hacer demasiadas críticas a su implementación. Una vez que comprendamos cómo funciona, estaremos en una buena posición para realizar algunas mejoras.

### 10.2.1 El proyecto zorros-y-conejos

Abra el proyecto *zorros-y-conejos-v1*. La Figura 10.1 muestra el diagrama de clases del proyecto.

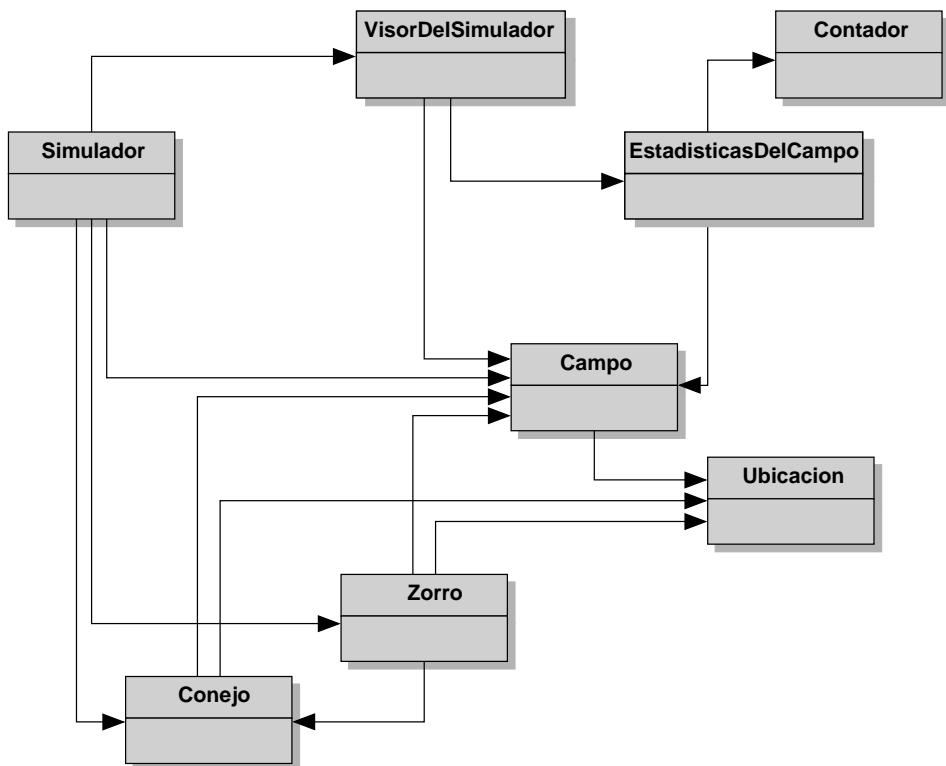
Las principales clases en las que centraremos nuestra discusión son *Simulador*, *Zorro* y *Conejo*. Las clases *Zorro* y *Conejo* proporcionan modelos sencillos del comportamiento de un predador y de una presa respectivamente. En esta implementación en particular, no pretendemos dar un modelo biológico, real y exacto de los zorros y de los conejos, simplemente tratamos de ilustrar los principios de las simulaciones del tipo predador-presa. La clase *Simulador* es la responsable de crear el estado inicial de la simulación y luego de controlarla y ejecutarla. La idea básica es sencilla: el simulador contiene una colección de zorros y conejos y lleva a cabo una secuencia de *pasos*. Cada paso permite mover a cada zorro y a cada conejo. Después de cada paso (cuando se movió cada animal) se despliega en la pantalla el estado actual del campo.

Podemos resumir el propósito de las restantes clases como sigue:

- *Campo* representa un terreno cerrado de dos dimensiones. El campo está compuesto por un número fijo de direcciones organizadas en filas y columnas. Cada dirección del campo puede ser ocupada por un animal como máximo. Cada dirección del campo puede contener una referencia a un animal o estar vacía.
- *Ubicacion* representa una posición bidimensional en el campo. La posición está determinada por un valor para la fila y un valor para la columna.

**Figura 10.1**

Diagrama de clases del proyecto zorros-y-conejos



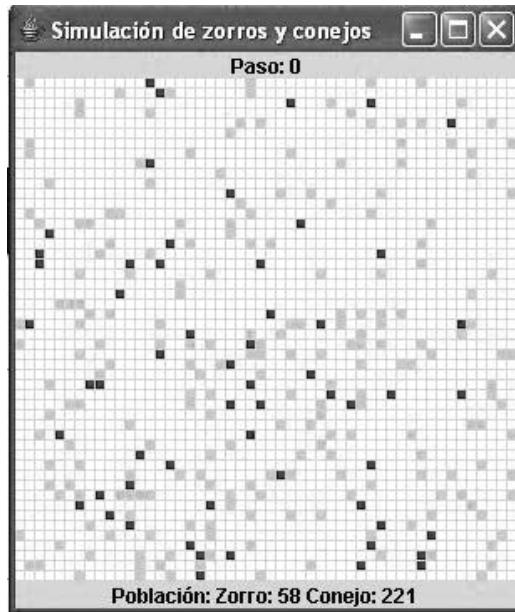
- Las clases `Simulador`, `Zorro`, `Conejo`, `Campo` y `Ubicacion` proporcionan en conjunto el modelo de la simulación. Determinan por completo el comportamiento de la simulación.
- Las clases `VisorDelSimulador`, `EstadisticasDelCampo` y `Contador` proveen una forma de mostrar la simulación de manera gráfica. El visor muestra una imagen del campo y de los contadores de cada especie (el número actual de conejos y de zorros).
- `VisorDelSimulador` ofrece una visualización gráfica del estado del campo. Se puede ver un ejemplo en la Figura 10.2.
- `EstadisticasDelCampo` proporciona los contadores del número de zorros y de conejos que hay en el campo para su visualización.
- Un `Contador` almacena la cantidad actual de un tipo de animal para colaborar con el conteo.

Trate de hacer los siguientes ejercicios para comprender cómo opera la simulación antes de leer sobre su implementación.

**Ejercicio 10.1** Cree un objeto `Simulador` mediante el constructor que no tiene parámetros y podrá ver el estado inicial de la simulación tal como se muestra en la Figura 10.2. Los cuadraditos más numerosos representan a los conejos. ¿Cambia el número de zorros si invoca una sola vez al método `simularUnPaso`?

**Figura 10.2**

Estado inicial de la simulación zorros-y-conejos



**Ejercicio 10.2** ¿Cambia el número de zorros en cada paso? ¿Qué proceso natural considera que estamos modelando, que provoca el aumento o la disminución del número de zorros?

**Ejercicio 10.3** Invoque el método `simular` para ejecutar la simulación continuamente durante un número significativo de pasos, por ejemplo 50 o 100 pasos. El número de zorros y de conejos, ¿aumenta o disminuye con tasas similares?

**Ejercicio 10.4** ¿Qué cambios observa si ejecuta la simulación un número relativamente grande de veces, por ejemplo 500 pasos? Para hacer esto puede usar el método `ejecutarSimulacionLarga`.

**Ejercicio 10.5** Use el método `inicializar` para volver al estado inicial de la simulación y luego ejecútela nuevamente. La simulación que se produce esta vez, ¿es idéntica a la anterior? Si no es así, ¿observa de todos modos que surja algún modelo similar?

**Ejercicio 10.6** Si ejecuta una simulación con un número de pasos suficientemente grande, ¿desaparecen por completo o mueren siempre todos los conejos o todos los zorros? De ser así, ¿puede precisar alguna razón sobre lo que puede estar ocurriendo?

En las siguientes secciones examinaremos la implementación inicial de las clases `Conejo`, `Zorro` y `Simulador`.

## 10.2.2 La clase `Conejo`

El Código 10.1 muestra el código fuente de la clase `Conejo`.

**Código 10.1**

La clase Conejo

```
/ se omitieron las sentencias import y el comentario de la clase
public class Conejo
{
    // Características compartidas por todos los conejos
    (campos estáticos)

    // La edad en que un conejo comienza a
    reproducirse.
    private static final int EDAD_DE_REPRODUCCION = 5;
    // La edad que puede vivir un conejo.
    private static final int EDAD_MAX = 50;
    // La probabilidad de reproducción de un conejo.
    private static final double
    PROBABILIDAD_DE_REPRODUCCION = 0.15;
    // El número máximo de nacimientos.
    private static final int MAX_TAMANIO_DE_CAMADA = 5;
    // Un número aleatorio para controlar la
    reproducción.
    private static final Random rand = new Random();

    // Características individuales (campos de instancia).

    // Edad del conejo.
    private int edad;
    // Si el conejo está vivo o no.
    private boolean vive;
    // La posición del conejo
    private Ubicacion ubicacion;
    /**
     * Crea un nuevo conejo. Se puede crear un conejo
    con edad
     * cero (un nuevo nacimiento) o con una edad por
    azar.
     *
     * @param edadPorAzar Si es true, el conejo tendrá
    una edad por azar.
    */
    public Conejo(boolean edadPorAzar)
    {
        // Se omite el cuerpo del constructor
    }

    /**
     * Esto es lo que hace el conejo la mayor parte
    del tiempo,
     * corre por todas partes. Algunas veces se
    reproducirá o morirá
     * de viejo.
    }
```

**Código 10.1  
(continuación)**  
La clase Conejo

```
        * @param campoActualizado El campo al que se
traslada.
        * @param nuevosConejos Una lista en la que se
agregan los nuevos
        *                                               conejos que nacen.
        */
    public void correr(Campo campoActualizado,
List<Conejo> conejosNuevos)
    {
        incrementarEdad();
        if(vive) {
            int nacimientos = reproducir();
            for(int n = 0; n < nacimientos; n++) {
                Conejo nuevoConejo = new
Conejo(false);
                conejosNuevos.add(nuevoConejo);
                Ubicacion ubi =
campoActualizado.direccionAdyacentePorAzar(ubicacion);
                nuevoConejo.setUbicacion(ubi);
                campoActualizado.ubicar(nuevoConejo,
ubi);
            }
            Ubicacion nuevaUbicacion =
campoActualizado.direccionAdyacenteLibre(ubicacion);
            // Sólo se traslada al campo actualizado
si la ubicación
            // está libre.
            if(nuevaUbicacion != null) {
                setUbicacion(nuevaUbicacion);
                campoActualizado.ubicar(this,
nuevaUbicacion);
            }
            else {
                // no se puede mover ni estar, superpoblación,
todas las
                // direcciones están ocupadas
                vive = false;
            }
        }
    }

/**
 * Aumenta la edad.
 * Podría dar por resultado la muerte del conejo.
 */
private void incrementarEdad()
{
    edad++;
}
```

**Código 10.1  
(continuación)**

La clase Conejo

```

        if(edad > EDAD_MAX) {
            vive = false;
        }
    }

    /**
     * Genera un número que representa el número de
     * nacimientos,
     * si es que el conejo se puede reproducir.
     * @return El número de nacimientos (puede ser
     cero).
    */
    private int reproducir()
    {
        int nacimientos = 0;
        if(sePuedeReproducir() && rand.nextDouble() <=
            PROBABILIDAD_DE_REPRODUCCION) {
            nacimientos =
rand.nextInt(MAX_TAMANIO_DE_CAMADA) + 1;
        }
        return nacimientos;
    }
    // Se omitieron los otros métodos
}

```

La clase Conejo contiene varias variables estáticas que definen la configuración de los valores que son comunes a todos los conejos. Esto incluye los valores de la edad máxima que puede vivir un conejo (definido como un número de pasos de la simulación) y el número máximo de hijos o de descendientes que se puede producir en cualquier paso. Cada conejo individual tiene tres variables de instancia que describen su estado: su edad medida en número de pasos, si aún sigue vivo y su ubicación actual en el campo.

El comportamiento del conejo se define en el método `correr` que internamente invoca a los métodos `reproducir` e `incrementarEdad`, e implementa el movimiento del conejo. En cada paso de la simulación, será invocado el método `correr` y un conejo aumentará su edad, se moverá y, si tiene edad suficiente, podrá también reproducirse. Tanto el comportamiento del movimiento como el de la reproducción tienen componentes aleatorios. La ubicación a la que se mueve el conejo se elige por azar y la reproducción ocurre aleatoriamente, controlada por el campo estático `PROBABILIDAD_DE_REPRODUCCION`.

Ya se pueden ver algunas de las simplificaciones que hemos hecho en nuestro modelo de conejos: por ejemplo, no hay ningún intento de distinguir entre masculinos y femeninos y un conejo puede, potencialmente, dar a luz una nueva prole en cada paso de la simulación.

**Ejercicio 10.7** ¿Considera que la omisión del género como un atributo de la clase Conejo conduce probablemente a una simulación incorrecta?

**Ejercicio 10.8** Comparado con la realidad, ¿piensa que existen otras simplificaciones en nuestra implementación de la clase Conejo? ¿Cree que estas simplificaciones pueden tener un impacto significativo en la exactitud de la simulación?

**Ejercicio 10.9** Experimente los efectos de alterar algunos o todos los valores de las variables estáticas de la clase Conejo. Por ejemplo, ¿Qué efecto tiene sobre una población de zorros y de conejos si se aumenta o disminuye la probabilidad de reproducción?

### 10.2.3 La clase Zorro

Hay una enorme similitud entre las clases Zorro y Conejo, de modo que solamente se muestran en el Código 10.2 los elementos distintivos.

#### Código 10.2

La clase Zorro

```
// Se omitieron las sentencias import y el comentario de la clase
public class Zorro
{
    // Características compartidas por todos los zorros
    // (campos estáticos)

    // La edad en que un zorro puede comenzar a
    // reproducirse.
    private static final int EDAD_DE_REPRODUCCION = 10;

    // Se omiten los restantes campos estáticos

    // Características individuales (campos de instancia)

    // La edad del zorro.
    private int edad;
    // Si el zorro está vivo o no.
    private boolean vive;
    // La posición del zorro.
    private Ubicacion ubicacion;
    // El nivel de comida del zorro que se incrementa
    // comiendo conejos.
    private int nivelDeComida;
    /**
     * Crea un zorro. Se puede crear un zorro mediante
     * un nuevo nacimiento
     * (edad cero y no tiene hambre) o con una edad por
     * azar.
     *
     * @param edadPorAzar Si es true, el zorro tendrá una
     * edad y un
     *                      nivel de hambre aleatorios.
     */
}
```

**Código 10.2  
(continuación)**

La clase Zorro

```

        public Zorro(boolean edadPorAzar)
        {
        // Se omite el cuerpo del constructor
        }

        /**
         * Esto es lo que hace el zorro la mayor parte del
         tiempo: caza
         * conejos. En el proceso, puede reproducirse, morir
         de hambre,
         * o morir de viejo.
         * @param campoActual El campo actualmente ocupado.
         * @param campoActualizado El campo al que se
         traslada.
         * @param zorrosNuevos Una lista en la que se
         agregan los nuevos zorros
         *                               que nacen.
         */
        public void cazar(Campo campoActual, Campo
campoActualizado,
                           List<Zorro> zorrosNuevos)
        {
            incrementarEdad();
            incrementarHambre();
            if(vive) {
                // Nacieron nuevos zorros en direcciones
                adyacentes.
                int nacimientos = reproducir();
                for(int n = 0; n < nacimientos; n++) {
                    Zorro nuevoZorro = new Zorro(false);
                    zorrosNuevos.add(nuevoZorro);
                    Ubicacion ubi =
campoActualizado.direccionAdyacentePorAzar(ubicacion);
                    nuevoZorro.setUbicacion(ubi);
                    campoActualizado.ubicar(nuevoZorro,
ubi);
                }
                // Se mueve hacia la fuente de comida,
                si es que la encuentra.
                Ubicacion nuevaUbicacion =
buscarComida(campoActualizado,
                ubicacion);
                if(nuevaUbicacion == null) {
                    // no encontró comida - se mueve
                    aleatoriamente
                    nuevaUbicacion =
campoActualizado.direccionAdyacenteLibre(ubicacion);
                }
            }
        }
    }
}

```

**Código 10.2  
(continuación)**

La clase Zorro

```
        if(nuevaUbicacion != null) {
            setUbicacion(nuevaUbicacion);
            campoActualizado.ubicar(this,
nuevaUbicacion);
        }
        else {
            // no puede moverse ni estar,
superpoblación, todas las
            // direcciones están ocupadas.
            vive = false;
        }
    }

    /**
     * Decirle al zorro que busque conejos adyacentes a
su ubicación actual.
     * Sólo come el primer conejo que encuentra vivo.
     * @param campo El campo en el que debe buscar.
     * @param ubicacion El lugar del campo en el que
está ubicado.
     * @return el lugar donde encontró comida, o null
si no encontró.
    */
    private Ubicacion buscarComida(Campo campo, Ubicacion
ubicacion)
{
    Iterator<Ubicacion> direccionesAdyacentes =
campo.direccionesAdyacentes(ubicacion);
    while(direccionesAdyacentes.hasNext()) {
        Ubicacion lugar =
direccionesAdyacentes.next();
        Object animal = campo.getObjectAt(lugar);
        if(animal instanceof Conejo) {
            Conejo conejo = (Conejo) animal;
            if(conejo.estaVivo()) {
                conejo.setComido();
                nivelDeComida =
VALOR_COMIDA_CONEJO;
                return lugar;
            }
        }
    }
    return null;
}
// Se omiten los restantes métodos
```

}

Para los zorros, el método `cazar` se invoca en cada paso y define su comportamiento. En cada paso, además de aumentar su edad y posiblemente reproducirse, un zorro busca comida (usando el método `buscarComida`). Si encuentra un conejo en una dirección adyacente entonces el conejo muere (es comido) y disminuye el nivel de comida del zorro.

**Ejercicio 10.10** Tal como lo hizo con los conejos, evalúe el grado en que hemos simplificado el modelo de los zorros y evalúe también si algunas de las simplificaciones realizadas pueden conducir probablemente a una simulación incorrecta.

**Ejercicio 10.11** El aumento de la edad máxima de los zorros en la simulación, ¿produce un número significativamente alto de zorros? O la población de conejos ¿es probable que resulte reducida a cero?

**Ejercicio 10.12** Experimente con diferentes combinaciones de valores iniciales para los zorros y para los conejos (edad de reproducción, edad máxima, probabilidad de reproducción, tamaño de la camada, etc.). ¿Siempre desaparecen por completo las especies en algunas configuraciones? ¿Existen configuraciones estables?

**Ejercicio 10.13** Experimente con diferentes tamaños de campo. (Puede hacer esto usando el segundo constructor del `Simulador`.) ¿Influye el tamaño del campo en la probabilidad de supervivencia de las especies?

**Ejercicio 10.14** Actualmente, un zorro comerá como máximo un conejo en cada paso. Modifique el método `buscarComida` de modo que los conejos ubicados en todas las direcciones adyacentes sean comidos en un solo paso. Evalúe el impacto de este cambio en los resultados de la simulación.

**Ejercicio 10.15** Cuando un zorro come un gran número de conejos en un solo paso, hay varias posibilidades diferentes sobre cómo podemos modelar su nivel de comida. Si sumamos todos los valores de comida del conejo, el zorro tendrá un nivel de comida muy alto, y será muy improbable que muera de hambre por un largo tiempo. Otra alternativa podría ser que impongamos un tope al nivel de comida del zorro, modelando el efecto de un predador que mata las presas sin tener en cuenta si tiene hambre o no. Evalúe los impactos de implementar esta elección en el resultado de la simulación.

## 10.2.4

### La clase `Simulador`: configuración

La clase `Simulador` es la parte central de la aplicación. El Código 10.3 ilustra algunas de sus características principales.

#### Código 10.3

Parte de la clase  
`Simulador`

```
/ Se omitieron las sentencias import y el comentario de la clase
public class Simulador
{
    // Se omiten las variables estáticas
    // Listas de los animales en el campo
    private List<Conejo> conejos;
```

**Código 10.3  
(continuación)**

Parte de la clase  
Simulador

```
private List<Zorro> zorros;
    // El estado actual del campo.
private Campo campo;
    // Un segundo campo que se usa para construir el
siguiente escenario
    // de la simulación.
private Campo campoActualizado;
    // El paso actual de la simulación.
private int paso;
    // Una vista gráfica de la simulación.
private VisorDelSimulador visor;

/**
 * Crea un campo de simulación de determinado
tamaño.
 * @param largo El largo del campo. Debe ser mayor
que cero.
 * @param ancho El ancho del campo. Debe ser mayor
que cero.
 */
public Simulador(int largo, int ancho)
{
    if(ancho <= 0 || largo <= 0) {
        System.out.println(
            "Las dimensiones deben ser mayores que
cero.");
        System.out.println("Uso de valores por
defecto.");
        largo = LARGO_POR_DEFECTO;
        ancho = ANCHO_POR_DEFECTO;
    }
    conejos = new ArrayList<Conejo>();
    zorros = new ArrayList<Zorro>();
    campo = new Campo(largo, ancho);
    campoActualizado = new Campo(largo, ancho);
    // Crea un visor del estado de cada ubicación
en el campo.
    visor = new VisorDelSimulador(largo, ancho);
    visor.setColor(Zorro.class, Color.blue);
    visor.setColor(Conejo.class, Color.orange);

    // Establece un punto de inicio válido.
    inicializar();
}

/**
 * Ejecuta la simulación un número determinado de
pasos a partir del
 * estado actual.
```

**Código 10.3  
(continuación)**

Parte de la clase  
Simulador

```
        * Se detiene antes del número dado de pasos si
deja de ser viable.
    */
public void simular(int numeroDePasos)
{
    for(int paso = 1; paso <= numeroDePasos &&

visor.esViable(campo); paso++) {
        simularUnPaso();
    }
}

/**
 * Ejecuta un solo paso de la simulación a partir
del estado actual.
 * Recorre el campo completo actualizando el estado
de cada zorro y
 * de cada conejo.
 */
public void simularUnPaso()
{

// Se omite el cuerpo del método

}

/**
 * Inicializa la simulación en un punto de inicio.
 */
public void inicializar()
{
    paso = 0;
    conejos.clear();
    zorros.clear();
    campo.limpiar();
    campoActualizado.limpiar();
    poblar(campo);

    // Muestra el estado inicial en el visor.
    visor.mostrarEstado(paso, campo);
}

/**
 * Puebla un campo con zorros y conejos.
 * @param campo El campo que se poblará.
 */
private void poblar(Campo campo)
{
    Random rand = new Random();
```

**Código 10.3****(continuación)**Parte de la clase  
Simulador

```

        campo.limpiar();
        for(int fila = 0; fila < campo.getLargo();
            fila++) {
            for(int col = 0; col < campo.getAncho();
                col++) {
                    if(rand.nextDouble() <=
PROBABILIDAD_DE_CREACION_DEL_ZORRO) {
                        Zorro zorro = new Zorro(true);
                        zorro.setUbicacion(fila, col);
                        zorros.add(zorro);
                        campo.ubicar(zorro, fila, col);
                    }
                    else if(rand.nextDouble() <=
PROBABILIDAD_DE_CREACION_DEL_CONEJO) {
                        Conejo conejo = new
Conejo(true);
                        conejo.setUbicacion(fila, col);
                        conejos.add(conejo);
                        campo.ubicar(conejo, fila,
col);
                    }
                    // de lo contrario, la ubicación
                    queda vacía.
                }
            }
        Collections.shuffle(conejos);
        Collections.shuffle(zorros);
    }
    // Se omiten los restantes métodos
}

```

La clase `Simulador` tiene tres partes importantes: su constructor, el método `poblar` y el método `simularUnPaso`. (El cuerpo de `simularUnPaso` se muestra más adelante.)

Cuando se crea un objeto `Simulador`, se construyen todas las otras partes de la simulación: el campo, las listas para contener los diferentes tipos de animales y la interfaz gráfica. Una vez que se han creado estas partes, se invoca al método `poblar` del simulador (indirectamente, mediante el método `inicializar`) para crear las poblaciones iniciales de zorros y de conejos. Se usan las diferentes probabilidades para decidir si una dirección en particular contendrá uno o más de estos animales. Observe que los animales creados al comienzo de la simulación tienen una edad inicial generada por azar que sirve para dos propósitos:

- Representa con más exactitud una población de edad promedio que será el estado normal de la simulación.
- Si todos los animales comenzaran con una edad cero, no se crearían nuevos animales hasta que las poblaciones iniciales hayan alcanzado sus respectivas edades de

reproducción. Con zorros que comen conejos sin tener en cuenta la edad de los zorros existe el riesgo de que la población de los conejos se extinga antes de que tengan la posibilidad de reproducirse o de que la población de zorros muera de hambre.

**Ejercicio 10.16** Modifique el método `poblar` del `Simulador` para determinar si resultaría catastrófico no configurar una edad inicial por azar para los zorros y los conejos.

**Ejercicio 10.17** Si la edad inicial se establece para los conejos pero no para los zorros, la población de conejos tenderá a crecer mientras que la población de zorros permanecerá muy pequeña. Una vez que los zorros tengan edad suficiente para reproducirse, la simulación ¿tendrá a comportarse nuevamente como en la versión original? ¿Qué sugiere esto sobre los tamaños relativos de las poblaciones iniciales y su impacto en los resultados de la simulación?

## 10.2.5

### La clase `Simulador`: un paso de simulación

La parte central de la clase `Simulador` es el método `simularUnPaso` que se muestra en el Código 10.4. Usa un ciclo separado para permitir que cada tipo de animal se mueva (y posiblemente se reproduzca o haga cualquiera de las cosas para las que está programado). La ejecución de simulaciones largas es trivial. Para lograrlo, se invoca repetidamente dentro de un sencillo ciclo el método `simularUnPaso`.

En vías de permitir que actúe cada animal, el simulador posee listas separadas de los diferentes tipos de animales. Aquí, no hacemos uso de la herencia y la situación nos trae reminiscencias de la primera versión del proyecto DoME en el que existían listas separadas de los distintos tipos de medios.

#### Código 10.4

Dentro de la clase `Simulador`: simular un paso

```
public void simularUnPaso()
{
    paso++;
    // Proporciona espacio para los conejos recién nacidos
    List<Conejo> conejosNuevos = new ArrayList<Conejo>();
    // Deja que todos los conejos actúen
    for(Iterator<Conejo> it = conejos.iterator();
    it.hasNext(); ) {
        Conejo conejo = it.next();
        conejo.correr(campoActualizado,
        conejosNuevos);
        if (!conejo.estaVivo ()) {
            it.remove();
        }
    }
    // Agrega los conejos recién nacidos a la lista de conejos
```

**Código 10.4  
(continuación)**

Dentro de la clase  
Simulador: simular  
un paso

```

        conejos.addAll(conejosNuevos);

        // Proporciona espacio para los zorros recién
        nacidos
        List<Zorro> zorrosNuevos = new
        ArrayList<Zorro>();
        // Deja que todos los zorros actúen
        for(Iterator<Zorro> it = zorros.iterator();
        it.hasNext(); ) {
            Zorro zorro = it.next();
            zorro.cazar(campo, campoActualizado,
            zorrosNuevos);
            if (!zorro.estaVivo ()) {
                it.remove();
            }
        }
        // Agrega los zorros recién nacidos a la lista de
        zorros
        zorros.addAll(zorrosNuevos);

    }

    // Intercambia el campo y el campoActualizado
    al final del paso.
    Campo temp = campo;
    campo = campoActualizado;
    campoActualizado = temp;
    campoActualizado.limpiar();
    // visualiza el nuevo campo en la pantalla
    visor.mostrarEstado(paso, campo);
}

```

Algo crucial para el progreso de la simulación a lo largo de un solo paso también es el uso de dos objetos Campo, referenciados mediante los atributos campo y campoActualizado del simulador. A medida que procesamos todos los animales del campo actual, cada uno se ubica en el campo campoActualizado después de su actuación. Esto facilita la eliminación de los animales muertos durante la simulación: simplemente no se los traslada al campo actualizado.

**Ejercicio 10.18** Cuando un conejo se mueve a una dirección que no está ocupada, se le ubica en el campo actualizado sólo si ya no existe un zorro en dicha dirección. ¿Cuál es el efecto sobre la población de zorros si se elimina esta restricción? ¿Y si se impone esta restricción sobre los conejos recién nacidos?

**Ejercicio 10.19** ¿Puede ocurrir que se intente mover dos zorros a la misma dirección del campo actualizado? Si es así, ¿se puede hacer algo para evitar esta situación?

### 10.2.6 Camino para mejorar la simulación

Ahora que ya hemos examinado cómo opera la simulación estamos en posición de realizar mejoras en su diseño interno y en su implementación. El foco de las secciones siguientes será realizar mejoras progresivas a través de la introducción de nuevas características de programación. Existen varios puntos por los que podríamos comenzar pero una de las debilidades más obvias es que no se han intentado explotar las ventajas de la herencia en la implementación de las clases *Zorro* y *Conejo* que comparten una gran cantidad de elementos comunes. En vías de hacer estas modificaciones presentamos el concepto de *clase abstracta*.

## 10.3

### Clases abstractas

El Capítulo 8 introdujo los conceptos de herencia y polimorfismo, que debemos ser capaces de explotar en la aplicación de la simulación. Por ejemplo, las clases *Zorro* y *Conejo* comparten varias características similares que sugieren que debieran ser subclases de una superclase común, tal como *Animal*. En esta sección comenzaremos a realizar cambios como éste con el objetivo de mejorar el diseño y la implementación de la simulación. Tal como en el ejemplo DoME del Capítulo 8, el uso de una superclase común evitaría la duplicación de código en las subclases y simplificaría el código de la clase cliente (en este caso: *Simulador*). Es importante resaltar que estamos llevando a cabo un proceso de refactorización y que estos cambios no debieran modificar las características esenciales de la simulación, vistas desde del ángulo del usuario.

**Ejercicio 10.20** Identifique las similitudes y las diferencias de las clases *Zorro* y *Conejo*. Escriba dos listas separadas de sus campos, métodos y constructores y distinga las variables de clase (campos estáticos) y las variables de instancia.

**Ejercicio 10.21** Los métodos candidatos a ser ubicados en la superclase son aquellos que son idénticos en todas las subclases. ¿Qué métodos son verdaderamente idénticos en las clases *Zorro* y *Conejo*? Para llegar a una conclusión, debería considerar el efecto de sustituir los valores de las variables de clase en los cuerpos de los métodos que las utilizan.

**Ejercicio 10.22** En la versión actual de la simulación, los valores de todas las variables de clase de nombres similares son diferentes. Si los dos valores de una variable de clase en particular (por ejemplo, *EDAD\_DE\_REPRODUCCION*) fueran iguales, ¿habría alguna diferencia en su evaluación de qué métodos son idénticos?

### 10.3.1 La superclase *Animal*

Para llevar a cabo el primer conjunto de cambios, moveremos los elementos idénticos de las clases *Zorro* y *Conejo* a la superclase *Animal*. El proyecto *zorros-y-conejos-v1* provee una copia de la versión base de la simulación para que pueda realizar los cambios que mencionaremos.

- Tanto Zorro como Conejo definen los atributos edad, vive y ubicacion, que pueden moverse a la superclase Animal, al igual que los métodos estaVivo y setUbicacion. Sus valores iniciales se establecen en el constructor de Animal.
- Al mover estos tres atributos a la clase Animal surge una pregunta respecto de la visibilidad que deberían tener. Por ejemplo, el método incrementarEdad necesita ser capaz tanto de obtener como de fijar el valor de la edad. Una posibilidad es declarar estos campos como protegidos ya que de esta manera las subclases tendrían acceso completo a ellos, pero, al mismo tiempo, se genera un alto acoplamiento entre estas clases. Podemos lograr un acoplamiento más bajo declarando estos campos como privados e implementando métodos de acceso y de modificación que las subclases podrán usar para inspeccionar y manipular los atributos.
- La clase Conejo define el método de modificación setComido que se usa en la clase Zorro dentro del método buscarComida. Sin embargo, ambas clases, Zorro y Conejo, necesitan asignar el valor false al atributo vive en otros lugares: incrementarEdad e incrementarHambre. En consecuencia, un cambio razonable y factible sería cambiar el nombre del método setComido por el nombre más general setMuerto y ubicarlo en la clase Animal, de manera tal que los métodos incrementar puedan usarlo.

**Ejercicio 10.23** ¿Qué tipo de estrategia de prueba de regresión podría establecerse antes de llevar a cabo el proceso de refactorización de la simulación? ¿Hay alguna prueba que se pueda automatizar convenientemente?

**Ejercicio 10.24** Cree la superclase Animal en su versión del proyecto. Realice los cambios discutidos anteriormente. Asegúrese de que la simulación continúa funcionando de manera similar a la anterior.

**Ejercicio 10.25** El uso de la herencia, ¿cómo podría mejorar aún más el proyecto? Argumente.

### 10.3.2 Métodos abstractos

Hasta ahora, la creación de la superclase Animal ayudó a evitar la duplicación de código en las clases Zorro y Conejo y facilitó el agregado de nuevos tipos de animales en el futuro. Como hemos visto en el Capítulo 8, el uso inteligente de la herencia debiera producir también la simplificación de la clase cliente. Investigaremos ahora este punto.

En la clase Simulador usamos dos listas separadas de tipos diferentes, una de conejos y otra de zorros, y escribimos código que recorre cada una de estas listas para implementar cada paso de la simulación. La parte de código relevante se muestra en el Código 10.4.

Ahora que tenemos la clase Animal podemos mejorar esta cuestión. Dado que todos los objetos en nuestras colecciones de animales son de un subtipo de Animal, podemos unirlas formando una sola colección y de aquí en adelante, la recorremos de una sola vez usando el tipo Animal. Sin embargo, evidenciamos un problema con esta solución de una única lista en el Código 10.5: pese a que sabemos que cada elemento de la lista es un Animal, debemos averiguar de qué tipo de animal se trata para poder invocar el método que realiza la acción correcta relacionada con dicho tipo de animal. Deter-

minamos el tipo mediante el uso del operador `instanceof`. El operador `instanceof` evalúa si un objeto determinado es una instancia de determinada clase, directa o indirectamente. La evaluación de la sentencia

```
obj instanceof MiClase
```

da resultado `true` si el tipo dinámico de `obj` es `MiClase` o cualquier subclase de `MiClase`.

#### Código 10.5

Una solución insatisfactoria con una lista única para lograr que los animales actúen

```
for(Iterator<Animal> iter = animales.iterator();
    iter.hasNext(); ) {
    Animal animal = iter.next();
    if(animal instanceof Conejo) {
        Conejo conejo = (Conejo)animal;
        conejo.correr(campoActualizado,
                      animalesNuevos);
    }
    else if(animal instanceof Zorro) {
        Zorro zorro = (Zorro)animal;
        zorro.cazar(campo, campoActualizado,
                     animalesNuevos);
    }
    else {
        System.out.println("se encontró un
                           animal desconocido");
    }
    // Elimina de la simulación los animales muertos.
    if(! animal.estaVivo()){
        iter.remove();
    }
}
```

El hecho de que en el Código 10.5 debe evaluarse y enmascararse cada tipo de animal separadamente y de que existe código especial para cada clase de animal es una buena señal de que no logramos, todavía, obtener las ventajas que ofrece la herencia. Si en cambio, aseguráramos que la superclase (`Animal`) tiene un método que permite que un animal actúe y este método se redefiniera en cada subclase, podríamos usar un método polimórfico para lograr que el animal actúe sin necesidad de evaluar los tipos específicos de los animales. Supongamos que hemos creado un método de estas características de nombre `actuar` e investiguemos el código resultante. El Código 10.6 muestra la implementación de esta solución.

#### Código 10.6

La solución mejorada para la acción del animal

```
// Permite que todos los animales actúen
for(Iterator<Animal> it = animales.iterator(); it.hasNext();
    ) {
    Animal animal = it.next();
```

### Código 10.6 (continuación)

La solución mejorada para la acción del animal

```
        animal.actuar(campo, campoActualizado,
animalesNuevos);
    // Elimina de la simulación los animales muertos.
    if(! animal.estaVivo()){
        it.remove();
    }
}
```

En este punto, son importantes varias observaciones:

- La variable que estamos usando para cada elemento de la colección (`animal`) es de tipo `Animal`. Esto es legal ya que todos los objetos de la colección son conejos o zorros, por lo que todos son subtipos de `Animal`.
- Asumimos que los métodos específicos de acción (`correr` para Conejo, `cazar` para Zorro) han sido renombrados como `actuar`. Esto es más adecuado: en lugar de decir exactamente lo que hace cada animal, estamos diciendo «`actuar`» y dejamos que el animal propiamente dicho decida exactamente lo que quiere hacer. Esto reduce el acoplamiento entre `Simulador` y las subclases individuales de los animales.
- El método `correr` de `Conejo` sólo tiene dos parámetros: `campoActualizado` y `animalesNuevos`. Hemos agregado un tercer parámetro, `campo`, para hacerlo consistente con el método `actuar` del zorro. Ahora, cada animal obtiene todos los parámetros que posiblemente necesite para implementar una acción flexible y cada clase puede elegir ignorar cualquiera de los parámetros.
- Dado que el tipo dinámico de la variable determina qué método es realmente ejecutado (como lo discutimos en el Capítulo 9), el método de acción del zorro se ejecutará para los zorros y el método de acción de los conejos, para los conejos.
- Dado que el control de tipos se realiza usando el tipo estático, este código compilará sólo si la clase `Animal` tiene un método `actuar` con la firma correcta.

El último de estos puntos es el único problema que falta solucionar. Dado que usamos la sentencia

```
    animal.actuar(campo, campoActualizado, animalesNuevos);
```

y la variable `animal` es de tipo `Animal`, hemos visto en el Capítulo 9 que este código compilará sólo si `Animal` define tal método. Sin embargo, la situación que tenemos aquí es algo diferente de la situación que encontramos con el método `imprimir` de la clase `Elemento` en el Capítulo 9. Allí, la versión de `imprimir` de la superclase tenía un trabajo útil para hacer: imprimir los campos definidos en la superclase. Aquí, pese a que cada animal en particular tiene un conjunto específico de acciones que realizar, no podemos describir detalladamente las acciones de los animales en general. Las acciones particulares dependen del subtipo específico.

Nuestro problema reside en decidir cómo debemos definir el método `actuar` de `Animal`.

El problema proviene del hecho de que nunca existirá una instancia de la clase `Animal`. No existe ningún objeto en nuestra simulación (o en la naturaleza) que sea un animal y que no sea al mismo tiempo una instancia de una subclase más específica. Este tipo de clases, que no se definen con la intención de crear objetos sino que sólo sirven

como superclases, se conocen como *clases abstractas*. Por ejemplo, en el caso de los animales, podemos decir que cada animal puede actuar pero no podemos describir exactamente cómo actúa sin hacer referencia a una subclase más específica. Esto es típico de las clases abstractas y se refleja en las construcciones de Java.

### Concepto

La definición de un **método abstracto** consiste en la signatura de un método sin su correspondiente cuerpo. Se indica mediante la palabra clave `abstract`.

En la clase `Animal` deseamos establecer que cada animal tiene un método `actuar` pero no podemos darle una implementación razonable dentro de la clase `Animal`. La solución en Java consiste en declarar el método como *abstracto*. Aquí hay un ejemplo del método abstracto `actuar`:

```
abstract public void actuar (Campo campoActual,
    Campo campoActualizado,
    List<Animal> animalesNuevos);
```

Un método abstracto se caracteriza por dos detalles:

- Está precedido por la palabra clave `abstract`.
- No tiene cuerpo y su encabezado termina con un punto y coma.

Dado que el método no tiene cuerpo, jamás podrá ser ejecutado, pero dejamos establecido que no queremos ejecutar el método `actuar` de `Animal`; por lo tanto, esto no es un problema.

Antes de que investiguemos en detalle los efectos del uso de métodos abstractos, presentaremos más formalmente el concepto de clase abstracta.

### 10.3.3 Clases abstractas

No sólo los métodos pueden declararse como abstractos, también las clases pueden declararse abstractas. El Código 10.7 muestra un ejemplo de la clase `Animal` como una clase abstracta. Las clases se declaran abstractas mediante la inserción de la palabra clave `abstract` en el encabezado de la clase.

### Código 10.7

`Animal` como una clase abstracta

```
public abstract class Animal
{
    // Se omiten los campos
    /*
     * Hace que este animal actúe, es decir: hace que haga
     * lo que quiera
     * o necesita hacer.
     * @param campoActual El campo que ocupa actualmente
     * @param campoActualizado El campo al que se trasladará
     * @param animalesNuevos Una lista para agregar los
     * animales recién
     * nacidos.
    */
    abstract public void actuar (Campo campoActual,
        Campo campoActualizado,
        List<Animal> animalesNuevos);
    // Se omiten los restantes métodos
}
```

### Concepto

Una **clase abstracta** es una clase de la que no se pretende crear instancias. Su propósito es servir como superclase a otras clases. Las clases abstractas pueden contener métodos abstractos.

Las clases que no son abstractas (todas las clases que hemos visto previamente) se denominan *clases concretas*.

La declaración de una clase abstracta sirve a varios propósitos:

- No se creará ninguna instancia de clases abstractas. El intento de uso de la palabra clave new con una clase abstracta es un error que se refleja en BlueJ: al hacer clic derecho sobre una clase abstracta en el diagrama de clases, no aparece ningún constructor en el menú contextual. Todo esto está en relación directa con nuestra intención discutida anteriormente: establecemos que no deseamos instancias creadas directamente a partir de la clase Animal: esta clase sólo sirve como superclase. La declaración de la clase como abstracta refuerza esta restricción.
- Sólo las clases abstractas pueden tener métodos abstractos. Esto asegura que siempre podrán ser ejecutados todos los métodos de las clases concretas. Si permitiéramos un método abstracto en una clase concreta, podríamos crear una instancia de una clase a la que le falta la implementación de un método.
- Las clases abstractas con métodos abstractos fuerzan a las subclases a sobrescribir una implementación de aquellos métodos declarados abstractos. Si una subclase no provee una implementación para un método abstracto heredado, es en sí misma abstracta, y no puede crearse ninguna instancia. Para que una subclase sea concreta, debe proveer implementaciones para *todos* los métodos abstractos heredados.

### Concepto

**Subclases abstractas.** Para que una subclase de una clase abstracta se convierta en una subclase concreta, debe proveer las implementaciones de todos los métodos abstractos heredados. De lo contrario, será propiamente abstracta.

Ahora podemos comenzar a ver el propósito de los métodos abstractos. Aunque no proveen una implementación, aseguran que todas las subclases tienen una implementación de este método. En otras palabras, aunque la clase Animal no implemente el método actuar, asegura que todos los animales existentes tienen implementado un método actuar. Esto se hace para asegurar que:

- no se pueda crear directamente ninguna instancia de la clase Animal y
- todas las subclases concretas deban implementar el método actuar.

Pese a que no podemos crear directamente una instancia de una clase abstracta, podemos usar una clase abstracta como un tipo de la manera habitual. Por ejemplo, las reglas normales del polimorfismo nos permiten manejar a los zorros y a los conejos como instancias de la clase Animal. Por lo tanto, aquellas partes de la simulación que no necesiten conocer si están tratando con una subclase específica pueden usar el tipo de la superclase.

**Ejercicio 10.26** Pese a que el cuerpo del ciclo en el Código 10.6 no opera más con los tipos Zorro y Conejo, todavía opera con el tipo Animal. ¿Por qué no es posible operar con cada objeto de la colección usando simplemente el tipo Object?

**Ejercicio 10.27** ¿Es necesario que una clase que tiene uno o más métodos abstractos se defina como abstracta? Si no está seguro, experimente con el código fuente de la clase Animal del proyecto zorros-y-conejos-v2.

**Ejercicio 10.28** ¿Es posible que una clase que no tiene métodos abstractos se defina como abstracta? Si no está seguro, cambie el método actuar en la clase Animal para que sea un método concreto, construyendo un cuerpo de método sin ninguna sentencia.

**Ejercicio 10.29** ¿Puede tener sentido la definición de una clase como abstracta si no tiene métodos abstractos? Discuta sobre este tema.

**Ejercicio 10.30** ¿Qué clases del paquete `java.util` son abstractas? Algunas de ellas tienen la palabra `abstract` en el nombre de la clase, pero ¿existe alguna otra forma de comunicarlo mediante la documentación? ¿Qué clases concretas las extienden?

**Ejercicio 10.31** ¿Puede deducir, a partir de la documentación API de una clase abstracta, cuáles de sus métodos son abstractos (si es que existe alguno)? ¿Necesita saber qué métodos son abstractos?

**Ejercicio 10.32** ¿Cuáles de las restantes clases de la simulación no necesitan saber si están operando específicamente con zorros o con conejos? ¿Podrían rescribirse de modo que usen la clase `Animal` en lugar de los tipos específicos? ¿Se obtendría algún beneficio al hacer esto?

**Ejercicio 10.33** Revise las reglas de sobrescritura de métodos y de campos discutidas en el Capítulo 9. ¿Por qué son particularmente importantes en nuestro intento de introducir herencia en esta aplicación?

**Ejercicio 10.34** Los cambios realizados en esta sección han eliminado las dependencias (acoplamientos) del método `simularUnPaso` respecto de las clases `Zorro` y `Conejo`. Sin embargo, la clase `Simulador` todavía está acoplada con `Zorro` y `Conejo` porque se hace referencia a estas clases en el método `poblar`. No hay manera de evitar esto; cuando creamos instancias de animales tenemos que especificar exactamente qué clase de animal vamos a crear.

Esta situación podría mejorarse dividiendo la clase `Simulador` en dos clases: una clase `Simulador` que ejecute la simulación y que esté completamente desacoplada de las clases concretas de animales y otra clase, `GenerarPoblacion` (creada e invocada por el simulador) que cree la población. De esta manera, sólo esta clase quedaría acoplada a las clases de animales concretos facilitando al programador la tarea de mantenimiento cuando deba encontrar los lugares en los que es necesario realizar un cambio si se extiende la aplicación. Trate de implementar este paso de refactorización. La clase `GenerarPoblacion` debiera definir también los colores para cada tipo de animal.

**Ejercicio 10.35** *Desafío.* Los métodos `puedeReproducir` de las clases `Zorro` y `Conejo` son textualmente idénticos, pero todavía no hemos elegido moverlos a la clase `Animal`. ¿Por qué? Intente mover estos métodos desde las clases `Zorro` y `Conejo` a la clase `Animal` y convertirlos en protegidos. ¿Existe alguna forma de lograr que las clases resultantes compilen? De ser así, ¿la simulación resultante funciona como debiera? ¿Cómo puede explicarlo?

El proyecto `zorros-y-conejos-v2` proporciona una implementación de nuestra simulación con las mejoras que hemos discutido en esta sección.

## 10.4

## Más métodos abstractos

Cuando creamos la superclase `Animal` en la Sección 10.3, lo hicimos identificando los elementos comunes de las subclases. Este camino podría ser sumamente conservador. Por ejemplo, ¿por qué no se mueve el método `puedeReproducir` a la clase `Animal`? La razón para no mover este u otros métodos es que, pese a que varios de los métodos restantes contienen cuerpos con sentencias textualmente idénticas, usan las variables de clase, y esto hace que no puedan moverse directamente a la superclase. En el caso de `puedeReproducir`, el problema está en la variable `EDAD_DE_REPRODUCCION`: si el método se mueve a la clase `Animal`, el compilador necesitará tener acceso al valor de la edad de reproducción en la clase `Animal`. Es tentador definir la variable `EDAD_DE_REPRODUCCION` en la clase `Animal` y asumir que su valor puede ser sobreescrito en las subclases por variables de nombres similares. Sin embargo, en Java, los campos se manejan de manera diferente que los métodos: los campos no pueden ser sobreescritos por las versiones de las subclases<sup>2</sup>.

Sin embargo, podemos usar esta idea si accedemos a la edad de reproducción mediante un método en lugar de acceder directamente a un campo. Este abordaje se muestra en el Código 10.8. El uso de este método de acceso nos permite mover los restantes métodos a la superclase.

### Código 10.8

El método `puedeReproducir` de la clase `Animal`

```
/*
 * Un animal puede reproducirse si alcanzó la edad de
 * reproducción.
 * @return true si el animal puede reproducirse
 */
public boolean puedeReproducir()
{
    return edad >= getEdadDeReproduccion();
}
```

El método `puedeReproducir` ha sido sobreescrito para que use el valor que retorna una llamada a un método en lugar del valor de una variable de clase. Para que esto funcione, se debe definir el método `getEdadDeReproduccion` en la clase `Animal`. Dado que no podemos especificar una edad de reproducción para los animales en general, podemos usar nuevamente un método abstracto en la clase `Animal` y concretar las redefiniciones en las subclases. Tanto `Zorro` como `Conejo` definirán sus propias versiones del método `getEdadDeReproduccion` para devolver sus particulares valores de la variable `EDAD_DE_REPRODUCCION`:

```
/**
 * @return La edad en que un conejo comienza a reproducirse
 */
public int getEdadDeReproduccion()
```

---

<sup>2</sup> Esta regla se aplica independientemente de si un campo es estático o no.

```
    return EDAD_DE_REPRODUCCION;
}
```

Este cambio posibilita que cada instancia use el valor adecuado a su tipo de subclase.

**Ejercicio 10.36** Usando su última versión del proyecto (o el proyecto *zorros-y-conejos-v2* en el caso de que no haya realizado todos los ejercicios), mueva el método `puedeReproducir` desde las clases Zorro y Conejo a la clase Animal y rescríbalo como muestra el Código 10.8. Provea versiones adecuadas del método `getEdadDeReproducción` en las clases Zorro y Conejo. ¿Son suficientes estos cambios para recompilar el proyecto? Si no es así, ¿qué le falta a la clase Animal?

**Ejercicio 10.37** Mueva el método `incremetarEdad` desde las clases Zorro y Conejo a la clase Animal proveyendo a la clase Animal de un método abstracto `getEdadMaxima` y de una versión concreta en Zorro y en Conejo.

**Ejercicio 10.38** ¿Puede moverse el método `reproducir` a la clase Animal? De ser así, realice este cambio.

**Ejercicio 10.39** A la luz de todos los cambios que ha realizado en estas tres clases, reconsideré la visibilidad de cada método y haga cualquier cambio que considere adecuado.

**Ejercicio 10.40** ¿Fue posible realizar estos cambios sin que tengan ningún impacto sobre las restantes clases del proyecto? Si la respuesta es afirmativa, ¿qué sugiere esta afirmación con respecto al grado de encapsulamiento y de acoplamiento que presentaba la versión original?

**Ejercicio 10.41 Desafío.** Defina en la simulación un tipo de animal completamente nuevo como una subclase de `Animal`. Necesitará decidir el impacto que tendrá la existencia del nuevo tipo de animal sobre los tipos de animales existentes. Por ejemplo, su animal podría competir con los zorros como un predador de la población de conejos, o su animal podría ser presa de los zorros pero no de los conejos. Probablemente encontrará que necesita experimentar bastante con la configuración que utilice. Necesitará modificar el método `poblar` de modo que pueda tener creados algunos de sus animales al comienzo de la simulación.

También deberá definir un nuevo color para su nueva clase de animal. Puede encontrar una lista de los nombres de los colores predefinidos en la documentación API de la clase `Color` en el paquete `java.awt`.

## 10.5

## Herencia múltiple

### 10.5.1

### La clase Actor

En esta sección discutimos algunas posibles futuras extensiones y algunas construcciones de programación para implementar estas extensiones.

La primera extensión obvia de nuestra simulación es que permita agregar nuevos animales. Si intentó realizar el Ejercicio 10.41, ya trabajó sobre este punto. Sin embargo, debemos

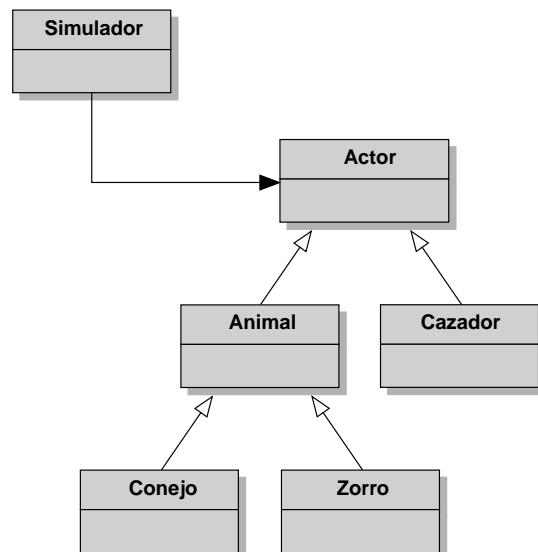
generalizar un poco más este asunto: podría ocurrir que no todos los participantes de la simulación sean animales. Nuestra estructura actual asume que todos los participantes que actúan en la simulación son animales y derivan de la superclase `Animal`. Una mejora que podríamos hacer es la introducción de predadores humanos en la simulación, como cazadores o colocadores de trampas. Estos actores no encararían con la asunción de que los actores están puramente basados en animales. Podríamos también extender la simulación para incluir plantas o el clima. El crecimiento de las plantas puede influir sobre la población de conejos y las plantas podrían estar influenciadas por el clima. Todos estos nuevos componentes deberían actuar en la simulación, pero claramente no son animales.

Si consideramos la potencialidad para introducir más actores en la simulación, aparece claramente la razón de nuestra elección de almacenar los detalles de los animales tanto en un objeto `Campo` como en las listas de objetos `Animal`. Esta elección claramente duplica información, lo que acarrea riesgos de creación de inconsistencias. Un motivo para esta decisión de diseño es que nos permite tener en cuenta a participantes de la simulación que no estén realmente dentro del campo; un ejemplo podría ser la representación del clima.

Para trabajar con estos actores más generales, parece ser una buena idea la introducción de una superclase `Actor`. La clase `Actor` podría servir como superclase para todos estos tipos de participantes de la simulación, independientemente de lo que son. La Figura 10.3 muestra un diagrama de clases de esta parte de la simulación. Las clases `Actor` y `Animal` son abstractas, mientras que `Conejo`, `Zorro` y `Cazador` son clases concretas.

**Figura 10.3**

Estructura de la simulación con la clase `Actor`



La clase `Actor` podría incluir las partes comunes a todos los actores. Una cosa común que es posible que tengan todos los actores es que llevan a cabo alguna clase de acción. Por lo que la única definición en la clase `Actor` es la de un método `actuar` abstracto.

```

// Se omiten todos los comentarios
public abstract class Actor
{
    abstract public void actuar(Campo campoActual,
                                Campo campoActualizado,
                                List<Actor> actoresNuevos);
}
  
```

Alcanzaría con describir el ciclo del actor en el Simulador (Código 10.6) usando la clase Actor en lugar de la clase Animal.

**Ejercicio 10.42** Introduzca la clase Actor en su simulación. Rescriba el método simularUnPaso en la clase Simulador para que use Actor en lugar de Animal. ¿Puede hacer este cambio aun cuando no haya introducido ningún tipo de participante nuevo? ¿Compila la clase Simulador? ¿O se necesita algo más en la clase Actor?

Esta nueva estructura es más flexible porque permite agregar fácilmente actores que no son animales. De hecho, podríamos describir la clase que lleva a cabo las estadísticas, EstadisticasDelCampo, como un Actor: también actúa una vez en cada paso. Su acción podría consistir en actualizar la cantidad actual de animales.

## 10.5.2 Flexibilidad a través de la abstracción

Acercándonos a la noción de la simulación como la responsable del manejo de los objetos actores, hemos logrado abstraer mucho más que en nuestro escenario original sumamente específico de zorros y conejos ubicados en un campo rectangular. Este proceso de abstracción ha brindado una flexibilidad creciente que nos permite ampliar el alcance de lo que podríamos hacer con un marco general de simulación más avanzado. Si pensamos en los requerimientos de otros escenarios de simulación similares, podríamos obtener ideas sobre las características adicionales que podríamos introducir.

Por ejemplo, podría ser útil simular otro escenario predador-presa como por ejemplo, una simulación marina que involucra peces y tiburones o peces y barcos pesqueros. Si la simulación marina involucrara modelar el aporte de alimento para los peces probablemente no queríamos visualizar poblaciones de plankton, ya sea porque el número de estas poblaciones es demasiado grande o porque su tamaño es demasiado pequeño. Otros ambientes de simulación podrían involucrar modelar el clima que, ya que es claramente un actor, también podría no requerir su visualización.

En la siguiente sección investigaremos, a modo de una extensión más avanzada de nuestro marco de simulación, la separación de la visualización a partir de la actuación.

## 10.5.3 Dibujo selectivo

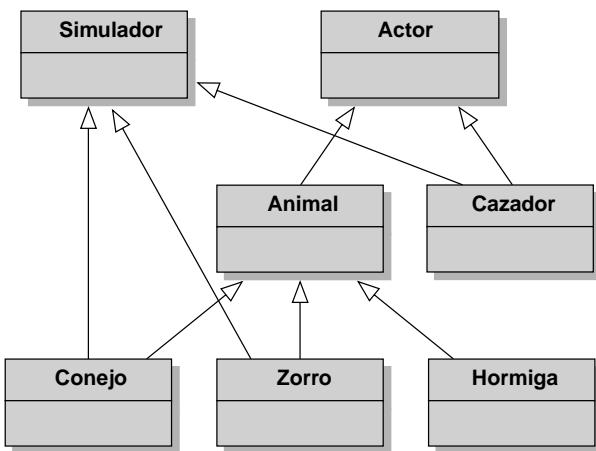
Una manera de implementar la separación de la visualización a partir de la actuación es modificar la forma en que ésta se lleva a cabo en la simulación. En cada momento, en lugar de recorrer el campo completo y dibujar los actores en cada posición, podríamos recorrer una colección separada de actores «dibujables». El código en la clase del simulador podría ser como sigue:

```
// permitir que todos los actores actúen
for(Actor actor : actores) {
    actor.actuar(...);
}
// dibujar todos los dibujables
for(Dibujable elemento : dibujables) {
    elemento.dibujar(...);
}
```

Todos los actores estarían en la colección actores, y aquellos actores que queremos mostrar en la pantalla también estarían en la colección dibujables. Para este trabajo necesitamos otra superclase de nombre Dibujable que declara un método abstracto dibujar. Los actores dibujables deben derivar tanto de la clase Actor como de Dibujable (la Figura 10.4 muestra un ejemplo en el que asumimos que tenemos hormigas que actúan pero que no están visibles en la pantalla).

**Figura 10.4**  
Jerarquía de herencia  
Actor con la clase  
Dibujable

**Concepto**  
**Herencia múltiple.**  
Una situación en la que una clase deriva de más de una superclase se denomina herencia múltiple.



### 10.5.4 Actores dibujables: herencia múltiple

El escenario aquí presentado usa una estructura que se conoce como *herencia múltiple*. La herencia múltiple existe en los casos en los que una clase deriva de más de una superclase. En consecuencia, la subclase tiene todas las características de ambas superclases y aquellas definidas en la subclase propiamente dicha.

La herencia múltiple es, en principio, muy fácil de comprender pero su implementación de un lenguaje de programación puede llegar a ser significativamente complicada. Los diferentes lenguajes orientados a objetos varían en cuanto a su tratamiento de la herencia múltiple: algunos lenguajes permiten la herencia múltiple de superclases y otros no. Java se encuentra en un lugar intermedio: no permite la herencia múltiple de clases pero proporciona otra construcción denominada «interfaces» que permite una forma limitada de herencia múltiple. Las interfaces se discuten en la próxima sección.

## 10.6 Interfaces

Hasta este momento, hemos usado en el libro el término «interfaz» en un sentido informal para representar la parte de una clase que se acopla con otras clases. Java captura este concepto más formalmente permitiendo definir los *tipos interfaces*.

En una primer mirada, las interfaces son similares a las clases, la diferencia más obvia radica en que sus definiciones de métodos no incluyen cuerpos. Por lo tanto, se parecen a las clases abstractas en las que todos sus métodos son abstractos.

### 10.6.1 La interfaz Actor

El Código 10.9 muestra Actor definida como un tipo de interfaz.

**Código 10.9**

La interfaz Actor

```

/**
 * La interfaz que será implementada por cualquier clase
 * que
 * que desee participar de la simulación.
 */
public interface Actor
{
    /**
     * Determina el comportamiento diario del actor. Traslada
     * al actor
     * al campoActualizado si es que participa en otros pasos
     * de la
     * simulación.
     * @param campoActual El estado actual del campo
     * @param ubicacion La ubicación del actor en el campo
     * actual
     * @param campoActualizado El estado actualizado del campo
     */
    void actuar(Campo campoActual, Ubicacion ubicacion,
                Campo campoActualizado);
}

```

**Concepto**

Una **interfaz** en Java es la especificación de un tipo (bajo la forma de un nombre de tipo y un conjunto de métodos) que no define ninguna implementación para los métodos.

Las interfaces en Java tienen una cantidad de características importantes:

- En el encabezado de la declaración se usa la palabra clave **interface** en lugar de **class**.
- Todos los métodos de una interfaz son abstractos; no se permiten métodos con cuerpo. No es necesaria la palabra clave **abstract**.
- Las interfaces no contienen ningún constructor.
- Todas las signatures de los métodos de una interfaz tienen visibilidad pública. No es necesario declarar la visibilidad: por ejemplo, no es necesario que cada método contenga la palabra clave **public**.
- En una interfaz, sólo se permiten los campos constantes (campo público, estático y final). Pueden omitirse las palabras clave **public**, **static** y **final** pero todos los campos, igualmente, serán tratados como públicos, estáticos y finales.

Una clase puede derivar de una interfaz de la misma manera en que deriva de una clase. Sin embargo, Java utiliza una palabra clave diferente, **implements**, para la herencia a partir de interfaces.

Se dice que una clase *implementa* una interfaz si incluye una *cláusula implements* en su encabezado. Por ejemplo:

```

public class Zorro extends Animal implements Dibujable
{
    ...
}

```

Como en este caso, si una clase extiende a una clase e implementa una interfaz, entonces la cláusula **extends** debe escribirse primero en el encabezado de la clase.

Dos de nuestras clases abstractas del ejemplo anterior, Actor y Dibujable, son buenas candidatas a ser reescritas como interfaces. Ambas contienen sólo la definición de un único método sin ninguna implementación. Por lo tanto, encajan perfectamente con la definición de una interfaz: no contienen campos, ni constructores, ni cuerpos de métodos.

La clase Animal es un caso diferente. Es una clase realmente abstracta que provee una implementación parcial (varios de los métodos tienen cuerpo) y sólo un único método abstracto.

**Ejercicio 10.43** Redefina la clase abstracta Actor en su proyecto como una interfaz. La simulación ¿aún compila? ¿Corre?

**Ejercicio 10.44** En la siguiente interfaz, ¿los campos son estáticos o de instancia?

```
public interface Examen
{
    int CORRECTO = 1;
    int INCORRECTO = 0;
    ...
}
```

¿Qué visibilidad tienen?

**Ejercicio 10.45** ¿Cuáles son los errores en la siguiente interfaz?

```
public interface Monitor
{
    private static final int UMBRAL = 50;

    public Monitor (int inicial);

    public int getUmbral()
    {
        return UMBRAL;
    }
    ...
}
```

## 10.6.2 Herencia múltiple de interfaces

Como mencionamos anteriormente, Java permite que cada clase extienda como máximo a otra clase, sin embargo permite que una clase implemente cualquier número de interfaces (además de la posibilidad de extender una clase). Por lo tanto, si definimos Actor y Dibujable como interfaces en lugar de como clases abstractas, podemos definir una clase Cazador (Figura 10.4) para implementar a ambas:

```
public class Cazador implements Actor, Dibujable
{
    ...
}
```

La clase `Cazador` hereda las definiciones de los métodos de todas las interfaces (en este caso, `actuar` y `dibujar`) como métodos abstractos. En consecuencia, se deben proveer definiciones para ambos métodos sobrescribiendo los métodos, de lo contrario la clase se declara abstracta.

La clase `Animal` muestra un ejemplo en el que una clase no implementa un método heredado de una interfaz. `Animal`, en nuestra nueva estructura de la Figura 10.4, hereda el método abstracto `actuar` de la clase `Actor`. No provee un cuerpo para este método, por lo que `Animal` es propiamente abstracta (y debe incluir la palabra clave `abstract` en su encabezado).

Por lo tanto, las subclases de `Animal` implementan el método `actuar` y así se convierten en clases concretas.

**Ejercicio 10.46 Desafío.** Agregue a la simulación un actor que no sea un animal. Por ejemplo, podría introducir una clase `Cazador` con las siguientes propiedades: los cazadores no tienen edad máxima y no se alimentan ni se reproducen. En cada paso de la simulación, un cazador se mueve a una ubicación aleatoria en cualquier lugar del campo y dispara un número fijo de tiros hacia objetivos ubicados en direcciones aleatorias del campo. Cualquier animal que se encuentre en una de las ubicaciones de estos objetivos pasará a estar muerto.

Ubique en el campo un pequeño número de cazadores, al comienzo de la simulación. Durante la simulación, los cazadores, ¿continúan estando en el campo o desaparecen en algún momento? Si desaparecen, ¿por qué podría ser? Esta situación, ¿representa un comportamiento real?

¿Qué otras clases requieren modificaciones como consecuencia de introducir cazadores? ¿Existe alguna necesidad de introducir un mayor desacoplamiento entre las clases?

### 10.6.3 Interfaces como tipos

Cuando una clase implementa una interfaz no hereda ninguna implementación de ella, pues las interfaces no pueden contener cuerpos de métodos. Entonces, la pregunta que cabe es: ¿qué ganamos realmente al implementar interfaces?

Cuando presentamos la herencia en el Capítulo 8 pusimos énfasis en dos grandes beneficios de la herencia:

- La subclase hereda el código (la implementación de métodos y campos) de la superclase. Esto permite la reutilización de código existente y evita la duplicación de código.
- La subclase se convierte en un subtipo de la superclase. Esto permite la existencia de variables polimórficas y la invocación polimórfica de métodos. En otras palabras, permite que los casos especiales de objetos (instancias de subclases) se traten de manera uniforme (como instancias del supertipo).

Las interfaces no brindan el primer beneficio (ya que no contienen ninguna implementación), pero sí ofrecen el segundo. Una interfaz define un tipo tal como lo hace una clase. Esto quiere decir que las variables pueden ser declaradas del tipo de la interfaz, aun cuando no pueda existir ningún objeto de tal tipo (sólo de los subtipos).

En nuestro ejemplo, aunque Actor ahora es una interfaz, todavía podemos declarar una variable de tipo Actor en la clase Simulador. El ciclo de la simulación aún continúa funcionando sin ningún cambio.

Las interfaces no pueden tener instancias directas pero sirven como supertipos para las instancias de otras clases.

#### 10.6.4 Interfaces como especificaciones

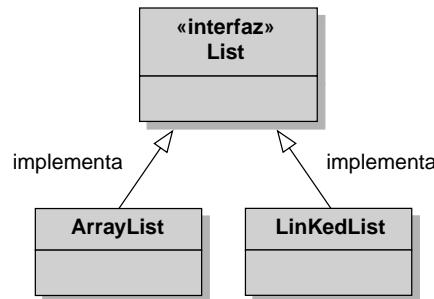
En este capítulo hemos introducido las interfaces con el sentido de implementar herencia múltiple en Java. Este es un uso importante de las interfaces, pero existen otros.

La característica más importante de las interfaces es que separan completamente la definición de la funcionalidad (la clase «interfaz» en el sentido más amplio de la palabra) de su implementación. Un buen ejemplo de cómo pueden usarse las interfaces en la práctica se puede encontrar en la jerarquía de las colecciones de Java.

La jerarquía de colecciones define, entre otros tipos, la interfaz `List` y las clases `ArrayList` y `LinkedList` (Figura 10.5). La interfaz `List` especifica la funcionalidad total de una lista sin aportar ninguna implementación. Las subclases `LinkedList` y `ArrayList` proveen dos implementaciones diferentes para la misma interfaz. Esto es interesante porque las dos implementaciones difieren enormemente en la eficiencia de algunas de sus funciones. Por ejemplo, el acceso aleatorio de elementos situados en el medio de una lista es mucho más rápido en el `ArrayList`, sin embargo la inserción o la eliminación de elementos puede ser mucho más rápida en la `LinkedList`.

**Figura 10.5**

La interfaz `List` y sus subclases



La decisión de cuál de las implementaciones resulta mejor para una aplicación determinada puede ser difícil de juzgar anticipadamente, depende mucho de la frecuencia relativa con que se lleven a cabo ciertas operaciones y algunos otros factores. En la práctica, la mejor forma de descubrir cuál es la mejor es probando: implementar la aplicación con ambas alternativas y medir el rendimiento.

La existencia de la interfaz `List` facilita esta prueba. Si en lugar de usar un `ArrayList` o una `LinkedList` como tipo de variable y tipo de parámetro usamos siempre `List`, nuestra aplicación funcionará independientemente del tipo específico de lista que estemos usando realmente. Debemos usar el nombre específico de la implementación seleccionada sólo cuando creamos una nueva lista. Por ejemplo, podemos escribir

```
private List<Tipo> miLista = new ArrayList<Tipo>();
```

Observe que el tipo del campo es justamente `List` de `Tipo`. De esta manera, podemos modificar toda la aplicación para que use una lista enlazada con sólo cambiar `ArrayList` por `LinkedList` en un único lugar: el lugar en el que se crea la lista.

**Ejercicio 10.47** ¿Qué métodos tienen `ArrayList` y `LinkedList` que no están definidos en la interfaz `List`? ¿Por qué cree que estos métodos no se incluyen en `List`?

**Ejercicio 10.48** Lea en el API la descripción de los métodos `sort` de la clase `Collections` en el paquete `java.util`. ¿Qué interfaces se mencionan en las descripciones?

**Ejercicio 10.49** Desafío. Investigue la interfaz `Comparable` que es una interfaz parametrizada. Defina una clase que implemente `Comparable`. Cree una colección que contenga objetos de esta clase y ordene la colección. *Pista:* la clase `EntradaLog` del proyecto *analizador-weblog* del Capítulo 4 implementa esta interfaz.

### 10.6.5 Otro ejemplo de interfaces

En la sección anterior hemos discutido cómo pueden usarse las interfaces para separar la especificación de un componente de su implementación, por lo que pueden «conectarse» diferentes implementaciones facilitando el reemplazo de los componentes de un sistema.

Esto se usa frecuentemente para separar partes de un sistema que están bajamente acopladas desde el punto de vista lógico. Un ejemplo en nuestra simulación es el visor. La simulación lógica (el campo y los actores) está bastante separada de la parte visual de la simulación. Podríamos imaginar maneras completamente diferentes de presentar la misma aplicación:

- El visor podría representar gráficamente la población de cada especie en el tiempo. El eje x del gráfico podría representar el tiempo (en pasos de simulación) mientras que el eje y mostraría el número de animales. Cada especie podría mostrarse con su propia curva en un color diferente.
- La salida de la simulación podría ser puramente textual: podríamos imprimir secuencias de texto en la terminal, una para cada paso de la simulación. Esto tendría la ventaja de que es muy fácil de implementar y que la salida puede ser, por ejemplo, grabada en un archivo. En oposición a la versión actual de la simulación, esta manera brindaría un registro del proceso en su totalidad.

Podemos llevar a cabo esta separación convirtiendo a `VisorDelSimulador` en una interfaz. Para definir esta interfaz podemos buscar en la clase `Simulador` para encontrar todos los métodos que se invocan desde su exterior. Estos son (en este orden):

```
visor.setColor(class, color);
visor.esViabile(campo);
visor.mostrarEstado(paso, campo);
```

Podemos ahora definir fácilmente la interfaz `VisorDelSimulador` completa:

```
import java.awt.Color;
public interface VisorDelSimulador
```

```

{
    void setColor(Class cl, Color color);
    boolean esViable(Campo campo);
    void mostrarEstado(int paso, Campo campo);
}

```

Nuestra clase actual `VisorDelSimulador` podría renombrarse como `VisorAnimado` (ya que provee una visión animada del simulador) y debería implementar la interfaz `VisorDelSimulador`:

```

public class VisorAnimado extends JFrame implements
VisorDeSimulador
{
    .
    .
}

```

Después de hacer estos cambios, se vuelve bastante fácil implementar y «conectar» otras vistas de la simulación.

**Ejercicio 10.50** Realice los cambios descritos anteriormente: renombre la clase `VisorDelSimulador` como `VisorAnimado` e implemente la interfaz `VisorDeSimulador`. Asegúrese de que en la clase `Simulador` el nombre `VisorAnimado` se use sólo una vez (cuando se crea el objeto visor); en todos los restantes lugares se usa la interfaz de nombre `VisorDelSimulador`.

**Ejercicio 10.51** Implemente una nueva clase `VisorDeTexto` que implemente `VisorDelSimulador`. `VisorDeTexto` proporciona una visión textual de la simulación: después de cada paso de la simulación, imprime una línea como la siguiente

Zorros: 121              Conejos: 266

Use el `VisorDeTexto` en lugar del `VisorAnimado` para realizar algunas pruebas. (No elimine las clases del `VisorAnimado`: queremos tener la capacidad de cambiar entre ambas vistas.)

**Ejercicio 10.52** ¿Puede hacer que ambas vistas estén activas al mismo tiempo?

## 10.6.6 ¿Clase abstracta o interfaz?

En algunas situaciones se tiene que elegir entre usar una clase abstracta o una interfaz. Algunas veces la elección es fácil: cuando se pretende que la clase contenga implementaciones para algunos métodos necesitamos usar una clase abstracta. En otros casos, tanto la clase abstracta como la interfaz pueden hacer el mismo trabajo.

Si tenemos que elegir, es preferible usar interfaces. Si proveemos un tipo mediante una clase abstracta, las subclases no pueden extender ninguna otra clase; dado que las interfaces permiten la herencia múltiple, el uso de una interfaz no crea tal restricción. Por lo tanto, el uso de interfaces da por resultado una estructura más flexible y más extensible.

## 10.7

### Resumen de herencia

En los últimos tres capítulos hemos discutido varios aspectos diferentes de las técnicas de herencia que incluyen herencia de código y subtipoado, así como la herencia a partir de interfaces, de clases abstractas y de clases concretas.

En general, distinguimos dos propósitos principales del uso de la herencia: podemos usarla para heredar código (código heredado) y podemos usarla para heredar el tipo (subtipoado). El primero es útil para reutilizar código, el segundo para el polimorfismo y la especialización.

Cuando heredamos a partir de clases concretas («extend») hacemos dos cosas: heredamos la implementación y el tipo. Cuando heredamos a partir de interfaces («implement») heredamos un tipo pero no la implementación. Para los casos en que ambas partes sean útiles podemos heredar a partir de clases abstractas; aquí, heredamos el tipo y una implementación parcial.

Cuando heredamos una implementación completa, podemos elegir agregar o sobrescribir métodos. Cuando no se hereda ninguna implementación de un tipo o se hereda parcialmente la implementación de un tipo, la subclase debe proveer la implementación antes de que pueda ser instanciada.

Algunos otros lenguajes orientados a objetos también proporcionan mecanismos para heredar código sin heredar el tipo. Java no provee este tipo de construcciones.

## 10.8

### Resumen

En este capítulo hemos discutido la estructura básica de las simulaciones por computadora.

Hemos usado este ejemplo para introducir clases abstractas e interfaces como construcciones que nos permiten crear abstracciones más avanzadas y desarrollar aplicaciones más flexibles.

Las clases abstractas son clases de las que no se tiene intención de tener ninguna instancia. Su propósito es servir como una superclase a otras clases. Las clases abstractas pueden tener tanto métodos abstractos (métodos que definen una firma pero no una implementación) como implementaciones de métodos. Las subclases concretas de clases abstractas deben sobrescribir los métodos abstractos para proveer implementaciones a los métodos.

Otra construcción para definir tipos en Java es la interfaz. Las interfaces de Java son similares a las clases totalmente abstractas: definen firmas de métodos pero no proveen ninguna implementación. Las interfaces definen tipos que pueden ser usados para las variables.

Las interfaces pueden usarse para proporcionar la especificación de una clase (o parte de una aplicación) sin establecer nada sobre la implementación concreta.

Java permite la herencia múltiple de interfaces (que se denominan relaciones «implements»), pero sólo herencia simple de clases (relaciones «extends»).

Términos introducidos en este capítulo

**método abstracto, clase abstracta, clase concreta, herencia múltiple, interfaz (construcción Java), implementa**

## Resumen de conceptos

- **método abstracto** Una definición de un método abstracto consiste en una firma de método sin un cuerpo. Se marca con la palabra clave **abstract**.
- **clase abstracta** Una clase abstracta es una clase de la que no se tiene intención de crear instancias. Su propósito es servir como una superclase a otras clases. Las clases abstractas pueden contener métodos abstractos.
- **subclases abstractas** Para que una subclase de una clase abstracta sea vuela concreta, debe proveer implementaciones para todos los métodos abstractos heredados; de lo contrario, es propiamente abstracta.
- **herencia múltiple** Una situación en la que una clase deriva de más de una superclase se denomina herencia múltiple.
- **interfaz** Una interfaz en Java es la especificación de un tipo (bajo la forma de un nombre de tipo y un conjunto de métodos) que no define ninguna implementación para ningún método.

**Ejercicio 10.53** ¿Puede una clase abstracta tener métodos concretos (no abstractos)? ¿Puede una clase concreta tener métodos abstractos? ¿Se puede tener una clase abstracta sin métodos abstractos? Justifique sus respuestas.

**Ejercicio 10.54** Observe el siguiente código. Se tienen cinco tipos (clases o interfaces) (U, G, B, Z y X) y una variable de cada uno de estos tipos.

```
U u;  
G g;  
B b;  
Z z;  
X x;
```

Las siguientes sentencias son todas legales (asuma que todas compilan).

```
u = z;  
x = b;  
g = u;  
x = u;
```

Las siguientes sentencias son todas ilegales (provocan errores de compilación).

```
u = b;  
x = g;  
b = u;  
z = u;  
g = x;
```

¿Qué puede decir sobre los tipos y sus relaciones? ¿Qué relaciones existen entre ellas?

**Ejercicio 10.55** Asuma que queremos modelar personas de una universidad para implementar un sistema de administración de cursos. Hay diferentes personas involucradas: miembros del personal, estudiantes, profesores, personal de mantenimiento, tutores, personal de soporte técnico y estudiantes técnicos. Los tutores y los estudiantes técnicos son interesantes: los tutores son estudiantes que han sido elegidos para enseñar algo y los estudiantes técnicos son estudiantes que han sido seleccionados para colaborar en el soporte técnico.

Dibuje una jerarquía de tipos (clases e interfaces) que represente esta situación. Indique qué tipos son clases concretas, clases abstractas e interfaces.

**Ejercicio 10.56** *Desafío.* Algunas veces, existen pares clase/interfaz en la biblioteca estándar de Java que definen exactamente los mismos métodos. Con frecuencia, el nombre de la interfaz finaliza con *Listener* y el nombre de la clase con *Adapter*. Un ejemplo es `PrintJobListener` y `PrintJobAdapter`. La interfaz define algunas signaturas de métodos y la clase adaptadora define los mismos métodos, cada uno con un cuerpo vacío. ¿Cuál podría ser el motivo de tener ambas clases?

**Ejercicio 10.57** La biblioteca de colecciones tiene una clase de nombre `TreeSet` que es un ejemplo de un conjunto ordenado. Los elementos de este conjunto se mantienen ordenados. Lea cuidadosamente la descripción de esta clase y luego escriba una clase `Persona` que pueda ser insertada en un `TreeSet`, que luego ordene los objetos `Persona` por edad.

# CAPÍTULO 11

## Construir interfaces gráficas de usuario

Principales conceptos que se abordan en este capítulo:

- construcción de IGU
- componentes de la interfaz
- esquemas de disposición de los componentes
- manejo de eventos

Construcciones Java que se abordan en este capítulo

JFrame, JLabel, JButton, JMenuBar, JMenu, JMenuItem, ActionEvent, Color, FlowLayout, BorderLayout, GridLayout, BoxLayout, Box, JOptionPane, EtchedBorder, EmptyBorder, clases internas anónimas

### 11.1

### Introducción

Hasta ahora, en este libro, nos hemos concentrado en escribir aplicaciones que tienen interfaces de usuario que utilizan exclusivamente texto. El motivo de usar estas interfaces textuales no es, en principio, que tengan una gran ventaja; de hecho, la única ventaja que tienen es que son fáciles de crear.

En realidad, no quisimos distraer mucho la atención de las cuestiones importantes del desarrollo de software, al dar los primeros pasos en el aprendizaje de la programación orientada a objetos: nos centramos en cuestiones relacionadas con la estructura y la interacción de los objetos, el diseño de clases y la calidad del código.

Las interfaces gráficas de usuario (IGU) también se construyen a partir de objetos que interactúan, pero tienen una estructura muy especializada y es por esto que evitamos introducirlas antes de aprender la estructura de los objetos en términos más generales. Sin embargo, ahora estamos preparados para dar una mirada a la construcción de las IGU.

Las IGU completan nuestras aplicaciones con una interfaz formada por ventanas, menús, botones y otros componentes gráficos, y hacen que la aplicación tenga una apariencia más similar a las típicas aplicaciones que la mayoría de la gente usa hoy en día.

Observe que estamos tropezando nuevamente con el doble significado de la palabra *interfaz*. Las interfaces de las que estamos hablando ahora no son las interfaces de las clases ni la construcción *interface* de Java. Hablamos de *interfaces de usuario*, la parte de una aplicación que está visible en la pantalla y que permite que un usuario interactúe con ella.

Una vez que sepamos cómo crear una IGU en Java, podremos desarrollar programas que tengan una mejor presentación visual.

## 11.2

# Componentes, gestores de disposición y captura de eventos

Los detalles involucrados en la creación de una IGU son numerosísimos. En este libro no cubriremos todos los detalles de todas las posibles cosas que se pueden hacer, sino que discutiremos los principios generales y un buen número de ejemplos.

En Java, toda la programación de una IGU se realiza mediante el uso de bibliotecas de clases estándares especializadas. Una vez que comprendemos los principios, podemos encontrar todos los detalles necesarios en la documentación de la biblioteca estándar.

Los principios que necesitamos comprender se pueden dividir en tres áreas:

- ¿Qué clase de elementos podemos mostrar en una pantalla?
- ¿Cómo podemos acomodar estos elementos?
- ¿Cómo podemos reaccionar ante una entrada del usuario?

Discutiremos estas cuestiones mediante los términos *componentes, gestores de disposición y manejo de eventos*.

### Concepto

Una IGU se construye mediante **componentes** que se ubican en la pantalla. Los componentes se representan mediante objetos.

### Concepto

La distribución de los componentes en la pantalla se lleva a cabo mediante **gestores de disposición**.

### Concepto

La terminología **manejo de eventos** hace referencia a la tarea de reaccionar a los eventos que produce el usuario como por ejemplo, hacer clic sobre el botón del ratón o ingresar algo por teclado.

Los *componentes* son las partes individuales a partir de las cuales se construye una IGU. Son cosas tales como botones, menús, elementos de menú, cajas de verificación, deslizadores, campos de texto, etc. La biblioteca de Java contiene una buena cantidad de componentes listos para usar y también podemos escribir los propios. Tendremos que aprender cuáles son los componentes importantes, cómo se crean y cómo hacer para que aparezcan en la pantalla tal cual deseamos verlos.

Los *gestores de disposición* participan de cuestiones relacionadas con la ubicación de los componentes en la pantalla. Los sistemas de IGU más viejos y primitivos manejaban coordenadas bidimensionales: el programador especificaba las coordenadas *x* e *y* (expresadas en píxeles) para determinar la posición y el tamaño de cada componente. En los sistemas de IGU más modernos, esta forma resulta demasiado simplista. Debemos tener en cuenta distintas resoluciones de pantalla, diferentes fuentes, ventanas que los usuarios pueden redimensionar, y muchos otros aspectos que vuelven mucho más difícil la distribución de los componentes. La solución será un esquema en el que podamos especificar la disposición de los componentes en términos más generales. Por ejemplo, podemos especificar que «este componente deberá estar debajo de este otro» o que «este componente se estrechará cuando la ventana cambie de tamaño pero ese otro tendrá un tamaño constante». Veremos que todo esto se logra mediante el uso de *gestores de disposición*.

El *manejo de eventos* se refiere a la técnica que usaremos para trabajar con las entradas del usuario. Una vez que hemos creado nuestros componentes y que los posicionamos en la pantalla, también tenemos que estar seguros de que ocurra algo cuando el usuario presione un botón. El modelo que usa la biblioteca de Java para lograr esto se basa en eventos: si un usuario activa un componente (por ejemplo, presiona un botón o selecciona un elemento de un menú) el sistema generará un evento. Entonces, nuestra aplicación puede recibir una notificación del evento (mediante la invocación de uno de sus métodos) y podemos llevar a cabo la acción adecuada.

Discutiremos cada una de estas áreas mucho más detalladamente en este capítulo. Primero, como siempre, introduciremos brevemente un poco más de terminología y de fundamento.

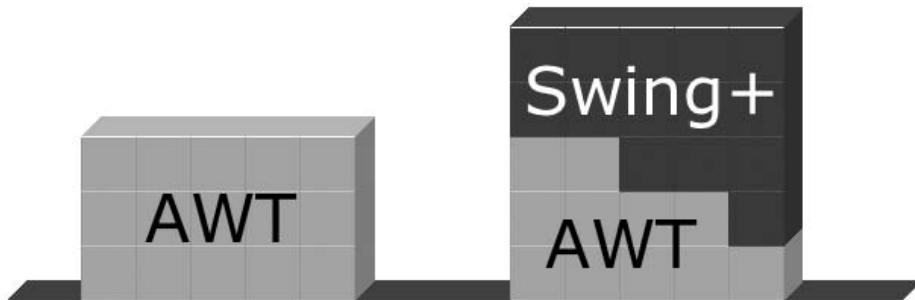
## 11.3

### AWT y Swing

Java tiene dos bibliotecas para la construcción de interfaces gráficas de usuario. La más antigua se denomina *AWT (Abstract Window Toolkit)* y fue introducida con el primer sistema Java original; más tarde, se agregó una biblioteca mucho mejor de nombre *Swing*.

**Figura 11.1**

AWT y Swing



Swing utiliza algunas de las clases de la biblioteca AWT, reemplaza algunas de las clases de AWT con sus propias versiones y agrega muchas clases nuevas (Figura 11.1).

En este libro, usaremos la biblioteca Swing; quiere decir que usaremos algunas de las clases AWT que todavía se utilizan en los programas Swing, pero usamos las versiones Swing de todas las clases que existen en ambas bibliotecas.

Como existen clases equivalentes en AWT y en Swing, las versiones Swing han sido identificadas mediante el agregado de la letra «J» al comienzo del nombre de la clase. Verá, por ejemplo, clases de nombre Button y JButton, Frame y JFrame, Menu y JMenu, y así sucesivamente. Las clases que comienzan con «J» son versiones Swing; son las únicas que usaremos.

Estos conceptos básicos alcanzan para empezar y ahora, veamos un poco de código.

## 11.4

### El ejemplo Visor de Imágenes

Como siempre, discutiremos los nuevos conceptos mediante un ejemplo. La aplicación que construiremos en este capítulo es un visor de imágenes (Figura 11.2). Es un pro-

grama que puede abrir y mostrar imágenes almacenadas en archivos con formato JPEG y PNG, puede realizar algunas transformaciones de las imágenes y grabarlas nuevamente en el disco.

**Figura 11.2**

Una aplicación sencilla para visualizar imágenes



### Concepto

#### Formato de imagen.

Las imágenes se pueden almacenar en diferentes formatos. Las diferencias tienen que ver principalmente con el tamaño del archivo y con la información que contienen.

En esta aplicación, usaremos nuestra propia clase de imagen para representar una imagen mientras permanece en memoria, implementaremos varios filtros para modificar el aspecto de la imagen y usaremos componentes Swing para construir una interfaz de usuario. Mientras vamos haciendo todo esto, centraremos nuestra discusión en las características de la IGU del programa.

Si tiene curiosidad por ver lo que construiremos, puede abrir y probar el proyecto *visor-de-imagen-1-0*: sólo debe crear un objeto *VisorDeImagen*; esta es la versión que se muestra en la Figura 11.2. Aquí comenzamos lentamente, al principio con algo muy simple y luego iremos mejorando paso a paso nuestra aplicación hasta llegar a la versión final.

#### 11.4.1

#### Primeros experimentos: crear una ventana

Casi todo lo que se puede ver en una IGU está contenido en un tipo de ventana del más alto nivel. Una ventana del nivel más alto es una ventana que está bajo el control del administrador de ventanas del sistema operativo y que típicamente puede moverse, cambiar de tamaño, minimizarse y maximizarse de manera independiente.

En Java, estas ventanas del más alto nivel se denominan *frames* y en Swing, se representan mediante la clase de nombre *JFrame*.

**Código 11.1**

Una primera versión  
de la clase  
*VisorDeImagen*

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
// Se omite el comentario
public class VisorDeImagen
{
    private JFrame ventana;

    /**
     * Crea un VisorDeImagen y lo muestra en la pantalla.
     */
    public VisorDeImagen ()
    {
        construirVentana();
    }

    /**
     * Crea la ventana Swing y su contenido.
     */
    private void construirVentana()
    {
        ventana = new JFrame("Visor de Imágenes");
        Container panelContenedor =
ventana.getContentPane();

        JLabel etiqueta = new JLabel("Soy una
etiqueta.");
        panelContenedor.add(etiqueta);
        ventana.pack();
        ventana.setVisible(true);
    }
}
```

Para obtener una IGU en la pantalla, lo primero que tenemos que hacer es crear y mostrar una ventana. El Código 11.1 presenta una clase completa que muestra una ventana en la pantalla (que ya tiene el nombre *VisorDeImagen* para prepararla para todo lo que sigue). Esta clase está disponible en los proyectos de este libro bajo el nombre *visor-de-imagen-0-1* (el número indica que es la versión 0.1 de la aplicación).

**Ejercicio 11.1** Abra el proyecto *visor-de-imagen-0-1*; este proyecto será la base para crear su propio visor de imágenes. Cree una instancia de la clase *VisorDeImagen*. Modifique el tamaño de la ventana que aparece en pantalla (agrándela). ¿Qué observa con respecto a la ubicación del texto en la ventana?

Ahora discutiremos más detalladamente sobre la clase *VisorDeImagen* que se muestra en el Código 11.1.

Las primeras tres líneas de dicha clase son sentencias de importación de todas las clases de los paquetes `java.awt`, `java.awt.event` y `javax.swing`<sup>1</sup>. Necesitamos varias de las clases de estos paquetes para todas las aplicaciones Swing que creemos, por lo que siempre importamos estos tres paquetes completos en todos los programas que construyan interfaces gráficas de usuario.

Observando el resto de la clase se ve rápidamente que toda la parte interesante está en el método `construirVentana`. Este método es el encargado de construir la IGU. El constructor de la clase contiene sólo una llamada a este método. Hemos hecho esto para que todo el código destinado a la construcción de la IGU esté en un lugar bien definido y más adelante, resulte más fácil encontrarlo (¡cohesión!). Haremos lo mismo en todos nuestros ejemplos de IGU.

La clase tiene una variable de instancia de tipo `JFrame` que se usa para contener a la ventana que necesita el visor para mostrar las imágenes en la pantalla.

Veamos más de cerca el método `construirVentana`.

La primer línea de este método es

```
ventana = new JFrame("Visor de Imágenes");
```

Esta sentencia crea una nueva ventana y la almacena en nuestra variable de instancia, para poder usarla más adelante.

Como principio general, en paralelo con el estudio de los ejemplos en este libro usted debería buscar la documentación de todas las clases que encontrremos. Esto es válido para todas las clases que usemos; no indicaremos esta cuestión nuevamente a partir de ahora, pero esperamos que lo haga.

**Ejercicio 11.2** Busque la documentación de la clase `JFrame`. ¿Cuál es la finalidad del parámetro «Visor de Imágenes» que se usa en la llamada al constructor?

#### Concepto

Los componentes se ubican en una ventana agregándolos a la **barra de menú** o al **panel contenedor**.

Una ventana consta de tres partes: la *barra del título*, una *barra de menú* opcional y un *panel contenedor* (Figura 11.3). La apariencia exacta de la barra del título depende del sistema operativo que se esté usando. Generalmente, contiene el título de la ventana y unos pocos controles para la ventana.

La barra de menú y el panel contenedor están bajo el control de la aplicación. Podemos agregar algunos componentes en ambos para crear una IGU. Nos concentraremos primero en el panel contenedor.

#### 11.4.2 Agregar componentes simples

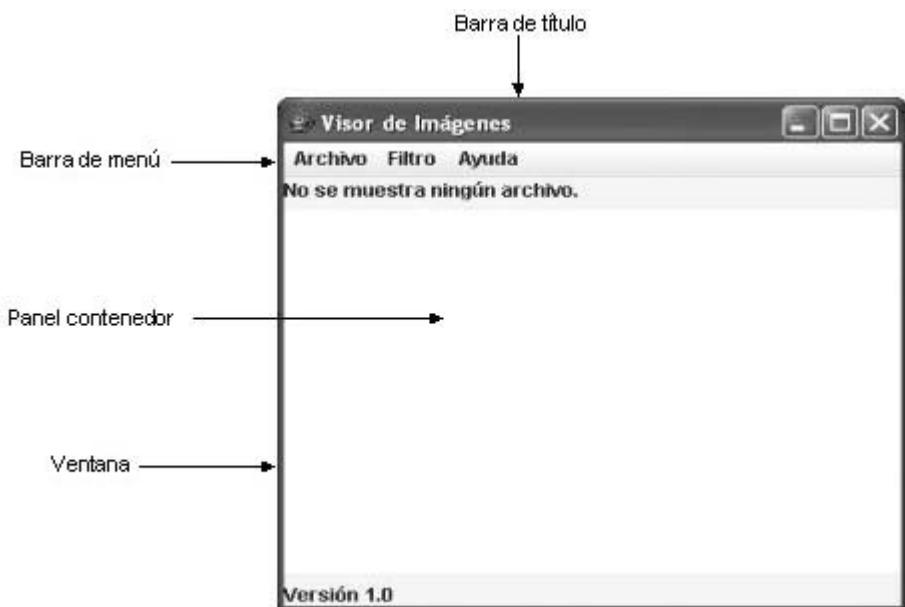
Inmediatamente después la creación del `JFrame`, la ventana no estará visible y su panel contenedor estará vacío. Continuamos el trabajo agregando una etiqueta al panel contenedor:

```
Container panelContenedor = ventana.getContentPane();
JLabel etiqueta = new JLabel("Soy una etiqueta.");
panelContenedor.add(etiqueta);
```

<sup>1</sup> En realidad, el paquete `swing` forma parte de un paquete denominado `javax` (termina con «x») y no `java`. La razón de este nombre es fundamentalmente histórica, no parece existir una explicación lógica para este nombre.

**Figura 11.3**

Diferentes partes de una ventana



La primera línea obtiene el panel contenedor de la ventana. Siempre debemos hacer esto: los componentes de la IGU se agregan a la ventana agregándolos al panel contenedor de la misma.

El panel contenedor es en sí mismo de tipo `Container`. Un contenedor es un componente Swing que puede contener grupos arbitrarios de otros componentes, prácticamente de la misma manera en que un `ArrayList` puede contener una colección arbitraria de objetos. Más adelante, hablaremos más detalladamente sobre los contenedores.

Luego, creamos un componente etiqueta (de tipo `JLabel`) y lo agregamos al panel contenedor. Una etiqueta es un componente que puede mostrar texto o alguna imagen, o ambas cosas a la vez.

Finalmente, tenemos las dos líneas

```
ventana.pack();
ventana.setVisible(true);
```

La primera línea hace que la ventana distribuya adecuadamente los componentes dentro de ella y les asigne el tamaño apropiado. Siempre tenemos que invocar el método `pack` sobre la ventana después de haber agregado o modificado el tamaño de sus componentes.

La última línea finalmente hace que la ventana se vuelva visible en la pantalla. Siempre comenzamos con una ventana que inicialmente es invisible, por lo que podemos acomodar todos los componentes dentro de ella sin que este proceso sea visible en la pantalla. Luego, cuando la ventana esté construida, podemos mostrarla en su estado completo.

**Ejercicio 11.3** Otro componente Swing que se usa con mucha frecuencia es el botón (de tipo `JButton`). Reemplace la etiqueta del ejemplo anterior por un botón.

**Ejercicio 11.4** ¿Qué ocurre cuando agrega dos etiquetas (o dos botones) al panel contenedor? ¿Puede explicar lo que observa? Experimente modificando el tamaño de la ventana.

### 11.4.3 Agregar menús

Nuestro próximo paso en la construcción de una IGU es agregar menús y elementos de menú. Esto es conceptualmente fácil pero contiene un detalle delicado: ¿cómo nos arreglaremos para reaccionar a las acciones del usuario como por ejemplo, a la selección de un elemento de un menú? Discutiremos esto luego.

Primero, creamos los menús. Las tres clases involucradas en esta tarea son:

- **JMenuBar** – Un objeto de esta clase representa una barra de menú que se puede mostrar debajo de la barra de título, en la parte superior de una ventana (véase la Figura 11.3). Cada ventana tiene un **JMenuBar** como máximo<sup>2</sup>.
- **JMenu** – Los objetos de esta clase representan un solo menú (como por ejemplo, los menús comunes «Archivo», «Edición» o «Ayuda»). Los menús frecuentemente están contenidos en una barra de menú; también pueden aparecer en menús emergentes, pero ahora no haremos esto.
- **JMenuItem** – Los objetos de esta clase representan un solo elemento de menú dentro de un menú, como por ejemplo, «Abrir» o «Grabar».

Para nuestro visor de imágenes, crearemos una barra de menú y varios menús y elementos de menú.

La clase **JFrame** tiene un método de nombre **setJMenuBar**. Podemos crear una barra de menú y usar este método para adjuntar nuestra barra de menú a la ventana:

```
JMenuBar barraDeMenu = new JMenuBar();
ventana.setJMenuBar(barraDeMenu);
```

Ahora estamos listos para crear un menú y agregarlo a la barra de menú:

```
JMenu menuArchivo = new JMenu(_Archivo_);
barraDeMenu.add(menuArchivo);
```

Estas dos líneas crean un menú con la etiqueta «Archivo» y lo insertan en la barra de menú. Finalmente, podemos agregar elementos al menú. Las siguientes líneas agregan dos elementos con las etiquetas «Abrir» y «Salir» al menú «Archivo».

```
JMenuItem elementoAbrir = new JMenuItem(_Abrir_);
menuArchivo.add(elementoAbrir);
JMenuItem elementoSalir = new JMenuItem(_Salir_);
menuArchivo.add(elementoSalir);
```

**Ejercicio 11.5** Agregue en su proyecto visor de imágenes, el menú y los elementos de menú mencionados en esta sección. ¿Qué ocurre cuando selecciona un elemento del menú?

---

<sup>2</sup> En el sistema operativo Mac, la forma nativa de mostrar es diferente: la barra de menú se ubica en la parte superior de la pantalla y no en la parte superior de cada ventana. En las aplicaciones Java, el comportamiento por defecto es adjuntar la barra de menú a la ventana. En las aplicaciones Java, puede ubicarse la barra de menú en la parte superior de la pantalla usando una propiedad específica del S.O. Mac.

**Ejercicio 11.6** Agregue otro menú de nombre «Ayuda» que contiene un elemento de menú con la etiqueta «Acerca del Visor de Imágenes». (Nota: para aumentar la legibilidad y la cohesión, puede ser una buena idea el mover la creación de los menús a un método separado, quizás bajo el nombre `construirBarraDeMenu`, que se invoque desde nuestro método `construirVentana`).

Hasta ahora, hemos llevado a cabo la mitad de nuestra tarea: podemos crear y mostrar menús pero falta la segunda mitad: todavía no ocurre nada cuando un usuario selecciona un menú. Ahora tenemos que agregar código para reaccionar a las selecciones del menú. Este es el tema que discutimos en la próxima sección.

#### 11.4.4 Manejo de eventos

Swing usa un modelo muy flexible para reaccionar ante los ingresos que se producen en la IGU: un modelo de *manejo de eventos* mediante *oyentes de eventos*.

El marco de trabajo Swing y algunos de sus componentes disparan eventos cuando ocurre algo en que otros objetos pueden estar interesados. Existen diferentes tipos de eventos provocados por diferentes tipos de acciones: cuando se presiona un botón o se selecciona un elemento de un menú, el componente dispara un *ActionEvent*; cuando se presiona un botón del ratón o se mueve el ratón, se dispara un *RatónEvent*; cuando se cierra una ventana o se la transforma en ícono, se genera un *WindowEvent*. Existen muchos otros tipos de eventos.

Cualquiera de nuestros objetos puede convertirse en oyente de cualquiera de estos eventos. Un objeto oyente se notificará de cualquiera de los eventos que es capaz de oír.

Un objeto se convierte en un oyente de eventos mediante la implementación de varias interfaces de oyentes que existen. Si implementa la interfaz correcta, puede registrarse a sí mismo como uno de los componentes al que quiere oír.

Veamos un ejemplo. Los elementos del menú (clase `JMenuItem`) disparan eventos de acción (*ActionEvents*) cuando son activados por un usuario. Los objetos que desean oír estos eventos deben implementar la interfaz `ActionListener` del paquete `java.awt.event`.

Hay dos estilos alternativos para la implementación de oyentes de eventos: un único objeto oye los eventos provenientes de varias fuentes diferentes o bien, a cada fuente de eventos diferente se le asigna su propio y único oyente. Discutiremos ambos estilos en las siguientes dos secciones.

**Concepto**  
Un objeto puede escuchar los eventos de los componentes implementando una interfaz **oyente de eventos**.

#### 11.4.5 Recepción centralizada de eventos

Para lograr que nuestro objeto `VisorDeImagen` se convierta en el único oyente de todos los eventos que provienen del menú tenemos que hacer tres cosas:

1. Debemos declarar, en el encabezado de la clase, que implementa la interfaz `ActionListener`.
2. Tenemos que implementar un método con la firma  
`public void actionPerformed(ActionEvent e)`

Este es el único método que se define en la interfaz `ActionListener`.

3. Debemos invocar al método `addActionListener` del elemento del menú para registrar al objeto `VisorDeImagen` como un oyente.

Los números 1 y 2, la implementación de la interfaz y la definición de su método, aseguran que nuestro objeto es un subtipo de `ActionListener`. Luego, el número 3 registra nuestro propio objeto como un oyente de los elementos del menú. El Código 11.2 muestra el código fuente para este contexto.

### Código 11.2

Agregar un oyente de acción a un elemento del menú

```
public class VisorDeImagen
    implements ActionListener
{
    // Se omiten los campos y el constructor

    public void actionPerformed(ActionEvent evento)
    {
        System.out.println("Elemento: " +
evento.getActionCommand());
    }

    /**
     * Crea la ventana Swing y su contenido.
     */
    private void construirVentana()
    {
        ventana = new JFrame("Visor de Imágenes");
        construirBarraDeMenu(ventana);

        // Se omite el resto de la construcción de la IGU
    }
    /**
     * Crea la barra de menú de la ventana.
     */
    private void construirBarraDeMenu(JFrame ventana)
    {
        JMenuBar barraDeMenu = new JMenuBar();
        ventana.setJMenuBar(barraDeMenu);

        // crea el menú Archivo
        JMenu menuArchivo = new JMenu("Archivo");
        barraDeMenu.add(menuArchivo);

        JMenuItem elementoAbrir = new JMenuItem("Abrir");
        elementoAbrir.addActionListener(this);
        menuArchivo.add(elementoAbrir);
        JMenuItem elementoSalir = new JMenuItem("Salir");
        elementoSalir.addActionListener(this);
        menuArchivo.add(elementoSalir);
    }
}
```

Observe especialmente las líneas

```
JMenuItem elementoAbrir = new JMenuItem("Abrir");  
elementoAbrir.addActionListener(this);
```

en el código del ejemplo. Aquí, se crea un elemento del menú y se registra el objeto actual (el propio objeto *VisorDeImagen*) como un oyente de acción, pasando al método *addActionListener* el parámetro *this*.

El efecto de registrar nuestro objeto como un oyente a través del elemento del menú, es que se invocará nuestro propio método *actionPerformed* mediante el elemento del menú, cada vez que se active este elemento. Cuando se invoque nuestro método, el elemento del menú será pasado como un parámetro de tipo *ActionEvent* que proporciona algunos detalles sobre el evento que ha ocurrido. Estos detalles incluyen el momento exacto del evento, el estado de las teclas modificadoras (control, shift y meta teclas) y una «cadena de comando», entre otras cosas.

La cadena de comando es una cadena que, de alguna manera, identifica al componente que produjo el evento. Para los elementos del menú, esta identificación se realiza, por defecto, mediante el texto de la etiqueta del elemento.

En nuestro ejemplo del Código 11.2, registramos el mismo objeto de acción para ambos elementos del menú. Esto quiere decir que ambos elementos del menú, cuando se activen, invocarán al mismo método *actionPerformed*.

En el método *actionPerformed*, simplemente imprimimos la cadena de comando del elemento para demostrar que este esquema funciona. Este es el lugar donde podríamos agregar el código adecuado para manejar la invocación del menú.

Este código de ejemplo, tal como lo hemos hecho hasta ahora, está disponible entre los proyectos que acompañan este libro bajo el nombre *visor-de-imagen-0-2*.

**Ejercicio 11.7** Implemente el código para manejar el menú que hemos discutido anteriormente en su propio proyecto del visor de imágenes. También tiene la alternativa de abrir el proyecto *visor-de-imagen-0-2* y examinar cuidadosamente su código. Describa por escrito y detalladamente la secuencia de eventos que se produce como resultado de activar el elemento *Salir* del menú.

**Ejercicio 11.8** Agregue otro elemento al menú de nombre *Grabar*.

**Ejercicio 11.9** Agregue tres métodos privados a su propia clase de nombres *abrirArchivo*, *grabarArchivo* y *salir*. Modifique el método *actionPerformed* para que invoque al método que corresponda cuando se active un elemento del menú.

**Ejercicio 11.10** Si resolvió el Ejercicio 11.6 (agregar el menú *Ayuda*), asegúrese de que este elemento de menú también funcione adecuadamente al activarse.

Vemos que este abordaje funciona.

Ahora podemos implementar métodos para manejar los elementos del menú de modo que realicen varias de las tareas de nuestro programa. Sin embargo, existe otro aspecto que debemos investigar: la solución actual no es muy buena en términos de mantenimiento y de extensibilidad.

Examine el código que escribió para el método `actionPerformed` en el Ejercicio 11.9. Existen varios problemas:

- Probablemente usó una sentencia `if` y el método `getActionCommand` para encontrar cuál es el elemento que se activó. Por ejemplo, pudo haber escrito:

```
if(evento.getActionCommand().equals("Abrir"))...
```

La dependencia de la cadena de la etiqueta del elemento para llevar a cabo la función correspondiente no es una buena idea. ¿Qué ocurre si se traduce la interfaz a otro idioma? Sólo un cambio en el texto del elemento del menú provocaría que el programa dejase de funcionar. (O bien, tendría que encontrar todos los lugares del código en los que se usó esta cadena y modificarla; un procedimiento muy tedioso y una gran fuente de errores.)

- El hecho de que el despacho de métodos esté centralizado (tal como lo hace nuestro `actionPerformed`) no es una buena estructura para nada. Esencialmente, construimos un único método sólo para luego escribir código tedioso en el que invocamos a los métodos separados correspondientes a cada elemento del menú. Esto no tiene sentido en términos de mantenimiento: para cada elemento adicional del menú tendremos que agregar una nueva sentencia `if` en el método `actionPerformed`. También parece ser un esfuerzo en vano. Sería mucho mejor si pudiéramos hacer que cada elemento del menú invoque directamente a cada método por separado.

En la próxima sección introducimos una nueva construcción del lenguaje que nos permite llevar a cabo la solución que sugerimos.

#### 11.4.6 Clases internas

Para solucionar los problemas que presenta el despacho centralizado de métodos que mencionamos anteriormente, usamos una nueva construcción que no hemos tratado con anterioridad: las *clases internas*. Las clases internas son clases que se declaran textualmente dentro de otra clase:

```
class ClaseEnvolvente
{
    ...
    class ClaseInterna
    {
        ...
    }
}
```

Las instancias de la clase interna se adjuntan a las instancias de la clase envolvente: sólo pueden existir dentro de una instancia que las envuelva. Las instancias de las clases internas conceptualmente existen en el *interior* de una instancia que las envuelve.

Un detalle interesante es que las sentencias de los métodos de la clase interna pueden ver y acceder a los campos y métodos privados de la clase envolvente. La clase interna se considera una parte de la clase externa que la envuelve, al igual que cualquiera de los métodos de la clase envolvente.

Ahora podemos usar esta construcción para armar una clase oyente de acción independiente para cada uno de los elementos del menú que queremos que oiga los eventos. Al ser clases independientes, cada una puede tener un método `actionPerformed` sepa-

rado, de modo tal que cada uno de estos métodos sólo maneje la activación de un único elemento. La estructura sería ésta:

```
class VisorDeImagen
{
    ...
    class AbrirActionListener implements ActionListener
    {
        public void actionPerformed(ActionEvent evento)
            // lleva a cabo la acción abrir
    }
}

class salirActionListener implements ActionListener
{
    public void actionPerformed(ActionEvent evento)
    {
        // lleva a cabo la acción de salir
    }
}
```

Como una guía de estilo, generalmente escribimos las clases internas al final de la clase envolvente, a continuación de los métodos.

Una vez que hemos escrito estas clases internas, podemos crear instancias de estas clases internas exactamente de la misma manera en que lo hacemos a partir de cualquier otra clase. Observe que `VisorDeImagen` no implementa más `ActionListener` pues hemos eliminado su método `actionPerformed`, pero sí lo hacen las dos clases internas. Esto nos permite usar instancias de las clases internas como oyentes de acción de los elementos del menú:

```
JMenuItem elementoAbrir = new JMenuItem("Abrir");
elementoAbrir.addActionListener(new AbrirActionListener());
...
JMenuItem elementoSalir = new JMenuItem("Salir");
elementoSalir.addActionListener(new SalirActionListener());
```

En resumen, en lugar de que el objeto visor de imagen sea el oyente de todos los eventos de acción, creamos objetos oyentes independientes para cada posible evento, donde cada oyente sólo escucha un único tipo de evento. Como cada oyente tiene su propio método `actionPerformed`, ahora podemos escribir el código específico necesario para manejar los eventos en estos métodos. Y como el alcance de las clases oyentes se extiende a la clase envolvente (pueden acceder a los campos privados de la clase envolvente y a sus métodos), podemos hacer un uso completo de la clase envolvente en la implementación de los métodos `actionPerformed`.

**Ejercicio 11.11** Implemente el manejo de eventos de los elementos del menú mediante clases internas, tal como lo discutimos aquí, en su propia versión del visor de imágenes.

Generalmente, se pueden usar las clases internas en algunos casos para mejorar la cohesión de proyectos grandes. Por ejemplo, el proyecto *zorros-y-conejos* del Capítulo 10 tiene una clase `VisorDelSimulador` que incluye a la clase interna `VisorDelCampo`. Podría estudiar este ejemplo para ampliar su comprensión sobre las clases internas.

### 11.4.7 Clases internas anónimas

La solución al problema del despacho de las acciones que utiliza clases internas es bastante buena pero queremos avanzar un poco más: podemos usar *clases internas anónimas*. El proyecto *visor-de-imagen-0-3* muestra una implementación que utiliza esta construcción.

**Ejercicio 11.12** Abra el proyecto *visor-de-imagen-0-3* y examínelo: pruebe y lea su código. No se preocupe si no comprende todo el código porque algunas de las características nuevas son temas de esta sección. ¿Qué observa sobre el uso de las clases internas para permitir que el `VisorDeImagen` escuche y maneje los eventos?

**Ejercicio 11.13** Habrá notado que ahora, al activar el elemento *Salir* del menú, el programa finaliza. Examine cómo lo hace. Busque la documentación de la biblioteca relacionada con todas las clases y métodos involucrados.

En el centro de las modificaciones de esta versión se encuentra la forma en que los oyentes de acción se configuran para que escuchen los eventos de acción de los elementos del menú. El código relevante es como sigue:

```
JMenuItem elementoAbrir = new JMenuItem("Abrir");
elementoAbrir.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        archivoAbrir();
    }
});
```

Este fragmento de código parece bastante misterioso cuando se lee por primera vez y, probablemente, tendrá algunas dudas sobre su interpretación, aun cuando haya comprendido todo lo que hemos discutido en este libro hasta ahora. Esta construcción, desde el punto de vista sintáctico, es probablemente el ejemplo más confuso que haya visto en lenguaje Java. Pero no se preocupe, lo investigaremos lentamente.

Lo que está viendo en este fragmento de código es una clase interna anónima. La idea de esta construcción está basada en la observación de nuestra versión anterior, que usó cada clase interna exactamente una y sólo una vez para crear una única instancia. En esta situación, las clases internas anónimas ofrecen un atajo sintáctico: nos permiten definir una clase y crear una sola instancia de ella, todo en un solo paso. El efecto es idéntico al de la clase interna de la versión anterior, con la diferencia de que no es necesario definir nombres para cada una de las clases oyentes, y que la definición del método oyente está más cerca de la registración del oyente del elemento del menú.

Cuando usamos una clase interna anónima, creamos una clase interna que no tiene *ningún nombre* e inmediatamente creamos una sola instancia de esa clase. En el código del oyente de acción anterior, esto se hace mediante el fragmento

```
new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        archivoAbrir();
    }
}
```

Se crea una clase interna anónima nombrando un supertipo (frecuentemente, una clase abstracta o una interfaz, en este caso, `ActionListener`), seguido de un bloque que contiene una implementación para sus métodos abstractos.

En este ejemplo, creamos una nueva subclase de `ActionListener` que implementa el método `actionPerformed`. Esta nueva subclase no recibe un nombre; en su lugar, la prefijamos con la palabra clave `new` para crear una sola instancia de esta clase.

En nuestro ejemplo, esta instancia es un objeto oyente de acción, ya que es un subtipo de `ActionListener`. Puede pasarse al método `addActionListener` del elemento del menú y luego invocar al método `archivoAbrir` de su clase envolvente, cuando se active el elemento del menú.

De la misma manera que las clases internas que tienen nombre, las clases internas anónimas pueden acceder a los campos y métodos de su clase envolvente. Además, dado que están definidas dentro de un método, pueden acceder a las variables locales y a los parámetros de dicho método. Sin embargo, una regla importante es que las variables locales accedidas de esta manera deben ser declaradas como `final`. Verá un ejemplo de estas variables en el proyecto *visor-de-imagen-2-0* que se discutirá en la Sección 11.6.

Es importante enfatizar algunas observaciones sobre las clases internas anónimas.

Primero, en nuestro problema concreto resulta de mucha utilidad el uso de las clases internas anónimas. Nos permiten eliminar por completo el método central `actionPerformed` de nuestra clase `VisorDeImagen`. En su lugar, creamos un oyente de acción independiente, hecho a medida (clase y objeto), para cada elemento del menú. Este oyente de acción puede invocar directamente al método para implementar la función correspondiente.

Esta estructura es mucho más cohesiva y extendible. Si necesitamos elementos adicionales en el menú, sólo agregamos código para crear el elemento y su respectivo oyente, y el método que maneje su función. No se requiere listarlo en un método central.

Segundo, el uso de clases internas anónimas hace que el código resulte bastante más difícil de leer. Recomendamos fuertemente usar estas clases sólo dentro de clases muy cortas y sólo para modismos de código bien determinados.

Tercero, con frecuencia usamos clases anónimas en los lugares en los que se requiere la implementación de una sola instancia: las acciones asociadas con cada elemento del menú son únicas para ese elemento en particular. Además, siempre se hará referencia a la instancia mediante su supertipo. Ambas razones quieren decir que el nombre de la nueva clase no es tan necesario, por lo tanto, puede ser anónima.

Para nosotros, la implementación de oyentes de eventos es el único ejemplo de este libro en el que usamos esta construcción<sup>3</sup>.

En todo nuestro trabajo siguiente, evitaremos el método central `actionListener` y usaremos en su lugar, clases internas anónimas. Por lo tanto, puede dejar de lado el proyecto *visor-de-imagen-0-2* y usar la estructura del proyecto *visor-de-imagen-0-3* como base para su futuro trabajo.

### Concepto

Las **clases internas anónimas** son construcciones muy útiles para implementar oyentes de eventos.

<sup>3</sup> Si quiere encontrar más información sobre las clases internas, puede recurrir a estas dos secciones del tutorial online de Java (en inglés): <http://java.sun.com/docs/books/tutorial/java/javaOO/nested.html> y <http://java.sun.com/docs/books/tutorial/java/javaOO/innerclasses.html>

## 11.5

# Visor de Imágenes 1.0: primera versión completa

Llegar al estado actual, es decir, mostrar una ventana con una etiqueta y algunos menús, fue un trabajo difícil y, en el camino, hemos discutido una gran cantidad de conceptos. ¡Será realmente más fácil de aquí en adelante! El detalle más dificultoso que hemos tenido que aplicar en nuestro ejemplo es, probablemente, el relacionado con la comprensión del manejo de los eventos de los elementos del menú.

Ahora trabajaremos en la creación de la primera versión completa, una versión que realmente pueda realizar la tarea principal: mostrar algunas imágenes en la pantalla.

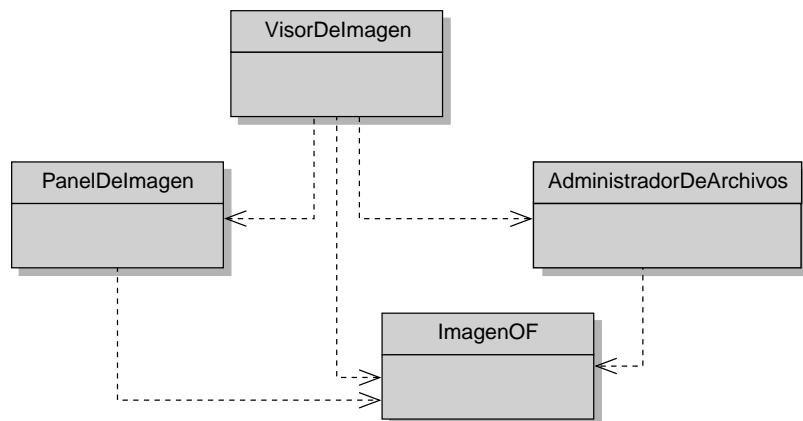
### 11.5.1

#### Clases para procesar imágenes

Encaminados hacia la solución, investigaremos una versión intermedia: *visor-de-imagen-0-4* cuya estructura de clases se muestra en la Figura 11.4.

**Figura 11.4**

Estructura de clases de la aplicación Visor de Imágenes



Como se puede ver, hemos agregado tres clases nuevas: `ImagenOF`, `PanelDeImagen` y `AdministradorDeArchivos`. `ImagenOF` es una clase que representa la imagen que queremos mostrar y manipular. `AdministradorDeArchivos` es una clase colaboradora que proporciona métodos estáticos para leer una imagen desde el disco (en formato JPEG o PNG) y devolverla en formato `ImagenOF`, y para grabar la `ImagenOF` nuevamente en el disco. `PanelDeImagen` es un componente Swing personalizado destinado a mostrar la imagen en nuestra IGU.

Discutiremos brevemente los aspectos más importantes de cada una de estas clases con un poco más de detalle. Sin embargo, no las explicaremos totalmente sino que las dejamos para que el lector curioso las investigue.

La clase `ImagenOF` es nuestro propio formato personalizado para representar una imagen en la memoria. Puede pensar en `ImagenOF` como un arreglo bidimensional de píxeles en el que, cada uno de los píxeles puede tener un color. Usamos la clase estándar `Color` (del paquete `java.awt`) para representar el color de cada píxel. (Dé una mirada a la documentación de la clase `Color`, la necesitaremos más adelante.)

ImagenOF está implementada como una subclase de la clase estándar de Java `BufferedImage` (del paquete `java.awt.image`). `BufferedImage` nos aporta la mayor parte de la funcionalidad que deseamos (también representa una imagen como un arreglo de dos dimensiones), pero no tiene métodos para configurar o tomar un píxel que usen un objeto `Color` (esta clase utiliza otros formatos que no queremos usar). Por este motivo construimos nuestra propia subclase que agrega estos dos métodos.

En este proyecto, puede considerar a `ImagenOF` como si fuera una clase de la biblioteca ya que no es necesario modificarla.

Los métodos de `ImagenOF` más importantes para nosotros son:

- `getPixel` y `setPixel` que permiten leer y modificar cada píxel individualmente.
- `getHeight` y `getWidth` cuya función es descubrir el tamaño de la imagen.

La clase `AdministradorDeArchivos` ofrece tres métodos: uno para leer desde el disco un archivo de imagen con nombre y devolverlo como un `ImagenOF`, uno para grabar un archivo `ImagenOF` en el disco y otro para abrir una caja de diálogo de selección de archivos que permite que el usuario seleccione la imagen que desea ver. Los métodos pueden leer archivos en los formatos estándares JPEG y PNG y el método de grabación sólo graba en formato JPEG. Esto se logra usando los métodos de entrada y salida de imágenes estándares de Java que se encuentran en la clase `ImageIO` (del paquete `javax.imageio`).

La clase `PanelDeImagen` implementa un componente Swing personalizado para mostrar nuestra imagen. Los componentes Swing personalizados pueden crearse fácilmente escribiéndolos como una subclase de algún componente existente y, como tal, puede insertarse en un contenedor Swing y mostrarse en nuestra IGU como cualquier otro componente Swing. `PanelDeImagen` es una subclase de `JComponent`.

Otro punto importante a tener en cuenta es que `PanelDeImagen` posee un método `setImagen` que tiene un parámetro `ImagenOF` para mostrar en pantalla cualquier `ImagenOF` que se le pase.

## 11.5.2 Agregar la imagen

Ahora que tenemos las clases preparadas para operar con las imágenes, es fácil agregar la imagen en la interfaz de usuario. El Código 11.3 muestra las diferencias importantes respecto de las versiones anteriores.

### Código 11.3

La clase  
`VisorDeImagen` con  
un `PanelDeImagen`

```
public class VisorDeImagen
{
    private JFrame ventana;
    private PanelDeImagen panelDeImagen;

    // Se omite el constructor y el método salir

    /**
     * Función Abrir: abre un selector de archivos para
     * elegir un nuevo
```

**Código 11.3  
(continuación)**

La clase  
VisorDeImagen con  
un PanelDeImagen

```

        * archivo de imagen.
    */
private void archivoAbrir()
{
    ImagenOF imagen =
AdministradorDeArchivos.getImagen();
    panelDeImagen.setImagen(imagen);
    ventana.pack();
}

/**
 * Crea la ventana Swing y su contenido.
 */
private void construirVentana()
{
    ventana = new JFrame("Visor de Imágenes");
    construirBarraDeMenu(ventana);

    Container panelContenedor =
ventana.getContentPane();

    panelDeImagen = new PanelDeImagen();
    panelContenedor.add(panelDeImagen);
    // terminó la construcción - acomoda los
componentes y los muestra
    ventana.pack();
    ventana.setVisible(true);
}

// Se omite el método construirBarraDeMenu
}

```

Cuando comparamos este código con el de la versión anterior, notamos que sólo hay dos pequeños cambios:

- En el método `construirVentana`, creamos y agregamos un componente `PanelDeImagen` en lugar de un `JLabel`. Agregar un panel no es más complicado que agregar una etiqueta. El objeto `PanelDeImagen` se almacena como un campo de instancia de modo que, más tarde, podamos acceder nuevamente a él.
- Nuestro método `archivoAbrir` se modificó para que realmente abra y muestre un archivo de imagen. Esto resulta fácil ahora que usamos nuestras clases de procesamiento de imágenes. La clase `AdministradorDeArchivos` tiene un método para seleccionar y abrir una imagen y el objeto `PanelDeImagen` tiene un método para mostrar dicha imagen. Algo que queremos destacar es que necesitamos invocar a `ventana.pack()` en el final del método `archivoAbrir` pues se modificó el tamaño de nuestro componente para mostrar la imagen. El método `pack` recalculará la disposición de los componentes en la ventana y dibujará nuevamente la ventana, por lo tanto, el cambio de tamaño se maneja adecuadamente.

**Ejercicio 11.14** Abra y pruebe el proyecto *visor-de-imagen-0-4*. La carpeta de los proyectos de este capítulo incluye también una carpeta con imágenes. En este lugar puede encontrar algunas imágenes de prueba que puede usar, aunque también puede usar sus propias imágenes.

**Ejercicio 11.15** ¿Qué ocurre cuando abre una imagen y luego cambia el tamaño de la ventana? ¿Qué ocurre si primero cambia el tamaño de la ventana y luego abre una imagen?

En esta versión hemos resuelto la parte central de la tarea: podemos abrir un archivo de imagen desde el disco y mostrarlo en la pantalla. Sin embargo, antes de denominar a nuestro proyecto «Versión 1.0» y declararlo como terminado por primera vez, queremos agregar algunas pequeñas mejoras (véase Figura 11.2):

- Queremos agregar dos etiquetas: una en la parte superior de la imagen, para mostrar el nombre del archivo de imagen y otra en la parte inferior, para mostrar un texto que indique el estado.
- Queremos agregar un menú *Filtro* que contenga algunos filtros que modifiquen la apariencia de la imagen.
- Queremos agregar un menú *Ayuda* que contenga un elemento *Acerca del Visor de Imágenes*. Al seleccionar este elemento del menú se mostrará una caja de diálogo con el nombre de la aplicación, el número de versión y la información sobre los autores.

### 11.5.3 Esquemas de disposición

Primeramente, trabajaremos en la tarea de agregar dos etiquetas en la interfaz: una ubicada en la parte superior de la imagen que se usa para mostrar el nombre del archivo de la imagen que se muestra actualmente y otra ubicada en la parte inferior que se usa para mostrar varios mensajes de estado.

La creación de estas etiquetas es fácil: ambas son simples instancias de `JLabel`. Las almacenamos en campos de instancia de modo que podamos acceder a ellas más tarde para cambiar el texto que muestran. La única cuestión que nos falta resolver es cómo acomodarlas en la pantalla.

Un primer intento (simplificado e incorrecto) podría ser este:

```
Container panelContenedor = ventana.getContentPane();
etiquetaNombreDeArchivo = new JLabel();
panelContenedor.add(etiquetaNombreDeArchivo);

panelDeImagen = new PanelDeImagen();
panelContenedor.add(panelDeImagen);

etiquetaEstado = new JLabel("Versión 1.0");
panelContenedor.add(etiquetaEstado);
```

La idea de este código es simple: tomamos el panel contenedor de la ventana y agregamos uno tras otro, los tres componentes que queremos mostrar. El único problema es que no hemos especificado exactamente cómo se ubicarán estos tres componentes. Podríamos querer que aparezcan uno cerca del otro, o uno debajo del otro, o alguna otra disposición posible. Como no hemos especificado ninguna disposición en especial, el contenedor (el panel contenedor) utiliza un comportamiento por defecto, y esto no es lo que queremos.

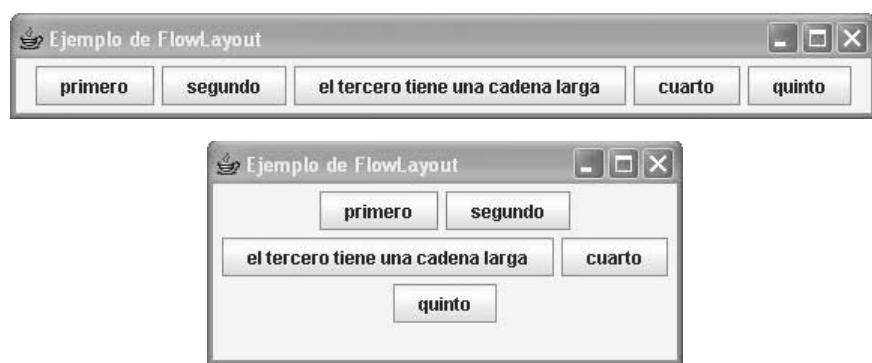
**Ejercicio 11.16** Continuando con su última versión del proyecto, use el fragmento de código que se muestra arriba para agregar las dos etiquetas. Pruebelo. ¿Qué observa?

Swing usa *gestores de disposición* para acomodar los componentes en una IGU. Cada contenedor que contiene componentes, por ejemplo, un panel, tiene un gestor de disposición asociado que se encarga de acomodar los componentes dentro del contenedor.

Swing proporciona varios gestores de disposición diferentes que soportan las diferentes preferencias de ubicación de los componentes. Los esquemas de disposición más importantes son: *FlowLayout*, *BorderLayout*, *GridLayout* y *BoxLayout*, cada uno de los cuales está representado por una clase Java en la biblioteca Swing y cada uno de los mismos dispone los componentes que tiene bajo su control de diferentes maneras.

Damos a continuación una breve descripción de cada esquema.

**Figura 11.5**  
FlowLayout



El gestor de disposición *FlowLayout* (Figura 11.5) acomoda todos los componentes secuencialmente, de izquierda a derecha. Deja cada componente en su tamaño preferido y los centra horizontalmente. Si el espacio horizontal no es suficiente para ajustar todos los componentes, los ubica en una segunda línea. También se puede configurar el esquema *FlowLayout* para alinear los componentes a la izquierda o a la derecha.

**Figura 11.6**  
BorderLayout



El BorderLayout (Figura 11.6) ubica cinco componentes con una disposición específica: uno en el centro y cada uno de los restantes en la parte superior, en la parte inferior, a la izquierda y a la derecha. Cada una de estas posiciones puede quedar vacía de modo que podría contener menos de cinco componentes. Los nombres de las cinco posiciones son: CENTRO, NORTE, SUR, ESTE y OESTE.

Este esquema podría parecer, en principio, muy especializado y uno podría preguntarse con qué frecuencia se necesita, pero en la práctica, es sorprendentemente útil y se usa en muchas aplicaciones. En BlueJ, por ejemplo, tanto la ventana principal como la ventana del editor usan BorderLayout como el principal gestor de disposición.

Cuando se cambia el tamaño de un BorderLayout, el componente central es el único que se modifica en ambas dimensiones. Los componentes ubicados al este y al oeste cambian su alto, pero no su ancho. Los componentes ubicados al norte y al sur mantienen su alto y sólo modifican su ancho.

**Figura 11.7**

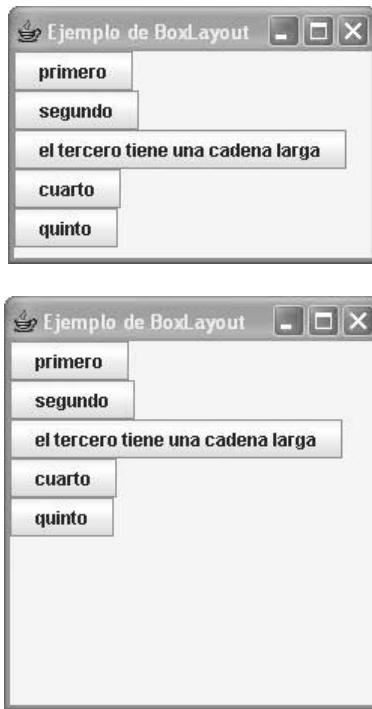
GridLayout



El esquema GridLayout (Figura 11.7), tal como su nombre sugiere, es muy útil para ubicar componentes en una grilla. Se puede especificar el número de filas y de columnas y el gestor de disposición GridLayout mantendrá siempre todos los componentes con el mismo tamaño. Puede ser útil por ejemplo, para forzar a que los botones tengan el mismo ancho. El ancho de las instancias de JButton se determina inicialmente mediante el texto del botón: cada botón se construye suficientemente ancho como para mostrar su texto completo. La inserción de botones en un GridLayout dará por resultado que todos los botones cambiarán de tamaño para que coincidan con el del botón más ancho.

BoxLayout ubica varios componentes vertical y horizontalmente. No arma otra línea cuando cambia el tamaño de los componentes (Figura 11.8). Mediante el anidado de varios esquemas BoxLayout, es decir, colocar uno dentro del otro, se pueden construir disposiciones de componentes en dos dimensiones, sofisticadas y alineadas.

**Figura 11.8**  
BoxLayout



#### 11.5.4 Contenedores anidados

Todas las estrategias de disposición de componentes discutidas previamente son sumamente simples. La clave para construir interfaces que tengan un buen aspecto y un buen comportamiento reside en un último detalle: se deben anidar los esquemas de disposición.

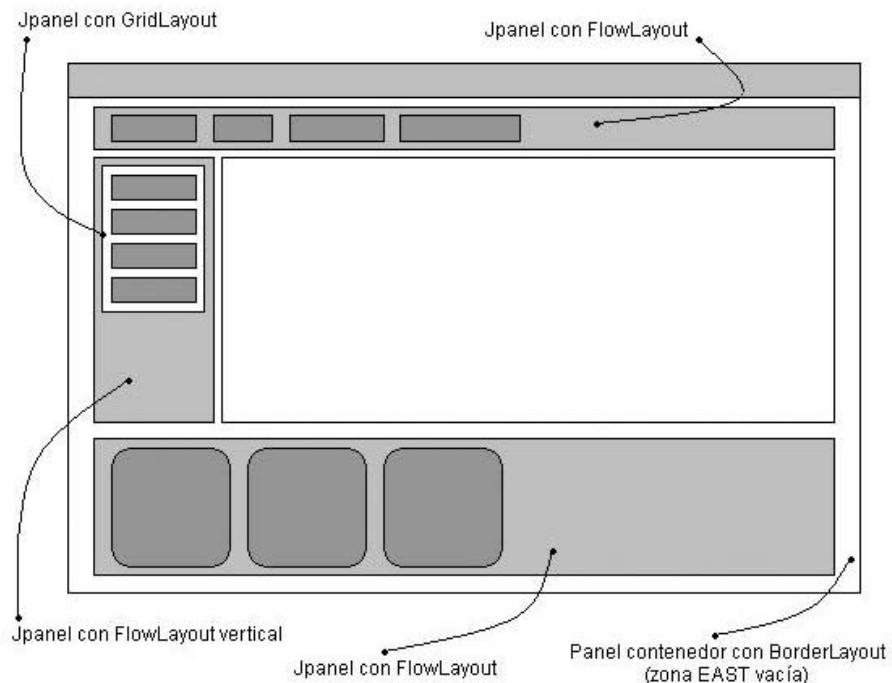
Algunos de los componentes Swing son *contenedores*. Desde afuera, los contenedores se presentan como si fueran componentes simples, pero pueden contener muchos otros componentes. Cada contenedor tiene asociado su propio gestor de disposición.

El contenedor que más se usa es el de la clase JPanel. Se puede insertar un JPanel en el panel contenedor de una ventana de la misma manera que un componente y luego, se pueden colocar más componentes dentro del JPanel. La Figura 11.9, por ejemplo, muestra una interfaz similar a la ventana principal de BlueJ. El panel contenedor de esta ventana usa el esquema BorderLayout, en el que no se utiliza la posición EAST. El área NORTH de este BorderLayout contiene un JPanel con un esquema FlowLayout horizontal que dispone sus componentes en una fila (podrían ser botones de una barra de herramientas). El área SOUTH es similar: otro JPanel con un FlowLayout asociado.

El grupo de botones de la zona WEST se ubicó primeramente en un JPanel con un GridLayout de una sola columna, para que todos los botones tengan el mismo tamaño. Luego, este JPanel se colocó dentro de otro JPanel con un FlowLayout vertical, de modo que la grilla no se extienda por encima del alto total de la zona WEST. Este JPanel exterior se insertó luego dentro del área WEST de la ventana.

**Figura 11.9**

Construcción de una interfaz mediante contenedores anidados



Observe la manera en que colaboran el contenedor y el gestor de disposición en la ubicación de los componentes. El contenedor contiene a los componentes, pero el gestor de disposición decide su ubicación exacta en la pantalla. Cada contenedor tiene un gestor de disposición que usa un esquema por defecto si es que no establecemos alguno explícitamente. El esquema por defecto es diferente para los diferentes contenedores: por ejemplo, el panel contenedor de un JFrame tiene asociado por defecto un BorderLayout mientras que un JPanel usa por defecto un FlowLayout.

**Ejercicio 11.17** Observe la IGU del proyecto calculadora que usamos en el Capítulo 6 (Figura 6.7 en página 188). ¿Qué tipo de contenedores y de gestores de disposición cree que se usaron? Después de responder por escrito, abra el proyecto *calculadora-gui* y controle su respuesta leyendo el código.

**Ejercicio 11.18** ¿Qué tipo de gestores de disposición habría que usar para crear el esquema de la ventana del editor de BlueJ?

**Ejercicio 11.19** En BlueJ, seleccione la función *Use Library Class* del menú *Tools*. Observe el diálogo que aparece en la pantalla. ¿Qué tipos de contenedores y gestores de disposición deben usarse para crear esta caja de diálogo? Para obtener información adicional, cambie el tamaño de la caja de diálogo y observe el comportamiento que presenta ante este cambio.

Es hora de ver nuevamente algo de código de la clase *VisorDeImagen* de nuestra aplicación. Nuestro objetivo es muy simple. Queremos visualizar tres componentes, uno debajo del otro: una etiqueta en la parte superior, la imagen en el medio y otra etiqueta en la parte inferior. Varios gestores de disposición pueden lograr este efecto. La decisión de cuál debemos elegir se aclara si pensamos en el comportamiento que tendrán los componentes ante el cambio de tamaño de la ventana. Cuando agrandemos la

ventana, querremos que las etiquetas mantengan su alto y que la imagen reciba todo el espacio restante. Esta descripción sugiere el uso de BorderLayout: las etiquetas pueden estar en las zonas NORTH y SOUTH y la imagen en la zona CENTER. El Código 11.4 muestra el código necesario para implementar esta distribución.

Hay dos detalles importantes que observar. Primero, el método `setLayout` se utiliza sobre el panel contenedor para establecer el gestor de disposición que se pretende usar<sup>4</sup>. El gestor de disposición es en sí mismo un objeto, de modo que creamos una instancia de `BorderLayout` y se la pasamos al método `setLayout`.

Segundo, cuando agregamos un componente en un contenedor con un `BorderLayout`, usamos un método `add` diferente, que tiene un segundo parámetro. El valor del segundo parámetro es una de las constantes públicas `NORTH`, `SOUTH`, `EAST`, `WEST` o `CENTER`, que están definidas en la clase `BorderLayout`.

#### Código 11.4

Uso de  
BorderLayout para  
acomodar los  
componentes

```
Container panelContenedor = ventana.getContentPane();
panelContenedor.setLayout(new BorderLayout());
etiquetaNombreDeArchivo = new JLabel();
panelContenedor.add(etiquetaNombreDeArchivo,
BorderLayout.NORTH);
panelDeImagen = new PanelDeImagen();
panelContenedor.add(panelDeImagen, BorderLayout.CENTER);
etiquetaEstado = new JLabel("Versión 1.0");
panelContenedor.add(etiquetaEstado, BorderLayout.SOUTH);
```

**Ejercicio 11.20** Implemente y pruebe el código que se muestra arriba en su versión del proyecto.

**Ejercicio 11.21** Experimente con otros gestores de disposición. Pruebe en su proyecto todos los gestores de disposición mencionados anteriormente y también pruebe si se comportan como se espera.

#### 11.5.5 Filtros de imagen

Aún nos resta hacer dos cosas antes de terminar nuestra primera versión del visor de imágenes: agregar algunos filtros de imagen y agregar el menú *Ayuda*. A continuación construiremos los filtros.

Los filtros son los primeros pasos en el procesamiento de imágenes. Eventualmente, no sólo queremos poder abrir y mostrar imágenes sino que también queremos ser capaces de procesarlas y grabarlas nuevamente en el disco.

Comenzaremos por agregar tres filtros simples. Un filtro es una función que se aplica a la imagen en su totalidad. (Aunque también se podría modificar el filtro para que se aplique a una parte de la imagen, pero no es lo que estamos haciendo.)

<sup>4</sup> Hablando estrictamente, la invocación a `setLayout` no es necesaria aquí pues el gestor por defecto del panel contenedor ya es `BorderLayout`. Hemos incluido esta llamada por claridad y legibilidad.

Los nombres de los tres filtros son *oscuro*, *claro* y *umbral*. El filtro *oscuro* hace que toda la imagen se oscurezca y el filtro *claro*, la ilumina. El filtro *umbral* cambia los colores de la imagen por una escala de grises mediante algunos tonos de gris preestablecidos. Elegimos implementar un filtro umbral de tres niveles, es decir, usaremos tres colores: blanco, negro y gris mediano. Todos los píxeles cuyos valores de brillo estén en el rango superior se volverán blancos, los que estén en el rango inferior se volverán negros y los del medio serán grises.

Para llevar a cabo esta tarea tenemos que hacer dos cosas:

- tenemos que crear dos elementos de menú, uno para cada filtro y cada uno asociado con un oyente del menú, y
- tenemos que implementar la operación del filtro actual.

Empezamos por los menús ya que no hay nada realmente nuevo en esta tarea. Es más de lo mismo, en cuanto al código de creación de menús que ya hemos escrito para los menús existentes.

Necesitamos agregar las siguientes partes:

- Creamos un nuevo menú (clase `JMenu`) de nombre *Filtro* y lo agregamos a la barra de menú.
- Creamos tres elementos de menú (clase `JMenuItem`) de nombres *oscuro*, *claro* y *umbral* y los agregamos a nuestro menú *Filtro*.
- Agregamos un oyente de acción para cada elemento del nuevo menú, usando los modismos de código relacionados con las clases anónimas que discutimos para los otros elementos del menú. Los oyentes de acción deberán invocar a los métodos `aplicarOscuro()`, `aplicarClaro()` y `aplicarUmbral()` respectivamente.

**Ejercicio 11.22** Agregue el nuevo menú y los elementos del menú en su versión del proyecto *visor-de-imagen-0-4* tal como se describió aquí. Con el fin de agregar los oyentes de acción, necesita crear los tres métodos privados `aplicarOscuro()`, `aplicarClaro()` y `aplicarUmbral()` en su clase *VisorDeImagen*. Estos métodos tendrán, inicialmente, cuerpos vacíos o simplemente pueden imprimir en pantalla algún texto que indique que han sido invocados.

Luego de haber agregado los menús y de haber creado los métodos (inicialmente vacíos) para manejar las funciones de los filtros, necesitamos implementar cada filtro.

Los tipos más simples de filtros incluyen el recorrido de una imagen y la realización de algún cambio del color de cada píxel. En el Código 11.5 se muestra un esquema de este proceso. Los filtros más complicados podrían usar los valores de los píxeles vecinos para ajustar el valor de un píxel.

#### Código 11.5

Esquema de un  
proceso de filtrado  
simple

```
int alto = getHeight();
int ancho = getWidth();
for(int y = 0; y < alto; y++){
    for(int x = 0; x < ancho; x++){
        Color pixel = getPixel(x,y);
        alterar el valor del color del pixel;
        setPixel(x, y, pixel);
    }
}
```

La función filtro opera sobre la imagen propiamente dicha, por lo tanto, siguiendo las pautas del diseño dirigido por responsabilidades, debe ser implementada en la clase `ImagenOF`. Por otro lado, el manejo de la invocación al menú también incluye código relacionado con la IGU (por ejemplo, cuando invocamos al filtro tenemos que controlar si existe una imagen abierta) que pertenece a la clase `VisorDeImagen`.

Como resultado de este razonamiento creamos dos métodos, uno en `VisorDeImagen` y otro en `ImagenOF` para compartir el trabajo (Código 11.6 y Código 11.7). Podemos ver que el método `aplicarOscuro` de `VisorDeImagen` contiene la parte de la tarea relacionada con la IGU (controlar que tenemos una imagen cargada, mostrar un mensaje de estado, repintar la ventana) mientras que el método oscuro de `ImagenOF` incluye el trabajo real de hacer que cada píxel de la imagen sea un poco más oscuro.

#### Código 11.6

El método del filtro en la clase `VisorDeImagen`

```
public class VisorDeImagen
{
    // se omiten campos, constructores y todos los métodos restantes

    /**
     * Función "Oscuro": oscurece la imagen
     */
    private void aplicarOscuro()
    {
        if(imagenActual != null) {
            imagenActual.oscuro();
            ventana.repaint();
            mostrarEstado("Filtro aplicado: Oscuro");
        }
        else {
            mostrarEstado("No hay ninguna imagen cargada");
        }
    }
}
```

**Ejercicio 11.23** ¿Qué hace la llamada a método `ventana.repaint()`, que se puede ver en el método `aplicarOscuro`?

**Ejercicio 11.24** Podemos ver una llamada al método `mostrarEstado` que es, claramente, una llamada a un método interno. A partir del nombre podemos suponer que este método debe mostrar un mensaje de estado usando la etiqueta de estado que hemos creado anteriormente. Implemente este método en su versión del proyecto `visor-de-imagen-0-4`. (*Pista:* busque el método `setText` en la clase `JLabel`.)

**Ejercicio 11.25** ¿Qué ocurre cuando se selecciona el elemento *Oscuro* del menú si no hay ninguna imagen cargada?

**Ejercicio 11.26** Explique detalladamente cómo funciona el método oscuro de `ImagenOF`. (*Pista:* contiene una llamada a otro método de nombre `darker`. ¿A qué clase pertenece este método? Investigue.)

**Código 11.7**

Implementación de un filtro en la clase `ImagenOF`

```
public class ImagenOF extends BufferedImage
{
    // se omiten campos, constructores y todos los restantes métodos

    /**
     * Oscurece un poco esta imagen
     */
    private void oscuro()
    {
        int alto = getHeight();
        int ancho = getWidth();
        for(int y = 0; y < alto; y++) {
            for(int x = 0; x < ancho; x++){
                setPixel(x, y, getPixel(x,y).darker());
            }
        }
    }
}
```

**Ejercicio 11.27** Implemente el filtro *Claro* en la clase `ImagenOF`.

**Ejercicio 11.28** Implemente el filtro *Umbral*. Para determinar el brillo de un píxel puede obtener sus valores de rojo, verde y azul y promediarlos. La clase `Color` define referencias estáticas que se ajustan a objetos de color negro, blanco y gris.

Puede encontrar una implementación de todo lo descrito anteriormente y que funciona, en el proyecto *visor-de-imagen-1-0*. Sin embargo, debería intentar primero hacer los ejercicios por su propia cuenta antes de ver la solución.

## 11.5.6 Diálogos

Nuestra última tarea para esta versión es agregar un menú *Ayuda* que contenga un elemento con la etiqueta *Acerca del Visor de Imágenes...*. Cuando se seleccione este elemento se desplegará una caja de diálogo con información sobre la aplicación.

**Ejercicio 11.29** Agregue nuevamente un menú *Ayuda* y un elemento en este menú con la etiqueta *Acerca del Visor de Imágenes...*

**Ejercicio 11.30** Agregue un método con su cuerpo vacío, de nombre `mostrarAcercaDe()` y agregue un oyente de acción para el elemento del menú *Acerca del Visor de Imágenes...* que invoque a este método.

Ahora tenemos que implementar el método `mostrarAcercaDe` de modo que muestre un diálogo del estilo «acerca de».

Una de las principales características de un diálogo es si es *modal* o no. Un diálogo modal bloquea todas las interacciones con las restantes partes de una aplicación hasta que se cierre. Esto obliga al usuario a que trate primero con el diálogo. Los diálogos no modales permiten la interacción con otras ventanas mientras están visibles.

Los diálogos se pueden implementar de manera similar a nuestro `JFrame` principal, aunque para mostrar la ventana usan con frecuencia la clase `JDialog`.

Sin embargo, para los diálogos modales con una estructura estándar, existen algunos métodos convenientes en la clase `JOptionPane` que facilitan mucho el trabajo de mostrar estos tipos de diálogos. `JOptionPane` tiene, entre otras cosas, métodos estáticos para mostrar tres tipos estándar de diálogos que son:

- *Diálogo de mensaje*: es un diálogo que muestra un mensaje y que tiene un botón `OK` para cerrar el diálogo.
- *Diálogo de confirmación*: este diálogo, generalmente, permite hacer preguntas al usuario y posee botones que el usuario puede utilizar para responder, por ejemplo: *Sí*, *No* y *Cancelar*.
- *Diálogo de entrada*: este diálogo incluye un campo de texto para que el usuario escriba algún texto.

Nuestra caja «acerca de» es un simple diálogo de mensaje. Buscando en la documentación de `JOptionPane` encontramos que existen métodos estáticos de nombre `showMessageDialog` para realizar esta tarea.

**Ejercicio 11.31** Busque la documentación de `showMessageDialog`. ¿Cuántos métodos hay con este nombre? ¿Cuáles son las diferencias entre ellos? ¿Cuál podríamos usar? ¿Por qué?

**Ejercicio 11.32** Implemente el método `mostrarAcercaDe` en su clase `VisorDeImagen` usando una invocación a un método `showMessageDialog`.

**Ejercicio 11.33** Los métodos `showInputDialog` del `JOptionPane` permiten solicitar al usuario el ingreso de algún dato, cuando se requiera. Por otra parte, el componente `JTextField` permite mostrar una zona permanente para el ingreso de texto en una IGU. Busque la documentación de esta clase. ¿Qué ingresos provocan que se notifique un `ActionListener` asociado con un `JTextField`? ¿Se puede impedir que el usuario edite el texto del campo? ¿Es posible que un oyente se notifique de cambios arbitrarios del campo de texto? (*Pista: ¿qué uso hace un `JTextField` de un objeto `Document`?*)

Puede encontrar un ejemplo de un `JTextField` en el proyecto `calculadora` del Capítulo 6.

Después de estudiar la documentación, podemos implementar nuestra caja «acerca de» mediante una llamada al método `showMessageDialog`. El código correspondiente se muestra en el Código 11.9. Observe que hemos introducido una constante de cadena de nombre `VERSION` que contiene el número de la versión actual.

#### Código 11.8

Mostrar un diálogo modal

```
private void mostrarAcercaDe()
{
    JOptionPane.showMessageDialog(ventana,
        "Visor de Imágenes\n" + VERSION,
        "Acerca del Visor de Imágenes",
        JOptionPane.INFORMATION_MESSAGE);
}
```

Esta fue la última tarea que debíamos hacer para completar la versión 1.0 de nuestra aplicación para visualizar imágenes. Si ya hizo todos los ejercicios, ahora tendrá una nueva versión del proyecto que puede abrir imágenes, aplicar filtros, mostrar mensajes de estado y mostrar un diálogo.

El proyecto *visor-de-imagen-1-0*, incluido en los proyectos de este libro, contiene una implementación de toda la funcionalidad discutida hasta ahora. Podría estudiar cuidadosamente este proyecto y compararlo con sus propias soluciones.

En este proyecto, también hemos mejorado el método `archivoAbrir` para incluir mejores notificaciones de error. Si el usuario selecciona un archivo que no es un archivo válido de imagen, ahora mostramos un mensaje de error adecuado. Esto resulta fácil de hacer ahora que conocemos los diálogos de mensaje.

## 11.6

## Visor de Imágenes 2.0: mejorar la estructura del programa

La versión 1.0 de nuestra aplicación tiene una IGU que se puede utilizar y que es capaz de mostrar imágenes en ella, también puede aplicar tres filtros básicos.

La siguiente idea obvia para mejorar nuestra aplicación es agregar algunos filtros más interesantes. Sin embargo, en lugar de hacerlo inmediatamente, vamos a pensar antes qué cosas involucra esta tarea.

Con la estructura actual de filtros, tenemos que hacer tres cosas para cada filtro:

1. agregar un elemento en el menú;
2. agregar un método que maneje la activación del menú en `VisorDeImagen` y
3. agregar una implementación del filtro en `ImagenOF`.

Los puntos 1 y 3 son inevitables, necesitamos un elemento en el menú y una implementación del filtro, pero el punto 2 es algo sospechoso. Si vemos estos métodos en la clase `VisorDeImagen` (el Código 11.9 muestra dos de ellos a modo de ejemplo), se presentan como una duplicación de gran cantidad de código. Estos métodos son esencialmente los mismos (excepto por algunos pequeños detalles) y lo que es peor, para cada nuevo filtro que queramos agregar, tenemos que agregar otro de estos métodos que es prácticamente igual a los anteriores.

### Código 11.9

Dos de los métodos que manejan filtros en `VisorDeImagen`

```
private void aplicarClaro()
{
    if (imagenActual != null) {
        imagenActual.claro();
        ventana.repaint();
        mostrarEstado("Filtro aplicado: Claro");
    }
    else {
        mostrarEstado("No hay ninguna imagen cargada");
    }
}
private void aplicarUmbral()
```

**Código 11.9**

(continuación)

Dos de los métodos que manejan filtros en VisorDeImagen

```

{
    if (imagenActual != null) {
        imagenActual.umbral();
        ventana.repaint();
        mostrarEstado("Filtro aplicado: Umbral");
    }
    else {
        mostrarEstado("No hay ninguna imagen cargada");
    }
}

```

Como sabemos, la duplicación de código es signo de un mal diseño y debemos evitarlo. Resolvemos el problema de la duplicación mediante la refactorización de nuestro código.

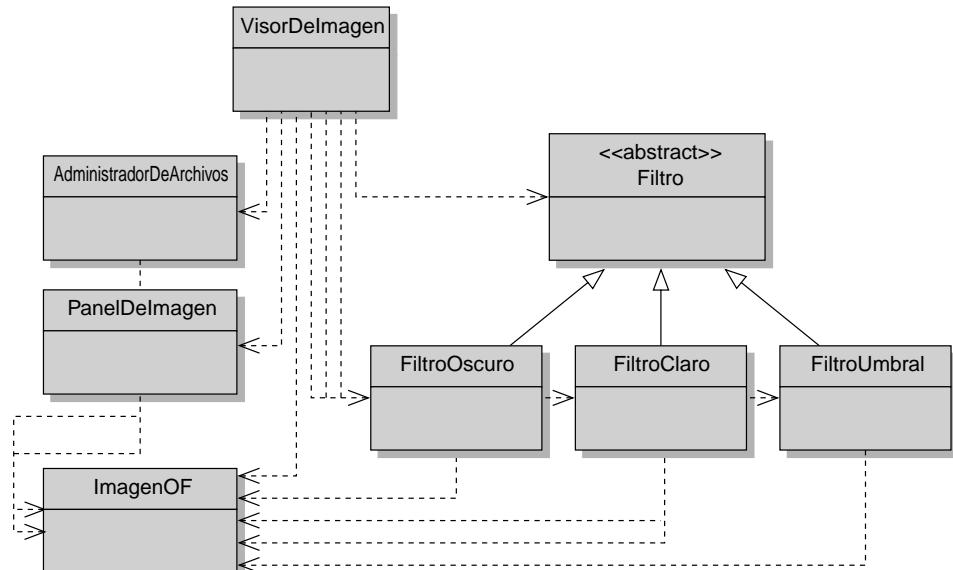
En este caso, queremos encontrar un diseño que nos permita agregar nuevos filtros sin tener que agregar cada vez un nuevo método que despache el filtro.

Para lograr lo que queremos hacer, necesitamos evitar la codificación de cada filtro en nuestra clase *VisorDeImagen*. En su lugar, usaremos una colección de filtros y luego escribimos una sola invocación al método del filtro que busque e invoque al filtro correcto.

En vías de hacer esto, los filtros en sí mismos se convierten en objetos. Si queremos almacenarlos en una colección común y aplicar los filtros directamente a partir de esta colección, todos los filtros necesitan una superclase en común que nombramos *Filtro* (Figura 11.10).

**Figura 11.10**

Estructura de clases con los filtros como objetos



Cada filtro tendrá un nombre y un método aplicar que aplica el filtro a una imagen. Podemos definir esto en la superclase Filtro (Código 11.10). Observe que ésta es una clase abstracta pues el método aplicar, en este nivel, tiene que ser abstracto, pero el método getNombre se puede implementar por completo.

### Código 11.10

Clase abstracta  
Filtro: la  
superclase para todos  
los filtros

```
public abstract class Filtro
{
    private String nombre;
    /**
     * Crea un nuevo filtro con un nombre determinado.
     */
    public Filtro(String nombre)
    {
        this.nombre = nombre;
    }

    /**
     * Devuelve el nombre de este filtro.
     *
     * @return El nombre de este filtro.
     */
    public String getNombre()
    {
        return nombre;
    }

    /**
     * Aplica este filtro a una imagen.
     *
     * @param imagen La imagen que cambiará mediante
     * este filtro.
     */
    public abstract void aplicar(ImagenOF imagen);
}
```

Una vez que tenemos la superclase escrita, no resulta difícil implementar filtros específicos como subclases. Todo lo que necesitamos hacer es proveer una implementación para el método aplicar que procese una imagen (pasada como parámetro) usando sus métodos getPixel y setPixel. El Código 11.11 muestra un ejemplo.

### Código 11.11

Implementación de  
una clase de filtro  
específica

```
// Se omiten todos los comentarios
public class FiltroOscuro extends Filtro{
    public FiltroOscuro(String nombre)
    {
        super(nombre);
    }
```

**Código 11.11  
(continuación)**

Implementación de una clase de filtro específica

```
public void aplicar(ImagenOF imagen)
{
    int alto = imagen.getHeight();
    int ancho = imagen.getWidth();
    for(int y = 0; y < alto; y++) {
        for(int x = 0; x < ancho; x++) {
            imagen.setPixel(x, y,
                imagen.getPixel(x, y).darker());
        }
    }
}
```

Un efecto colateral de esta refactorización es que la clase `ImagenOF` se vuelve mucho más simple ya que se pueden eliminar todos los métodos de los filtros. Ahora define solamente los métodos `setPixel` y `getPixel`.

Una vez que hemos definido nuestros filtros, podemos crear objetos filtro y almacenarlos en una colección (Código 11.12).

**Código 11.12**

Agregar una colección de filtros

```
public class VisorDeImagen
{
    // Se omiten los restantes campos

    private List<Filtro> filtros;

    public VisorDeImagen()
    {
        filtros = crearFiltros();
        ...
    }
    private List<Filtro> crearFiltros()
    {
        List<Filtro> listaDeFiltros = new ArrayList<Filtro>();
        listaDeFiltros.add(new FiltroOscuro("Oscuro"));
        listaDeFiltros.add(new FiltroClaro("Claro"));
        listaDeFiltros.add(new FiltroGris("Umbral"));

        return listaDeFiltros;
    }

    // Se omiten los restantes métodos
}
```

Una vez que tenemos esta estructura, podemos hacer los últimos dos cambios necesarios:

- Cambiamos el código que crea los elementos del menú para los filtros de modo que recorra la colección de filtros. Para cada filtro, se crea un elemento de menú y se usa el método `getNombre` para determinar la etiqueta del elemento correspondiente.
- Una vez que tenemos este código, podemos escribir un método genérico `aplicarFiltro` que recibe un filtro como parámetro y lo aplica sobre la imagen actual.

El proyecto *visor-de-imagen-2-0* incluye la implementación completa de estos cambios.

**Ejercicio 11.34** Abra el proyecto *visor-de-imagen-2-0*. Estudie el código del nuevo método para crear y aplicar filtros en la clase `VisorDeImagen`. Preste especial atención a los métodos `construirBarraDeMenu` y `aplicarFiltro`. Explique detalladamente cómo funciona la creación de los elementos del menú para los filtros y su respectiva activación. Dibuje un diagrama de objetos para los filtros. Observe en particular, que la variable `filtro` en `construirBarraDeMenu` se ha declarado `final`, tal como lo hemos mencionado en la Sección 11.4.7. Asegúrese de que comprende el motivo de esta declaración.

**Ejercicio 11.35** ¿Qué necesita cambiar para agregar un nuevo filtro en su visor de imágenes?

En esta sección hemos realizado un proceso de refactorización pura. No hemos cambiado la funcionalidad de la aplicación para nada, pero hemos trabajado exclusivamente en mejorar la estructura de la implementación de modo que los cambios futuros resulten más fáciles de hacer.

Ahora, luego de terminar con la refactorización, debemos probar que toda la funcionalidad existente todavía funciona como se espera.

En todos los desarrollos de proyectos necesitamos fases como ésta. No siempre realizamos decisiones de diseño perfectas desde el comienzo y las aplicaciones crecen y cambian sus requerimientos. Aunque nuestra principal tarea en este capítulo es trabajar con las IGU, necesitamos volver un paso atrás y refactorizar nuestro código antes de proceder. Este trabajo se compensará, a lo largo del camino, facilitando los futuros cambios.

Algunas veces resulta tentador dejar las estructuras tal como están, sin embargo reconocemos que no es bueno. Colocar un poco de código duplicado puede ser más fácil en el corto plazo que hacer una cuidadosa refactorización, pero en los proyectos que pretenden sobrevivir por un tiempo largo, esto está ligado a crear problemas. Como regla general: ¡tómese su tiempo, mantenga su código prolífico!

Ahora que ya hemos refactorizado nuestra aplicación, estamos listos para agregar más filtros.

**Ejercicio 11.36** Agregue un filtro *escala de grises* a su proyecto. El filtro convierte una imagen de color en una imagen en blanco y negro, formada por tonos de grises. Puede hacer que cada píxel tome un tono de gris asignando el mismo valor a los tres componentes del color (rojo, verde y azul). El brillo de cada píxel debiera permanecer sin cambios.

**Ejercicio 11.37** Agregue un filtro *espejo* que invierte horizontalmente la imagen. El píxel del extremo izquierdo se moverá al extremo derecho y viceversa, produciendo el efecto de ver la imagen reflejada en un espejo.

**Ejercicio 11.38** Agregue un filtro *invertir* que invierte cada color. «Invertir» un color significa reemplazar cada valor  $x$  del color por un valor  $255 - x$ .

**Ejercicio 11.39** Agregue un filtro *alisar* que «alisa la imagen». Un filtro alisar reemplaza cada valor del píxel por el promedio de los valores de sus píxeles vecinos incluyendo al propio píxel considerado (nueve píxeles en total). Debe ser muy cuidadoso con los bordes de la imagen donde pueden no existir algunos píxeles vecinos. También se debe asegurar de trabajar con una copia temporal de la imagen mientras la procesa ya que el resultado no es correcto si trabaja sobre una única imagen. (¿Por qué?) Puede obtener fácilmente una copia de la imagen creando un nuevo objeto `ImagenOF` pasando la imagen original a su constructor como parámetro.

**Ejercicio 11.40** Agregue un filtro *solarizar*. La solarización es un efecto que se puede crear manualmente sobre negativos fotográficos mediante la reexposición del negativo. Podemos simular este filtro reemplazando cada componente del color de cada píxel que tiene un valor  $v$  menor que 128 por el valor  $255 - v$ . Los componentes del brillo (de valor 128 o mayor) deben quedar sin cambios. (Este es un algoritmo de solarización muy sencillo, puede encontrar descripciones de algoritmos más sofisticados en la bibliografía específica.)

**Ejercicio 11.41** Implemente un filtro *detector de bordes*. Haga esto analizando los nueve píxeles de una cuadrícula de tres por tres alrededor del píxel (similar al filtro alisar) y luego asigne al valor del píxel del medio, la diferencia entre el mayor y el menor valor encontrado. Haga esto para cada componente del color (rojo, verde, azul). Produce un buen efecto si, al mismo tiempo, también invierte la imagen.

**Ejercicio 11.42** Experimente con sus filtros sobre diferentes imágenes. Trate de aplicar varios filtros, uno después del otro.

Una vez que haya implementado algunos otros filtros propios, deberá cambiar el número de versión de su proyecto para que pase a ser la «versión 2.1».

## 11.7

### Visor de Imágenes 3.0: más componentes de interfaz

Antes de dar por terminado el proyecto del visor de imágenes queremos agregar unas últimas mejoras y en el proceso, ver dos componentes IGU más: botones y bordes.

#### 11.7.1

##### Botones

Ahora queremos agregar funcionalidad al visor de imágenes para que permita cambiar el tamaño de la imagen. Lo hacemos proporcionando dos funciones: *agrandar*, que duplica el tamaño de la imagen y *achicar*, que lleva el tamaño de la imagen a su mitad. (Para ser exactos: duplicamos o achicamos tanto el alto como el ancho, pero no el área de la imagen.)

Una forma de proveer estas funciones es mediante la implementación de filtros pero decidimos no hacerlo de esta manera. Hasta ahora, los filtros nunca cambian el tamaño de la imagen y queremos dejarlo así. En lugar de filtros, introducimos una barra de herramientas a la izquierda de la ventana, con dos botones con las etiquetas *Agrandar* y *Achicar* (Figura 11.11). Este camino también nos da la oportunidad de experimentar un poco con botones, contenedores y gestores de disposición.

**Figura 11.11**

Visor de imágenes con una barra de herramientas con botones



Hasta ahora, nuestra ventana usa un *BorderLayout*, en donde la zona WEST está vacía. Podemos usar esta zona para agregar nuestros botones de la barra de herramientas. Sin embargo, hay un pequeño problema: la zona WEST de un *BorderLayout* puede tener sólo un componente, pero en este caso, tenemos dos botones.

La solución es simple. Agregamos un *JPanel* en el área WEST de la ventana (como ya sabemos, un *JPanel* es un contenedor) y luego pegamos los dos botones dentro de él. El Código 11.13 muestra el código necesario para hacer esta tarea.

**Código 11.13**

Agregar un panel para armar una barra de herramientas con dos botones

```
// Crear una barra de herramientas con botones
JPanel barraDeHerramientas = new JPanel();

botonAchicar = new JButton("Achicar");
barraDeHerramientas.add(botonAchicar);

botonAgrandar = new JButton("Agrandar");
barraDeHerramientas.add(botonAgrandar);
panelContenedor.add(barraDeHerramientas, BorderLayout.WEST);
```

**Ejercicio 11.43** Agregue, en su última versión del proyecto, dos botones con las etiquetas *Agrandar* y *Achicar* respectivamente, usando código similar al que mostramos. Pruébelo. ¿Qué observa?

Cuando probamos esta modificación, vemos que parcialmente funciona, pero no aparece de la manera en que esperábamos. El motivo es que un JPanel usa por defecto un FlowLayout y un FlowLayout dispone los componentes horizontalmente y nosotros queremos acomodarlos verticalmente.

Podemos lograrlo usando otro gestor de disposición: un GridLayout hace lo que queremos. Cuando se crea un GridLayout, los parámetros del constructor determinan cuántas filas y columnas queremos que tenga. Un valor cero tiene un significado especial, es interpretado como «tantas filas y columnas como sea necesario». Por lo tanto, podemos crear un GridLayout con una sola columna usando 0 como el número de filas y 1 como el número de columnas. Luego, podemos aplicar este GridLayout en nuestro JPanel usando el método `setLayout` del panel, inmediatamente después de crearlo.

```
JPanel barraDeHerramientas = new JPanel();
barraDeHerramientas.setLayout(new GridLayout(0, 1));
```

Alternativamente, el gestor de disposición también puede especificarse como un parámetro del constructor del contenedor:

```
JPanel barraDeHerramientas = new JPanel(new GridLayout(1, 0));
```

**Ejercicio 11.44** Cambie su código de modo que su panel para la barra de herramientas utilice un GridLayout, tal como lo discutimos en el párrafo anterior. Pruébelo. ¿Qué observa?

Si probamos todo este código, podemos ver que estamos más cerca de la solución, pero todavía no tenemos lo que queremos. Nuestros botones, ahora, son mucho más grandes de lo que pretendíamos.

El motivo es que, en el esquema BorderLayout, un contenedor (en este caso, nuestra barra de herramientas JPanel) siempre cubre el área completa en que está ubicado (el área WEST en nuestra ventana) y un GridLayout siempre modifica el tamaño de sus componentes hasta llenar la totalidad del contenedor.

Un FlowLayout no hace esto y queda muy bien dejar un poco de espacio alrededor de los componentes. Por lo tanto, nuestra solución es usar ambos gestores: el GridLayout para acomodar los botones en una columna y un FlowLayout para dejar un poco de espacio entre los botones. Terminamos teniendo un panel GridLayout dentro de un panel FlowLayout dentro de un BorderLayout. El Código 11.14 muestra esta solución. Las construcciones de este estilo son muy comunes. Frecuentemente deberá anidar varios contenedores dentro de otros contenedores para crear exactamente lo que desea ver.

#### Código 11.14

Uso de un contenedor GridLayout anidado dentro de un contenedor FlowLayout

```
// Crea una barra de herramientas con botones
JPanel barraDeHerramientas = new JPanel();
barraDeHerramientas.setLayout(new GridLayout(0, 1));
botonAchicar = new JButton("Achicar");
barraDeHerramientas.add(botonAchicar);
botonAgrandar = new JButton("Agrandar");
```

**Código 11.14  
(continuación)**

Uso de un contenedor GridLayout anidado dentro de un contenedor FlowLayout

```
barraDeHerramientas.add(botonAgrandar);  
  
// Agrega la barra en un panel con un FlowLayout para  
espaciar  
JPanel panelFlow = new JPanel();  
panelFlow.add(barraDeHerramientas);  
  
panelContenedor.add(panelFlow, BorderLayout.WEST);
```

Nuestros botones ahora aparecen muy próximos y esto es lo que pretendíamos. Antes de agregar los últimos retoques podemos trabajar para lograr que funcionen los botones.

Necesitamos agregar dos métodos de nombres, por ejemplo, agrandar y achicar que realicen efectivamente el trabajo y necesitamos agregar oyentes de acción para que los botones invoquen a estos métodos.

**Ejercicio 11.45** En su proyecto, agregue dos métodos de nombres agrandar y achicar. Inicialmente, coloque simplemente una sentencia `println` dentro de sus cuerpos para ver si los métodos son invocados. Los métodos deben ser privados.

**Ejercicio 11.46** Agregue oyentes de acción a los dos botones de modo que invoquen a los dos nuevos métodos. El agregar oyentes de acción a los botones es idéntico al agregar oyentes de acción a los elementos del menú. Esencialmente, puede copiar el código base desde allí. Pruébelo. Asegúrese de que los métodos agrandar y achicar se invocan al activar los botones.

**Ejercicio 11.47** Implemente adecuadamente los métodos agrandar y achicar. Para hacerlo tiene que crear una nueva `ImagenOF` con un tamaño diferente, copiar los píxeles de la imagen actual (mientras aumenta o reduce la escala) y luego asignar la nueva imagen a la imagen actual. Al final de su método deberá invocar al método `pack` de la `ventana` para reordenar los componentes con el tamaño modificado.

**Ejercicio 11.48** Todos los componentes Swing cuentan con un método `setEnabled(boolean)` que habilita o deshabilita el componente. Los componentes inhabilitados se muestran generalmente grisados y no reaccionan. Cambie su visor de imágenes de modo que los dos botones de la barra de herramientas estén inicialmente inhabilitados. Se habilitarán cuando se abra una imagen, y cuando se cierre, se deshabilitarán nuevamente.

## 11.7.2 Bordes

El último retoque que queremos dar a nuestra interfaz es agregar algunos bordes internos. Se pueden usar bordes para agrupar componentes o sólo para agregar espacio entre ellos. Cada componente Swing puede tener un borde.

Algunos gestores de disposición también aceptan parámetros en el constructor que definen sus espacios y luego, el gestor de disposición se encarga de crear el espacio requerido entre los componentes.

Los bordes más usados son `BevelBorder`, `CompoundBorder`, `EmptyBorder`, `EtchedBorder` y `TitledBorder`. En este caso, deberá familiarizarse con estos bordes por sus propios medios.

Podemos hacer tres cosas para mejorar el aspecto de nuestra IGU:

- agregar espacio alrededor de la parte exterior de la ventana;
- agregar espacio entre los componentes de la ventana y
- agregar una línea alrededor de la imagen.

El código necesario para hacer estas tres cosas se muestra en el Código 11.15. La llamada al método `setBorder` del panel contenedor con el parámetro `EmptyBorder` agrega espacio alrededor del borde exterior de la ventana. Observe que ahora convertimos el panelContenedor en un JPanel pues el supertipo Container no posee el método `setBorder`.

#### Código 11.15

Agregar espacio con huecos y bordes

```
JPanel panelContenedor = (JPanel)ventana.getContentPane();
panelContenedor.setBorder(new EmptyBorder(6, 6, 6, 6));

// Especifica el gestor de disposición con un buen espaciado
panelContenedor.setLayout(new BorderLayout(6, 6));
panelDeImagen = new PanelDeImagen();
panelDeImagen.setBorder(new EtchedBorder());
panelContenedor.add(panelDeImagen, BorderLayout.CENTER);
```

La creación del `BorderLayout` con dos parámetros de tipo entero agrega espacio entre los componentes que dispone. Y finalmente, el determinar un `EtchedBorder` para el `panelDeImagen` agrega una línea con apariencia de «grabado» alrededor de la imagen.

Los bordes se definen en el paquete `javax.swing.border`; tenemos que agregar e importar sentencias de este paquete.

Todas las mejoras discutidas en esta sección han sido implementadas en la última versión de esta aplicación en los proyectos de este libro: *visor-de-imagen-3-0*. En esta versión, también hemos agregado una función *Grabar Como* en el menú *Archivo* de modo que se puedan grabar las imágenes nuevamente en el disco.

Además, hemos agregado otro filtro bajo el nombre *Ojo de Pez* para darle algunas ideas adicionales sobre lo que se puede hacer. Pruébelo. Funciona especialmente bien cuando se aplica sobre retratos.

## 11.8

### Otras extensiones

La programación de interfaces gráficas de usuario mediante Swing es un tema bastante extenso. Swing ofrece varios tipos diferentes de componentes, varios contenedores diferentes y gestores de disposición, cada uno de los cuales posee varios atributos y métodos.

Familiarizarse con toda la biblioteca Swing lleva tiempo y no es algo que se pueda hacer en unas pocas semanas. Generalmente, mientras trabajamos con IGU continuamos

leyendo detalles que no conocíamos y con el tiempo, nos vamos convirtiendo en expertos.

El ejemplo discutido en este capítulo, pese a que contiene una gran cantidad de detalles, es sólo una breve introducción a la programación de IGU. Hemos tratado los conceptos más importantes pero todavía existe una gran cantidad de funcionalidad por descubrir, de la cual, la mayoría está fuera del alcance de este libro.

Existen variadas fuentes de información disponibles para que continúe leyendo sobre el tema. Tendrá que buscar con frecuencia en la documentación API de las clases Swing. No es posible trabajar sin ella.

También existen muchos tutoriales disponibles sobre programación de IGU y Swing, tanto impresos como la web.

Un buen punto de inicio es el Tutorial de Java, disponible en línea públicamente en el sitio de Sun Microsystems, que contiene una sección titulada *Creating a GUI with JFC/Swing*<sup>5</sup> (<http://java.sun.com/docs/books/tutorial/uiswing/index.html>) para crear interfaces gráficas de usuario utilizando la biblioteca Swing.

Esta sección tiene varios apartados interesantes, uno de los más útiles puede ser la sección *Using Swing Components*, y en ella, el apartado *How to* que contiene títulos tales como *How to Use Buttons, Check Boxes, and Radio Buttons; How to Use Labels; How to Make Dialogs; How to Use Panels*, etc.

De manera similar, la sección de más alto nivel *Laying Out Components Within a Container* también tiene una sección *How to* que trata sobre todos los gestores de disposición disponibles.

**Ejercicio 11.49** Busque la sección *Creating e GUI with JFC/Swing* del Tutorial de Java (en el sitio web, las secciones se denominan *trails*) y márquelo.

**Ejercicio 11.50** Escriba una lista de todos los gestores de disposición que existen en Swing.

**Ejercicio 11.51** ¿Qué es un *deslizador (slider)*? Busque una descripción y resúmala. Escriba un ejemplo breve en código Java para crear y usar un *deslizador*.

**Ejercicio 11.52** ¿Qué es un *panel tabulado (tabbed pane)*? Busque una descripción y resúmala. Dé ejemplos de posibles usos de un *panel tabulado*.

**Ejercicio 11.53** ¿Qué es un *cuadro de recorrido (spinner)*? Busque una descripción y resúmala.

**Ejercicio 11.54** Busque la aplicación de ejemplo *ProgressBarDemo* que utiliza una *barra de progreso*. Ejecútela en su computadora. Describa lo que hace.

Es aquí donde dejamos la discusión del ejemplo visor de imágenes aunque los lectores interesados pueden extender esta aplicación en varias direcciones. Mediante la información del tutorial en línea, se pueden agregar numerosos componentes de interfaz.

Los siguientes ejercicios aportan algunas ideas y obviamente, existen muchas otras posibilidades.

---

<sup>5</sup> N. del T. Existen algunas publicaciones en español en Internet, aunque no son oficiales de Sun. Una dirección en la que se puede encontrar este tutorial traducido es <http://www.programacion.com/tutorial/swing/>

**Ejercicio 11.55** Implemente la función *deshacer* en su visor de imágenes. Esta función revierte la última operación.

**Ejercicio 11.56** Deshabilite los elementos del menú que no debieran usarse cuando no se muestra ninguna imagen.

**Ejercicio 11.57** Implemente la función *recargar* que descarta todos los cambios de la imagen actual y carga nuevamente la imagen original desde el disco.

**Ejercicio 11.58** La clase `JMenu` es, actualmente, una subclase de `JMenuItem`. Esto quiere decir que los menús anidados se pueden crear ubicando un `JMenu` dentro de otro. Agregue un menú *Ajustar* en la barra de menú. Anide dentro de él un menú *Rotar* que permita que la imagen rote 90 o 180 grados, en sentido horario o en sentido antihorario. Implemente esta funcionalidad. El menú *Ajustar* también podría contener, por ejemplo, elementos de menú que invoquen a la funcionalidad que ya existe para agrandar y achicar las imágenes.

**Ejercicio 11.59** La aplicación siempre cambia el tamaño de la ventana para asegurar que se visualice la imagen completa. El hecho de tener una ventana grande no siempre es deseable. Lea la documentación de la clase `JScrollPane`. En lugar de agregar directamente el `PanelDeImagen` en el panel contenedor, ubique el panel en un `JScrollPane` y agréguelo al panel contenedor. Muestre una imagen grande y experimente con el cambio de tamaño de la ventana. ¿Qué diferencias presenta el hecho de tener un panel de desplazamiento? ¿Le permite mostrar imágenes que de otra manera serían demasiado grandes para la pantalla?

**Ejercicio 11.60** Modifique su aplicación de modo que se puedan abrir varias imágenes al mismo tiempo, pero que muestre una sola imagen por vez. Luego agregue un menú desplegable (usando la clase `JComboBox`) para seleccionar la imagen a mostrar.

**Ejercicio 11.61** Como una alternativa al uso de un `JComboBox`, tal como se hace en el Ejercicio 11.60, utilice un *panel tabulado* (clase `JTabbedPane`) que pueda contener varias imágenes abiertas.

**Ejercicio 11.62** Implemente una función para construir una presentación de diapositivas que permita seleccionar imágenes de una carpeta y luego muestre cada imagen durante una cierta cantidad de tiempo (por ejemplo, cinco segundos).

**Ejercicio 11.63** Una vez que tenga la presentación de diapositivas, agregue un *deslizador* (clase `JSlider`) para seleccionar una imagen de la presentación moviéndolo. Mientras se ejecuta la presentación, el *deslizador* deberá moverse para indicar su progreso.

## 11.9

### Otro ejemplo: reproductor de sonido

Hasta ahora, en este capítulo, hemos discutido detalladamente un ejemplo de la interfaz de usuario de una aplicación. Ahora queremos introducir una segunda aplicación para aportar otro ejemplo a partir del cual se pueda aprender algo más. Este programa introduce algunos componentes IGU adicionales.

Este segundo ejemplo es una aplicación para reproducir sonidos. No ofreceremos demasiados detalles ya que sólo pretende ser una base para que estudie el código por su propia cuenta y una fuente de fragmentos de código que puede copiar y modificar.

Aquí, en este capítulo, sólo señalaremos algunos pocos aspectos de esta aplicación en los que vale la pena concentrarse.

**Ejercicio 11.64** Abra el proyecto *sonidos-simples*. Cree una instancia de `ReproductorDeSonidoIGU` y experimente con la aplicación.

El reproductor de sonido busca y ejecuta fragmentos de sonido almacenados en la carpeta *audio* ubicada en la carpeta del proyecto. Puede reproducir sonidos almacenados en los formatos AIFF, AU y WAV. Tenga en cuenta que el formato WAV usa diversas formas diferentes de codificación y sólo algunas de ellas pueden ser ejecutadas en nuestro reproductor. Si tiene archivos propios de sonido del formato correcto, podrá reproducirlos llevándolos a la carpeta *audio* del proyecto *sonidos-simples*.

El reproductor de sonido está implementado mediante dos clases: `ReproductorDeSonidoIGU` y `MotorDeSonido`. Intentamos estudiar aquí solamente la primera. La clase `MotorDeSonido` se puede usar esencialmente como una clase de la biblioteca. Conviene que se familiarice con esta interfaz pero no es necesario que comprenda o modifique su implementación. (Será bienvenido, por supuesto, el hecho de que estudie esta clase tanto como quiera, pero en ella se aplican conceptos que no discutiremos en este libro.)

Seguidamente, realizamos algunas observaciones relevantes sobre este proyecto.

### *Separación Modelo/Vista*

Esta aplicación presenta una mejor separación entre el modelo y la vista que la del ejemplo anterior. Esto quiere decir que la funcionalidad de la aplicación (el modelo) está claramente separada de la interfaz de usuario (la IGU). Cada una de estas dos partes, el modelo y la vista, pueden estar compuestas por varias clases, pero cada clase deberá estar ubicada claramente en uno o en otro grupo para llevar a cabo una clara separación. En nuestro ejemplo, cada parte cuenta con una única clase.

Separar la funcionalidad de la aplicación de la interfaz de usuario es señal de buena cohesión: hace que el programa sea más fácil de comprender, de mantener y de adaptar a diferentes requerimientos (especialmente a diferentes interfaces de usuario). Por ejemplo, podría resultar bastante fácil la escritura de una interfaz para el reproductor de sonido que utilice sólo texto, reemplazando efectivamente la clase `ReproductorDeSonidoIGU` y dejando la clase `MotorDeSonido` sin modificaciones.

### *Derivar de JFrame*

En este ejemplo, demostramos una versión popular y diferente de creación de ventanas. Nuestra clase IGU no instancia un objeto `JFrame` sino que extiende la clase `JFrame`.

El resultado de esta extensión es que todos los métodos de `JFrame` que se necesitan (tales como `getContentPane`, `setJMenuBar`, `pack`, `setVisible`, etc.) ahora pueden ser invocados como métodos internos (heredados).

No existe una razón fuerte para preferir un estilo (usar una instancia de `JFrame`) sobre el otro (derivar de `JFrame`); la elección del estilo es, mayormente, una cuestión de preferencia personal.

### ***Mostrar imágenes estáticas***

Es muy común que se quiera mostrar una imagen en una IGU. La forma más fácil de hacerlo es incluyendo un `JLabel` en la interfaz que tenga un gráfico como etiqueta (un `JLabel` puede mostrar tanto texto como gráfico, o ambos al mismo tiempo). El reproductor de sonido incluye un ejemplo para hacer esta clase de etiquetas.

El código relevante es

```
JLabel imagen = new JLabel(new ImageIcon("titulo.jpg"));
```

Esta sentencia carga un archivo de imagen de nombre «`titulo.jpg`» desde la carpeta del proyecto, crea un ícono con dicha imagen y luego crea un `JLabel` que muestra este ícono.

El término «ícono» parece sugerir aquí que estamos hablando solamente de imágenes pequeñas, pero la imagen puede, de hecho, ser de cualquier tamaño. Este método funciona con imágenes JPEG, GIF y PNG.

### ***Cuadros combinados***

El reproductor de sonido presenta un ejemplo de uso de un `JComboBox`. Un cuadro combinado posee un conjunto de valores predefinidos, de los cuales se puede seleccionar uno en cualquier momento. Se muestra el valor seleccionado y se puede acceder a la selección a través de un menú desplegable. En el reproductor de sonido, el cuadro combinado se usa para seleccionar los formatos específicos de sonido.

Un `JComboBox` también puede ser editable, en cuyo caso no están predefinidos todos los valores sino que el usuario puede escribir algún valor que no esté en la lista.

### ***Listas***

El programa también incluye un ejemplo de una lista (clase `JList`) para mostrar la lista de sonidos disponibles. Una lista puede contener un número arbitrario de valores, de los cuales se puede seleccionar uno o más. Los valores de la lista de este ejemplo son cadenas, pero es posible que sean de otros tipos. Una lista no posee automáticamente una barra de desplazamiento.

### ***Barras de desplazamiento***

Otro componente que se demuestra en este ejemplo es el uso de las barras de desplazamiento.

Se pueden crear las barras de desplazamiento mediante un contenedor especial: una instancia de la clase `JScrollPane`. Los objetos IGU de cualquier tipo se pueden ubicar dentro de un panel de desplazamiento y luego, este panel, si contiene objetos demasiado grandes para mostrar dentro del espacio disponible, provee las barras de desplazamiento necesarias.

En nuestro ejemplo, hemos ubicado nuestra lista de sonidos en un panel de desplazamiento. Luego, el panel de desplazamiento se ubica dentro de su contenedor relacionado.

Otros elementos que se demuestran en este ejemplo son el uso de un deslizador y el uso del color para cambiar el aspecto de una aplicación. Cada uno de los elementos IGU tiene varios métodos para modificar la apariencia del componente o su comportamiento; tendrá que buscar en la documentación de cualquier componente que le interese y experimentar con él modificando algunas propiedades del mismo.

**Ejercicio 11.65** Modifique el reproductor de sonido de modo que muestre una imagen diferente en su centro. Busque una imagen en la web o cree una propia para usar en este ejercicio.

**Ejercicio 11.66** Cambie los colores de los restantes componentes (los colores del fondo y del texto) para que combinen con la nueva imagen principal.

**Ejercicio 11.67** Agregue un método «Recargar» al reproductor de sonido que relea los archivos de sonido de la carpeta *audio*. Luego, podrá dejar un nuevo archivo de sonido en la carpeta y cargarlo sin tener que salir del reproductor.

**Ejercicio 11.68** Agregue una función «Abrir» al menú *Archivo*. Cuando se active, presenta un diálogo de selección de archivos que permite al usuario seleccionar el archivo de sonido que desea abrir. Si el usuario selecciona una carpeta, el reproductor de sonido abre todos los archivos de sonido de dicha carpeta (tal como hace con la carpeta *audio*).

**Ejercicio 11.69** Modifique el deslizador de modo que el inicio y el final (y otras posibles marcas) estén etiquetadas con números. El inicio podría ser cero y el final podría ser la duración del archivo de sonido, expresada en segundos.

**Ejercicio 11.70** Modifique el reproductor de sonido de modo que al hacer doble clic sobre un elemento de la lista de sonidos, comience a ejecutarse el sonido seleccionado.

**Ejercicio 11.71** Mejore la apariencia del botón. Todos los botones que no cumplen ninguna función en determinado momento deberían estar inhabilitados, y debieran habilitarse sólo cuando puedan ser usados correctamente.

**Ejercicio 11.72** La clase *MotorDeSonido* provee un método para ajustar el volumen. Agregue un deslizador en algún lugar de la interfaz de usuario, para que se pueda ajustar el volumen.

## 11.10

## Resumen

En este capítulo hemos ofrecido una introducción a la programación IGU usando AWT y Swing. Hemos tratado las tres principales áreas conceptuales: crear componentes IGU, gestores de disposición y manejo de eventos.

Hemos visto que la construcción de una IGU generalmente comienza con la creación de una ventana de nivel alto, tal como un *JFrame*. Luego, la ventana se rellena con varios componentes que proveen información y funcionalidad al usuario. Entre estos componentes encontramos menús, elementos de menú, botones, etiquetas y bordes, entre otros.

Los componentes se acomodan en la pantalla con la ayuda de contenedores y de gestores de disposición. Los contenedores contienen colecciones de componentes y cada contenedor tiene un gestor de disposición que asume el trabajo de acomodar los componentes dentro del área del contenedor en la pantalla.

Los componentes interactivos (aquellos que pueden reaccionar a los ingresos del usuario) generan eventos que son activados por el usuario. Otros objetos se convierten en oyentes de eventos y pueden notificarse de tales eventos mediante la implementa-

ción de interfaces estándares. Cuando el objeto oyente se notifica, puede tomar la acción adecuada para operar con el evento del usuario.

Hemos introducido el concepto de clases propias anónimas como una técnica modular y extendible para escribir oyentes de eventos.

Y finalmente, hemos indicado una referencia en línea y un tutorial que pueden usarse para aprender más detalles no cubiertos en este capítulo.

**Ejercicio 11.73** Agregue una IGU al proyecto *world-of-zuul* del Capítulo 7.

Cada habitación deberá tener asociada una imagen que se mostrará cuando el jugador ingresa en ella. Debiera haber zonas de texto no editables para mostrar las salidas textuales. Para el ingreso de los comandos puede elegir entre diferentes posibilidades: puede dejar el ingreso mediante texto y usar un campo de texto (clase *JTextField*) para escribir los comandos o bien, puede usar botones para la entrada de los comandos.

**Ejercicio 11.74** Agregue sonidos al juego *world-of-zuul*. Puede asociar sonidos individuales con las habitaciones, con los elementos o con los personajes.

**Ejercicio 11.75** Diseñe y construya una IGU para un editor de textos. Los usuarios debieran tener la posibilidad de ingresar texto, editarlo, desplazarlo, etc. Considere funciones de formato (fuentes, estilos y tamaño) y funciones estadísticas como cantidad de palabras o de caracteres. No es necesario que implemente funciones para cargar y grabar el texto; tal vez prefiera esperar a leer el próximo capítulo.

Términos introducidos en este capítulo

**IGU, AWT, Swing, componente, gestor de disposición, evento, manejo de evento, oyente de evento, ventana, barra de menú, elemento de menú, panel contenedor, diálogo modal, clase interna anónima**

## Resumen de conceptos

- **componentes** Una IGU se construye mediante la ubicación de componentes en la pantalla. Los componentes están representados por objetos.
- **gestor de disposición** La distribución de los componentes en la pantalla se logra mediante el uso de gestores de disposición.
- **manejo de eventos** Los términos manejo de eventos hacen referencia a la tarea de reaccionar ante los eventos del usuario, tales como presionar el botón del ratón o pulsar una tecla.
- **formatos de imagen** Las imágenes se pueden almacenar en diferentes formatos. Las diferencias afectan principalmente al tamaño del archivo y a la información que contienen.
- **barra de menú, panel contenedor** Los componentes se ubican en una ventana agregándolos a la barra de menú de la ventana o al panel contenedor.

- **oyente de evento** Un objeto puede escuchar los eventos de los componentes implementando una interfaz de oyente de eventos.
- **clases internas anónimas** Las clases internas anónimas son una construcción muy útil para implementar oyentes de eventos.



# CAPÍTULO 12

## Manejo de errores

Principales conceptos que se abordan en este capítulo:

- programación defensiva
- lanzamiento y manejo de excepciones
- informe de errores
- procesamiento simple de archivos

Construcciones Java que se abordan en este capítulo

`TreeMap`, `TreeSet`, `SortedMap`, `assert`, excepción, `throw`, `throws`, `try`, `catch`, `FileReader`, `FileWriter`, `Scanner`, flujo

En el Capítulo 6 hemos visto que los errores lógicos de los programas son más difíciles de descubrir que los errores sintácticos porque el compilador no los detecta. Los errores lógicos surgen por diversos motivos y en algunas situaciones pueden estar encubiertos:

- La solución de un problema puede estar implementada incorrectamente. Por ejemplo, un problema que genera algunas estadísticas sobre los datos se puede haber programado de tal manera que calcula el valor de la media en lugar del valor de la mediana (el valor del medio).
- Se puede haber solicitado a un objeto que haga algo que es incapaz de hacer. Por ejemplo, se puede haber invocado al método `get` de una colección de objetos con un índice que está fuera del rango válido.
- Se puede haber usado un objeto de maneras tales que no coinciden con las anticipadas por el diseñador de la clase, dejando al objeto en un estado inapropiado o inconsistente. Esto ocurre con frecuencia cuando se reutiliza una clase en un ambiente diferente de su ambiente original, probablemente mediante herencia.

Aunque las distintas estrategias de prueba discutidas en el Capítulo 6 nos pueden ayudar a identificar y eliminar muchos errores lógicos antes de que nuestros programas estén listos para su uso, la experiencia nos sugiere que continuarán ocurriendo fallos en el programa. Además, aun cuando un programa se pruebe exhaustivamente puede fallar debido a circunstancias que están más allá del control del programador. Considere, por ejemplo, el caso de un navegador al que se le pide que muestre un sitio web que no

existe, o el de un programa que intenta grabar en un disco que no tiene más espacio. Estos problemas no son consecuencias de errores lógicos, pero pueden fácilmente hacer que un programa falle si es que no se anticipó la posibilidad de que surjan.

En este capítulo veremos cómo anticiparse y responder a las posibles situaciones de error que pueden surgir durante la ejecución de un programa. Además, ofrecemos algunas sugerencias sobre la manera de informar de los errores cuando éstos ocurren. También brindamos una breve introducción sobre los procesos de entrada y salida de texto como una de las situaciones en la que pueden aparecer fácilmente errores durante el tratamiento de los archivos.

## 12.1

### El proyecto *libreta-de-direcciones*

Usaremos la familia de proyectos *libreta-de-direcciones* para ilustrar algunos de los principios de informe y manejo de los errores que surgen en muchas aplicaciones. Los proyectos representan una aplicación que almacena datos de contacto (nombre, dirección y número de teléfono) de un número arbitrario de personas. En la libreta, los datos de los contactos se ordenan alfabéticamente tanto por nombre como por número de teléfono. Las clases principales que discutiremos son *LibretaDeDirecciones* (Código 12.1) y *DatosDelContacto*. Además, se proporciona la clase *LibretaDeDireccionesDemo* como un medio conveniente de preparar una libreta de direcciones con algunos datos de ejemplo.

#### Código 12.1

La clase  
*LibretaDeDirecciones*

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;
import java.util.SortedMap;
import java.util.TreeMap;
import java.util.TreeSet;
/**
 * Una clase para mantener un número arbitrario de
 * contactos.
 * Los datos de los contactos se ordenan por nombre y
 * por
 * número de teléfono.
 * @author David J. Barnes and Michael Kölling.
 * @version 2006.03.30
 */
public class LibretaDeDirecciones
{
    // Espacio para almacenar un número arbitrario de
    contactos.
    private TreeMap<String, DatosDelContacto> libreta;
    private int numeroDeEntradas;
    /**
     * Inicializa la libreta de direcciones.
     */
```

**Código 12.1  
(continuación)**

La clase  
LibretaDeDirecciones

```
public LibretaDeDirecciones()
{
    libreta = new TreeMap<String,
DatosDelContacto>();
    numeroDeEntradas = 0;
}

/**
 * Busca un nombre o un número de teléfono y
devuelve
 * los correspondientes datos de ese contacto.
 * @param clave El nombre o el número a buscar.
 * @return Los datos del contacto correspondiente a
la clave.
 */
public DatosDelContacto getContacto(String clave)
{
    return libreta.get(clave);
}
/**
 * Return si la clave actual está o no en uso.
 * @param clave El nombre o el teléfono a buscar.
 * @return true si la clave está en uso, false en
caso contrario.
 */
public boolean claveEnUso(String clave)
{
    return libreta.containsKey(clave);
}
/**
 * Agrega un nuevo contacto a la libreta de
direcciones.
 * @param contacto Los datos de contacto asociados
con una persona.
 */
public void agregarContacto(DatosDelContacto contacto)
{
    libreta.put(contacto.getNombre(), contacto);
    libreta.put(contacto.getTelefono(), contacto);
    numeroDeEntradas++;
}

/**
 * Cambia los datos del contacto almacenados
previamente bajo
 * la clave dada.
 * @param claveVieja Una de las claves que se usó
para almacenar los
 *
 * datos del contacto.

```

**Código 12.1  
(continuación)**

La clase  
LibretaDeDirecciones

```
        * @param contacto Los datos del contacto que
reemplazarán a los
        *                               existentes.
        */
    public void modificarContacto(String claveVieja,
        DatosDelContacto contacto)
    {
        eliminarContacto(claveVieja);
        agregarContacto(contacto);
    }

    /**
     * Busca todos los datos de los contactos
almacenados bajo
     * una clave que comienza con un prefijo
determinado.
     * @param prefijo El prefijo a buscar entre las
claves.
     * @return Un arreglo con los contactos que se
encontraron.
     */
    public DatosDelContacto[] buscar(String prefijo)
    {
        List<DatosDelContacto> coincidencias =
new LinkedList<DatosDelContacto>();
        // Busca las claves iguales o mayores que el
prefijo dato.
        SortedMap<String, DatosDelContacto> cola =
libreta.tailMap(prefijo);
        Iterator<String> it = cola.keySet().iterator();
        boolean finDeBusqueda = false;
        while(!finDeBusqueda && it.hasNext()) {
            String clave = it.next();
            if(clave.startsWith(prefijo)) {
                coincidencias.add(libreta.get(clave));
            }
            else {
                finDeBusqueda = true;
            }
        }
        DatosDelContacto[] resultados =
new
        DatosDelContacto[coincidencias.size()];
        coincidencias.toArray(resultados);
        return resultados;
    }
    /**

```

**Código 12.1  
(continuación)**

La clase  
LibretaDeDirecciones

```
        * @return El número de entradas que hay
actualmente en la libreta.
    */
public int getNumeroDeEntradas()
{
    return numeroDeEntradas;
}
/**
     * Elimina de la libreta, la entrada que tiene la
clave dada.
     * @param clave Una de las claves de entrada a
eliminar.
    */
public void eliminarContacto(String clave)
{
    DatosDelContacto contacto = libreta.get(clave);
    libreta.remove(contacto.getNombre());
    libreta.remove(contacto.getTelefono());
    numeroDeEntradas--;
}
/**
     * @return Los datos de todos los contactos, en el
orden que
     *          los almacena la clase
DatosDelContacto.
    */
public String listarContactos()
{
    // Dado que cada entrada se almacena mediante
dos claves,
    // es necesario construir un conjunto de
DatosDelContacto que
    // elimina los contactos duplicados.

    StringBuffer todasLasEntradas = new
StringBuffer();
    Set<DatosDelContacto> contactosOrdenados =
        new
TreeSet<DatosDelContacto>(libreta.values());
    for(DatosDelContacto contacto :
contactosOrdenados) {
        todasLasEntradas.append(contacto);
        todasLasEntradas.append('\n');
        todasLasEntradas.append('\n');
    }
    return todasLasEntradas.toString();
}
}
```

Se pueden almacenar nuevos contactos en la libreta mediante el método `agregarContacto`. Este método asume que los datos representan un nuevo contacto y no la modificación de los datos de un contacto que ya existe. Para cubrir este último caso, el método `modificarContacto` elimina una entrada anterior y la reemplaza por los datos revisados. La libreta de direcciones proporciona dos maneras de obtener los datos de los contactos: el método `getContacto`, que toma un nombre o un número de teléfono como clave y devuelve los datos del contacto que coincide con la clave, y el método `buscar`, que devuelve un arreglo con todos los contactos que comienzan con determinada cadena de búsqueda. Por ejemplo, la cadena de búsqueda «08459» devolverá todas las entradas cuyos números de teléfono tengan ese prefijo de área.

Hay dos versiones introductorias del proyecto *libreta-de-direcciones* que se pueden explorar, ambas proporcionan acceso a la misma versión de la clase `LibretaDeDirecciones` que se muestra en el Código 12.1. El proyecto *libreta-de-direcciones-v1t* proporciona una interfaz de usuario basada en texto, de estilo similar al de la interfaz del juego *zuul* que tratamos en el Capítulo 7. Los comandos actualmente disponibles en esta interfaz son los que permiten listar el contenido de la libreta, buscar algún contacto y agregar una nueva entrada. Probablemente, la interfaz de la versión *libreta-de-direcciones-v1g* sea más interesante ya que incorpora una IGU sencilla. Experimente con ambas versiones para obtener un poco de experiencia sobre la funcionalidad de la aplicación.

**Ejercicio 12.1** Abra el proyecto *libreta-de-direcciones-v1g* y cree un objeto `LibretaDeDireccionesDemo`. Invoque su método `mostrarInterfaz` para visualizar la IGU e interactuar con la libreta de direcciones de ejemplo.

**Ejercicio 12.2** Repita su experimentación utilizando la interfaz de texto del proyecto *libreta-de-direcciones-v1t*.

**Ejercicio 12.3** Examine la implementación de la clase `LibretaDeDirecciones` y evalúe si considera que está bien escrita o no. ¿Tiene alguna crítica específica acerca de esta clase?

**Ejercicio 12.4** La clase `LibretaDeDirecciones` usa varias clases del paquete `java.util`; si no está familiarizado con algunas de ellas, busque la documentación API para completar los baches que pueda tener. ¿Piensa que se justifica el uso de tantas clases de utilidad diferentes? ¿Se podría usar un `HashMap` en lugar de un `TreeMap`?

**Ejercicio 12.5** Modifique las clases `PalabrasComando` y `LibretaDeDireccionesInterfazDeTexto` del proyecto *libreta-de-direcciones-v1t* de modo que proporcionen acceso interactivo a los métodos `getContacto` y `eliminarContacto` de `LibretaDeDirecciones`.

**Ejercicio 12.6** La clase `LibretaDeDirecciones` define un atributo para registrar el número de entradas. ¿Considera que sería más adecuado calcular este valor a partir del número de entradas en el `TreeMap`? Por ejemplo, ¿encuentra alguna situación en la que el siguiente cálculo no produciría el mismo valor?

```
return libreta.size() / 2;
```

## 12.2 Programación defensiva

### 12.2.1 Interacción cliente-servidor

`LibretaDeDirecciones` es un objeto servidor típico pues no inicia ninguna acción por su propia cuenta sino que toda su actividad se dirige a satisfacer las peticiones del cliente. Los implementadores pueden adoptar como mínimo dos puntos de vista posibles al diseñar e implementar un servidor:

- Pueden asumir que los objetos cliente sabrán lo que están haciendo y requerirán servicios sólo de una manera sensata y bien definida.
- Pueden asumir que el servidor operará en un ambiente esencialmente hostil, en el que se deben tomar todas las medidas posibles para prevenir que los objetos cliente usen el servidor incorrectamente.

Estas visiones representan claramente extremos opuestos; en la práctica, la mayoría de las situaciones asumirán posiciones intermedias. La mayoría de las interacciones del cliente será razonable excepto algún intento ocasional de uso del servidor de manera incorrecta, ya sea como resultado de un error lógico de programación o de un concepto erróneo del programador del cliente. Estos diferentes puntos de vista proporcionan una base muy útil para discutir asuntos del estilo:

- ¿Cuántas verificaciones de las solicitudes del cliente deben realizar los métodos del servidor?
- ¿Cómo debe informar el servidor, los errores a sus clientes?
- ¿Cómo puede un cliente anticipar un fallo en una solicitud al servidor?
- ¿Cómo puede tratar un cliente el fallo de una solicitud?

Si examinamos la clase `LibretaDeDirecciones` con estas cuestiones en mente, veremos que la clase se escribió confiando plenamente en que sus clientes la usarán adecuadamente. El Ejercicio 12.7 ilustra un ejemplo que confirma la afirmación anterior y permite detectar los posibles problemas que se pueden presentar.

**Ejercicio 12.7** En el proyecto *libreta-de-direcciones-v1g* cree un nuevo objeto `LibretaDeDirecciones` en el banco de objetos. La libreta estará completamente vacía, no contendrá ningún contacto. A continuación invoque al método `eliminarContacto` con cualquier cadena para la clave. ¿Qué ocurre? ¿Puede explicar por qué ocurre esto?

**Ejercicio 12.8** La respuesta más fácil de un programador ante una situación de error que surja es permitir que el programa finalice (es decir, «se cae»). Trate de pensar algunas situaciones en las que permitir simplemente que un programa finalice puede ser muy peligroso.

**Ejercicio 12.9** Muchos programas que se venden comercialmente contienen errores que no están manejados adecuadamente y que provocan que el programa se caiga. ¿Es esto inevitable? ¿Es aceptable? Discútalo.

El problema que tiene el método `eliminarContacto` es que asume que la clave que recibe es válida para la libreta y utiliza esa supuesta clave para recuperar los datos asociados con el contacto:

```
DatosDelContacto contacto = libreta.get(clave);
```

Sin embargo, si la clave no tiene un objeto asociado, la variable `contacto` contendrá el valor `null`. Esto no es en sí mismo un error, pero se genera el error a partir de la siguiente sentencia en la que asumimos que `contacto` hace referencia a un objeto válido:

```
libreta.remove(contacto.getNombre());
```

No es posible invocar un método sobre el valor `null` y el resultado de esta invocación es un error en tiempo de ejecución. BlueJ informa esta situación como un `NullPointerException` y resalta la sentencia que lo produjo. Más adelante, en este capítulo, discutiremos las excepciones en detalle, pero por ahora, simplemente podemos decir que si ocurriera en la ejecución de una aplicación un error de este estilo, la aplicación finalizará prematuramente antes de que se haya completado la tarea.

Aquí existe claramente un problema, pero ¿de quién es la culpa? ¿Del objeto cliente por invocar al método con un argumento erróneo? ¿O es del objeto servidor por no manejar adecuadamente esta situación? El escritor de la clase cliente podría argumentar que no existe nada en la documentación del método que indique que la clave debe ser válida. Recíprocamente, el escritor de la clase servidor podría argumentar que es obviamente erróneo tratar de eliminar los datos de un contacto con una clave no válida. Nuestro compromiso en este capítulo no es resolver tales disputas sino, primordialmente, impedir que se disparen tales errores. Comenzaremos viendo el manejo del error desde el punto de vista de la clase servidor.

**Ejercicio 12.10** Grabe con otro nombre una copia de uno de los proyectos `libreta-de-direcciones-v1` para trabajar sobre ella. Modifique el método `eliminarContacto` para evitar que se genere un `NullPointerException` cuando la clave no tiene, en la libreta, una entrada que le corresponda. Si la clave no es válida, el método no debe hacer nada.

**Ejercicio 12.11** En una llamada a `eliminarContacto`, ¿es necesario informar el uso de una clave no válida? De ser así, ¿cómo podría informarlo?

**Ejercicio 12.12** ¿Existen otros métodos en la clase `LibretaDeDirecciones` que sean vulnerables a errores similares? De ser así, trate de corregirlos en su copia del proyecto. ¿Es posible, en todos los casos, que un método simplemente no haga nada cuando sus argumentos no son los adecuados? ¿Es necesario informar los errores de alguna manera? De ser así, ¿cómo debiera hacerlo? ¿Todos los errores se deben informar de la misma forma?

## 12.2.2 Validar argumentos

Un objeto servidor es más vulnerable cuando su constructor y sus métodos reciben los valores de los argumentos a través de sus parámetros. Los valores que se pasan a un constructor se utilizan para establecer el estado inicial de un objeto; los valores que se pasan a un método se usarán para influir sobre el efecto general de la llamada al método

y quizás también sobre el resultado que produce. Por lo tanto, es vital que un servidor sepa si puede confiar en que los valores de los argumentos son válidos o si necesita verificar su validez por sí mismo. La situación actual en las clases `DatosDelContacto` y `LibretaDeDirecciones` es que no existe ningún control sobre los valores de los argumentos. Como hemos visto con el método `eliminarContacto`, esta falta de control puede conducir a la ocurrencia de un error fatal en tiempo de ejecución.

Es relativamente fácil impedir que se genere un `NullPointerException` en `eliminarContacto` y el Código 12.2 ilustra cómo puede hacerse. Observe que además de haber mejorado el código del método también hemos actualizado el comentario del método para que documente el hecho de que se ignoran las claves desconocidas.

### Código 12.2

Resguardo contra una clave no válida en `eliminarContacto`

```
/** Elimina la entrada de la libreta con la clave dada.  
 * Si la clave no existe, no hace nada.  
 * @param clave Una de las claves de la entrada a  
 eliminar  
 */  
public void eliminarContacto(String clave)  
{  
    if (claveEnUso(clave)) {  
        DatosDelContacto contacto = libreta.get(clave);  
        libreta.remove(contacto.getNombre());  
        libreta.remove(contacto.getTelefono());  
        numeroDeEntradas--;  
    }  
}
```

Si examinamos todos los métodos de `LibretaDeDirecciones` encontramos que existen otros lugares en los que podríamos implementar mejoras similares:

- El método `agregarContacto` debe controlar que su argumento no sea el valor `null`.
- El método `modificarContacto` debe controlar que su clave vieja sea una de las que están en uso y que los nuevos datos no son `null`.
- El método `buscar` debe controlar que su clave no sea `null`.

Estos cambios han sido implementados en la versión de la aplicación que se encuentra en los proyectos `libreta-de-direcciones-v2g` y `libreta-de-direcciones-v2t`.

**Ejercicio 12.13** ¿Por qué cree que consideramos innecesario realizar cambios similares en los métodos `getContacto` y `claveEnUso`?

**Ejercicio 12.14** Al trabajar sobre los errores de los argumentos, no hemos impreso ningún mensaje de error. ¿Considera que `LibretaDeDirecciones` debe imprimir un mensaje de error cuando recibe un argumento erróneo en alguno de sus métodos? ¿Existen algunas situaciones en las que sería inadecuada la impresión de un mensaje de error?

**Ejercicio 12.15** ¿Existe alguna otra validación de los argumentos de los restantes métodos que considera que se debe hacer para evitar que el objeto `LibretaDeDirecciones` funcione incorrectamente?

## 12.3

### Informar de errores del servidor

Una vez que el servidor impide llevar a cabo una operación ilegal a través de parámetros con valores incorrectos podríamos considerar el punto de vista de que esto es todo lo que el escritor de la clase servidor necesita hacer. Sin embargo, idealmente y en primer lugar, debemos evitar que se produzcan tales situaciones de error. Además, es frecuente el caso en el que se aporta un parámetro incorrecto como resultado de algún error de programación en el cliente. En consecuencia, en lugar de simplemente programar alrededor del problema en el servidor y dejar el problema localizado allí, es una buena práctica hacer que el servidor realice algún esfuerzo para indicar que ha surgido un problema, ya sea del propio cliente o de un usuario humano o del programador. En este sentido, existe la posibilidad de que funcione bien un cliente escrito incorrectamente. ¿Cuál es la mejor manera de que un servidor informe de los problemas cuando éstos ocurren? No hay una sola respuesta a esta pregunta y generalmente, la respuesta más adecuada dependerá del contexto particular en el que se use el objeto servidor. En las siguientes secciones exploraremos un conjunto de opciones para informar errores mediante un servidor.

**Ejercicio 12.16** ¿De cuántas maneras diferentes se puede indicar que un método ha recibido valores incorrectos en sus parámetros o que es incapaz de completar su tarea? Considere tantos tipos diferentes de aplicaciones como pueda. Por ejemplo, las que tienen una IGU, las que tienen una interfaz de texto y un usuario humano, las que no tienen ningún tipo de interactividad con el usuario como por ejemplo, el software de los sistemas que dirigen el motor de un automóvil.

#### 12.3.1

#### Notificar al usuario

La manera más obvia en que un objeto puede tratar de responder cuando detecta algo erróneo es intentar notificar al usuario de la aplicación de alguna forma. Las principales opciones son imprimir un mensaje de error usando `System.out` o mostrar una ventana de mensaje de error.

Los dos problemas principales que tiene este abordaje son los siguientes:

- Asumen que la aplicación será usada por un usuario humano que verá el mensaje de error. Hay muchas aplicaciones que corren de manera completamente independiente de un usuario humano, en las que un mensaje de error o una ventana de error será completamente pasada por alto. En rigor de verdad, la computadora en la que se ejecuta la aplicación podría no tener ningún dispositivo visual conectado para mostrar estos mensajes de error.
- Aun cuando exista un humano que pueda ver el mensaje de error, es raro que dicho usuario esté en posición de hacer algo con respecto al problema. ¡Imagine a un usuario de un cajero automático enfrentado a un `NullPointerException`! Solamente en aquellos casos en los que la acción directa del usuario conduzca al problema (como aportar un ingreso no válido a la aplicación) puede estar capacitado para tomar alguna medida correctiva adecuada.

Los programas que imprimen mensajes de error inapropiados tienden más a confundir al usuario que a tener alguna utilidad para el mismo. Por lo tanto, excepto en un muy limitado conjunto de circunstancias, la notificación al usuario no es, en general, una solución al problema del informe de errores.

### 12.3.2 Notificar al objeto cliente

Un enfoque radicalmente diferente al que abordamos hasta ahora consiste en que el servidor ofrezca alguna indicación al objeto cliente de que algo anduvo mal. Hay dos maneras de hacer esto:

- Un servidor puede usar el valor de retorno de un método para devolver una bandera que indique si fue exitoso o si ocurrió un fallo en la llamada a dicho método.
- Un servidor puede *lanzar una excepción* desde el método servidor si algo anda mal. Esto introduce una nueva característica de Java que se encuentra también en otros lenguajes de programación. Describiremos esta característica detalladamente en la Sección 12.4.

Ambas técnicas tienen el beneficio de asegurar que el programador del cliente tenga en cuenta que puede fallar una llamada a un método sobre otro objeto. Sin embargo, sólo la decisión de lanzar una excepción evita activamente que el programador del cliente ignore las consecuencias del fallo del método.

El primer enfoque es fácil de introducir en un método que tiene un tipo de retorno `void`, como es el caso de `eliminarContacto`. Si el tipo `void` se reemplaza por el tipo `boolean`, el método puede devolver `true` para indicar que la eliminación fue exitosa y `false` para indicar que falló por algún motivo (Código 12.3).

#### Código 12.3

Tipo de retorno  
`boolean` para indicar  
éxito o fracaso

```
/**  
 * Elimina la entrada de la libreta con la clave dada.  
 * La clave debe ser una de las que está actualmente en uso.  
 * @param clave Una de las claves de la entrada a  
eliminar  
 * @return true Si la entrada se eliminó exitosamente,  
 *             falso en caso contrario.  
 */  
public boolean eliminarContacto(String clave)  
{  
    if (claveEnUso(clave)) {  
        DatosDelContacto contacto = libreta.get(clave);  
        libreta.remove(contacto.getNombre());  
        libreta.remove(contacto.getTelefono());  
        numeroDeEntradas--;  
        return true;  
    }  
    else {  
        return false;  
    }  
}
```

Esto permite que un cliente use una sentencia `if` para salvaguardar sentencias que dependen del éxito de la eliminación de una entrada:

```
if(contactos.eliminarContacto(_....)) {
    // Entrada exitosamente eliminada. Continúa normalmente.
    ...
}
else {
    // La eliminación falló. Intenta recuperarse, si es posible.
    ...
}
```

Cuando un método servidor ya tenga un tipo de retorno distinto de `void` (para evitar efectivamente que se retorne un valor de diagnóstico `boolean`) todavía existe alguna forma de indicar que ha ocurrido un error mediante el tipo de retorno. Este será el caso si se dispone de un valor en el rango del tipo de retorno que actúe como un valor de diagnóstico de error. Por ejemplo, el método `getContacto` devuelve el objeto `DatosDelContacto` correspondiente a una clave dada y el siguiente ejemplo asume que una clave en particular ubicará un conjunto válido de datos del contacto:

```
// Envía un mensaje de texto a David.
DatosDelContacto contacto = contactos.getContacto(_David_);
String telefono = contacto.getTelefono();
...
```

Una manera en que el método `getContacto` puede indicar si una clave no es válida o si no está en uso es devolviendo el valor de retorno `null` en lugar de devolver un objeto `DatosDelContacto` (Código 12.4).

#### Código 12.4

Retornar un valor de diagnóstico de error fuera de los límites

```
/**
 * Busca un nombre o un número de teléfono y devuelve
los
 * datos del contacto correspondiente.
 * @param clave El nombre o número a buscar.
 * @return Los datos correspondientes a la clave o null
 *         si la clave no está en uso
 */
public DatosDelContacto getContacto(String clave)
{
    if(claveEnUso(clave)) {
        return libreta.get(clave);
    }
    else {
        return null;
    }
}
```

Esto podría permitir que un cliente examine el resultado de la llamada y luego continúe con el control normal del flujo o intente recuperarse del error:

```
DatosDelContacto contacto =
contactos.getContacto(_David_);
```

```
if (contacto != null) {
    // Envía un mensaje de texto a David.
    String telefono = contacto.getTelefono();
    ...
}
else {
    // Falló al buscar la entrada. Intenta recuperarse,
    si es posible.
    ...
}
```

Es común que los métodos que retornan referencias a objetos utilicen el valor `null` para indicar un fallo o un error. En los métodos que retornan valores de tipos primitivos, se suele devolver algún valor fuera de los límites válidos que cumple un rol similar: por ejemplo, el método `indexOf` de la clase `String` devuelve un valor negativo para indicar que falló en encontrar el carácter buscado.

**Ejercicio 12.17** Use una copia del proyecto *libreta-de-direcciones-v2t* para realizar los cambios en la clase `LibretaDeDirecciones` en los lugares adecuados, de modo que proporcione información de los fallos a un cliente, cuando un método reciba valores incorrectos en sus parámetros o cuando le resulte imposible completar su tarea.

**Ejercicio 12.18** ¿Considera que los diferentes estilos de interfaces de los proyectos *v2g* y *v2t* implican que debieran diferenciarse también en la manera en que se informen los errores a los usuarios?

**Ejercicio 12.19** ¿Existen algunas combinaciones de valores de los argumentos que considera inapropiados para pasar al constructor de la clase `DatosDelContacto`?

**Ejercicio 12.20** ¿Considera que una llamada al método `buscar` que no encuentre coincidencias requiere una notificación de error? Justifique su respuesta.

**Ejercicio 12.21** ¿Tiene un constructor alguna manera de indicar al cliente que no pudo preparar adecuadamente el estado de un nuevo objeto? ¿Qué debiera hacer un constructor si recibe argumentos inapropiados?

Claramente, este enfoque no se puede usar en aquellos lugares en los que todos los valores del tipo de retorno ya tienen significados válidos para el cliente. En tales casos, generalmente será necesario pasar a la técnica alternativa de *lanzar una excepción* (véase Sección 12.4) que, de hecho, ofrece importantes ventajas. Para ayudarlo a apreciar estas ventajas, es valioso considerar dos cuestiones asociadas al uso de los valores de retorno como indicadores de fracaso o de error:

- Desafortunadamente, no hay manera de requerir al cliente que controle el valor de retorno en relación a sus propiedades de diagnóstico. En consecuencia, un cliente podría fácilmente actuar como si nada hubiera ocurrido y luego terminar con un `NullPointerException`, o peor todavía, podría usar el valor de retorno de diagnóstico como si fuera un valor de retorno normal, creando un error lógico difícil de diagnosticar.

- En algunos casos, podríamos usar el valor de diagnóstico con dos propósitos muy diferentes. Es lo que ocurre en los métodos revisados `eliminarContacto` (Código 12.3) y `getContacto` (Código 12.4). Un propósito es notificar al cliente si su petición fue exitosa o no. El otro es indicar que hubo algún error en su solicitud, como por ejemplo, que se pasó un valor incorrecto como argumento.

En muchos casos, una solicitud no exitosa no representa un error lógico de programación sino que se hizo una solicitud incorrecta. Debemos esperar respuestas muy diferentes de un cliente en estos dos casos. No existe una manera satisfactoria y general de resolver este conflicto usando simplemente valores de retorno.

## 12.4

### Principios del lanzamiento de excepciones

#### Concepto

Una **excepción** es un objeto que representa los detalles de un fallo de un programa. Se lanza una excepción para indicar que ha ocurrido un fallo.

El lanzamiento de una excepción es la manera más efectiva que tiene un objeto servidor para indicar que es incapaz de completar la solicitud del cliente. Una de las mayores ventajas que tiene esta técnica es que usa un valor especial de retorno que hace casi imposible que un cliente ignore el hecho de que se ha lanzado una excepción y continúe indiferente. El fracaso del cliente al manejar una excepción dará por resultado que la aplicación termine inmediatamente. Además, el mecanismo de la excepción es independiente del valor de retorno de un método y se puede usar en todos los métodos, más allá del tipo de valor que retornan.

#### 12.4.1 Lanzar una excepción

El Código 12.5 muestra cómo se lanza una excepción usando una *sentencia throw* dentro de un método. El método `getContacto` lanza una excepción para indicar que no tiene sentido el pasaje de un valor `null` para la clave.

#### Código 12.5

Lanzar una excepción

```
/*
 * Busca un nombre o un número de teléfono y devuelve
 * los
 *   * datos del contacto correspondiente.
 *   * @param clave El nombre o número a buscar.
 *   * @return Los datos correspondientes a la clave o null
 *           si no hay coincidencias.
 *   * @throws NullPointerException si la clave es null.
 */
public DatosDelContacto getContacto(String clave)
{
    if(clave == null) {
        throw new NullPointerException(
            "clave null en getContacto");
    }
    return libreta.get(clave);
}
```

El lanzamiento de una excepción tiene dos etapas: primero se crea un objeto excepción (en este caso un objeto `NullPointerException`) y luego se lanza el objeto excepción usando la palabra clave `throw`. Estas dos etapas se combinan casi invariamente en una única sentencia:

```
throw new TipoDeExcepcion ("cadena opcional de diagnóstico");
```

Cuando se crea un objeto excepción, se puede pasar una cadena de diagnóstico a su constructor. Esta cadena estará disponible para el receptor de la excepción mediante el método de acceso `getMessage` del objeto excepción o de su método `toString`.

El Código 12.5 ilustra también que se puede expandir la documentación de un método para que incluya los detalles de cualquier excepción que lance mediante la etiqueta `@throws` del documentador de java (`javadoc`).

## 12.4.2 Clases `Exception`

Un objeto excepción es siempre una instancia de una clase de una jerarquía de herencia especial. Podemos crear nuevos tipos de excepciones creando subclases en esta jerarquía (Figura 12.1). Hablando estrictamente, las clases de excepciones siempre son subclases de la clase `Throwable` que está definida en el paquete `java.lang`. Seguiremos la convención de definir y usar las clases de excepciones como subclases de la clase `Exception`, también definida en `java.lang`<sup>1</sup>. El paquete `java.lang` define varias clases de excepciones que se ven comúnmente y con las que es posible que ya se haya encontrado pues se pueden haber ejecutado inadvertidamente durante el desarrollo de los programas como por ejemplo: `NullPointerException`, `IndexOutOfBoundsException` y `ClassCastException`.

Java divide las clases de excepciones en dos categorías: *excepciones comprobadas y no comprobadas*. Todas las subclases de la clase estándar de Java `RunTimeException` son excepciones no comprobadas, todas las restantes subclases de `Exception` son excepciones comprobadas.

Sumamente simplificado, la diferencia es ésta: las excepciones comprobadas están pensadas para aquellos casos en los que el cliente debe esperar que una operación falle (por ejemplo: cuando grabamos en un disco, sabemos que el disco puede estar lleno). En estos casos, el cliente está obligado a comprobar si la operación fue exitosa. Las excepciones no comprobadas están pensadas para aquellos casos que no deben fallar en una operación normal; generalmente indican un error en el programa.

Desafortunadamente, saber qué categoría de excepción conviene lanzar en una circunstancia en particular no es una ciencia exacta pero podemos ofrecer las siguientes sugerencias:

- Una regla a priori que se puede aplicar es usar excepciones no comprobadas en las situaciones que podrían producir un fallo en el programa, típicamente porque se sospecha la existencia de un error lógico en el programa que le impedirá continuar funcionando. Se desprende que las excepciones comprobadas deben usarse cuando

---

<sup>1</sup> La clase `Exception` es una de las dos subclases directas de `Throwable`; la otra es `Error`. Las subclases de `Error` se reservan, generalmente, para los errores en tiempo de ejecución antes que para los errores sobre los que el programador tiene control.

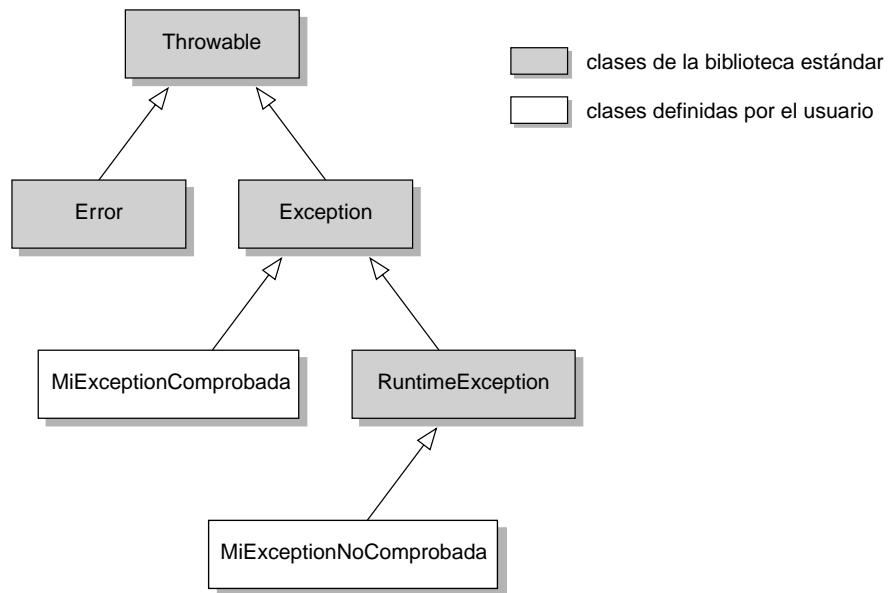
ocurrió un problema pero existe alguna posibilidad de que el cliente efectúe alguna recuperación. Un problema con esta política es que asume que el servidor es suficientemente consciente del contexto en el que se está usando como para ser capaz de determinar si es probable que la recuperación del cliente sea posible.

- Otra regla a priori es usar excepciones no comprobadas en aquellas situaciones que pueden ser razonablemente evitadas. Por ejemplo, el uso de un índice no válido para acceder a un arreglo es el resultado de un error lógico de programación que es completamente evitable y el hecho de que la excepción `ArrayIndexOutOfBoundsException` no es comprobada encaja con este modelo. Se desprende que las excepciones no comprobadas deben usarse para situaciones de fallos que están bajo el control del programador como por ejemplo, que un disco se llene cuando se intenta grabar un archivo.

Las reglas formales de Java que gobiernan el uso de las excepciones son significativamente diferentes para las excepciones comprobadas y para las no comprobadas y describiremos las diferencias en detalle en las secciones 12.4.4 y 12.5.1 respectivamente. En términos simples, las reglas aseguran que un objeto cliente que llama a un método que puede disparar una excepción comprobada puede contener tanto código para anticipar la posibilidad de un problema como código para intentar manejar el problema cuando éste ocurra<sup>2</sup>.

**Figura 12.1**

Jerarquía de clases de excepción



**Ejercicio 12.22** Enumere tres tipos de excepciones del paquete `java.io`.

**Ejercicio 12.23** La excepción `SecurityException` del paquete `java.lang`, ¿es una excepción comprobada o una no comprobada? ¿Y la excepción `NoSuchMethodException`?

<sup>2</sup> De hecho, es aún demasiado fácil para el escritor del cliente adherir en principio a las reglas, pero no intentar una recuperación apropiada del problema.

### 12.4.3 El efecto de una excepción

¿Qué ocurre cuando se lanza una excepción? En realidad, hay dos efectos a considerar: el efecto en el método en que se lanzó la excepción y el efecto en el invocador.

Cuando se lanza una excepción, la ejecución del método que la disparó termina inmediatamente (no continúa hasta el final del cuerpo). Una consecuencia particular de esto es que no se requiere un método con un tipo de retorno distinto de `void` para ejecutar una sentencia `return` en la ruta en que se lanza una excepción. Esto es razonable porque el lanzamiento de una excepción es una indicación de la incapacidad del método disparador para continuar con la ejecución normal, que incluye la imposibilidad de retornar un resultado válido. Podemos ilustrar este principio con la siguiente versión alternativa del cuerpo del método que se muestra en el Código 12.5:

```
if (key == null) {
    throw new NullPointerException("clave null en getContacto");
}
else {
    return libreta.get(clave);
}
```

La ausencia de una sentencia `return` en la ruta en que se dispara una excepción es aceptable. En su lugar, el compilador indicará un error si se han escrito sentencias a continuación de la sentencia `throw` porque podrían no ejecutarse nunca.

El efecto de una excepción en el sitio del programa que invocó al método es un poco más complejo. En particular, el efecto completo depende de si se ha escrito o no código para *capturar* la excepción. Considere la siguiente llamada a `getContacto`:

```
DatosDelContacto datos = libreta.getContacto(null);
// La siguiente sentencia no será encontrada
String telefono = datos.getTelefono();
```

Podemos decir que en todos los casos la ejecución de estas sentencias quedará incompleta: la excepción lanzada por `getContacto` interrumpirá la ejecución de la primera sentencia y no se realizará ninguna asignación a la variable `datos`. En consecuencia, la segunda sentencia tampoco se ejecutará.

Este ejemplo ilustra claramente el poder de las excepciones para impedir que un cliente continúe sin tener en cuenta el hecho de que haya surgido un problema. Lo que realmente ocurre a continuación de una excepción depende de si se la captura o no. Si no se captura la excepción, el programa simplemente terminará con la indicación de que se ha lanzado una `NullPointerException` sin capturar. Discutiremos cómo capturar una excepción en la Sección 12.5.2.

### 12.4.4 Excepciones no comprobadas

#### Concepto

Las **excepciones no comprobadas** son un tipo de excepción cuyo uso no requerirá controles por parte del compilador.

Las excepciones no comprobadas son las más fáciles de usar desde el punto de vista del programador, porque el compilador impone muy pocas reglas para su uso. Este es el sentido de «no comprobadas»: el compilador no aplica ningún control especial sobre el método en el que se lanza una excepción no comprobada, ni tampoco en el lugar desde donde se invocó dicho método. Una clase excepción es no comprobada si es una subclase de la clase `RuntimeException` definida en el paquete `java.lang`. Todos los ejemplos que hemos usado hasta ahora para ilustrar el lanzamiento de excepciones han sido de excepciones no comprobadas, por lo tanto, hay muy poco para agregar

sobre cómo lanzar una excepción no comprobada: simplemente usar una sentencia *throw*.

Si seguimos también la convención de que las excepciones no comprobadas deben usarse en aquellas situaciones en las que esperamos que el resultado sea la terminación del programa, (es decir, que no se va a capturar la excepción), entonces tampoco hay más para discutir sobre lo que debe hacer el método invocador puesto que no hará nada y dejará que el programa falle. Sin embargo, si existe la necesidad de capturar una excepción no comprobada, entonces se puede escribir un manejador de dicha excepción, exactamente de la misma manera que para una excepción comprobada. La forma de hacer esto se describe en la Sección 12.5.2.

Una excepción no comprobada, que se usa comúnmente es `IllegalArgumentExcep-`  
`tion`, es lanzada por un constructor o un método para indicar que los valores de sus argumentos no son los adecuados. Por ejemplo, el método `getContacto` podría dis-  
`parar` esta excepción cuando la cadena pasada para la clave es vacía (Código 12.6).

#### Código 12.6

Verificar si un argumento es ilegal

```
 /**
 * Busca un nombre o un número de teléfono y devuelve
 * los datos de contacto correspondientes.
 * @param clave El nombre o el número que a
buscar.
 * @throws NullPointerException si la clave es
null.
 * @throws IllegalArgumentException si la clave
está vacía.
 * @return Los datos correspondientes a la clave
dato o
 *         null si no hay ninguna coincidencia.
 */
public DatosDelContacto getContacto(String clave)
{
    if(clave == null) {
        throw new NullPointerException(
"clave null en getContacto");
    }
    if(clave.trim().length() == 0){
        throw new IllegalArgumentException(
"Se pasó clave vacía a getContacto");
    }
    return libro.get(clave);
}
```

Es valioso tener un método que conduzca una serie de controles de validez de sus argumentos antes de proceder con el propósito principal del método. Esto hace menos probable que un método ejecute parte de sus acciones antes de lanzar una excepción debida a valores incorrectos en sus argumentos. Una razón particular para evitar esta situación es que la modificación parcial de un objeto probablemente lo deje en un estado inconsistente para su futuro uso. Si una operación falla por alguna razón, idealmente,

el objeto deberá quedar en el estado en que estaba antes de que se intentara realizar la operación.

**Ejercicio 12.24** Revise todos los métodos de la clase `LibretaDeDirecciones` y decida si algunos de ellos deben lanzar una `IllegalArgumentException`. De ser así, agregue los controles y las sentencias `throw` necesarios.

**Ejercicio 12.25** Si todavía no lo ha hecho, agregue documentación javadoc para describir cualquier excepción lanzada por los métodos de la clase `LibretaDeDirecciones`.

#### 12.4.5 Impedir la creación de un objeto

Un uso importante de las excepciones es impedir que se creen objetos cuando no se los puede preparar con un estado inicial válido. Generalmente, este será el resultado del pasaje al constructor de argumentos inadecuados. Podemos ilustrar este punto con la clase `DatosDelContacto`: el constructor actualmente está pasando por alto los valores de los argumentos que recibe: no rechaza valores `null` sino que los reemplaza por cadenas vacías. Sin embargo, la libreta de direcciones necesita como mínimo un nombre o un número de teléfono por cada entrada, para usarlo como valor de índice único, por lo tanto, será imposible usar como índice una entrada que tenga simultáneamente el nombre y el teléfono vacíos. En tal caso, podemos reflejar este requerimiento impidiendo la construcción de un objeto `DatosDelContacto`. El proceso de lanzamiento de una excepción desde un constructor es exactamente el mismo que el lanzamiento desde un método. El Código 12.7 muestra el constructor revisado que impedirá que una entrada tenga el nombre y el teléfono simultáneamente vacíos.

##### Código 12.7

El constructor de la clase  
`DatosDelContacto`

```
/*
 * Prepara los datos del contacto. A todos los datos se
 * les elimina los espacios en blanco al comienzo y al
 * final.
 * El nombre y el teléfono no pueden ser simultáneamente
 * cadenas vacías.
 * @param nombre El nombre.
 * @param telefono El número de teléfono.
 * @param direccion La dirección.
 * @throws IllegalStateException Si el nombre y el
 * teléfono están vacíos.
 */
public DatosDeContacto(String nombre, String telefono,
String direccion)
{
    // Usa cadenas vacías si alguno de los argumentos
    es null.
    if(nombre == null) {
        nombre = "";
    }
    if(telefono == null) {
        telefono = "";
    }
}
```

**Código 12.7  
(continuación)**

El constructor de la clase  
DatosDelContacto

```

        }
        if(direccion == null) {
            direccion = "";
        }
        this.nombre = nombre.trim();
        this.telefono = telefono.trim();
        this.direccion = direccion.trim();

        if(this.nombre.length() == 0 && this.telefono.length()
== 0) {
            throw new IllegalStateException(
                "El nombre y el teléfono no
pueden estar vacíos.");
        }
    }
}

```

Una excepción que se lanza desde un constructor tiene el mismo efecto sobre el cliente que una excepción que se lanza desde un método. En consecuencia, el siguiente intento de crear un objeto DatosDelContacto no válido fallará completamente; no dará por resultado que se almacene un valor null en la variable:

```
DatosDelContacto datosErroneos = new DatosDelContacto("","","");
```

**12.5****Manejo de excepciones**

Los principios del lanzamiento de excepciones se aplican tanto para las excepciones comprobadas como para las no comprobadas, pero las reglas particulares de Java indican que el manejo de una excepción se convierte en un requerimiento sólo en el caso de excepciones comprobadas. Una clase de excepción comprobada es una subclase de `Exception` pero no de `RuntimeException`. Existen varias reglas que se deben seguir cuando se usan excepciones comprobadas porque el compilador obliga a tener controles tanto en los métodos que lanzan una excepción comprobada como en el invocador de dicho método.

**12.5.1****Excepciones comprobadas: la cláusula throws**

El primer requerimiento del compilador es que un método que lanza una excepción comprobada debe declarar que lo hace mediante una *cláusula throws* que se agrega en su encabezado. Por ejemplo, un método que lanza una `IOException` comprobada del paquete `java.io` debe tener el siguiente encabezado<sup>3</sup>:

```
public void grabarEnArchivo(String archivoDestino)
    throws IOException
```

Si bien se permite el uso de la cláusula *throws* para las excepciones no comprobadas, el compilador no lo requiere. Recomendamos que se use una cláusula *throws* solamente para enumerar las excepciones comprobadas que lanza un método.

**Concepto**

Las **excepciones comprobadas** son un tipo de excepción cuyo uso requiere controles adicionales del compilador. En particular, las excepciones comprobadas en Java requieren el uso de cláusulas *throws* y de sentencias *try*.

<sup>3</sup> Observe que aquí la palabra clave es *throws* y no *throw*.

Es importante distinguir entre la cláusula *throws* en el encabezado de un método y la etiqueta que se utiliza en el comentario que precede al método; la última es completamente opcional para ambos tipos de excepción. Sin embargo, recomendamos que se incluya la etiqueta *throws* en la documentación *javadoc* para ambos tipos de excepciones, comprobadas y no comprobadas. De esta manera, se pone a disposición de alguien que quiera usar ese método en particular tanta información como sea posible.

### 12.5.2

### Captura de excepciones: la sentencia *try*

El segundo requerimiento es que el invocador de un método que lanza una excepción comprobada debe proveer un tratamiento para dicha excepción. Esto generalmente implica escribir un *manejador de excepción* bajo la forma de una *sentencia try*. Las sentencias *try* más prácticas tienen la forma general que se muestra en el Código 12.8. Esta sentencia introduce dos nuevas palabras clave de Java, *try* y *catch*, que marcan un *bloque try* y un *bloque catch* respectivamente.

#### Código 12.8

Los bloques *try* y *catch* en un manejador de excepción

```
try {
    Aquí se protege una o más sentencias.
}
catch(Exception e) {
    Aquí se informa y se recupera de la excepción
}
```

#### Concepto

El código de un programa que protege sentencias que podrían lanzar una excepción se denomina **manejador de excepción**. El código proporciona información y/o código para recuperarse del error.

El Código 12.9 ilustra una sentencia *try* formando parte de un método que graba el contenido de una libreta de direcciones en un archivo. Se le pide al usuario de alguna manera el nombre del archivo (quizás mediante una ventana de diálogo IGU) y se invoca al método *grabarEnArchivo* de la libreta para grabar la lista de contactos en un archivo. Dado que el proceso de escritura puede fallar con una excepción, la llamada a *grabarEnArchivo* debe encerrarse en un bloque *try*. Observe que se puede incluir cualquier número de sentencias dentro del bloque *try*. El bloque *catch* intentará capturar las excepciones de cualquiera de las sentencias que están dentro del bloque *try* precedente.

#### Código 12.9

Un manejador de excepción

```
String nombreDeArchivo = null;
try {
    nombreDeArchivo = nombre-que-se-pide-al-usuario;
    libreta.grabarEnArchivo(nombreDeArchivo);
}
catch(IOException e) {
    System.out.println("Imposible grabar en " +
nombreDeArchivo);
}
```

En vías de comprender cómo funciona un manejador de excepción es esencial apreciar que una excepción impide que el invocador continúe con el control normal del flujo. Una excepción interrumpe la ejecución de la sentencia del invocador que la causó y de aquí en adelante, tampoco se ejecutará cualquier sentencia que esté inmediatamente a continuación de la sentencia que produjo el problema. La pregunta que surge entonces es, «¿Dónde continúa la excepción en el invocador?». La sentencia *try* proporciona la respuesta: si se dispara una excepción desde una sentencia invocada dentro del bloque *try*, la ejecución continúa en el correspondiente bloque *catch*. En consecuencia, si consideramos el ejemplo del Código 12.9, el efecto de que se lance una *IOException* en la llamada a *grabarEnArchivo* será que el control se transferirá desde el bloque *try* hacia el bloque *catch*, tal como se muestra en el Código 12.10.

Las sentencias ubicadas dentro de un bloque *try* se conocen como *sentencias protegidas*. Si no se dispara ninguna excepción durante la ejecución de las sentencias protegidas, entonces se saltará el bloque *catch* cuando se llegue al final del bloque *try*. La ejecución continuará con cualquier sentencia que esté a continuación de la sentencia *try* completa.

1. La excepción se lanza desde aquí                    2. El control se transfiere aquí

```
Código 12.10  
Transferencia del  
control en una  
sentencia try
```

```
try {
    libreta.grabarEnArchivo(nombreDeArchivo);
    probarNuevoamente = false;
}
catch(IOException e) {
    System.out.println("Imposible grabar en " +
        nombreDeArchivo);
    probarNuevoamente = true;
}
```

El bloque *catch* nombra el tipo de excepción que tiene designado tratar dentro de un par de paréntesis inmediatamente a continuación de la palabra *catch*. Así como el nombre del tipo de la excepción, también incluye un nombre de variable (tradicionalmente, simplemente «*e*») que se puede usar para hacer referencia al objeto excepción que fue lanzado. Una referencia a este objeto puede ser muy útil para proporcionar la información que se podrá usar para recuperarse del problema. Una vez que se completó el bloque *catch*, el control *no* retorna a la sentencia que causó la excepción.

**Ejercicio 12.26** El proyecto *libreta-de-direcciones-v3* incluye algunos lanzamientos de excepciones no comprobadas cuando los valores de los argumentos son *null*. El proyecto también incluye la clase de excepción comprobada *NoCoincidenciaException* que actualmente no se usa. Modifique el método *eliminarContacto* de *LibretaDeDirecciones* de modo que lance esta excepción si su argumento clave no es una clave que está en uso. Agregue un manejador de excepción al método *eliminar* de *LibretaDeDireccionesInterfazDeTexto* para capturar e informar las ocurrencias de esta excepción.

**Ejercicio 12.27** Utilice la excepción `NoCoincideContactoException` en el método `modificarContacto` de la `LibretaDeDirecciones`. Mejore la interfaz de usuario de modo que puedan modificarse los datos de una entrada existente. Capture e informe las excepciones en `LibretaDeDireccionesInterfazDeTexto` que surgen del uso de una clave que no coincide con ninguna de las entradas existentes.

**Ejercicio 12.28** ¿Por qué el siguiente código no es una manera sensata de usar un manejador de excepción?

```
Persona p;
try {
    p = baseDeDatos.buscar(contacto);
}
catch(Exception e) {
}
System.out.println("Los datos pertenecen a: " + p);
```

### 12.5.3 Lanzar y capturar varias excepciones

Algunas veces, un método lanza más de un tipo de excepción para indicar diferentes tipos de problemas. Cuando se trate de excepciones comprobadas deben enumerarse todas en la cláusula `throws` del método, separadas por comas. Por ejemplo:

```
public void procesar()
    throws EOException, FileNotFoundException
```

Un manejador de excepción debe capturar todas las excepciones comprobadas que se lanzan desde sus sentencias protegidas, de modo que una sentencia `try` puede contener varios bloques `catch`, tal como se muestra en el Código 12.11. Observe que se puede usar el mismo nombre de variable en cada caso, para el objeto excepción.

#### Código 12.11

Varios bloques `catch` en una sentencia `try`

```
try {
    ...
    ref.procesar();
    ...
}
catch(EOException e) {
    // Tomar las medidas apropiadas para una excepción
    fin-de-archivo
    ...
}
catch(FileNotFoundException e) {
    // Tomar las medidas apropiadas para una excepción
    archivo-no-encontrado
    ...
}
```

Cuando se lanza una excepción mediante una llamada a método dentro de un bloque `try`, los bloques `catch` se evalúan en el orden en que están escritos hasta que se encuentra una coincidencia en el tipo de excepción. Por lo tanto, si se lanza una

`EOFException` entonces el control se transferirá al primer bloque *catch* y si se lanza una `FileNotFoundException` el control se transferirá al segundo. Una vez que se llega al final de un único bloque *catch*, la ejecución continúa debajo del último bloque *catch*.

Si se desea, se puede usar polimorfismo para evitar la escritura de varios bloques *catch*. Sin embargo, esto puede ser a expensas de ser capaz de tomar medidas de recuperación de un tipo específico. En el Código 12.12, el único bloque *catch* manejará *cualquier* excepción lanzada por las sentencias protegidas. Esto es así porque el proceso de coincidencia de excepciones que busca un bloque *catch* adecuado controla simplemente que el objeto excepción sea una instancia del tipo nombrado en el bloque. Como todas las excepciones son subtipos de la clase `Exception`, el único bloque *catch* las capturará a todas, ya sean comprobadas o no comprobadas. Del proceso natural de coincidencias se desprende que es importante el orden de los bloques *catch* en una única sentencia *try* y que un bloque *catch* para un tipo de excepción en particular no puede estar a continuación de uno de sus supertipos; porque el bloque del supertipo anterior siempre coincidirá antes con el bloque del subtipo que se controla.

**Código 12.12**

Capturar todas las excepciones en un solo bloque *catch*

```
try {  
    ...  
    ref.procesar();  
    ...  
}  
catch(Exception e) {  
    // Tomar las medidas adecuadas para todas las  
    excepciones  
    ...  
}
```

**Ejercicio 12.29** Mejore las sentencias *try* que escribió como soluciones de los ejercicios 12.26 y 12.27 de modo que incluyan el manejo de excepciones comprobadas y no comprobadas en diferente bloques *catch*.

**Ejercicio 12.30** ¿Qué está mal en la siguiente sentencia *try*?

```
try {  
    Persona p = baseDeDatos.buscar(datos);  
    System.out.println(_Los datos pertenecen a: _ + p);  
}  
catch(Exception e) {  
    // Maneja cualquiera de las excepciones comprobadas  
    ...  
}  
catch(RuntimeException e) {  
    // Maneja cualquiera de las excepciones no comprobadas  
    ...  
}
```

## 12.5.4 Propagar una excepción

Hasta ahora, hemos sugerido que una excepción debe ser capturada y manejada en la oportunidad más temprana posible. Es decir, una excepción lanzada en un método *procesar* debe ser capturada y manejada en el método que invocó a procesar. En la realidad, este no es estrictamente el caso ya que Java permite que una excepción se *propague* desde el método receptor hasta su invocador y posiblemente, más allá. Un método propaga una excepción simplemente al no incluir un manejador de excepción para proteger la sentencia que puede lanzarla. Sin embargo, para una excepción comprobada, el compilador requiere que el método propagador incluya una cláusula *throws* aun cuando no lance en sí mismo una excepción. Si la excepción es no comprobada, la cláusula *throws* es opcional y preferimos omitirla.

La propagación es común en los lugares en que el método invocador es incapaz de tomar una medida de recuperación o bien, no necesita ninguna, pero esto podría ser posible o necesario dentro de llamadas de nivel más alto.

## 12.5.5 La cláusula *finally*

Una sentencia *try* puede incluir un tercer componente que es opcional: la cláusula *finally* (Código 12.13) que se omite con frecuencia. La cláusula *finally* se proporciona para sentencias que se deben ejecutar cuando se lanza una excepción desde sentencias protegidas o desde sentencias no protegidas. Si el control alcanza el final del bloque *try* entonces se saltea el bloque *catch* y se ejecuta la cláusula *finally*. Recíprocamente, si se lanza una excepción a partir del bloque *try*, entonces se ejecuta el bloque *catch* apropiado y luego se sigue con la ejecución de la cláusula *finally*.

### Código 12.13

Una sentencia *try* con una cláusula *finally*

```
try {
    Aquí se protegen una o más sentencias
}
catch (Exception e) {
    Aquí se informa la excepción y se recupera de la misma
}
finally {
    Se realizan acciones comunes, se haya o no
    lanzado una excepción.
}
```

A primera vista, una cláusula *finally* puede parecer redundante. El siguiente código ¿no ilustra el mismo control de flujo que el Código 12.13?

```
try {
    Aquí se protegen una o más sentencias
}
catch (Exception e) {
    Aquí se informa la excepción y se recupera de la misma
}
Se realizan acciones comunes, se haya o no
lanzado una excepción.
```

De hecho, existen por lo menos dos casos en los que estos dos ejemplos tendrán efectos diferentes:

- Se ejecuta una cláusula *finally* aun si se ejecuta una sentencia *return* en los bloques *try* o *catch*.
- Si se lanza una excepción en el bloque *try* pero no se captura, entonces también se ejecuta la cláusula *finally*.

En el último caso, la excepción no capturada podría ser una excepción no comprobada que no requiere un bloque *catch*, por ejemplo. Sin embargo, también podría ser una excepción comprobada que no se maneja mediante un bloque *catch* pero que se propaga desde un método. En tal caso, la cláusula *finally* aún podría ser ejecutada. Como consecuencia, es posible que no se tenga ningún bloque *catch* en una sentencia *try* que tiene un bloque *try* y una cláusula *finally*:

```
try {
    Aquí se protegen una o más sentencias
}
finally {
    Se realizan acciones comunes, se haya o no
    lanzado una excepción.
}
```

## 12.6

## Definir nuevas clases de excepción

Cuando las clases estándares de excepciones no describen satisfactoriamente la naturaleza del problema, se pueden definir nuevas clases más descriptivas usando herencia. Las nuevas clases de excepciones comprobadas pueden definirse como subclases de una clase de excepción comprobada existente (tal como `Exception`) y las nuevas excepciones no comprobadas debieran ser subclases de la jerarquía `RuntimeException`.

Todas las clases de excepción existentes soportan la inclusión de una cadena de diagnóstico que se pasa al constructor. Sin embargo, una de las principales razones para definir nuevas clases de excepción es la inclusión de más información dentro del objeto excepción para brindar el diagnóstico de error y de recuperación. Por ejemplo, algunos métodos en la aplicación libreta de direcciones, tal como `modificarContacto`, tiene un argumento clave que debe coincidir con una entrada existente. Si no se puede encontrar ninguna entrada que coincida, esto representa un error de programación ya que el método no puede completar su tarea. En el informe de la excepción es muy útil incluir detalles de la clave que causó el error. El Código 12.14 muestra una nueva clase de excepción comprobada que está definida en el proyecto `libreta-de-direcciones-v3t`. Recibe la clave en su constructor y luego la vuelve disponible a través de dos maneras, a través de la cadena de diagnóstico y de un método de acceso dedicado. Si esta excepción fuera capturada por un manejador de excepciones, la clave debiera estar disponible para las sentencias que intentan recuperarse del error.

### Código 12.14

Una clase excepción con información adicional de diagnóstico

```
/**
 * Captura una clave que falló al buscar una coincidencia
 * con una entrada en la libreta de direcciones.
 */
```

**Código 12.14  
(continuación)**

Una clase excepción con información adicional de diagnóstico

```

 * @author David J. Barnes and Michael Kölling.
 * @version 2006.03.30
 */
public class NoCoincideContactoException extends Exception
{
    // La clave que no tiene coincidencias.
    private String clave;
    /**
     * Almacena los datos erróneos.
     * @param clave La clave que no coincide.
     */
    public NoCoincidenContactoException(String clave)
    {
        this.clave = clave;
    }
    /**
     * @return La clave errónea.
     */
    public String getClave()
    {
        return clave;
    }

    /**
     * @return Una cadena de diagnóstico que contiene
     * la clave errónea.
     */
    public String toString()
    {
        return "No se encontraron datos que coincidan
con : " + clave ;
    }
}

```

El principio de incluir información que podría colaborar en la recuperación del error debe tenerse en cuenta particularmente cuando se definen nuevas clases de excepción comprobadas. La definición de los parámetros formales del constructor de una excepción ayudará a asegurar que la información de diagnóstico esté disponible. Además, cuando la recuperación no sea posible o no se intente, asegura que se sobrescriba el método `toString` de la excepción de modo que incluya la información adecuada y de esta manera, ayudará a diagnosticar el motivo del error.

**Ejercicio 12.31** En el proyecto *libreta-de-direcciones-v3t* defina una nueva clase de excepción comprobada: `ClaveDuplicadaException`. Debe ser lanzada por el método `agregarContacto` si cualquiera de los campos clave no vacíos de sus argumentos está actualmente en uso. La clase excepción debe almacenar los datos de la clave que se intentó usar. Realice cualquier otro cambio que sea necesario en la clase de la interfaz de usuario, para capturar e informar la excepción.

**Ejercicio 12.32** ¿Le parece que `ClaveDuplicadaException` debiera ser comprobada o no comprobada? Justifique los motivos de su respuesta.

## 12.7

## Usar aserciones

### 12.7.1 Controlar la consistencia interna

Cuando diseñamos o implementamos una clase frecuentemente tenemos un sentido intuitivo de las cosas que deben ser ciertas en un punto dado de la ejecución, pero raramente las establecemos formalmente. Por ejemplo, podemos esperar que el objeto `DatosDelContacto` siempre contenga como mínimo un campo no vacío o que, cuando se invoque el método `eliminarContacto` con una clave particular, esperamos que esa clave no esté más en uso al finalizar el método. Típicamente estas son condiciones que deseamos establecer mientras desarrollamos una clase, antes de liberarla. En un sentido, los tipos de pruebas que discutimos en el Capítulo 6 son un intento de determinar si hemos implementado una representación correcta de lo que la clase o un método debe hacer. Las características de ese estilo de prueba es que las pruebas son *externas* a la clase que está siendo probada. Si una clase se modifica, entonces es el momento de ejecutar pruebas regresivas en vías de establecer que aún funciona como debe y esto es muy fácil de olvidar. La práctica de controlar los argumentos que hemos introducido en este capítulo cambia ligeramente el énfasis desde el control completamente externo hacia una combinación de control interno y externo.

Sin embargo, el control de argumentos se intenta primordialmente para proteger a un objeto servidor de ser usado incorrectamente por un objeto cliente. Esto deja aún de lado la cuestión de si debemos incluir algunos controles internos para asegurar que el objeto servidor se comporte como es debido.

Una manera en que podríamos implementar el control interno durante el desarrollo sería a través del mecanismo normal de lanzamiento de excepciones. En la práctica debemos usar excepciones no comprobadas porque no podemos esperar que las clases cliente regulares incluyan manejadores de excepciones para aquellos casos que son esencialmente errores internos del servidor. Nos enfrentamos con la cuestión de eliminar estos controles internos una vez que se completó el proceso de desarrollo para evitar el potencialmente alto costo de estos controles en tiempo de ejecución que casi seguro van a pasar desapercibidos.

### 12.7.2 La sentencia assert

Para satisfacer la necesidad de llevar a cabo controles eficientes de la consistencia interna que puedan permanecer activos durante el desarrollo del código pero desactivados cuando se lo libera, se introdujo la *facilidad de aserción* en la versión 1.4 del SDK de Java. El proyecto *libreta-de-direcciones-assert* es una versión desarrollada de los proyectos *libreta-de-direcciones* que ilustra cómo se utilizan las aserciones. El Código 12.15 muestra el método `eliminarContacto`, que contiene dos formas de la sentencia `assert`.

**Código 12.15**

Usar asercciones para controlar la consistencia interna

```
/*
 * Elimina una entrada de la libreta de direcciones con
 * la clave dada.
 * La clave debe ser una de las que están actualmente en
 * uso.
 * @param clave Una de las claves de entrada a eliminar.
 * @throws IllegalArgumentException Si la clave es null.
 */
public void eliminarContacto(String clave)
{
    if(clave == null){
        throw new IllegalArgumentException(
            "Se pasó clave null a
eliminarContacto.");
    }
    if(claveEnUso(clave)) {
        DatosDelContacto contacto = libreta.get(clave);
        libreta.remove(contacto.getNombre());
        libreta.remove(contacto.getTelefono());
        numeroDeEntradas--;
    }
    assert !claveEnUso(clave);
    assert tamanioConsistente() :
        "El tamaño de la libreta es inconsistente
en eliminarContacto";
}
```

**Concepto**

Una **aserción** es la afirmación de un hecho que debe ser verdadero en la ejecución normal del programa. Podemos usar asercciones para establecer explícitamente lo que asumimos y para detectar errores de programación más fácilmente.

La palabra clave `assert` va seguida de una expresión booleana. El propósito de la sentencia es afirmar algo que debe ser verdadero en este punto del método. Por ejemplo, la primera sentencia `assert` en el Código 12.15 afirma que `claveEnUso` debe retornar, en este punto, un valor `false` ya sea porque la clave no estaba en uso en el primer lugar o bien porque no está más en uso pues se eliminaron de la libreta los datos asociados a ella. Esta afirmación aparentemente obvia es más importante de lo que podría parecer en un principio; observe que el proceso de eliminación no involucra realmente el uso de la clave con la libreta de direcciones.

De esta manera, una sentencia `assert` cumple con dos propósitos. Por un lado, expresa explícitamente lo que asumimos como verdadero en un punto determinado de la ejecución y por lo tanto, aumenta la legibilidad tanto del desarrollador actual como la del futuro programador de mantenimiento y, por otro lado, realmente realiza el control de modo que nos notifica si el valor que asumimos no fue el correcto. Esta sentencia puede ser de gran ayuda para encontrar errores temprana y fácilmente.

Si la expresión booleana en una sentencia `assert` se evalúa `true`, entonces la sentencia `assert` no tiene más efecto; si la sentencia se evalúa `false` se lanzará un `AssertionError`. Este último es una subclase de `Error` (véase Figura 12.1) y forma parte de la jerarquía que representa errores irrecuperables: no se debe proveer ningún manejador a los clientes.

La segunda sentencia *assert* en el Código 12.15 ilustra la forma alternativa de una sentencia *assert*. La cadena seguida de un punto y coma se pasará al constructor de *AssertionError* para ofrecer una cadena de diagnóstico. La segunda expresión no tiene por qué ser una cadena explícita, puede ser cualquier expresión con un valor determinado que se convertirá en un *String* antes de ser pasada al constructor.

La primera sentencia *assert* muestra que una aserción usará frecuentemente métodos que ya existen en la clase (*claveEnUso*). El segundo ejemplo ilustra que puede ser de utilidad proporcionar un método específico para los fines de llevar a cabo una prueba de aserción (en este ejemplo, *tamanioConsistente*). Esta modalidad podría usarse cuando el control involucra cálculos significativos. El Código 12.16 muestra el método *tamanioConsistente* cuyo propósito es asegurar que el campo *numeroDeEntradas* represente correctamente el número de contactos que hay en la libreta de direcciones.

### Código 12.16

Control de consistencia interna en la libreta de direcciones

```
/*
 * Controla que el campo numeroDeEntradas sea consistente
 * con
 *   * el número de entradas actualmente almacenadas en la
 *     libreta.
 *   * @return true si el campo es inconsistente, false en
 *     caso contrario.
 */
private boolean tamanioConsistente()
{
    Collection<DatosDelContacto> todasLasEntradas =
libreta.values();
    // Elimina los duplicados ya que se usan claves
múltiples.
    Set<DatosDelContacto> entradasUnicas =
new HashSet<DatosDelContacto>(todasLasEntradas);
    int cantidadActual = entradasUnicas.size();
    return numeroDeEntradas == cantidadActual;
}
```

### 12.7.3 Pautas para usar aserciones

Las aserciones están pensadas primordialmente para ofrecer una forma de realizar controles de consistencia durante las fases de desarrollo y de prueba de un proyecto. No están pensadas para ser usadas en el código liberado. Es por este motivo que el compilador de Java incluirá las sentencias *assert* en el código compilado sólo si se lo solicitamos. Se desprende que las sentencias *assert* nunca deben usarse para implementar la funcionalidad normal de un programa. En la libreta de direcciones sería incorrecto combinar aserciones con la eliminación de contactos, como se hace en el siguiente ejemplo:

```
// Error: no utilice sentencias assert en un procesamiento
//        normal!
assert libreta.remove(contacto.getNombre()) != null;
assert libreta.remove(contacto.getTelefono()) != null;
```

**Ejercicio 12.33** Abra el proyecto *libreta-de-direcciones-assert*. Recorra la clase *LibretaDeDirecciones* e identifique todas las sentencias *assert*, para asegurarse de que comprende lo que controlan y por qué lo hacen.

**Ejercicio 12.34** La clase *LibretaDeDireccionesDemo* contiene varios métodos de prueba que invocan a métodos de *LibretaDeDirecciones* que contienen sentencias *assert*. Recorra el código de *LibretaDeDirecciones-Demo* para verificar que comprende las pruebas y luego ejecute cada uno de los métodos de prueba. ¿Se generó algún error de aserción? De ser así, ¿comprende por qué?

**Ejercicio 12.35** El método *modificarContacto* de *LibretaDeDirecciones* no tiene actualmente sentencias *assert*. Una aserción que podríamos hacer en este método es que la libreta debe contener al finalizar el método, el mismo número de entradas que al principio. Agregue una sentencia *assert* (y cualquier otra sentencia que necesite) para controlar esta cuestión. Después de realizar la modificación, ejecute el método *testModificar* de *LibretaDeDireccionesDemo*. ¿Considera que este método debería incluir el control de la consistencia del tamaño?

**Ejercicio 12.36** Suponga que decidimos permitir que la libreta de direcciones se indexe por dirección, así como por nombre y por número de teléfono. Si simplemente agregamos la siguiente sentencia al método *agregarContacto*

```
libreta.put(contacto.getDireccion(), contacto);
```

¿Puede anticipar cuáles de las asercciones fracasarán? Pruébelo. Realice en *LibretaDeDirecciones* cualquier otro cambio que necesite para asegurarse de que todas las asercciones sean exitosas.

**Ejercicio 12.37** Los objetos *DatosDelContacto* son inmutables, es decir, no tienen métodos de modificación. ¿Qué importancia tiene este hecho con respecto a la consistencia interna de *LibretaDeDirecciones*? Por ejemplo, suponga que la clase *DatosDelContacto* tiene un método *setTelefono*. ¿Puede anticipar algunas pruebas para ilustrar los problemas que podría causar la inclusión de este método?

## 12.7.4 Aserciones y el marco de trabajo de unidades de prueba de BlueJ

En el Capítulo 6 presentamos el soporte que ofrece BlueJ para el marco de trabajo de unidades de prueba JUnit. Este soporte se basa en la facilidad de aserción que hemos discutido en esta sección. Los métodos del marco de trabajo, como por ejemplo *assertEquals*, están construidos con sentencias *assert* que contienen una expresión booleana que se arma a partir de sus argumentos. Cuando se usan las clases de prueba de JUnit para probar clases que contienen sus propias sentencias de aserción, se informan los errores de aserción que surgen de estas sentencias en la ventana de resultados de pruebas, con los fracasos de aserción de la clase de prueba. El proyecto *libreta-de-direcciones-junit* contiene una clase de prueba para ilustrar esta combinación. El método *testAgregarContactoError* de *LibretaDeDireccionesTest* disparará un error de aserción porque no debe usarse *agregarContacto* para modificar los datos de un contacto que ya existe (véase Ejercicio 12.31).

## 12.8

# Recuperarse del error y anularlo

Hasta ahora, el foco principal de este capítulo ha sido el problema de la identificación de errores en un objeto servidor y la seguridad de que se informe cualquier problema al cliente, si es apropiado. Existen dos cuestiones complementarias al informe de errores: la recuperación del error y la anulación del error.

### 12.8.1

#### Recuperarse del error

El primer requerimiento de una recuperación exitosa del error es que los clientes tomen nota de cualquier notificación de error que reciban. Esto puede sonar obvio, pero no es poco común que algunos programadores asuman que una llamada a método no fallará y por lo tanto, no se preocupen por controlar el valor que retorna. Aunque es difícil que se ignoren los errores cuando se usan excepciones, hemos visto con frecuencia segmentos de código equivalentes al siguiente abordaje del manejo de las excepciones:

```
DatosDelContacto contacto = null;
try{
    contacto = libreta.getContacto(...);
}
catch(Excepction e) {
    System.out.println("Error: " + e);
}
String telefono = contacto.getTelefono();
```

Se captura y se informa la excepción pero no se toma ninguna medida respecto del hecho de que probablemente sea incorrecto continuar indiferente.

La sentencia *try* de Java es la clave para proporcionar un mecanismo de recuperación del error cuando se lanza una excepción. La recuperación de un error implicará generalmente, tomar alguna medida correctiva dentro del bloque *catch* y luego, probar nuevamente. Se pueden repetir los intentos ubicando la sentencia *try* dentro de un ciclo. Se muestra un ejemplo de este enfoque en el Código 12.17 que es una versión expandida del Código 12.9. Los esfuerzos para armar un nombre de archivo alternativo involucran, por ejemplo, tratar una lista de posibles carpetas, o solicitar interactivamente al usuario diferentes nombres.

#### Código 12.17

Un intento de recuperación del error

```
// Se intenta grabar la libreta de direcciones.
boolean exito = false;
int intentos = 0;
do {
    try {
        libreta.grabarEnArchivo(nombreDeArchivo);
        exito = true;
    }
    catch(IOException e) {
        System.out.println("Imposible grabar en " +
nombreDeArchivo);
        intentos++;
    }
}
```

**Código 12.17****(continuación)**

Un intento de recuperación del error

```

        if(intentos < MAX_INTENTOS) {
            nombreDeArchivo = un nombre de archivo
            alternativo;
        }
    } while (!exito && intentos < MAX_INTENTOS);
if (!exito) {
    Informar el problema y rendirse.
}

```

Aunque este ejemplo ilustra la recuperación para una situación específica, los principios que ilustra son más generales:

- La anticipación de un error y la recuperación del mismo, generalmente requerirán un control de flujo más complejo que si el error no pudiera ocurrir.
- Las sentencias del bloque *catch* son la clave para preparar el intento de recuperación.
- La recuperación frecuentemente implica probar nuevamente.
- No se puede garantizar el éxito de la recuperación.
- Debe haber algunas rutas de escape que eviten que el intento de recuperación se realice desesperada y eternamente.

No siempre habrá un usuario humano al que se le pueda pedir un ingreso alternativo. Registrar el error debiera ser responsabilidad del cliente.

## 12.8.2 Anular el error

Debe quedar claro que llegar a una situación que lanza una excepción será, en el peor de los casos, fatal para la ejecución de un programa y, en el mejor caso, difícil de recuperar desde el cliente. En primer lugar, puede ser más simple tratar de evitar el error, pero esto generalmente requiere de la colaboración entre el servidor y el cliente.

Muchos de los casos en los que se fuerza al objeto *LibretaDeDirecciones* a lanzar una excepción involucran el pasaje de argumentos con valores null a sus métodos. Esto representa errores lógicos de programación en el cliente que deben claramente evitarse con anterioridad mediante pruebas. Los argumentos null son generalmente el resultado de asumir cuestiones no válidas en el cliente. Considere el siguiente ejemplo:

```

String clave = codigoPostalBaseDeDatos.buscar(codigoPostal);
DatosDelContacto universidad = libreta.getContacto(clave);
...

```

Si la búsqueda en la base de datos fracasa, entonces la clave puede retornar en blanco o null. El pasaje de este resultado directamente al método *getContacto* producirá una excepción en tiempo de ejecución. Sin embargo, usando una simple prueba del resultado de la búsqueda puede evitarse la excepción y se puede registrar el problema real de un fracaso en la búsqueda del código postal de la siguiente manera:

```

String clave = codigoPostalBaseDeDatos.buscar(codigoPostal);
If (clave != null) && clave.length() > 0) {

```

```

        DatosDelContacto universidad = libreta.getContacto(clave);
        ...
    }
    else {
        Tratar el error del código postal...
    }
}

```

En este caso el cliente puede decidir por su cuenta si puede ser inapropiado invocar al método del servidor, cosa que no siempre es posible y en algunos casos, el cliente debe anotarse para contar con la ayuda del servidor.

El Ejercicio 12.31 estableció el principio de que el método agregarContacto no debe aceptar un nuevo conjunto de datos si una de las claves ya está en uso por otro conjunto. En vías de evitar una llamada inapropiada, el cliente debería usar el método claveEnUso de la libreta de direcciones de la siguiente manera:

```

// Agregar cómo debe ser un nuevo conjunto de datos para
// la libreta.
if (libreta.claveEnUso(contacto.getNombre()) {
    libreta.modificarContacto(contacto.getNombre(), contacto);
}
else if (libreta.claveEnUso(contacto.getTelefono())) {
    libreta.modificarContacto(contacto.getTelefono(), contacto);
}
else {
    Agregar el contacto...
}

```

El uso de este abordaje hace claramente posible que se evite por completo el lanzamiento de una `DuplicateKeyException` en `agregarContacto`, lo que sugiere que se debe pasar de una excepción comprobada a una no comprobada.

Este ejemplo en particular ilustra algunos principios generales importantes:

- Si el control de validación del servidor y los métodos de prueba del estado están visibles para un cliente, el cliente, generalmente, estará capacitado para evitar las causas que producen que el servidor lance una excepción.
- Si se puede evitar una excepción de esta manera, entonces la excepción que se lance realmente representa un error lógico de programación en el cliente. Esto sugiere el uso de excepciones no comprobadas para tales situaciones.
- Usar excepciones no comprobadas significa que el cliente no tiene que usar una sentencia `try` cuando ya se estableció que no se lanzará la excepción. Esta es una ganancia significativa porque tener que escribir sentencias `try` para situaciones que «no pueden ocurrir» es molesto para el programador y hace menos probable que se tenga seriamente en cuenta la provisión de una recuperación adecuada para las situaciones genuinas de error.

Sin embargo, los efectos no son todos positivos. Aquí hay algunas razones por las que este enfoque no siempre es práctico:

- Hacer que los controles de validación del servidor y los métodos de prueba de estado sean públicamente visibles para sus clientes, podría representar una pérdida significativa del encapsulamiento y dar por resultado un grado de acoplamiento más alto entre el servidor y el cliente que no es deseable.

- Probablemente no sea seguro que un servidor asuma que sus clientes *harán* los controles necesarios para evitar una excepción. Como resultado, esos controles frecuentemente estarán duplicados en ambos, cliente y servidor. Si los controles son computacionalmente «caros» de hacer entonces la duplicación puede ser indeseable o prohibitiva. Sin embargo, desde nuestro punto de vista, es mejor sacrificar la supuesta eficiencia en función de programación más segura, cuando esta elección sea posible.

## 12.9

## Estudio de caso: entrada/salida de texto

Un área importante de programación en la que no se puede ignorar la recuperación del error es la relacionada con las operaciones de entrada/salida (E/S), pues el programador de una aplicación puede tener menos control directo sobre el ambiente externo en el que se ejecuta. Por ejemplo, el archivo de datos que requiere cierta aplicación puede haber sido borrado accidentalmente o se puede haber corrompido de alguna manera, antes de que la aplicación se ejecute; o puede frustrarse un intento de guardar resultados en el sistema de archivos porque excede el límite de archivos posibles. Existen varias maneras en las que puede fallar una operación de E/S.

El API de Java incluye el paquete `java.io` que contiene numerosas clases para implementar operaciones de E/S independientes de la plataforma en que se realicen. El paquete define la clase de excepción comprobada `IOException`, como un indicador general de que algo anduvo mal en una operación de E/S. Otras clases de excepción proveen información de diagnóstico más detallada, como por ejemplo: `EOFException` y `FileNotFoundException`.

Está fuera del alcance de este libro la descripción completa de las diferentes clases del paquete `java.io` pero ofreceremos un estudio de caso corto sobre cómo se podrían agregar operaciones de E/S de texto en la aplicación libreta de direcciones. A través de este estudio podrá obtener suficiente conocimiento como para que pueda experimentar con E/S en sus propios proyectos. En particular, mediante el proyecto *libreta-de-direcciones-io* ilustraremos las siguientes tareas comunes:

- escribir salida de texto en un archivo mediante la clase `FileWriter`;
- leer entradas de texto desde un archivo mediante las clases `FileReader` y `BufferedReader`;
- anticipar el lanzamiento de excepciones `IOException` desde las clases de E/S.

Además, el proyecto incluye métodos para leer y escribir versiones binarias de los objetos `LibretaDeDirecciones` y `DatosDelContacto` para que pueda explorar la característica de *serialización* de Java.

Para avanzar en la lectura sobre E/S en Java recomendamos el Tutorial de Sun que se puede encontrar online en:

<http://java.sun.com/docs/books/tutorial/essential/io/index.html>

### 12.9.1

### Lectores, escritores y flujos

Varias de las clases del paquete `java.io` se ubican dentro de dos categorías principales: aquellas que operan con archivos de texto y las que operan con archivos binarios. Podemos

pensar en los archivos de texto como archivos que contienen datos de manera similar al tipo `char` de Java; típicamente contienen líneas de información alfanumérica simple, legible para los humanos. Los archivos binarios son más variados: uno de los ejemplos comunes son los archivos de imagen aunque también lo son los programas ejecutables, como por ejemplo, los procesadores de texto. Las clases comprometidas con archivos de texto se conocen como *lectores* y *escritores* mientras que las comprometidas con los archivos binarios se conocen como *manejadores de flujo*<sup>4</sup>. En este estudio de caso, haremos foco exclusivamente sobre los lectores y los escritores.

## 12.9.2 El proyecto *libreta-de-direcciones-io*

El proyecto *libreta-de-direcciones-io* es una versión de la aplicación libreta de direcciones a la que, por razones de simplicidad, se le eliminó la interfaz de usuario. Incluye la clase adicional `LibretaManejadorDeArchivos`, parte de la cual se muestra en Código 12.18, cuyo único propósito es proporcionar operaciones de manejo de archivos sobre el objeto `LibretaDeDirecciones`. Las operaciones de manejo de archivo incluyen cargar el contenido de la libreta de direcciones desde un archivo, grabar nuevamente su contenido y grabar los resultados de una operación de búsqueda en la libreta de direcciones.

### Código 12.18

La clase  
`LibretaManejador-`  
`DeArchivos`

```
import java.io.*;
import java.net.URISyntaxException;
import java.net.URL;
/**
 * Proporciona algunas operaciones de manejo de archivos
 * sobre
 *   * la LibretaDeDirecciones.
 *   * Estos métodos demuestran algunas características
 * básicas del paquete
 *   * java.io
 *
 *   * @author David J. Barnes and Michael Kölling.
 *   * @version 2006.03.30
 */
public class LibretaManejadorDeArchivos
{
    // Libreta de direcciones sobre la que se realizarán
    // las
    // operaciones de E/S.
    private LibretaDeDirecciones libreta;
    // Nombre del archivo que se usa para almacenar los
    // resultados de
    // la búsqueda.
    private static final String ARCHIVO_RESULTADOS =
"resultados.txt";
```

<sup>4</sup> N. del T. En Java, un flujo se conoce bajo el término *stream* y es una abstracción que representa todo aquello que consume o produce información.

**Código 12.18  
(continuación)**

La clase  
LibretaManejador-  
DeArchivos

```
    /**
     * Constructor de objetos de la clase
     LibretaManejadorDeArchivos
     * @param libreta La libreta de direcciones que se
     va a usar.
     */
    public LibretaManejadorDeArchivos(LibretaDeDirecciones
libreta)
    {
        this.libreta = libreta;
    }

    /**
     * Graba los resultados de una búsqueda en la
     libreta en
     * el archivo "resultados.txt" situado en la carpeta
     del proyecto.
     * @param prefijo El prefijo de la clave a buscar.
     */
    public void grabarResultadoDeBusqueda(String prefijo)
throws IOException
{
    File archivoResultados =
        crearNombreDeArchivoAbsoluto(ARCHIVO_RESULTADOS);
    DatosDelContacto[] resultados =
libreta.buscar(prefijo);
    FileWriter escritor = new
FileWriter(archivoResultados);
    for(DatosDelContacto contacto : resultados) {
        escritor.write(resultados[i].toString());
        escritor.write('\n');
        escritor.write('\n');
    }
    escritor.close();
}

/**
 * Muestra los resultados de la llamada más reciente a
 * grabarResultadoDeBusqueda. La salida es en la
consola,
 * se informa cualquier problema directamente desde
este método.
 */
public void mostrarResultadoDeBusqueda()
{
    BufferedReader lector = null;
    try {
        File archivoResultados =
crearNombreDeArchivoAbsoluto(ARCHIVO_RESULTADOS);
```

**Código 12.18  
(continuación)**

La clase  
LibretaManejador-  
DeArchivos

```

lector = new BufferedReader(
                    new
FileReader(archivoResultados));
                    System.out.println("Resultados ...");
                    String linea;
                    linea = lector.readLine();
                    while(linea != null) {
                        System.out.println(linea);
                        linea = lector.readLine();
                    }
                    System.out.println();
}
catch(FileNotFoundException e) {
    System.out.println("Imposible encontrar el
archivo: " +
ARCHIVO_RESULTADOS);
}
catch(IOException e) {
    System.out.println("Se encontró un error
al leer el archivo: " +
ARCHIVO_RESULTADOS);
}
finally {
    if(lector != null) {
        // Captura cualquier excepción pero
no se puede
        // hacer nada con ella.
        try {
            lector.close();
        }
        catch(IOException e) {
            System.out.println("Error al
cerrar: " +
ARCHIVO_RESULTADOS);
        }
    }
}
// Se omiten los restantes métodos...
}

```

La clase para manejar archivos está fuertemente acoplada con la clase para la libreta de direcciones, por lo que se podría intuir que estas dos clases debieran conformar una sola clase. Sin embargo, manteniéndolas en clases separadas, se logra que cada una de ellas resulte más cohesiva. Además, al no embeber las operaciones de E/S directamente dentro de `LibretaDeDirecciones` se facilita la creación de un conjunto de soluciones de E/S alternativas que se podrían requerir.

Las siguientes secciones describen las maneras en que se usan las clases del paquete `java.io` para grabar y mostrar los resultados de una búsqueda en la libreta de direcciones.

### 12.9.3 Salida de texto con `FileWriter`

Hay tres pasos involucrados en el almacenamiento de datos en un archivo:

1. Se abre el archivo.
2. Se escriben los datos.
3. Se cierra el archivo.

La naturaleza de la salida por archivo implica que cualquiera de estos pasos podría fallar por distintos motivos, muchos de ellos completamente ajenos al control del programador de la aplicación. En consecuencia, será necesario anticipar las excepciones que se lanzarán en cada paso.

Cuando se trata de escribir un archivo de texto, es habitual la creación de un objeto `FileWriter` cuyo constructor toma el nombre del archivo sobre el que se escribirá. El nombre del archivo puede ser una cadena o un objeto `File`. La creación de un `FileWriter` tiene el efecto de abrir el archivo externo y prepararlo para recibir alguna salida. Si el intento de abrir el archivo fracasa por algún motivo, entonces el constructor lanzará una `IOException`. Los motivos del fracaso podrían ser que los permisos del sistema de archivos impiden que un usuario escriba sobre determinados archivos o que el nombre del archivo no coincide con una ubicación válida en el sistema de archivos.

Una vez que el archivo se abrió satisfactoriamente, se puede usar el método `write` del escritor para guardar caracteres, generalmente en forma de cadenas, en el archivo. Podría fallar cualquier intento de escritura, aun cuando el archivo se haya abierto exitosamente; estos fallos son raros, pero no imposibles.

Una vez que se ha escrito toda la salida, es importante cerrar formalmente el archivo. Esto asegura que todos los datos hayan sido realmente escritos en el sistema externo de archivos y generalmente, tiene el efecto de liberar algunos recursos internos o externos. Nuevamente, aunque en raras ocasiones, podría fallar el intento de cerrar un archivo.

El modelo básico que surge de la discusión anterior podría ser:

```
try {
    FileWriter escritor = new FileWriter("...nombre del
    archivo... ");
    while (hay más texto para escribir) {
        ...
        escritor.write(siguiente parte de texto);
        ...
    }
    escritor.close();
}
catch(IOException e) {
    algo anduvo mal al acceder al archivo
}
```

La cuestión principal que surge es cómo tratar cualquier excepción que se lance durante los tres pasos. La excepción que se lanza cuando se intenta abrir un archivo es realmente la única en la que es posible hacer algo y sólo si existe alguna forma de generar un nombre de archivo alternativo para intentar nuevamente. Debido a que esta alternativa requerirá generalmente de la intervención de un usuario humano de la aplicación, las posibilidades de tratar la excepción exitosamente son obviamente específicas de la aplicación y del contexto. Si fracasa un intento de escribir en el archivo, es poco probable que la repetición del intento resulte exitosa. De manera similar, el fracaso al cerrar un archivo, generalmente, no merece la pena ningún esfuerzo de recuperación y la consecuencia será probablemente un archivo incompleto.

La dificultad de recuperación de una excepción lanzada durante la salida a un archivo es el principal motivo por el que el método `grabarResultadoDeBusqueda` que se muestra en el Código 12.18 simplemente propaga la excepción a su invocador, ya que sería apropiado intentar una recuperación en un nivel más alto de la aplicación.

#### 12.9.4 Entrada de texto con `FileReader`

El complemento de la salida de texto mediante un `FileWriter` es la entrada de texto mediante un `FileReader`. Tal como se podría esperar, para la entrada de texto se requiere un conjunto complementario de tres pasos: abrir el archivo, leerlo y cerrarlo. Mientras que las unidades naturales para la escritura de texto son los caracteres y las cadenas, las unidades naturales para la lectura de texto son los caracteres y las líneas. Sin embargo, pese a que la clase `FileReader` contiene un método para leer un solo carácter<sup>5</sup>, no contiene ningún método para leer una línea. El problema con la lectura de líneas desde un archivo reside en que no hay un límite predefinido para la longitud de una línea. Esto quiere decir que cualquier método que devuelva la línea siguiente completa desde el archivo, debe ser capaz de leer un número arbitrario de caracteres. Por este motivo, generalmente se envuelve un objeto `FileReader` con un objeto `BufferedReader` que define un método `readLine` para leer una línea. Este método siempre elimina el carácter de terminación de línea en la cadena que retorna y se usa el valor `null` para indicar el fin de archivo.

Esto sugiere el siguiente modelo básico para leer el contenido de un archivo de texto:

```
try {
    BufferedReader lector = new BufferedReader(
        new FileReader("...nombre del archivo..."));
    String linea = lector.readLine();
    while (linea != null) {
        hacer algo con la linea
        linea = lector.readLine();
    }
    lector.close();
}
catch(FileNotFoundException e) {
    no se encontró el archivo especificado
}
```

---

<sup>5</sup> En realidad, su método `read` devuelve cada carácter como un valor entero `int` en lugar de un valor `char` porque usa un valor adicional `-1` fuera de los límites posibles, para indicar el fin del archivo.

```

    }
    catch(IOException e) {
        algo anduvo mal al leer o al cerrar el archivo
    }
}

```

Tal como en la salida, la cuestión que surge es qué hacer con cualquier excepción que se lanza durante todo el proceso. La clase `File` ofrece métodos que hacen posible reducir la probabilidad de que fracase la operación de apertura del archivo. Por ejemplo, define métodos de consulta tales como `exists` y `canRead` que permiten controlar el estado de un archivo antes de abrirlo. Tales controles no son aplicables generalmente cuando se trata de escribir en un archivo porque un archivo no debe existir antes de ser escrito.

La clase `LibretaManejadorDeArchivos` contiene dos ejemplos diferentes del uso de los objetos `FileReader` y `BufferedReader`. En particular, el método `mostrarResultadoDeBusqueda` que se muestra en el Código 12.18 incluye un ejemplo de cómo puede fracasar un intento de cerrar un archivo pero sólo si, en primer lugar, el archivo fue abierto exitosamente. Observe que la variable `lector` ha sido definida *fuera* del bloque `try` de modo que esté disponible para la cláusula `finally`. Observe también que cualquier excepción que se produzca a partir del intento de cerrar el archivo requiere una sentencia `try` adicional en la cláusula `finally`.

**Ejercicio 12.38** Lea la documentación API de la clase `File` del paquete `java.io`. ¿Qué tipo de información está disponible sobre archivos?

**Ejercicio 12.39** ¿Cómo puede decidir si un nombre de archivo representa un archivo ordinario o un directorio (carpeta)?

**Ejercicio 12.40** ¿Es posible determinar algo sobre el contenido de un archivo en particular a partir de la información almacenada en un objeto `File`?

## 12.9.5

### Scanner: leer entradas desde la terminal

Regularmente hemos usado llamadas a los métodos `print` y `println` de `System.out` para escribir texto en la ventana terminal de BlueJ. `System.out` es de tipo `java.io.PrintStream` y se corresponde con lo que frecuentemente se denomina un destino de *salida estándar*. De manera similar, existe la correspondiente fuente de *entrada estándar* disponible en `System.in`, que es de tipo `java.io.InputStream`. Normalmente, no se usa directamente un `InputStream` cuando se necesitan leer entradas del usuario desde la terminal porque entrega la entrada de a un carácter por vez. En su lugar, generalmente se pasa un `System.in` al constructor de un `Scanner`, definida en el paquete `java.util`. La clase `LectorDeEntrada` del proyecto *soprotecnico-completo* del Capítulo 5 utiliza este abordaje para leer las preguntas del usuario:

```

Scanner lector = new Scanner(System.in);
...
String linea = lector.nextLine();

```

El método `nextLine` de `Scanner` retorna la siguiente línea completa desde la entrada estándar (sin la inclusión del carácter final `newLine`).

La clase `Scanner` no se limita a aportar entradas desde `System.in`; incluye un constructor que toma un parámetro `File` y suministra entradas leídas desde dicho archivo.

**Ejercicio 12.41** Revise la clase LectorDeEntrada del proyecto *soporte-tecnico-completo* para verificar que comprende cómo utiliza la clase Scanner.

**Ejercicio 12.42** Lea la documentación API de la clase Scanner en el paquete `java.util`. ¿Qué métodos «next» posee, además de `nextLine`?

**Ejercicio 12.43** Revise la clase Analizador del proyecto *zuul-mejorado* para ver cómo utiliza también la clase Scanner. Tenga en cuenta que la utiliza de dos maneras ligeramente diferentes.

**Ejercicio 12.44** ¿Por qué considera que podría ser de utilidad el hecho de que la clase Scanner tenga un constructor que toma un parámetro `String`?

Probablemente, la característica más importante de la clase Scanner es su habilidad para «analizar» entradas; en otras palabras, para identificar si la entrada de texto tiene una estructura con algún sentido. Por ejemplo, una invocación al método `nextInt` de Scanner produce que se lea una secuencia de caracteres y se los convierta en su correspondiente valor entero. Esto nos evita leer una entrada como un texto y luego convertirla en números (o en otro tipo de datos) por nuestros propios medios. Usamos esta característica en la clase SeparadorDeLineaLog del proyecto *analizador-weblog* en el Capítulo 4:

```
Scanner separador = new Scanner(lineaLog);
for(int i = 0; i < lineaDeDatos.length; i++) {
    lineaDeDatos[i] = separador.nextInt();
}
```

En este caso, se le aporta al Scanner una línea leída del archivo y la convierte en números enteros individuales.

## 12.9.6

### Serialización de objetos

#### Concepto

La **serialización** permite leer y escribir objetos completos y jerarquías de objetos en una única operación. Cada objeto involucrado debe ser de una clase que implemente la interfaz `Serializable`.

Tal como lo mencionamos en la introducción de la Sección 12.9, la clase `Libreta-ManejadorDeArchivos` incluye métodos para leer y escribir versiones binarias de los objetos `LibretaDeDirecciones` y `DatosDelContacto`. Utiliza una característica de Java que se conoce como *serialización*. En términos simples, la serialización permite que se escriba un objeto completo en un archivo externo mediante una sola operación de escritura y recuperarlo en un paso posterior usando una sola operación de lectura<sup>6</sup>. Esto funciona con estos dos objetos simples y con objetos de múltiples componentes como son las colecciones. Es una característica importante que evita, por ejemplo, tener que leer y escribir objetos campo por campo. Es particularmente útil en el proyecto *libreta-de-direcciones* porque permite que se graben todas las entradas creadas en una sesión y luego leerlas nuevamente en otra sesión.

Para ser elegida para participar en la serialización, una clase debe implementar la interfaz `Serializable` que se define en el paquete `java.io`. Sin embargo, es valioso notar que esta interfaz no define ningún método. Quiere decir que el proceso de serialización es manejado automáticamente por el sistema en tiempo de ejecución y requiere

<sup>6</sup> Esta es una simplificación porque los objetos también pueden ser escritos y leídos a través de la red, por ejemplo, y no sólo dentro de un sistema de archivos.

que se escriba poco código definido por el usuario. En nuestro ejemplo, ambas clases, `LibretaDeDirecciones` y `DatosDelContacto` implementan esta interfaz, por lo tanto, pueden grabarse en un archivo.

**Ejercicio 12.45** Modifique la clase `Contestador` del proyecto *soporte-tecnico* del Capítulo 5 de modo que lea las palabras clave y las respuestas desde un archivo de texto. Esto permite el perfeccionamiento externo y la configuración del sistema sin tener que modificar las fuentes.

**Ejercicio 12.46** Modifique el proyecto *world-of-zuul* del Capítulo 7 de tal manera que escriba las entradas del usuario en un archivo de texto, a modo de registro del juego. Luego realice otras modificaciones para que se pueda jugar nuevamente a partir del archivo grabado.

## 12.10

## Resumen

Cuando dos objetos interactúan, siempre existe la posibilidad de que algo ande mal por diversos motivos. Por ejemplo:

- El programador del cliente podría no haber comprendido el estado o las capacidades de un objeto servidor en particular.
- Un objeto servidor puede ser incapaz de cumplimentar la solicitud de un cliente debido a un conjunto de circunstancias externas.
- Un cliente podría haber sido programado incorrectamente provocando el paso de argumentos inapropiados al método del servidor.

Si algo anda mal, es probable que un programa termine prematuramente (es decir, ¡se caiga!) o que produzca efectos incorrectos y no deseables. Podemos encaminarnos para evitar muchos de estos problemas usando el mecanismo de lanzamiento de excepciones. Este mecanismo proporciona una manera claramente definida para que un objeto informe a un cliente si algo anduvo mal. Las excepciones impiden que un cliente simplemente ignore el problema y estimula a los programadores a tratar de encontrar un curso alternativo de acción como un rodeo si algo anda mal.

Al desarrollar una clase se pueden usar las sentencias `assert` para proporcionar control de la consistencia interna. Estas sentencias típicamente se omiten en el código de producción.

Términos introducidos en este capítulo

**excepción, excepción no comprobada, excepción comprobada, manejador de excepciones, aserción, serialización**

### Resumen de conceptos

- **excepción** Una excepción es un objeto que representa los detalles del fallo de un programa. Se lanza una excepción para indicar que ha ocurrido un fallo.
- **excepción no comprobada** Las excepciones no comprobadas son un tipo de excepción cuyo uso no requiere controles del compilador.

- **excepción comprobada** Las excepciones comprobadas son un tipo de excepción cuyo uso requerirá controles adicionales del compilador. En particular, las excepciones comprobadas en Java requieren el uso de cláusulas *throws* y de sentencias *try*.
- **manejador de excepción** Es el código de un programa que protege las sentencias desde las cuales podría lanzarse una excepción. Proporciona código para la información y/o recuperación, una vez que se lanzó la excepción.
- **aserción** Una aserción es la afirmación de un hecho que debe ser verdadero en la ejecución normal de un programa. Podemos usar aserciones para establecer explícitamente las cuestiones que asumimos y para detectar errores de programación más fácilmente.
- **serialización** La serialización permite que se graben y lean objetos completos y jerarquías de objetos en una sola operación. Cada objeto involucrado debe ser de una clase que implemente la interfaz `Serializable`.

# CAPÍTULO **13**

## Diseñar aplicaciones

Principales conceptos que se abordan en este capítulo:

- descubrir clases
- diseñar interfaces
- tarjetas CRC
- patrones de diseño

Construcciones Java que se abordan en este capítulo

(En este capítulo no se introduce ninguna construcción nueva de Java.)

En los capítulos anteriores de este libro hemos descrito la manera en que se pueden escribir clases de buena calidad. Hemos discutido sobre cómo diseñarlas, cómo volverlas mantenibles y robustas y cómo hacer para que interactúen. Todo esto es importante, pero hemos omitido un aspecto de la tarea: encontrar las clases.

En todos nuestros ejemplos anteriores hemos asumido que más o menos sabemos las clases que debemos usar para resolver nuestros problemas. En un proyecto real de software, la decisión de cuáles son las clases que se necesitan para implementar la solución de un problema puede ser una de las tareas más difíciles. En este capítulo discutimos este aspecto del proceso de desarrollo.

Los pasos iniciales en el desarrollo de un sistema de software se conocen, generalmente, como las etapas de *análisis y diseño*: analizamos el problema y luego diseñamos una solución. El primer paso del diseño es de un nivel más alto que el diseño de clases que tratamos en el Capítulo 7: pensamos qué clases se deben crear para resolver nuestro problema y cómo deben interactuar exactamente. Una vez que tenemos una solución para este problema podemos continuar con el diseño de las clases individuales y comenzar a pensar en su implementación.

### **13.1**

### **Análisis y diseño**

El análisis y el diseño de sistemas de software es un área de problemas amplia y compleja, cuyo tratamiento detallado está fuera de los alcances de este libro. Existe bibliografía específica en la que se describen muchas metodologías diferentes que se usan,

en la práctica, para esta tarea. En este capítulo, nuestra meta es ofrecer sólo una introducción a los problemas que se encuentran durante el proceso.

Usaremos un método bastante simple para orientar la tarea, que funcionará bien en problemas relativamente pequeños. Para descubrir las clases iniciales usamos el *método verbo/sustantivo* y luego usaremos *tarjetas CRC* para llevar adelante el diseño inicial de la aplicación.

### 13.1.1

#### El método verbo/sustantivo

##### Concepto

Las clases de un sistema se corresponden aproximadamente con los **sustantivos** que aparecen en la descripción del mismo; los métodos se corresponden con los **verbos**.

Este método trata de identificar las clases y los objetos, y las asociaciones e interacciones entre ellos. En el lenguaje humano, los sustantivos describen «cosas» como por ejemplo: personas, edificios, etc. y los verbos describen «acciones» como escribir, comer, etc.

A partir de estos conceptos del lenguaje natural podemos ver que, en la descripción de un problema de programación los nombres generalmente se corresponden con las clases y con los objetos mientras que los verbos se corresponden con las cosas que hacen esos objetos, es decir, con los métodos. No precisamos una descripción más larga para ilustrar esta técnica; generalmente, la descripción necesita contener sólo unos pocos párrafos.

El ejemplo que usaremos para tratar este proceso es el diseño de un sistema de reserva de entradas para el cine.

### 13.1.2

#### El ejemplo de reserva de entradas para el cine

Esta vez, a diferencia de los ejemplos anteriores, no comenzaremos con la extensión de un proyecto existente sino que asumimos que estamos en una situación en la que nuestra tarea es crear una aplicación desde el principio. La tarea es crear un sistema que pueda ser usado por una empresa operadora de cines, para manejar la reserva de entradas a las distintas salas. Es muy frecuente que la gente llame por teléfono al cine para reservar entradas para determinada función. La aplicación debiera ser capaz de encontrar asientos vacíos para la función solicitada y reservarlos para el cliente que lo solicita.

Asumiremos que hemos tenido varios encuentros con los operadores de cine, durante los cuales nos han descrito la funcionalidad que esperan del sistema. (La comprensión de la funcionalidad esperada, su descripción y el acuerdo con un cliente conforman un problema importante que está fuera del alcance de este libro y que se puede estudiar en otros cursos y en otros libros.)

La descripción que escribimos para nuestro sistema de reserva de entradas es la siguiente:

*El sistema de reserva de entradas para el cine debe almacenar reservas para varias salas. En cada sala, los asientos están ubicados en filas. Los clientes pueden reservar entradas y se les da un número de fila y un número de asiento. El cliente puede requerir la reserva de varios asientos consecutivos.*

*Cada entrada es para una función en particular, es decir, para la exhibición de una determinada película en un cierto horario. Las funciones se realizan a determinada fecha y hora en la sala designada para exhibirlas. El sistema almacena el número de teléfono del cliente.*

Una vez que tenemos una descripción razonablemente clara como ésta, podemos hacer un primer intento de descubrir las clases que conformarán al sistema y sus métodos convenientes, mediante la identificación de los sustantivos y de los verbos que aparecen en el texto.

### 13.1.3 Descubrir clases

El primer paso en la identificación de las clases es recorrer la descripción y marcar todos los sustantivos y verbos que aparecen en el texto; en esta tarea, encontramos los siguientes sustantivos y verbos. (Los sustantivos se muestran en el orden en que aparecen en el texto y los verbos se muestran asociados a los sustantivos a los que se refieren.)

Sustantivos	Verbos
sistema de reserva de entradas para el cine	<i>almacena</i> (reservas de entradas) <i>almacena</i> (número de teléfono)
reserva de entrada sala	<i>tiene</i> (asientos)
asiento fila cliente	<i>reserva</i> (asientos) <i>se le da</i> (número de fila, número de asiento) <i>solicita</i> (reserva de entrada)
número de fila número de asiento función	<i>se designa</i> (una sala)
película fecha hora número de teléfono	

Los sustantivos que identificamos nos dan una primera aproximación de las clases de nuestro sistema. En un primer paso, podemos pensar en una clase por cada sustantivo. Este no es un método exacto, más adelante podríamos encontrar que necesitamos algunas clases adicionales o que algunos de nuestros sustantivos no son necesarios, pero esta es una cuestión que controlaremos más adelante. Es importante no excluir ninguno de los sustantivos de la descripción ya que todavía no tenemos información suficiente como para tomar una buena decisión.

Probablemente habrá notado que todos los sustantivos se han escrito en singular y esto se debe a que lo típico es que los nombres de las clases estén en singular y no en plural; por ejemplo, es preferible que una clase tenga el nombre *Cine* antes que el nombre *Cines*. Esto se debe a que se logra la multiplicidad mediante la creación de varias instancias de una clase.

**Ejercicio 13.1** Revise los proyectos de los capítulos anteriores de este libro. ¿Existen casos en que el nombre de la clase esté en plural? De ser así, ¿se justifican esas situaciones por alguna razón en particular?

### 13.1.4 Usar tarjetas CRC

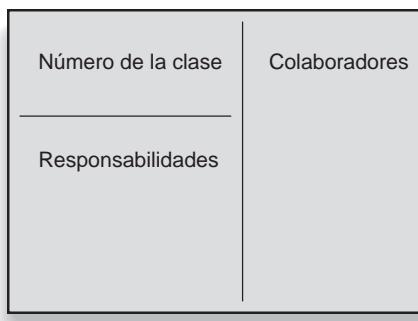
El próximo paso en nuestro proceso de diseño es trabajar con las interacciones entre nuestras clases; para hacerlo usaremos un método denominado *tarjetas CRC*<sup>1</sup>.

La denominación CRC se basa en la tríada Clase/Responsabilidades/Colaboradores. La idea consiste en tomar tarjetas de cartulina y usar una tarjeta para cada clase. Es importante para esta actividad usar tarjetas reales y físicas y no una computadora o una simple hoja de papel.

Cada tarjeta está dividida en tres áreas: en el área de la izquierda se escribe el nombre de la clase; en el área ubicada debajo de la anterior se escriben las responsabilidades de dicha clase y en el área de la derecha se escriben los colaboradores de esta clase (las clases que usa esta clase). La Figura 13.1 ilustra el esquema de una tarjeta CRC.

**Figura 13.1**

Una tarjeta CRC



**Ejercicio 13.2** Cree tarjetas CRC para cada una de las clases del sistema de reserva de entradas para el cine. En este paso sólo necesita completar los nombres de las clases.

### 13.1.5 Escenarios

#### Concepto

Se pueden usar los **escenarios** (también conocidos como «casos de uso») para comprender las interacciones de las clases en el sistema.

Ya tenemos una primera aproximación de las clases necesarias para nuestro sistema y una representación física de ellas mediante tarjetas CRC. Con el objetivo de deducir las interacciones necesarias entre las clases de nuestro sistema jugaremos con varios *escenarios*. Un escenario es un ejemplo de la actividad que el sistema tiene que llevar adelante o proporcionar. En algunas ocasiones, se hace referencia a los escenarios como *casos de uso*. No usamos este término aquí porque generalmente se usa para denotar una manera más formal de la descripción de los escenarios.

<sup>1</sup> Las tarjetas CRC fueron descritas por primera vez en un documento escrito por Kent Beck y Ward Cunningham, titulado *A Laboratory For Teaching Object-Oriented Thinking*. Merece la pena leer este documento para obtener información adicional a la que aporta este capítulo. Puede encontrarlo online en <http://c2.com/doc/oops1a89/paper.html> o buscándolo en Internet por su título.

El juego con los escenarios resulta mejor cuando se realiza en grupo. A cada miembro del grupo se le asigna una clase (o un número pequeño de clases) y esa persona cumple su rol diciendo en voz alta lo que la clase está actualmente haciendo. Mientras se juega con un escenario, los miembros registran en las tarjetas CRC cada cosa que se descubre sobre la clase en acción: cuáles deben ser sus responsabilidades y qué otras clases colaboran con ella.

Comenzamos con un ejemplo de un escenario sencillo. Un cliente llama al cine y quiere reservar dos asientos para ver *The Shawshank Redemption* esta noche. El empleado del cine comienza a usar el sistema de reservas para encontrar y reservar un asiento.

Dado que el usuario humano interactúa con el sistema de reservas, representado mediante la clase `SistemaDeReserva`, este es el lugar donde comienza el escenario. Esto es lo que podría ocurrir a continuación:

- El usuario (el empleado del cine) quiere encontrar todas las funciones de *The Shawshank Redemption* que se dan esta noche. De modo que podemos anotar en la tarjeta CRC `SistemaDeReserva` como una responsabilidad: *Debe encontrar las funciones por título y por día*. También podemos tomar nota de que la clase `Funcion` es un colaborador.
- Nos debemos preguntar: ¿cómo encuentra la función el sistema? ¿Quién lo solicita? Una solución podría ser que el `SistemaDeReserva` almacene una colección de funciones, lo que nos da por resultado una clase adicional: la colección. (Esta clase se podría implementar más adelante mediante la clase `ArrayList`, `LinkedList`, `HashSet` o alguna otra forma de colección; podemos tomar esta decisión más tarde, por ahora sólo tomamos nota de una colección.) Este es un ejemplo de cómo se podrían presentar clases adicionales durante el juego con los escenarios. Podría ocurrir que debamos agregar clases por razones de implementación que inicialmente hemos pasado por alto. Agregamos en las responsabilidades de la tarjeta `SistemaDeReserva`: *Almacena una colección de funciones* y en su lista de colaboradores, a la clase `Coleccion`.

**Ejercicio 13.3** Cree una tarjeta CRC para la clase colección recientemente identificada y agréguela a su sistema.

- Asumimos que habrá tres funciones: una a las 17:30, otra a las 21:00 y otra a las 23:30. El empleado informa los horarios al cliente y éste elige el de las 21:00. El empleado quiere verificar los detalles de esa función (si los asientos están totalmente vendidos, en qué sala se exhibe, etc.). En consecuencia, el `SistemaDeReserva` debe ser capaz de recuperar y mostrar los detalles de las funciones. La persona que cumple el rol del sistema de reserva deberá solicitar a la persona que cumple el rol de la función que le informe los detalles requeridos. Luego, anotamos en la tarjeta `SistemaDeReserva`: *Recupera y muestra los detalles de la función* y en la tarjeta `Funcion`: *Proporciona los detalles sobre la sala y el número de asientos disponibles*.
- Asumamos que hay muchos asientos disponibles. El cliente elige los asientos 13 y 14 de la fila 12. El empleado hace la reserva. Anotamos en la tarjeta `SistemaDeReserva`: *Acepta del usuario la reserva de asientos*.
- Ahora tenemos que ver a través de un escenario cómo funciona exactamente la reserva de asientos. La reserva de un asiento está claramente asociada a una función en particular. Por lo que el `SistemaDeReserva` probablemente deberá

informar de la función para la que se realiza la reserva, pero delega la tarea de hacer la reserva al objeto `Funcion`. Podemos anotar en la clase `Funcion`: *Puede reservar asientos.* (Habrá observado que la noción de objetos y de clases es borrosa durante el juego con los escenarios CRC. En efecto, la persona que representa una clase también es representante de todas sus instancias. Esto es intencional y generalmente no es un problema.)

- Ahora pasamos a la clase `Funcion` que ha recibido un pedido de reserva de un asiento. ¿Qué debe hacer exactamente? Debe ser capaz de almacenar las reservas de asientos, debe tener una representación de los asientos en la sala. Por lo tanto, asumimos que cada función está vinculada con un objeto sala. (Anote esto en la tarjeta: *Almacena sala*. Esta clase también es un colaborador.) Probablemente, la sala conozca el número exacto de asientos y su ubicación en ella. (Podemos anotar también en nuestras mentes, o en un trozo de papel aparte, que cada función debe tener su propia copia del objeto sala, dado que se pueden asignar varias funciones a la misma sala y la reserva de un asiento en una función no reserva el mismo asiento para otra función. Esto es algo que deberemos tener en cuenta cuando se creen los objetos `Funcion`. Pensaremos sobre este asunto más tarde cuando trabajemos con otro escenario: calendarizar nuevas funciones.) Por lo tanto, la manera en que opera una función para tratar la reserva de asientos es, probablemente, pasando esta solicitud de reserva a la sala.
- Ahora, la sala ha recibido la solicitud de reserva de un asiento. (Anote esto en la tarjeta: *Acepta la solicitud de reserva*.) ¿Cómo trata esta solicitud? La sala puede tener una colección de asientos o puede tener una colección de filas (cada fila sería un objeto separado) y las filas contienen asientos. ¿Cuál es la mejor alternativa? Si pensamos en otros posibles escenarios podríamos decidir avanzar con la idea de almacenar filas. Si, por ejemplo, un cliente requiere cuatro asientos juntos en la misma fila, podría ser más fácil encontrar asientos adyacentes si los tenemos todos ubicados en la misma fila. Anotamos en la tarjeta `Sala`: *Almacena filas* y ahora `Fila` es un colaborador.
- Anotamos en la clase `Fila`: *Almacena una colección de asientos* y un nuevo colaborador: `Asiento`.
- Volvemos a la clase `Sala`. Todavía no hemos trabajado sobre cómo debe reaccionar exactamente ante la solicitud de una reserva de asientos. Asumimos que hace dos cosas: primero encuentra la fila requerida y luego solicita al objeto `Fila` la reserva con el número de asiento a reservar.
- A continuación anotamos en la tarjeta `Fila`: *Acepta el pedido de reserva de un asiento*. Luego debe encontrar el objeto `Asiento` correcto (podemos anotar como una responsabilidad: *Debe encontrar asientos por número*) y debe reservar ese asiento. Podría hacerlo informándole al objeto `Asiento` que ahora está reservado.
- Ahora podemos agregar en la tarjeta `Asiento`: *Acepta reservas*. El asiento puede recordar por sí mismo si está reservado. Anotamos en la tarjeta `Asiento`: *Almacena el estado de la reserva (disponible/reservado)*.

**Ejercicio 13.4** Juegue con este escenario utilizando sus propias tarjetas (si es posible, hágalo con un grupo de personas). Agregue cualquier otra información que le parezca que falta en esta descripción.

El asiento, ¿debe también almacenar información sobre quién lo ha reservado? ¿Debe almacenar el nombre del cliente o el número de teléfono? ¿O puede ser que debamos crear un objeto cliente tan pronto como alguien haga una reserva y almacenar el objeto cliente con el asiento una vez que el asiento haya sido reservado? Estas son preguntas interesantes y trataremos de trabajar para encontrar la mejor solución jugando con más escenarios.

Este fue sólo un primer escenario simple. Necesitamos jugar con muchos más escenarios para obtener una mejor comprensión de cómo debería funcionar el sistema.

El trabajo con los escenarios funciona mejor cuando un grupo de personas se sienta alrededor de una mesa y mueve las tarjetas sobre ella. Las tarjetas que cooperan cercanamente se pueden ubicar juntas y más próximas para dar alguna impresión sobre el grado de acoplamiento del sistema.

Otros escenarios con los que se podría jugar a continuación podrían incluir lo siguiente:

- Un cliente solicita cinco asientos juntos. En este caso hay que trabajar sobre cómo se logra encontrar los cinco asientos consecutivos.
- Un cliente llama y dice que olvidó el número de asientos asignados en la reserva que hizo ayer. ¿Podría buscar los números de asiento nuevamente?
- Un cliente llama para cancelar una reserva; puede darnos su nombre y la función, pero olvidó los números de asiento.
- Llama un cliente que ya hizo una reserva y quiere saber si puede reservar otro asiento cercano a los que ya tiene.
- Se canceló una función. El cine quiere llamar a todos los clientes que han hecho reservas para dicha función.

Estos escenarios deberían aportarnos una buena comprensión de la forma en que se realiza la búsqueda de asientos y de la parte del sistema que se ocupa de reservar efectivamente los asientos. Luego, necesitamos otro grupo de escenarios: aquellos que traten con la configuración de la sala y el calendario de las funciones. Aquí hay algunos posibles escenarios:

- Se debe configurar el sistema para un nuevo cine. El cine tiene dos salas de diferentes tamaños. La sala A tiene 26 filas, con 18 asientos en cada fila. La sala B tiene 32 filas; en esta sala, las primeras seis filas tienen 20 asientos, las siguientes 10 filas tienen 22 asientos y las restantes filas tienen 26 asientos.
- Se exhibirá una nueva película durante las próximas dos semanas, tres veces por día (a las 16:40, a las 18:30 y a las 20:30). Se deben agregar las funciones al sistema. Todas las funciones se dan en la sala A.

**Ejercicio 13.5** Juegue con estos escenarios. Anote todas las preguntas que queden sin responder en una hoja de papel aparte. Tome nota de todos los escenarios con los que ha trabajado.

**Ejercicio 13.6** ¿Qué otros escenarios se le ocurren? Escriba una lista de escenarios y luego juegue con ellos.

El juego con los escenarios requiere un poco de paciencia y un poco de práctica. Es importante emplear el tiempo suficiente para hacerlo. El juego con los escenarios mencionados aquí tomará varias horas.

Es muy común que los principiantes tomen atajos y no cuestionen y registren cada detalle de la ejecución de un escenario. ¡Esto es peligroso! Pasan rápidamente a desarrollar el sistema en Java y si han quedado sin responder algunos detalles, es muy probable que las decisiones *ad hoc* se tomen en tiempo de implementación, lo que más tarde producirá malas elecciones.

También es común que los principiantes olviden algunos escenarios. El olvido de una parte del sistema antes de iniciar el diseño de las clases y su implementación puede acarrear una gran cantidad de trabajo más adelante, cuando el sistema esté parcialmente implementado y deba ser modificado.

**Ejercicio 13.7** Haga un diseño de clases para un sistema de simulación de control de un aeropuerto. Use tarjetas CRC y escenarios. Aquí está una descripción posible del sistema:

*El programa es un sistema de simulación de un aeropuerto. Necesitamos, para nuestro nuevo aeropuerto, conocer si podemos operar con dos pistas de aterrizaje o si necesitamos tres. El aeropuerto funciona de esta manera:*

*El aeropuerto tiene varias pistas de aterrizaje. Los aviones despegan y aterrizan en pistas de aterrizaje. Los controladores del tráfico aéreo coordinan el tráfico y le dan permisos a los aviones para despegar o aterrizar. Algunas veces, los controladores dan el permiso directamente, otras veces le dicen a los aviones que esperen. Los aviones deben mantener una cierta distancia entre ellos. El propósito del programa es simular el aeropuerto en operación.*

## 13.2

## Diseño de clases

Ahora es el momento del siguiente gran paso: convertir las tarjetas CRC en clases de Java. Durante el ejercicio con las tarjetas CRC se logra una buena comprensión de la estructura de la aplicación y de la manera en que cooperan las clases para resolver las distintas tareas del programa. Al atravesar los diferentes casos surge la necesidad de introducir clases adicionales (que generalmente se trata de clases que representan estructuras internas de datos) y también puede ocurrir que quede alguna tarjeta que representa una clase que jamás se usó, en cuyo caso, se puede eliminar.

A esta altura, el reconocimiento de las clases que se deben implementar es trivial: las tarjetas nos muestran el conjunto completo de las clases que se necesitan. La decisión de la interfaz de cada clase, es decir, el conjunto de métodos públicos que debe tener una clase, es un poco más difícil, pero hemos dado un importante paso hacia adelante que es bueno. Si el juego con los escenarios estuvo bien hecho, las responsabilidades anotadas en cada clase describen los métodos públicos de dichas clases y quizás también, algunos campos de instancia. Se deben evaluar las responsabilidades de cada clase de acuerdo con los principios del diseño de clases discutidos en el Capítulo 7: diseño dirigido por responsabilidades, acoplamiento y cohesión.

### 13.2.1 Diseñar interfaces de clases

Antes de comenzar a escribir el código de nuestra aplicación en Java, podemos usar nuevamente las tarjetas para avanzar otro paso más en el diseño, mediante la traduc-

ción de las descripciones informales de las invocaciones a los métodos y el agregado de los parámetros necesarios.

Para llegar a una descripción más formal podemos jugar nuevamente con los distintos escenarios, pero esta vez, en términos de llamadas a métodos, parámetros y valores de retorno. La lógica y la estructura de la aplicación no debiera cambiar más, pero tratamos de anotar la información completa de las signaturas de los métodos y los campos de instancia. Llevamos a cabo esta tarea con un nuevo conjunto de tarjetas.

**Ejercicio 13.8** Cree un nuevo conjunto de tarjetas CRC que representen las clases que ha identificado. Atraviese nuevamente los escenarios. Esta vez, anote los nombres exactos de los métodos que se invoquen desde otra clase y especifique detalladamente (tipo y nombre) todos los parámetros que se deben pasar y los valores de retorno de los métodos. Las signaturas de los métodos se escriben en las tarjetas CRC en el área que corresponde a las responsabilidades. En la parte posterior de la tarjeta anote los campos de instancia que contiene cada clase.

Una vez que se resolvió el último ejercicio propuesto, resulta fácil escribir la interfaz de cada clase. Podemos traducir las tarjetas directamente a Java. Típicamente, se deben crear todas las clases y se deben incluir *métodos stubs* para todos los métodos públicos que se deban escribir. Un *método stub* es un método que tiene la signatura correcta y el cuerpo vacío<sup>2</sup>.

A muchos estudiantes les resulta tedioso realizar esta tarea detalladamente pero, al finalizar el proyecto, apreciarán el valor de estas actividades. Muchos equipos de desarrollo de software han enfatizado que el tiempo que se ahorra en la etapa de diseño, muchas veces se emplea en la corrección de errores u omisiones que no se descubrieron con la anterioridad suficiente.

Generalmente, los programadores inexpertos sienten que la escritura del código es la «parte real de la programación». Si bien no llegan a considerar como superflua la construcción del diseño inicial, les parece molesta y que no pueden esperar a terminarla, por lo que comienzan el trabajo real. Esta es una visión muy alejada del buen camino.

El diseño inicial es una de las partes más importantes del proyecto. El tiempo que se empleará en el diseño se debe planificar, como mínimo, similar al tiempo que se empleará en la implementación. El diseño de una aplicación no es anterior a la programación, ¡es la parte más importante de la programación!

Los errores del código propiamente dichos pueden solucionarse de manera bastante fácil. Los errores del diseño pueden ser, en el mejor de los casos, muy caros de corregir y en el peor de los casos, fatales para la aplicación una vez terminada. En algunos casos desafortunados, pueden ser prácticamente incorregibles (hay que parar y comenzar todo de nuevo).

---

<sup>2</sup> Si se desea, se pueden incluir sentencias `return` triviales en los cuerpos de los métodos cuyo tipo de retorno es distinto de `void`. Sólo se retorna un valor `null` en los métodos que retornan objetos y un cero o un valor `false` en los métodos que retornan tipos primitivos.

### 13.2.2 Diseño de la interfaz de usuario

Hasta ahora, hemos dejado fuera de la discusión, el diseño de la interfaz de usuario<sup>3</sup>. En este punto, tenemos que decidir detalladamente lo que los usuarios verán en la pantalla y las maneras en que interactuarán con nuestro sistema.

En una aplicación bien diseñada, la interfaz de usuario es muy independiente de la lógica subyacente de la aplicación, por lo que puede diseñarse independientemente del diseño de la estructura de clases del resto del proyecto. Como vimos en el Capítulo 6, BlueJ nos da la posibilidad de interactuar con nuestra aplicación antes de que se disponga de una interfaz para el usuario final, por lo que podemos elegir trabajar primero con la estructura interna.

La interfaz de usuario puede ser una IGU (una interfaz gráfica de usuario) con menús y botones, puede ser una interfaz basada en texto o podemos decidir ejecutar la aplicación usando el mecanismo de invocación de métodos de BlueJ.

Por ahora, ignoraremos el diseño de la interfaz de usuario y usaremos el método de invocación de BlueJ para trabajar con nuestro programa.

## 13.3 Documentación

Después de identificar las clases y sus interfaces, y antes de comenzar con la implementación de los métodos de una clase, se debe documentar la interfaz. Esto implica escribir un comentario de clase y comentarios de métodos en cada clase del proyecto. Los comentarios deben ser descriptivos con la cantidad de detalle suficiente como para que se puedan identificar los principales propósitos de cada clase y de cada método.

Al igual que el análisis y el diseño, la documentación es un área frecuentemente desdenada por los principiantes. No es fácil que los programadores inexpertos vean los motivos por los que la documentación es tan importante. La razón es que los programadores inexpertos generalmente trabajan sobre proyectos que tienen sólo unas pocas clases y que se escriben en un período de unas pocas semanas o meses. Un programador puede contar con documentación pobre cuando trabaja sobre estos miniproyectos.

Sin embargo, aún los programadores experimentados frecuentemente se preguntan cómo es posible escribir la documentación antes que la implementación. Esto es así porque fallan al apreciar que la buena documentación hace foco en cuestiones de alto nivel, tales como qué hace una clase o un método, antes que en cuestiones de bajo nivel tales como exactamente cómo lo hace. Esto es, generalmente, un síntoma de ver a la implementación como más importante que el diseño.

Si un desarrollador de software quiere avanzar hacia problemas más interesantes y comenzar a trabajar profesionalmente en aplicaciones de la vida real, no es poco usual que trabaje con docenas de personas sobre una aplicación durante varios años. La solución *ad hoc* de sólo «tener la documentación en su cabeza» no funcionará nunca más.

<sup>3</sup> En este lugar, ¡observe con cuidado el doble significado del término «diseñar interfaces»! Antes, hablábamos de las interfaces de las clases (el conjunto de los métodos públicos); ahora, hablamos de la interfaz del usuario, lo que el usuario ve en la pantalla para interactuar con la aplicación. Ambas son cuestiones muy importantes y desafortunadamente se utiliza en ambos casos el término interfaz.

**Ejercicio 13.9** Cree un proyecto en BlueJ para el sistema de reserva de entradas para el cine. Cree las clases necesarias. Cree los métodos stub de todos los métodos.

**Ejercicio 13.10** Documente todas las clases y los métodos. Si trabajó en grupo, asigne las responsabilidades de las clases a diferentes miembros del grupo. Use el documentador de java (javadoc) para el formato de los comentarios, con las etiquetas javadoc adecuadas para documentar los detalles.

## 13.4

## Cooperación

**Programación por parejas** Tradicionalmente, la implementación de las clases se hace a solas. La mayoría de los programadores trabajan en sus propias clases escribiendo el código y se contratan a otras personas solamente después de que se terminó la implementación, para que prueben o revisen el código.

Recientemente, se ha sugerido la programación por parejas como una alternativa que intenta producir código de mejor calidad (código con mejor estructura y menos fallos). La programación por parejas es también uno de los elementos de una técnica que se conoce como programación extrema. Busque en Internet «programación por parejas» o «programación extrema» para encontrar más información.

El desarrollo de software, generalmente, se hace en equipo. Un abordaje puro, orientado a objetos, proporciona un fuerte soporte al trabajo en equipo porque permite la separación del problema en componentes bajamente acopladas (clases) que pueden ser implementadas independientemente.

Aunque el trabajo de diseño inicial es mejor cuando se realiza en grupo, llega el momento de dividirlo. Si la definición de las interfaces de las clases y la documentación está bien hecha, debe ser posible implementar las clases de manera independiente. Se pueden asignar las clases a los programadores, quienes pueden trabajar a solas o en parejas.

En el resto de este capítulo no discutiremos los detalles de la fase de implementación del sistema de reserva de entradas para el cine. Esa fase involucra mayormente los tipos de tareas que hemos estado haciendo a lo largo de este libro en los capítulos anteriores y esperamos que ahora los lectores puedan determinar por sí mismos cómo continuar a partir de aquí.

## 13.5

## Prototipos

En lugar de diseñar y luego construir la aplicación completa en un paso enorme, se usan los prototipos para investigar partes de un sistema.

Un prototipo es una versión de la aplicación en la que se simula una parte de ella, en vías de experimentar con las restantes partes. Por ejemplo, se podría implementar un prototipo para probar una interfaz gráfica de usuario. En este caso, la lógica de la aplicación podría no estar implementada apropiadamente y en cambio, podríamos escribir implementaciones simples de aquellos métodos que simulan la tarea. Por ejemplo, cuando se invoque un método que busca un asiento disponible en el sistema del cine,

**Concepto**

Armar un prototipo significa construir un sistema que funciona parcialmente en el que se simulan algunas de las funciones de la aplicación. Sirve para proporcionar mayor compresión, en una etapa temprana del proceso de desarrollo, de la manera en que funcionará el sistema.

el método podría devolver siempre, a modo de resultado, *asiento 3, fila 15* en lugar de implementar realmente la búsqueda. Los prototipos nos permiten desarrollar rápidamente un sistema ejecutable (pero no totalmente funcional), de modo que podamos investigar en la práctica distintas partes de la aplicación.

Los prototipos también son útiles para las clases independientes y ayudan al equipo en el proceso de desarrollo. Frecuentemente, cuando diferentes miembros de un equipo trabajan sobre diferentes clases, no todas las clases insumen la misma cantidad de tiempo en terminarse. En algunos casos, una clase dejada de lado, puede retrasar la continuación del desarrollo y la prueba de otras clases. En esos casos puede ser beneficioso escribir una clase prototipo. El prototipo tiene implementaciones de todos los métodos pero en lugar de contener implementaciones finales y completas, el prototipo sólo simula la funcionalidad. La escritura de un prototipo puede ser posible rápidamente y el desarrollo de las clases cliente puede continuar usando el prototipo hasta que la clase se haya implementado en su totalidad.

Tal como lo discutimos en la Sección 13.6, un beneficio adicional de los prototipos es que puede brindar a los desarrolladores conocimientos profundos de cuestiones y problemas que no fueron considerados en un estado anterior.

**Ejercicio 13.11** Esquematice un prototipo para el ejemplo del sistema de reserva de entradas para el cine. ¿Qué clases deben implementarse primero y cuáles pueden permanecer en estado de prototipo?

**Ejercicio 13.12** Implemente un prototipo del sistema de reserva de entradas para el cine.

## 13.6

### Crecimiento del software

Para la construcción del software, existen varios modelos que se pueden aplicar, uno de los más comúnmente utilizados es el *modelo de cascada*, denominado así porque la actividad progresó de un nivel al siguiente, tal como el agua de una cascada, y no hay vueltas atrás.

#### 13.6.1

##### Modelo de cascada

En el modelo de cascada, las distintas fases del desarrollo del software se realizan siguiendo una secuencia determinada:

- análisis del problema
- diseño del software
- implementación de los componentes del software
- prueba unitaria
- prueba integral
- entrega del sistema al cliente

Si se presenta algún problema en cualquiera de las fases, deberíamos regresar a la fase anterior para solucionarlo; por ejemplo, si alguna prueba demuestra la existencia de un fallo regresamos a la implementación, pero no existe ningún plan para revisitar las fases anteriores.

Este es, probablemente, el modelo más tradicional y conservador de desarrollo de software y se ha usado extensamente durante largo tiempo. Sin embargo, a lo largo de los años, se han descubierto numerosos inconvenientes en este modelo. Dos de las principales grietas son que asume que los desarrolladores comprenden por completo y detalladamente la funcionalidad del sistema desde el principio y que el sistema no va a cambiar con posterioridad a su entrega al cliente.

En la práctica, ambas presunciones, generalmente, no son ciertas. Es muy común que el diseño de la funcionalidad de un software no sea perfecto desde el principio, frecuentemente porque el cliente, quien conoce el dominio del problema, no sabe mucho de computación y los ingenieros de software, quienes saben cómo programar, tienen sólo un conocimiento limitado del dominio del problema.

### 13.6.2 Desarrollo iterativo

Una solución posible a los problemas que acarrea el modelo de cascada es el uso temprano de prototipos y la interacción frecuente con el cliente durante el proceso de desarrollo. Se construyen los prototipos de los sistemas, que no hacen demasiado pero que nos dan una impresión de cómo se podría presentar el sistema y de lo que podría hacer, y regularmente, los clientes realizan comentarios sobre el diseño y la funcionalidad. Esta solución conduce a un proceso más circular que el modelo de cascada, el desarrollo del software se retroalimenta pasando varias veces por el circuito análisis-diseño-implementación de prototipo-cliente.

Otro enfoque captura la noción de que un buen software no se diseña sino que *crece*. La idea subyacente es diseñar inicialmente un sistema pequeño y prolífico, y ponerlo en funcionamiento para que pueda ser usado por usuarios finales. Luego, se van agregando gradualmente las características adicionales de una manera controlada (el software crece) y se alcanzan estados «finales» repetidamente y con bastante frecuencia (es decir, estados en los que el software es completamente usable y puede ser entregado a los clientes).

En realidad, el crecimiento del software no se contradice con el diseño del software; se diseña cuidadosamente cada etapa de crecimiento. Lo que se trata de hacer es no diseñar el sistema completo y correcto desde el inicio, aún más: ¡la noción de un sistema de software completo no existe en absoluto!

El modelo de cascada tradicional tiene como objetivo principal la liberación de un sistema completo. El modelo de crecimiento del software asume que no existen los sistemas completos que se usan indefinidamente y sin modificaciones; sólo hay dos cosas que le pueden ocurrir a un sistema de software: es continuamente mejorado y adaptado o desaparecerá.

Esta discusión es central en este libro porque influye fuertemente en la visión de las tareas y las habilidades que se requieren de un programador o de un ingeniero de software. Se podría decir que los autores de este libro están fuertemente a favor del modelo de crecimiento del software por encima del modelo de cascada<sup>4</sup>.

---

<sup>4</sup> Un libro excelente que describe los problemas del desarrollo del software y algunas aproximaciones posibles para solucionarlos es *The Mythical Man-Month* de Frederick P. Brooks Jr, Addison-Wesley. Pese a que la edición original tiene 25 años de antigüedad, su lectura es muy entretenida y muy esclarecedora.

En consecuencia, ciertas tareas y habilidades cobran mucha más importancia de la que podrían tener en el modelo de cascada. El mantenimiento del software, la lectura de código (en lugar de sólo su escritura), el diseño preparado para la extensibilidad, la documentación, la codificación que apunta a la legibilidad y muchas otras cuestiones que hemos mencionado en este libro resultan importantes a partir del hecho de que sabemos que vendrán otros después de nosotros que tendrán que adaptar y extender nuestro código.

La visión de una pieza de software como algo que continuamente crece, cambia y se adapta en lugar de ser una pieza estática de texto que se escribe y se preserva como una novela, determina nuestra visión sobre cómo debe escribirse un código de buena calidad. Todas las técnicas que hemos discutido a lo largo de este libro apuntan a esto.

**Ejercicio 13.13** ¿De qué maneras se podría adaptar o extender en el futuro el sistema de reserva de entradas de cine? ¿Qué cambios son más probables? Escriba una lista de las posibles modificaciones futuras.

**Ejercicio 13.14** ¿Existen otras organizaciones que podrían usar un sistema de reservas similar al que hemos discutido? ¿Qué diferencias significativas existen entre estos sistemas?

**Ejercicio 13.15** ¿Considera que sería posible diseñar un sistema de reservas «genérico» que se podría adaptar o personalizar como para que pueda ser usado en un amplio rango de organizaciones diferentes con necesidades de reservas? Si fuera a crear un sistema como éste, ¿en qué punto del proceso de desarrollo del sistema de cine introduciría modificaciones? ¿O le parece que sería mejor tirar todo y comenzar desde el principio?

## 13.7

## Usar patrones de diseño

En los capítulos anteriores hemos tratado en detalle algunas técnicas para reutilizar una parte de nuestro trabajo y lograr que nuestro código resulte más comprensible para otros. Hasta ahora, una gran parte de estas discusiones ha permanecido en el nivel del código fuente de las clases.

### Concepto

Un **patrón de diseño** es la descripción de un problema computacional común y la descripción de un pequeño conjunto de clases y su estructura de interacción que ayuda a resolver dicho problema.

A medida que nos volvemos más expertos y diseñamos sistemas de software de mayor envergadura, la implementación de las clases deja de ser el problema más dificultoso. La estructura del sistema, la complejidad de las relaciones entre las clases, se vuelve más complicada de diseñar y de comprender que el código de las clases individuales.

Es lógico que tratemos de alcanzar los mismos objetivos para las estructuras de las clases que los que planteamos para el código: queremos reutilizar buena parte de nuestro trabajo y queremos permitir que otros comprendan lo que hemos hecho.

A nivel de las estructuras de clases, ambos objetivos se pueden lograr usando *patrones de diseño*.

Un patrón de diseño describe un problema común, que ocurre regularmente en el desarrollo del software y luego describe una solución general del problema que se puede usar en varios contextos diferentes. La solución de los patrones de diseño de software

consiste, típicamente, en la descripción de un conjunto de clases y sus respectivas interacciones.

Los patrones de diseño colaboran en nuestra tarea de dos maneras. Primero, documentan buenas soluciones a problemas planteados, por lo tanto, estas soluciones se pueden reutilizar más adelante en problemas similares. En este caso, la reutilización no es a nivel código sino a nivel estructura de clases.

Segundo, los patrones de diseño tienen nombres y de esta manera establecen un vocabulario que ayuda a los diseñadores de software a hablar sobre sus diseños. Cuando los diseñadores experimentados discuten sobre la estructura de una aplicación, uno de ellos podría decir «Creo que aquí deberíamos usar un Singleton». Singleton es el nombre de un patrón de diseño ampliamente conocido por lo que si ambos diseñadores están familiarizados con este patrón, serán capaces de hablar sobre él a ese nivel, ahorrándose explicaciones de muchos detalles. De esta manera, el patrón de lenguaje introducido por los patrones de diseño comúnmente conocidos introduce otro nivel de abstracción, uno que nos permite sobrellevar la complejidad en sistemas cada vez más complejos.

Los patrones de diseño de software se hicieron populares a partir de un libro publicado en 1995 que describe un conjunto de patrones, sus aplicaciones y beneficios<sup>5</sup>. Este libro es, aún hoy en día, uno de los trabajos más importantes sobre patrones de diseño. En esta sección no intentamos ofrecer una visión completa de los patrones de diseño sino que discutimos un pequeño número de patrones para brindar a los lectores una idea sobre los beneficios del uso de patrones de diseño y luego, dejar que el lector continúe con el estudio de los patrones mediante la bibliografía específica.

### 13.7.1 Estructura de un patrón

Las descripciones de los patrones se registran, generalmente, mediante una plantilla que contiene un mínimo de información. La descripción de un patrón no sólo contiene información sobre la estructura de algunas clases sino que también incluye una descripción del problema(s) que este patrón resuelve y argumentos a favor o en contra del uso del patrón.

La descripción de un patrón incluye como mínimo:

- un *nombre* que se puede utilizar para hablar sobre el patrón convenientemente;
- una descripción del **problema** que resuelve el patrón (frecuentemente dividido en secciones como intento, motivación, pertinencia);
- una descripción de la **solución** (frecuentemente se describe la estructura, los participantes y los colaboradores);
- las **consecuencias** del uso del patrón, incluyendo los resultados y lo que se deja de lado.

En la siguiente sección discutiremos brevemente algunos patrones usados comúnmente.

<sup>5</sup> *Design Patterns: Elements of Reusable Object-Oriented Software* de Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, Addison-Wesley, 1995.

### 13.7.2 Decorador

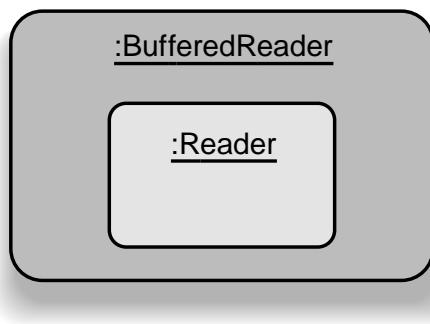
El patrón *Decorador* trata el problema de agregar funcionalidad a un objeto que ya existe. Asumimos que queremos un objeto que responda a las mismas llamadas a método (es decir, que tiene la misma interfaz) pero con un comportamiento adicional o alterado. También quisiéramos agregarlo a la interfaz existente.

Un camino sería utilizando herencia: una subclase puede sobrescribir la implementación de métodos y agregar métodos adicionales. Pero el uso de la herencia produce una solución estática: una vez que se crean los objetos no pueden cambiar su comportamiento.

Una solución más dinámica es el uso de un objeto Decorador. El Decorador es un objeto que encapsula un objeto existente y que puede usarse en lugar del original (generalmente implementa la misma interfaz). Luego, los clientes pueden comunicarse con el Decorador en lugar de hacerlo directamente con el objeto original (sin necesidad de conocer esta sustitución). El Decorador pasa las llamadas a método al objeto encapsulado pero puede llevar a cabo acciones adicionales. Podemos encontrar un ejemplo en la biblioteca de entrada/salida de Java donde se usa un `BufferedReader` como un Decorador de un `Reader` (Figura 13.2). El `BufferedReader` implementa la misma interfaz que un `Reader` y se puede usar en lugar de éste que no utiliza un buffer, pero le agrega el comportamiento básico del `Reader`. En contraste con el uso de la herencia, los decoradores se pueden agregar a objetos ya existentes.

**Figura 13.2**

Estructura del patrón  
Decorador



### 13.7.3 Singleton

Una situación común en muchos programas es la de tener un objeto del que debe existir sólo una instancia. Por ejemplo, en nuestro juego *world-of-zuul* queremos contar sólo con un único analizador. Si escribimos un entorno de desarrollo de software podríamos querer un único compilador o un único depurador.

El patrón *Singleton* asegura que se creará una única instancia de una clase y que ésta proporcionará acceso unificado a la misma. En Java, se puede definir un *Singleton* mediante un constructor privado. Esto asegura que no pueda ser invocado fuera de la clase y por lo tanto, las clases cliente no pueden crear nuevas instancias. Podemos luego escribir código en la clase *Singleton* propiamente dicha para crear una única instancia y ofrecer acceso a ella (el Código 13.1 ilustra esta característica para una clase *Analizador*).

**Código 13.1**

El patrón Singleton

```
class Analizador
{
    private static Analizador instancia = new Analizador();

    public static Analizador getInstance()
    {
        return instancia;
    }

    private Analizador()
    {
        ...
    }
}
```

En este patrón:

- El constructor es privado, por lo que las instancias se pueden crear sólo mediante la clase propiamente dicha y tiene que ser en la parte estática de la clase (inicializaciones de campos estáticos o de métodos estáticos) ya que no existirá ninguna otra instancia.
- Se declara e inicializa un campo estático y privado con la única instancia del analizador.
- Se define el método estático `getInstance` para proporcionar acceso a la instancia única.

Ahora, los clientes de Singleton pueden usar este método estático para tener acceso al objeto analizador:

```
Analizador analizador = Analizador.getInstance();
```

## 13.7.4

## Método Fábrica

El patrón *método Fábrica* provee una interfaz para crear objetos pero deja que las subclases decidan la clase específica de objeto que se crea. Típicamente, el cliente espera una superclase o una interfaz del objeto actual y el método Fábrica provee las especializaciones.

Los iteradores de las colecciones son un ejemplo de esta técnica. Si tenemos una variable de tipo `Collection` podemos solicitar un iterador (usando el método `iterator`) y luego trabajar con dicho iterador (Código 13.2). En este ejemplo, el método `iterator` es el método Fábrica.

**Código 13.2**

Uso de un método Fábrica

```
public void procesar(Collection<Tipo> col)
{
    Iterator<Tipo> it = col.iterator();
}
```

Desde el punto de vista del cliente (en el código que se muestra en Código 13.2) estamos operando con objetos de tipo `Collection` e `Iterator`. En realidad, el tipo (dinámico) de la colección podría ser `ArrayList`, en cuyo caso el método `iterator` retorna un objeto de tipo `ArrayListIterator`; o podría ser un `HashSet` en donde `iterator` retorna un `HashSetIterator`. El método Fábrica se especializa en las subclases para retornar instancias especializadas del tipo de retorno «oficial».

Podemos hacer uso de este patrón en nuestra simulación *zorros-y-conejos* para desacoplar la clase `Simulador` de las clases específicas de animales. (Recuerde: en nuestra versión, `Simulador` está acoplada a las clases `Zorro` y `Conejo` porque crea las instancias iniciales.)

En su lugar, podemos introducir la interfaz `FabricaDeActor` e implementar esta interfaz para cada actor (por ejemplo `FabricaDeZorro` y `FabricaDeConejo`). La clase `Simulador` podría almacenar simplemente una colección de `FabricaDeActor` y debería solicitar que cada uno de ellos produzca un cierto número de actores. Por supuesto que cada fábrica debe producir un tipo diferente de actor, pero el `Simulador` habla con ellos a través de la interfaz `FabricaDeActor`.

### 13.7.5 Observador

En la discusión de varios de los proyectos de este libro hemos intentado separar el modelo interno de la aplicación de la manera en que se presenta en la pantalla (la vista). El patrón *Observador* proporciona una manera de llevar a cabo esta separación modelo-vista.

En términos más generales: el patrón *Observador* define una relación uno a varios, por lo tanto, cuando un objeto cambia su estado podrían modificarse muchos otros objetos. Logra este efecto con un grado muy bajo de acoplamiento entre los observadores y el objeto observado.

Podemos ver a partir de esto que el patrón *Observador* no sólo soporta una vista desacoplada del modelo sino que también permite varias vistas diferentes (ya sean alternativas o simultáneas). A modo de ejemplo podemos usar nuevamente nuestra simulación *zorros-y-conejos*.

En la simulación, presentamos en la pantalla las poblaciones de animales mediante una grilla animada de dos dimensiones. Existen otras posibilidades: podríamos haber preferido mostrar las poblaciones mediante un gráfico de líneas que represente el números de pobladores en función del tiempo o mediante un diagrama de barras animado (Figura 13.3). Podríamos aún querer visualizar todas las representaciones al mismo tiempo.

Para implementar el patrón *Observador* usamos dos clases abstractas: `Observable` y `Observer`<sup>6</sup>. La entidad observable (en nuestra simulación: el `Campo`) extiende la clase `Observable` y el observador (`VisorDelSimulador`) extiende la clase `Observer` (Figura 13.4).

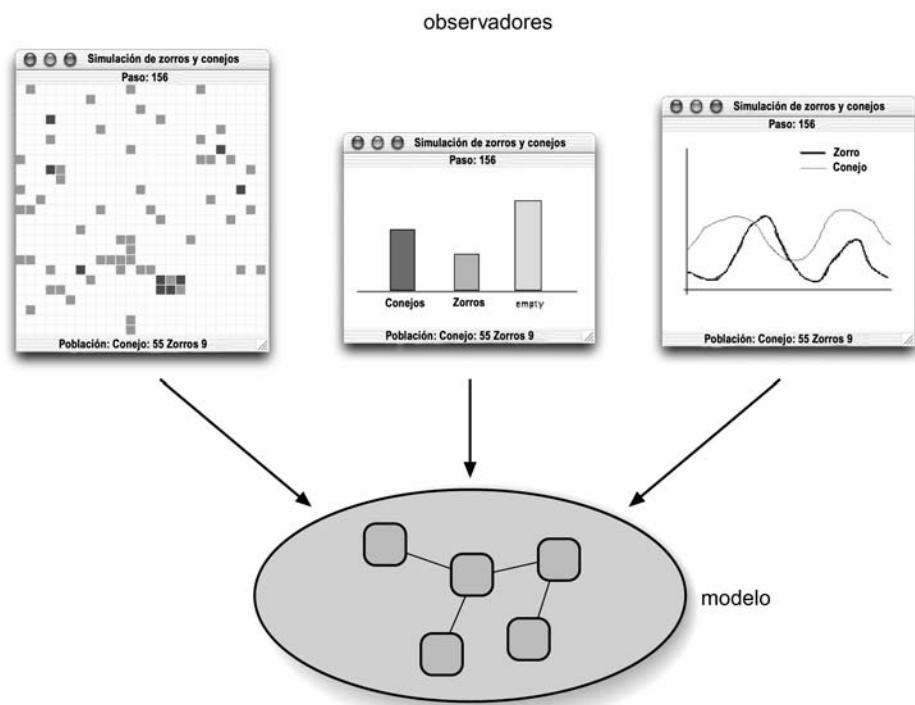
La clase `Observable` proporciona métodos a los observadores que les permite asociarse a la entidad observada. Esto asegura que el método `update` de los observadores se invoque cada vez que la entidad observada (el campo) invoca su método heredado `notify`. Los observadores actuales (los espectadores) pueden obtener un estado nuevo y actualizado del campo y mostrarlo nuevamente en la pantalla.

---

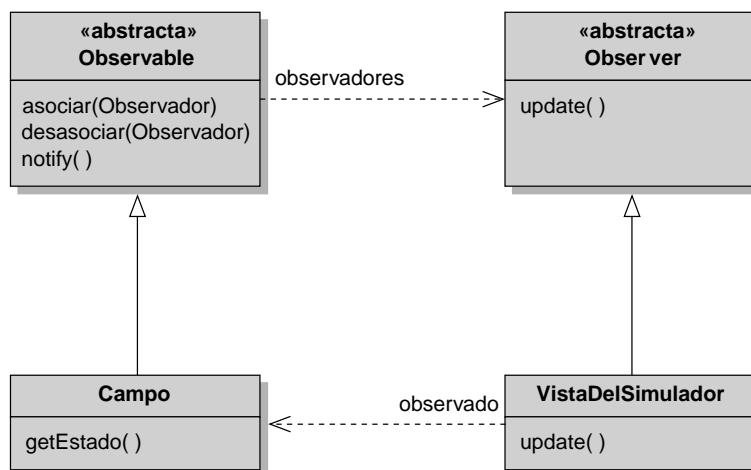
<sup>6</sup> En el paquete `java.util`, `Observer` es una interfaz con un único método: `update`.

**Figura 13.3**

Varias vistas de un mismo asunto

**Figura 13.4**

Estructura del patrón Observador



El patrón Observador también se puede usar para otros problemas distintos del que presenta la separación modelo-vista. Se puede aplicar siempre que el estado de uno o más objetos dependa del estado de otro objeto.

### 13.7.6

### Resumen de patrones

La discusión detallada sobre los patrones de diseño y sus aplicaciones está fuera del alcance de este libro. En esta sección hemos presentado sólo una breve idea de qué

son los patrones de diseño y hemos ofrecido descripciones informales de algunos de los patrones más comunes.

Sin embargo, esperamos que esta discusión sirva para mostrar hacia dónde ir a partir de aquí. Una vez que comprendemos cómo crear buenas implementaciones de clases con funcionalidad bien definida, podemos concentrarnos en decidir qué tipo de clases debemos tener en nuestra aplicación y cómo deben cooperar. Las buenas soluciones no siempre son obvias y por eso los patrones de diseño describen estructuras que han demostrado ser útiles una y otra vez para resolver tipos de problemas que se repiten.

A medida que adquiera más experiencia como desarrollador de software, empleará más tiempo en pensar sobre las estructuras de alto nivel en lugar de pensar en la implementación de métodos.

**Ejercicio 13.16** Otros tres patrones que se usan comúnmente son Estado, Estrategia y Visitante. Busque las descripciones de cada uno de ellos e identifique como mínimo un ejemplo de aplicación en el que considere que puede utilizarse cada patrón.

**Ejercicio 13.17** Al finalizar el desarrollo de un proyecto, encuentra que dos equipos que han trabajado independientemente en dos partes de una aplicación han implementado clases incompatibles. La interfaz de varias de las clases implementadas por uno de los equipos es algo diferente de la interfaz que el otro equipo espera para usar. Explique la manera en que el patrón Adaptador podría ayudar en esta situación, para evitar la reescritura de cualquiera de las clases existentes.

## 13.8

## Resumen

En este capítulo hemos avanzado un paso en términos de niveles de abstracción, pasamos de pensar sobre el diseño como una sola clase (o cooperación entre dos clases) al diseño de una aplicación como un todo. La decisión de qué clases se deben implementar y las estructuras de comunicación entre dichas clases es central en el diseño de un sistema de software orientado a objetos.

Algunas clases son bastante obvias y fáciles de descubrir. Hemos usado un método para identificar sustantivos y verbos en una descripción textual del problema como punto de partida. Después de descubrir las clases podemos usar tarjetas CRC y jugar con escenarios para diseñar las dependencias y los detalles de comunicación entre las clases y ajustar los detalles de las responsabilidades de cada una. Para los diseñadores menos experimentados, ayuda el atravesar los escenarios en grupo.

Se pueden usar las tarjetas CRC para refinar el diseño descendente de la definición de nombres de métodos y sus parámetros. Una vez que esto se ha logrado, se pueden codificar en Java las clases con métodos stubs y se pueden documentar las interfaces de las clases.

El seguir un proceso organizado como éste sirve a varios propósitos: asegura que los problemas potenciales con las primeras ideas de diseño se descubran antes de que se haya invertido mucho tiempo en la implementación. También permite que los programadores trabajen sobre la implementación de varias clases de manera independiente

sin tener que esperar a que se termine la implementación de una clase para comenzar a implementar otra.

Las estructuras de clases flexibles y extensibles no siempre son fáciles de diseñar. Los patrones de diseño se usan generalmente para documentar buenas estructuras que han demostrado ser útiles en la implementación de diferentes tipos de problemas. A través del estudio de patrones de diseño, un ingeniero de software puede aprender mucho sobre buenas estructuras de aplicación y mejorar las habilidades de diseño de una aplicación.

El mayor problema, el más importante es que la aplicación tenga una buena estructura. Cuando un ingeniero de software se vuelve más experimentado, empleará más tiempo en diseñar las estructuras de la aplicación y menos tiempo en escribir código.

Términos introducidos en este capítulo

**análisis y diseño, método sustantivo/verbo, tarjeta CRC, escenario, caso de uso, método stub, patrón de diseño**

## Resumen de conceptos

- **sustantivo/verbo** En un sistema, las clases se corresponden aproximadamente con los sustantivos de la descripción del problema; los métodos se corresponden con los verbos.
- **escenarios** Los escenarios (conocidos también como «casos de uso») se pueden usar para comprender las interacciones en un sistema.
- **prototipo** La construcción de un prototipo es la construcción de un sistema que funciona parcialmente, en el que algunas funciones de la aplicación están simuladas. Sirve para brindar comprensión sobre cómo funcionará realmente el sistema en las fases iniciales del proceso de desarrollo.
- **patrón de diseño** Un patrón de diseño es la descripción de un problema computacional común y la descripción de un pequeño conjunto de clases y su estructura de interacción que ayuda a resolver dicho problema.

**Ejercicio 13.18** Asuma que tiene un sistema de administración escolar para su escuela en el que existe una clase denominada **BaseDeDatos** (una clase bastante central) que contiene objetos tipo **Estudiante**; cada estudiante tiene una dirección contenida en un objeto **Dirección** (es decir, cada objeto **Estudiante** contiene una referencia a un objeto **Dirección**).

Desde la clase **BaseDeDatos**, se necesita acceder a la calle, la ciudad y el código postal de un estudiante. La clase **Direccion** tiene métodos de acceso para estos datos. Para diseñar la clase **Estudiante** se tienen dos opciones: implementar los métodos **getCalle**, **getCiudad** y **getCodigoPostal** en la clase **Estudiante** de tal manera que sólo pasen la llamada al objeto **Direccion** y luego manipular el resultado que éstos devuelven o bien, implementar un método **getDireccion** en la clase **Estudiante** que retorne a la clase **BaseDeDatos** el objeto **Direccion** completo y luego

permitir que el objeto `BaseDeDatos` invoque directamente a los métodos del objeto `Direccion`.

¿Cuál de estas alternativas es mejor? ¿Por qué? Confeccione un diagrama de clases para cada situación y enuncie los argumentos que justifican cada elección.

## CAPÍTULO

# 14

## Un estudio de caso

Principales conceptos que se abordan en este capítulo:

- desarrollo de una aplicación completa

Construcciones Java que se abordan en este capítulo

(En este capítulo no se introduce ninguna construcción nueva de Java.)

En este capítulo reunimos muchos de los principios de orientación a objetos que hemos introducido en este libro mediante la presentación de un extenso estudio de caso. Emprenderemos el estudio desde la fase inicial de la discusión del problema, a través del descubrimiento de las clases, el diseño y un proceso iterativo de implementación y prueba. A diferencia de los capítulos anteriores, no es nuestra intención introducir nuevos temas, sino que intentamos reforzar los temas presentados en la segunda mitad del libro tales como herencia, técnicas de abstracción, manejo de errores y diseño de una aplicación.

### 14.1

#### **El estudio de caso**

El estudio de caso que usaremos es el desarrollo de un modelo para una compañía de taxis. La compañía está considerando expandir sus operaciones a nuevas zonas de una cierta ciudad. La compañía opera con taxis y con minibuses. Los taxis dejan a sus pasajeros en sus respectivos destinos antes de recoger nuevos pasajeros mientras que los minibuses pueden recoger varios pasajeros en distintas ubicaciones durante el mismo viaje y trasladarlos a direcciones similares (por ejemplo, recogen varios huéspedes de distintos hoteles y los trasladan a diferentes terminales del aeropuerto). Basado en estimaciones del número de los clientes potenciales que tiene dicha zona, la compañía desea saber si será beneficioso expandirse y de ser así, cuántos taxis necesitarían para operar efectivamente.

##### 14.1.1 Descripción del problema

El siguiente párrafo presenta una descripción informal de los procedimientos de operación de la compañía de taxis, a la que se llegó tras varios encuentros con sus integrantes.

*La compañía opera tanto con taxis como con minibuses. Los taxis se usan para transportar a un individuo (o a un grupo pequeño de personas) desde una ubicación de la ciudad a otra; los minibuses se usan para recoger individuos en distintas ubicaciones y transportarlos a sus diferentes destinos. Cuando la compañía recibe la llamada proveniente de una persona, de un hotel, de un lugar de entretenimiento o de una organización turística, trata de asignar un vehículo para cumplimentar el viaje solicitado. Si no tiene vehículos disponibles, no implementa ninguna forma de sistema de espera. Cuando un vehículo llega a una determinada ubicación de salida para recoger un pasajero, el conductor lo notifica a la compañía; de manera similar, cuando se deja a un pasajero en su destino, el conductor también lo notifica a la compañía.*

Como hemos sugerido en el Capítulo 10, uno de los objetivos comunes del modelado es que nos ayude a aprender algo sobre la situación que se modela. Resulta útil identificar tempranamente qué es lo que deseamos aprender porque estos objetivos pueden tener mucha influencia sobre el diseño que producimos. Por ejemplo, si buscamos responder cuestiones relacionadas con la rentabilidad que se podría obtener operando con los taxis en esa zona, debemos asegurarnos de que podremos obtener información a partir del modelo que nos ayudará a evaluar la rentabilidad. Por lo tanto, debemos considerar estas dos cuestiones: la frecuencia con que se pierden pasajeros potenciales debido a que no hay vehículos disponibles para recogerlos y en el extremo opuesto, la cantidad de tiempo que los taxis permanecen ociosos por falta de pasajeros. Estas influencias no se encuentran en la descripción básica de la manera en que opera normalmente la compañía de taxis sino que representan escenarios que tendremos que atravesar en el momento en que construyamos el diseño.

Por lo tanto, podríamos agregar el siguiente párrafo a la descripción:

*El sistema almacena información sobre los pedidos de los pasajeros que no se pueden satisfacer; también proporciona información de la cantidad de tiempo que invierten los vehículos en cada una de las siguientes actividades: trasladar pasajeros, ir a las ubicaciones en las que se solicita un taxi y estar ociosos.*

Sin embargo, para desarrollar nuestro modelo nos centraremos en la descripción original de los procedimientos de la compañía y dejaremos las características adicionales como ejercicios para el lector.

**Ejercicio 14.1** ¿Considera que existe alguna información adicional que sería útil obtener a partir del modelo? De ser así, agregue estos requerimientos a las descripciones dadas y úselos en sus propias extensiones del proyecto.

## 14.2

## Análisis y diseño

Tal como lo hemos sugerido en el Capítulo 13, comenzaremos tratando de identificar las clases y sus interacciones en la descripción del sistema, mediante el método sustantivo/verbo.

### 14.2.1 Descubrir clases

Los siguientes sustantivos están presentes en la descripción, en su forma singular: compañía, taxi, minibús, individuo, persona, ubicación, destino, hotel, lugar de entre-

tenimiento, organización turística, vehículo, ubicación de salida, viaje, conductor y pasajero.

El primer punto a tener en cuenta es que sería un error armar un conjunto de clases directamente a partir de esta lista de sustantivos; las descripciones informales raramente se escriben de forma tal que se ajusten a esta correspondencia directa.

Generalmente, se hace necesario un primer refinamiento que consiste en identificar algunos *sinónimos* en la lista de sustantivos, es decir, palabras diferentes que se usan para nombrar la misma entidad. Por ejemplo, en este contexto, «individuo», «persona» y «viaje» funcionan como sinónimos de pasajero.

Un segundo refinamiento es la eliminación de aquellas entidades que realmente no es necesario que se modelen en el sistema. Por ejemplo, la descripción identifica varias maneras en que la compañía de taxis puede ser contactada: por individuos, por hoteles, por lugares de entretenimiento, por organizaciones turísticas. ¿Será realmente necesario contemplar estas distinciones? La respuesta dependerá de la información que queremos obtener a partir del modelo. Sería importante distinguirlas si, por ejemplo, quisieramos acordar descuentos a los hoteles que proveen un gran número de clientes o enviar material de publicidad a los lugares de entretenimiento que aún no solicitan el servicio. Si no se requiere este nivel de detalle, podemos simplificar el modelo «inyectando» pasajeros en él según algún patrón estadístico razonable.

**Ejercicio 14.2** Considere la simplificación del número de sustantivos asociados con los vehículos. En este contexto, los sustantivos «vehículo» y «taxi» ¿son sinónimos? ¿Es necesario la diferenciación entre «minibús» y «taxi»? ¿Qué ocurre con el sustantivo «conductor»? Justifique sus respuestas.

**Ejercicio 14.3** En este contexto, ¿es posible eliminar alguno de los siguientes sustantivos: «ubicación», «destino», «ubicación de salida» y considerarlos como sinónimos?

**Ejercicio 14.4** Identifique los sustantivos de alguna de las extensiones que agregó al sistema y realice las simplificaciones que considere necesarias.

### 14.2.2 Usar tarjetas CRC

La Figura 14.1 contiene un resumen de todos los sustantivos y los verbos asociados que quedaron después de llevar a cabo algunas simplificaciones en la descripción original. Ahora, cada uno de los sustantivos podría asignarse a una tarjeta CRC, preparada para registrar sus responsabilidades y colaboradores identificados.

A partir de este resumen, es claro que taxi y minibús son especializaciones de una clase más general de vehículo. La diferencia principal entre un taxi y un minibús es que un taxi siempre tiene el compromiso de recoger y transportar a un solo pasajero o a un grupo pequeño de pasajeros, pero un minibús trabaja simultáneamente con múltiples pasajeros *independientes*. La relación entre estas tres clases sugiere una jerarquía de herencia, en la que taxi y minibús representan subtipos de vehículo.

**Ejercicio 14.5** Cree tarjetas CRC concretas para los sustantivos/clases identificados en esta sección, con el propósito de atravesar los escenarios que se sugieren en la descripción del proyecto.

**Figura 14.1**

Asociaciones de sustantivos y verbos en la compañía de taxis

Sustantivos	Verbos
compañía	opera con taxis y con minibuses recibe una llamada asigna un vehículo
taxi	transporta un pasajero
minibús	transporta uno o más pasajeros
pasajero	
ubicación	
pasajero-fuente	llama a la compañía recoge un pasajero llega a la ubicación de salida notifica a la compañía la llegada
vehículo	notifica a la compañía que dejó al pasajero

**Ejercicio 14.6** Haga el mismo trabajo con alguna de sus propias extensiones para continuar con la próxima etapa.

### 14.2.3 Escenarios

La compañía de taxis no representa, realmente, una aplicación demasiado compleja. Encontraremos que gran parte de la interacción total del sistema se explora al considerar el escenario fundamental de tratar de satisfacer la solicitud de un pasajero para ir de una ubicación en la ciudad hacia otra. En la práctica, este escenario simple se puede descomponer en un conjunto de pasos que se siguen secuencialmente, desde la llamada inicial hasta el final del viaje:

- Hemos decidido que un pasajero-fuente sea el encargado de crear todos los nuevos objetos pasajero del sistema. Por lo tanto, una responsabilidad de *PasajeroFuente* es *Crear un pasajero* y *Pasajero* funciona como un colaborador.
- El pasajero-fuente llama a la compañía de taxis para solicitar que se recoja a un pasajero. Anotamos a *CompaniaDeTaxis* como un colaborador de *Pasajero-Fuente* y agregamos como responsabilidad *Pedir que se recoja un pasajero*; correspondientemente, agregamos en *CompaniaDeTaxis* la responsabilidad *Recibir el pedido de recoger un pasajero*. Asociado con el pedido, habrá un pasajero y una ubicación de salida; por lo tanto la *CompaniaDeTaxis* tiene como colaboradores a *Pasajero* y a *Ubicacion*. Cuando el pasajero-fuente llama a la compañía para realizar el pedido, se podría pasar al pasajero y a la ubicación de salida como objetos separados, sin embargo, es preferible asociarlos estrechamente. Por lo tanto, *Ubicacion* es un colaborador de *Pasajero* y será una responsabilidad del *Pasajero* *Proveer la ubicación de salida*.
- ¿En dónde se origina la ubicación de salida del pasajero? La ubicación de salida y el destino pueden decidirse en el momento en que se crea el pasajero. Por lo que agregamos a *PasajeroFuente* la responsabilidad *Generar la ubicación de salida y el destino de un pasajero*, teniendo como colaborador a *Ubicacion* y agregamos a *Pasajero* la responsabilidad *Recibir las ubicaciones de salida y de destino* y *Proveer la ubicación del destino*.

- Al recibir un pedido, la CompaniaDeTaxis tiene la responsabilidad de *Asignar un vehículo* lo que sugiere que otra de sus responsabilidades es *Almacenar una colección de vehículos* y sus colaboradores son Coleccion y Vehiculo. Dado que el pedido puede fallar (puede que no haya vehículos disponibles) se debe devolver al pasajero-fuente la indicación del éxito o del fracaso de la solicitud.
- No hay nada que indique si la compañía realiza distinciones entre taxis y minibuses cuando asigna un vehículo, por lo que no necesitamos tomar en cuenta este aspecto. Sin embargo, se puede asignar un vehículo sólo si está disponible, lo que significa que una responsabilidad del Vehiculo será *Indicar si está disponible*.
- Cuando se ha sido identificado un vehículo disponible, se debe dirigir a la ubicación de salida. La CompaniaDeTaxis tiene la responsabilidad de *Dirigir el vehículo a la ubicación de salida* con la correspondiente responsabilidad del Vehiculo de *Recibir la ubicación de salida*. Se agrega Ubicacion como un colaborador de Vehiculo.
- Al recibir una ubicación de salida, el comportamiento de los taxis y de los minibuses será bien diferente. Un taxi estará disponible cuando no está en camino a una ubicación de salida o está situado en la ubicación del destino; por lo tanto, la responsabilidad del Taxi es *Ir a la ubicación de salida*. Por el contrario, un minibus tiene que tratar con múltiples pasajeros; cuando recibe una ubicación de salida puede ocurrir que tenga que elegir entre varias ubicaciones alternativas posibles para dirigirse a la más cercana. Por lo tanto, agregamos al Minibus la responsabilidad de *Elegir la ubicación más cercana* con una Coleccion como colaborador, para mantener un conjunto de ubicaciones de destino posibles y poder elegir entre ellas. El hecho de que un vehículo se mueve entre ubicaciones sugiere que tiene la responsabilidad de *Mantener su ubicación actual*.
- Al arribar a una ubicación de salida, el Vehiculo debe *Notificar a la compañía la llegada a la ubicación de salida* teniendo como colaborador a CompaniaDeTaxis y a su vez, CompaniaDeTaxis debe *Recibir la notificación del arribo a la ubicación de salida*. En la vida real, un taxi encuentra a su pasajero por primera vez cuando arriba a la ubicación de salida, por lo que es el punto natural en que el vehículo puede recibir a su próximo pasajero. En el modelo, esta acción la realiza la compañía que recibió originalmente la ubicación de salida desde el pasajero-fuente. Responsabilidad de CompaniaDeTaxis: *Pasar pasajero al vehículo*; responsabilidad del Vehiculo: *Recibir pasajero* con Pasajero como otro colaborador de Vehiculo.
- Ahora, el vehículo solicita el destino pretendido por el pasajero. Responsabilidad del Vehiculo: *Solicitar ubicación del destino* y responsabilidad del Pasajero: *Proveer ubicación del destino*. Nuevamente en este punto el comportamiento de los taxis y de los minibuses es diferente. Un Taxi simplemente tiene la responsabilidad de *Ir al destino del pasajero*; un Minibus va a *Agregar ubicación a la colección de ubicaciones de destino* y seleccionará la más próxima.
- Al arribar al destino del pasajero, un Vehiculo tiene las responsabilidades de *Des cargar al pasajero* y *Notificar a la compañía el arribo del pasajero*. La CompaniaDeTaxis debe *Recibir la notificación del arribo del pasajero*.

Los pasos que hemos esquematizado representan la actividad fundamental de la compañía de taxis, que se repite una y otra vez cuando cada nuevo pasajero solicita el servicio. Un punto importante a destacar, sin embargo, es que nuestro modelo computa-

cional necesita ser capaz de reiniciar la secuencia para cada nuevo pasajero tan pronto como se recibe un nuevo pedido, aun cuando no se haya completado un pedido anterior. En otras palabras, dentro de un paso del programa, un vehículo podría estar dirigiéndose a una ubicación de salida mientras que otro podría estar llegando al destino del pasajero y un nuevo pasajero podría estar requiriendo un viaje.

**Ejercicio 14.7** Revise la descripción del problema y el escenario que hemos trabajado. ¿Existen algunos otros escenarios que se necesiten tener en cuenta antes de comenzar con el diseño de las clases? ¿Hemos cubierto adecuadamente lo que ocurre cuando no hay un vehículo disponible en el momento en que se recibe un pedido, por ejemplo? Si considera que hay más tareas que realizar, complete los escenarios de análisis.

**Ejercicio 14.8** ¿Considera que hemos descrito el escenario con un nivel correcto de detalle? Por ejemplo, ¿hemos incluido muy poco o demasiado detalle en la discusión de las diferencias entre los taxis y los minibuses?

**Ejercicio 14.9** ¿Considera que es necesario en esta etapa, tomar nota de cómo se mueven los vehículos entre las distintas ubicaciones?

**Ejercicio 14.10** ¿Considera que surgirá la necesidad de otras clases cuando se desarrolle la aplicación (clases a las que la descripción del problema no hace referencia inmediata)? De ser así, ¿por qué sería este el caso?

## 14.3

### Diseño de clases

En esta sección comenzaremos a movernos desde el diseño abstracto, de alto nivel y en papel hacia el esquema del diseño concreto de un proyecto BlueJ.

#### 14.3.1

##### Diseñar las interfaces de las clases

En el Capítulo 13 hemos sugerido que nuestro siguiente paso es la creación de un nuevo conjunto de tarjetas CRC para convertir las responsabilidades de cada clase en un conjunto de signaturas de métodos. No deseamos disminuir el énfasis de la importancia de este paso, pero lo dejamos en manos del lector y nos moveremos directamente al esquema del proyecto BlueJ, que contiene métodos y clases stub. Este esquema debiera proporcionar una buena idea de la complejidad del proyecto y también debiera servir para constatar que no nos hemos olvidado algo crucial en los pasos que hemos dado hasta ahora.

Es valioso tener en cuenta que, en cada etapa del ciclo de vida del proyecto, esperamos encontrar errores o cabos sueltos en lo que hemos hecho en las etapas anteriores. Esto no implica necesariamente que haya debilidades en nuestras técnicas o habilidades, es más una reflexión sobre el hecho de que el desarrollo del proyecto es generalmente un proceso de descubrimiento. Solamente explorando y probando cosas obtenemos mayor comprensión y conocimiento de lo que estamos tratando de lograr. Por lo tanto, ¡el descubrimiento de omisiones es realmente algo que habla de un aspecto positivo del proceso que estamos usando!

### 14.3.2 Colaboradores

Una vez que hemos identificado las colaboraciones entre las clases, una cuestión que necesitaremos registrar con frecuencia es la manera en que un objeto en particular obtiene referencias de sus colaboradores. Hay generalmente tres maneras distintas en que esto ocurre y representan con frecuencia tres patrones de interacción de objetos:

- Se recibe un colaborador como un argumento en un constructor. Un colaborador como éste generalmente estará almacenado en uno de los campos del nuevo objeto por lo que estará disponible durante la vida de dicho nuevo objeto. De esta manera, el colaborador podría ser compartido por varios objetos diferentes. Ejemplo: un objeto `PasajeroFuente` recibe al objeto `CompaniaDeTaxis` a través de su constructor.
- Se recibe un colaborador como un argumento en un método. La interacción con este colaborador es generalmente transitoria, sólo por el período de ejecución del método, sin embargo el objeto receptor puede elegir almacenar la referencia en uno de sus campos para una interacción más prolongada. Ejemplo: `CompaniaDeTaxis` recibe un colaborador `Pasajero` a través de sus métodos para manejar el pedido de un viaje.
- El objeto construye al colaborador por sí mismo. El colaborador será de uso exclusivo del objeto que lo construye a menos que se lo pase a otro objeto de una de las maneras descritas anteriormente. Si se construye dentro de un método, la colaboración será generalmente por un lapso corto, por la duración del bloque en el que se construye. Sin embargo, si el colaborador se almacena en un campo entonces la colaboración se mantiene probablemente durante el tiempo de vida total del objeto creador. Ejemplo: `CompaniaDeTaxis` crea una colección para almacenar sus vehículos.

**Ejercicio 14.11** Como en la próxima sección tratamos el proyecto `compania-de-taxis-esquema`, preste particular atención a los lugares en los que se crean los objetos y a la manera en que los objetos colaboradores toman conocimiento de los otros objetos. Trate de identificar como mínimo un ejemplo más de cada uno de los patrones que hemos descrito.

### 14.3.3 El esquema de implementación

El proyecto `compania-de-taxis-esquema` contiene un esquema de la implementación de las clases, las responsabilidades y las colaboraciones que hemos descrito durante el proceso de diseño. Le invitamos a recorrer el código fuente y asociar las clases concretas con las descripciones correspondientes a la Sección 14.2.3. El Código 14.1 muestra un esquema de la clase `Vehiculo` del proyecto.

#### Código 14.1

Un esquema de la clase `Vehiculo`

```
/*
 * Captura un esquema de los detalles de un vehículo.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2006.03.30
 */
public abstract class Vehiculo
```

**Código 14.1  
(continuación)**

Un esquema de la clase Vehiculo

```
{  
    private CompaniaDeTaxis compania;  
    // Lugar donde está ubicado el vehículo.  
    private Ubicacion ubicacion;  
    // Lugar hacia donde se dirige el vehículo.  
    private Ubicacion ubicacionDelDestino;  
  
    /**  
     * Constructor de la clase Vehiculo  
     * @param compania La compañía de taxis que no debe  
     * ser null.  
     * @param ubicacion El punto de partida del vehículo,  
     * no debe ser null.  
     * @throws NullPointerException Si la compañía o la  
     * ubicación es null.  
     */  
    public Vehiculo(CompaniaDeTaxis compania, Ubicacion  
    ubicacion)  
    {  
        if(compania == null) {  
            throw new NullPointerException("compañía");  
        }  
        if(ubicacion == null) {  
            throw new NullPointerException("ubicación");  
        }  
        this.compania = compania;  
        this.ubicacion = ubicacion;  
        ubicacionDelDestino = null;  
    }  
  
    /**  
     * Notificar a la compañía nuestra llegada a la  
     * ubicación de la salida.  
     */  
    public void notificarLlegadaASalida()  
    {  
        compania.llegadaASalida(this);  
    }  
  
    /**  
     * Notificar a la compañía nuestro llegada al destino  
     * del pasajero.  
     */  
    public void notificarLlegadaDelPasajero(Pasajero pasajero)  
    {  
        compania.llegadaADestino(this, pasajero);  
    }  
  
    /**  
     * Recibir una ubicación de salida.  
    }
```

**Código 14.1  
(continuación)**

Un esquema de la clase Vehiculo

```
* El manejo de la ubicación depende del tipo de vehículo.  
 * @param ubicacion La ubicación de la salida.  
 */  
public abstract void setUbicacionDeSalida(Ubicacion  
ubicacion);  
  
/**  
 * Recibir un pasajero.  
 * El manejo del pasajero depende del tipo de vehículo.  
 * @param pasajero El pasajero que será recogido.  
 */  
public abstract void recoger(Pasajero pasajero);  
  
/**  
 * @return Si el vehículo está o no está libre.  
 */  
public abstract boolean estaLibre();  
  
/**  
 * Dejar aquellos pasajeros cuyo destino es la  
 * ubicación actual.  
 */  
public abstract void dejarPasajero();  
  
/**  
 * @return Lugar en el que el vehículo está  
actualmente ubicado.  
 */  
public Ubicacion getUbicacion()  
{  
    return ubicacion;  
}  
  
/**  
 * Asignar la ubicación actual.  
 * @param ubicacion El lugar en el que está. No  
debe ser null.  
 * @throws NullPointerException Si la ubicación es null.  
 */  
public void setUbicacion(Ubicacion ubicacion)  
{  
    if(ubicacion != null) {  
        this.ubicacion = ubicacion;  
    }  
    else {  
        throw new NullPointerException();  
    }  
}  
  
/**
```

**Código 14.1  
(continuación)**

Un esquema de la clase Vehículo

```

        * @return Si este vehículo actualmente se dirige
hacia algún
        *                 destino o null si está ocioso.
        */
public Ubicacion getUbicacionDelDestino()
{
    return ubicacionDelDestino;
}

/**
 * Asignar la ubicación del destino.
 * @param ubicacion Hacia donde se dirige, no debe
ser null.
 * @throws NullPointerException Si la ubicación es
null.
 */
public void setUbicacionDelDestino(Ubicacion ubicacion)
{
    if(ubicacion != null) {
        ubicacionDelDestino = ubicacion;
    }
    else {
        throw new NullPointerException();
    }
}

/**
 * Blanquear la ubicación del destino.
 */
public void limpiarUbicacionDelDestino()
{
    ubicacionDelDestino = null;
}
}

```

Del proceso de creación del esquema del proyecto emergen varias cuestiones; aquí hay algunas de ellas:

- Es esperable encontrar algunas diferencias entre el diseño y la implementación, debidas a la naturaleza diferente de los lenguajes de diseño y de implementación. Por ejemplo, la discusión de escenarios sugirió que el PasajeroFuente debe tener la responsabilidad de *Generar una ubicación de salida y de destino para un pasajero* y que el Pasajero debe tener la responsabilidad de *Recibir las ubicaciones de salida y de destino*. En lugar de hacer corresponder estas responsabilidades con invocaciones a métodos individuales, la implementación más natural en Java es escribir algo similar a

```
new Pasajero(new Ubicacion( ... ), new Ubicacion( ... ))
```

- Nos hemos asegurado de que el esquema de nuestro proyecto esté suficientemente completo como para que compile exitosamente. Esto no siempre es necesario en

esta etapa, pero su consecuencia es que la tarea de desarrollo incremental de la próxima etapa será un poco más fácil. Sin embargo, tiene la correspondiente desventaja de olvidar algunos tramos de código que serán potencialmente más difíciles de encontrar porque el compilador no señalará los cabos sueltos.

- Los elementos compartidos y los distintivos de las clases `Vehiculo`, `Taxi` y `Minibus` sólo comienzan a tomar forma realmente cuando nos movemos hacia su implementación. Por ejemplo, las diferentes maneras en que los taxis y los minibuses responden a la solicitud de un viaje se refleja en el hecho de que `Vehiculo` define `setUbicacionDelDestino` como un método abstracto, que tendrá implementaciones concretas y diferentes en las subclases. Por otro lado, aun cuando los taxis y los minibuses tienen diferentes formas de decidir hacia dónde se dirigen, pueden compartir el concepto de tener una única ubicación de destino. Esto se ha implementado en la superclase mediante el campo `ubicacionDelDestino`.
- En dos puntos del escenario, se espera que un vehículo notifique a la compañía su arribo ya sea a un lugar de salida o a uno de destino. Existen por lo menos dos maneras posibles de organizar esto en la implementación. La manera directa es que un vehículo almacene una referencia de su compañía, lo que significa que debiera existir una asociación explícita entre las dos clases en el diagrama de clases.

Una alternativa es el uso del patrón *Observador* introducido en el Capítulo 13 con `Vehiculo` extendiendo a la clase `Observable` y `CompaniaDeTaxis` implementando la interfaz `Observer`. Se reduce el acoplamiento directo entre `Vehiculo` y `CompaniaDeTaxis` pero el acoplamiento implícito aún permanece y el proceso de notificación es un poco más complejo de programar.

- Llegado a este punto, no ha habido ninguna discusión sobre el número de pasajeros que puede trasladar un minibús. Presumiblemente ¿podría haber minibuses de diferentes tamaños? Este aspecto de la aplicación ha sido diferido para resolverlo más adelante.

No existe ninguna regla absoluta que indique hasta dónde se debe llegar exactamente con el esquema de implementación de una aplicación en particular. El propósito del esquema de implementación no es crear un proyecto que funcione completamente sino registrar el diseño de la estructura del esquema de la aplicación (que ha sido desarrollado anteriormente mediante las actividades con las tarjetas CRC). Si revisa las clases del proyecto `compania-de-taxis-esquema` puede considerar que en este caso hemos ido muy lejos o puede ser que le parezca que no hemos ido suficientemente lejos. Desde el lado positivo, al intentar la creación de una versión que por lo menos compile, encontramos ciertamente que nos vimos forzados a pensar en la jerarquía de herencia `Vehiculo` con algún nivel de detalle, en especial en aquellos métodos que debieron implementarse en la superclase y que hubiera sido mejor dejarlos como abstractos. Desde el lado negativo, siempre existe el riesgo de tomar decisiones de implementación demasiado anticipadas: por ejemplo, comprometerse con alguna clase de estructura de datos que podría ser mejor dejarla para más adelante o, tal como hicimos aquí, elegir desechar el patrón *Observer* en función de un abordaje más directo.

**Ejercicio 14.12** Para cada una de las clases del proyecto, busque la interfaz y escriba una lista de las pruebas unitarias que se deberían usar para probar la funcionalidad de la clase.

**Ejercicio 14.13** El proyecto `compania-de-taxis-esquema` define una clase `Demo` para crear un par de objetos `PasajeroFuente` y `CompaniaDeTaxis`. Cree un

objeto `Demo` y pruebe su método `recogerTest`. ¿Por qué el objeto `Compania-DeTaxis` es incapaz de satisfacer la solicitud de un viaje en esta etapa?

**Ejercicio 14.14** ¿Le parece que deberíamos haber desarrollado más el código para permitir por lo menos que una solicitud de viaje fuera exitosa? De ser así, ¿cuán lejos cree que debería ir el desarrollo?

#### 14.3.4 Prueba

Una vez comenzada la implementación, no debemos ir mucho más allá antes de empezar a considerar cómo probaremos la aplicación. No queremos cometer el error de idear las pruebas una vez que se complete la implementación. Ya podemos poner algunas pruebas en su lugar que evolucionarán gradualmente a medida que evolucione la implementación. Intente hacer los siguientes ejercicios para percibir por qué es posible escribir las pruebas en esta temprana etapa.

**Ejercicio 14.15** El proyecto `compania-de-taxis-esquema-prueba` incluye tres clases JUnit sencillas de prueba que contienen algunas pruebas iniciales, experimente con ellas. Agregue cualquier otra prueba que considere apropiada en esta etapa del desarrollo para sentar las bases de un conjunto de pruebas que se usarán durante el futuro desarrollo. ¿Tiene importancia el hecho de que las pruebas que creamos fallen en esta etapa?

**Ejercicio 14.16** La clase `Ubicación` actualmente no contiene campos ni métodos. La extensión del desarrollo de esta clase, ¿de qué manera es probable que afecte a las pruebas de las clases existentes?

#### 14.3.5 Algunos asuntos pendientes

Uno de los asuntos más importantes que aún no hemos intentando abordar es cómo organizar la secuencia de varias actividades: las solicitudes de los pasajeros, los movimientos de los vehículos, etc. Otro es que no se les ha dado a las ubicaciones una forma concreta y detallada por lo que el movimiento no tienen ningún efecto. A medida que desarrollemos la aplicación emergerán las resoluciones de estos asuntos y de algunos otros.

### 14.4

## Desarrollo iterativo

Obviamente, aún tenemos un largo camino que recorrer desde el desarrollo del esquema de la implementación hasta la versión final, sin embargo, en lugar de sentirnos desbordados por la magnitud del trabajo podemos hacer cosas más manejables identificando algunos pasos discretos para llegar al objetivo último y seguir un proceso de desarrollo iterativo.

#### 14.4.1 Pasos del desarrollo

El planificar algunos pasos para el desarrollo nos ayuda a considerar cómo podemos dividir un problema grande en varios problemas más pequeños. Individualmente, estos problemas pequeños es probable que sean menos complejos y más manejables que un solo gran problema, pero todos juntos se combinarán para formar un todo. A medida

que resolvemos los pequeños problemas podremos encontrarnos con que necesitamos dividirlos aún más. Además, podríamos encontrar que algunas de nuestras suposiciones originales eran erróneas o que nuestro diseño es inadecuado de alguna manera. Este proceso de descubrimiento, cuando se combina con un enfoque de desarrollo iterativo, significa que obtenemos retroalimentación valiosa para nuestro diseño y para las decisiones que tomamos en una etapa suficientemente temprana como para permitirnos incorporarlas nuevamente en un proceso flexible y evolutivo.

El considerar los pasos en los que se dividirá el problema tiene la ventaja adicional de ayudar a identificar algunos de los modos en que están interconectadas las partes de la aplicación. En un proyecto grande, esto nos ayuda a identificar las interfaces entre los componentes. Identificar los pasos también nos ayuda a planificar los tiempos del proceso de desarrollo.

Es importante que cada paso del desarrollo iterativo represente un punto claramente identificable en la evolución de la aplicación en vistas de los requerimientos totales. En particular, necesitamos ser capaces de determinar cuándo se ha completado cada paso. La finalización podría marcarse mediante la ejecución de un conjunto de pruebas y la revisión de los logros obtenidos en la etapa, a modo de ser capaces de incorporar en los siguientes pasos cualquier lección que se haya aprendido.

Esta es una serie posible de pasos para el desarrollo de la aplicación de la compañía de taxis:

- Habilitar la parte del sistema correspondiente a recoger un único pasajero y conducirlo a su destino para un único taxi.
- Proporcionar suficientes taxis como para permitir que varios pasajeros independientes sean recogidos y conducidos a sus destinos simultáneamente.
- Habilitar la parte que permite que se recoja un único pasajero y se lo conduzca a su destino mediante un único minibús.
- Asegurarse de que se registre la información de aquellos pasajeros para los que no hay vehículo disponible.
- Habilitar el sistema para que un minibús recoja varios pasajeros y los conduzca simultáneamente a sus respectivos destinos.
- Proveer una IGU para mostrar las actividades de todos los vehículos y los pasajeros activos en la simulación.
- Asegurarse de que los taxis y los minibuses sean capaces de operar en simultáneo.
- Proveer toda la funcionalidad restante, incluyendo todos los datos estadísticos.

No discutiremos la implementación de estos pasos detalladamente sino que completaremos la aplicación hasta el punto en el que usted mismo podría ser capaz de agregar el resto de la funcionalidad.

**Ejercicio 14.17** Evalúe críticamente la lista de pasos que hemos esquematizado con las siguientes cuestiones en mente. ¿Considera que el orden es el adecuado? En cuanto al nivel de complejidad de cada paso, ¿considera que es demasiado alto, demasiado bajo o que es adecuado? ¿Falta algún paso? Revise la lista de manera que satisfaga su propia visión del proyecto.

**Ejercicio 14.18** Los criterios de terminación de cada etapa (finalizar con las pruebas), ¿son suficientemente obvios? De ser así, documente algunas pruebas para cada etapa.

#### 14.4.2 La primera etapa

En la primera etapa queremos ser capaces de crear un único pasajero que sea recogido por un único taxi y dejarlo en su destino. Esto quiere decir que tendremos que trabajar sobre varias clases: seguramente sobre `Ubicacion`, `Taxi` y `CompaniaDeTaxis` y posiblemente con algunas otras más. Además, tendremos que ingeniarlos para simular el tiempo que transcurre a medida que el taxi se mueve por la ciudad. Esto sugiere que podríamos reutilizar algunas de las ideas que involucran a los actores y que hemos visto en el Capítulo 10.

El proyecto `compania-de-taxis-etapa-uno` contiene una implementación de los requerimientos de esta primera etapa. Las clases han sido desarrolladas hasta el punto en que un taxi recoge y deja un pasajero en su destino. El método `ejecutar` de la clase `Demo` opera con este escenario. Sin embargo, las clases de prueba son la parte realmente más importante de esta etapa: `UbicacionTest`, `PasajeroTest`, `PasajeroFuenteTest` y `TaxiTest`, y las discutiremos en la Sección 14.4.3.

En lugar de discutir detalladamente este proyecto, describiremos simplemente algunas de las cuestiones que surgen de su desarrollo a partir de la versión previa del esquema. usted deberá suplementar esta discusión con la lectura del código.

Los objetivos de la primera etapa fueron deliberadamente determinados para que sean bastante modestos, aunque relevantes para la actividad fundamental de la aplicación: recoger y trasladar pasajeros. Hubo buenos motivos para esto: al establecer un objetivo modesto la tarea parece ser posible de ser llevada a cabo en un tiempo razonablemente breve. Al establecer un objetivo relevante, el trabajo nos va acercando a completar el proyecto. Estos factores nos ayudan a mantener nuestra motivación alta.

Nos apropiamos del concepto de actores del proyecto `zorros-y-conejos` del Capítulo 10. Para esta etapa, sólo los taxis necesitan ser actores a través de su superclase `Vehiculo`. En cada paso, un taxi se mueve hacia la ubicación de un destino o bien permanece ocioso (Código 14.2). Aunque todavía no registramos ninguna estadística en esta etapa, es simple y conveniente registrar el número de pasos en que permanecen ociosos los vehículos. Esto anticipa parte del trabajo de nuestras siguientes etapas.

#### Código 14.2

La clase `Taxi` como un actor

```
 /**
 * Un taxi puede trasladar un solo pasajero.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2006.03.30
 */
public class Taxi extends Vehiculo
{
    private Pasajero pasajero;

    /**

```

**Código 14.2****(continuación)**

La clase Taxi como un actor

```
* Constructor de objetos de la clase Taxi
 * @param compania La compañía de taxis que no debe
ser null.
 * @param ubicacion El punto de salida del vehículo
que no debe ser null.
 * @throws NullPointerException Si la compañía o la
ubicación es null.
 */
public Taxi(CompaniaDeTaxis compania, Ubicacion ubicacion)
{
    super(compania, ubicacion);
}

/**
 * Lleva a cabo las acciones de un taxi.
 */
public void actuar()
{
    Ubicacion destino = getUbicacionDelDestino();
    if(destino != null) {
        // Busca hacia donde se moverá a continuación.
        Ubicacion siguiente =
getUbicacion().siguienteUbicacion(destino);
        setUbicacion(siguiente);
        if(siguiente.equals(destino)) {
            if(pasajero != null) {

notificarLlegadaDePasajero(pasajero);
                dejarPasajero();
            }
            else {
                notificarLlegadaASalida();
            }
        }
    }
    else {
        incrementarContadorDeOcio();
    }
}

/**
 * @return Si el taxi está libre o no.
 */
public boolean estaLibre()
{
    return getUbicacionDelDestino() == null &&
pasajero == null;
}

/**
```

**Código 14.2  
(continuación)**

La clase Taxi como un actor

```
        * Recibir la ubicación de comienzo de un viaje. Se
        convierte
        * en la ubicación del destino.
        * @param ubicacion La ubicación de la salida del
        viaje.
        */
public void setUbicacionDeSalida(Ubicacion ubicacion)
{
    setUbicacionDelDestino(ubicacion);
}

/**
 * Recibir un pasajero.
 * Asigna el destino del pasajero como la ubicación
del destino del taxi.
 * @param pasajero El pasajero.
 */
public void recoger(Pasajero pasajero)
{
    this.pasajero = pasajero;
setUbicacionDelDestino(pasajero.getDestino());
}

/**
 * Dejar un pasajero.
 */
public void dejarPasajero()
{
    pasajero = null;
limpiarUbicacionDelDestino();
}

/**
 * Retorna los detalles del taxi, en este caso dónde
está ubicado.
 * @return Una cadena de representación del taxi.
 */

public String toString()
{
    return "Taxi en " + getUbicacion();
}
}
```

La necesidad de modelar el movimiento requiere que la clase Ubicacion se implemente de manera más completa que en el esquema. En apariencia, debiera ser un contenedor relativamente simple de una posición bidimensional en una grilla rectangular. Sin embargo, en la práctica, también se necesita proveer a la clase de una evaluación de la coincidencia entre dos ubicaciones (`equals`) y de una manera para que un vehículo

encuentre hacia dónde debe moverse a continuación, basándose en su ubicación actual y en su destino (`ubicacionSigiente`). En esta etapa, no se impusieron límites a la zona de la grilla (excepto que las coordenadas sean positivas) pero surge la necesidad, en una etapa posterior, de que algo registre los límites de la zona en la que opera la compañía.

Una de las cuestiones más importantes a la que apuntamos fue la manera de manejar la asociación entre un pasajero y un vehículo, entre la solicitud de un viaje y el punto de arribo del vehículo. Pese a que se requería manejar un único taxi y un único pasajero, intentamos tener en mente que finalmente habrá múltiples solicitudes de viajes en cualquier momento. En la Sección 14.2.3 decidimos que un vehículo recibiría a su pasajero cuando notifica a la compañía que arribó al punto en que lo recogerá. Por lo tanto, cuando se recibe una notificación, la compañía necesita ser capaz de reconocer qué pasajero ha sido asignado a qué vehículo. La solución que elegimos fue que la compañía almacene el par vehículo-pasajero en un mapa. Cuando el vehículo notifica a la compañía que llegó a la ubicación de salida del viaje, la compañía le pasa el correspondiente pasajero. Sin embargo existen varios motivos por los que esta solución no es perfecta y exploraremos estas cuestiones en los siguientes ejercicios.

Una situación de error que hemos apuntado fue que podría no haberse encontrado a ningún pasajero cuando el vehículo llegó al punto de salida y esto podría ser el resultado de un error de programación, por lo que definimos la clase `PasajeroPerdido-Exception` que corresponde a una excepción no comprobada.

Como se requirió un solo pasajero en esta etapa, el desarrollo de la clase `Pasajero-Fuente` fue diferido a una etapa posterior. En su lugar, los pasajeros se crean directamente en la clase `Demo` y en las clases de prueba.

**Ejercicio 14.19** Si todavía no lo hizo, dé una mirada a la implementación del proyecto `compania-de-taxis-etapa-uno`. Asegúrese de que comprende cómo se efectúa el movimiento del taxi mediante su método `actuar`.

**Ejercicio 14.20** ¿Considera que el objeto `CompaniaDeTaxis` debería mantener listas separadas de aquellos vehículos que están libres y de los que no, para mejorar la eficiencia de su asignación? ¿En qué puntos se debería mover un vehículo entre dichas listas?

**Ejercicio 14.21** La siguiente etapa planificada de la implementación es proporcionar múltiples taxis para trasladar simultáneamente a múltiples pasajeros. Revise la clase `CompaniaDeTaxis` con este objetivo en mente. ¿Considera que ya soporta esta funcionalidad? Si no es así, ¿qué cambios se requieren?

**Ejercicio 14.22** Revise la manera en que se almacenan las asociaciones `vehiculo-pasajero` en el mapa `asignaciones` de `CompaniaDeTaxis`. ¿Puede ver alguna debilidad en este abordaje? ¿Soporta el hecho de que se recoja más de un pasajero en la misma ubicación? ¿Puede ocurrir que un vehículo necesite registrar múltiples asociaciones?

**Ejercicio 14.23** Si observa algún problema en la manera en que se almacenan las asociaciones `vehiculo-pasajero`, ¿sería de ayuda la creación de una identificación única para cada asociación, por ejemplo, un número de registro? De ser así, ¿es necesario modificar alguna de las signaturas de los métodos de la jerarquía `Vehiculo`? Implemente una versión mejorada que soporte los requerimientos de todos los escenarios existentes.

#### 14.4.3 Probar la primera etapa

Como parte de la implementación de la primera etapa desarrollamos dos clases de prueba: `UbicacionTest` y `TaxiTest`. La primera controla la funcionalidad básica de la clase `Ubicacion` que es crucial para el movimiento correcto de los vehículos. La segunda está diseñada para probar que se recoge al pasajero y se le conduce a su destino en el número de pasos correcto y que el taxi queda libre inmediatamente después de que deja a su pasajero. Con el objetivo de desarrollar el segundo conjunto de pruebas, se mejoró la clase `Ubicacion` con el método `distancia` que proporciona el número de pasos requeridos para moverse entre dos ubicaciones<sup>1</sup>.

En la operación normal, la aplicación se ejecuta silenciosamente y sin una IGU no existe forma visual de monitorear el progreso de un taxi. Un abordaje podría consistir en agregar sentencias de impresión en los métodos más importantes de las clases `Taxi` y `CompaniaDeTaxis`. Sin embargo, BlueJ ofrece la alternativa de fijar un punto de interrupción, por ejemplo, en el método `actuar` de la clase `Taxi` de manera que sería posible «observar» el movimiento de un taxi mediante su inspección.

Una vez que se alcance un nivel de confianza razonable en la etapa actual de la implementación, simplemente dejamos las sentencias de impresión en los métodos de notificación de `CompaniaDeTaxis` para proporcionar un mínimo de retroalimentación al usuario.

Como testimonio del valor de desarrollar las pruebas en paralelo con la implementación, es valioso registrar que las clases de prueba existentes nos permiten identificar y corregir dos serios errores en nuestro código.

**Ejercicio 14.24** Revise las pruebas implementadas en las clases de prueba de `compania-de-taxis-etapa-uno`. ¿Es posible usar estas pruebas como pruebas de regresión durante las siguientes etapas, o se requieren cambios sustanciales?

**Ejercicio 14.25** Implemente pruebas adicionales y otras clases de prueba que considere necesarias para incrementar su nivel de confianza en la implementación actual. Solucione cualquier error que descubra durante este proceso.

#### 14.4.4 Una etapa de desarrollo más avanzada

No es nuestra intención discutir la manera de completar el desarrollo de la aplicación de la compañía de taxis ya que sería poco lo que usted ganaría con esto. En cambio, presentaremos brevemente la aplicación en un estado más avanzado y le animamos a que complete el resto a partir de allí.

Esta etapa más avanzada se puede encontrar en el proyecto `compania-de-taxis-etapa-avanzada` que maneja varios taxis y varios pasajeros y en el que la IGU proporciona una visión progresiva de los movimientos de ambos (Figura 14.2). Aquí presentamos un esquema de algunos de los principales desarrollos de esta versión a partir de la primera.

---

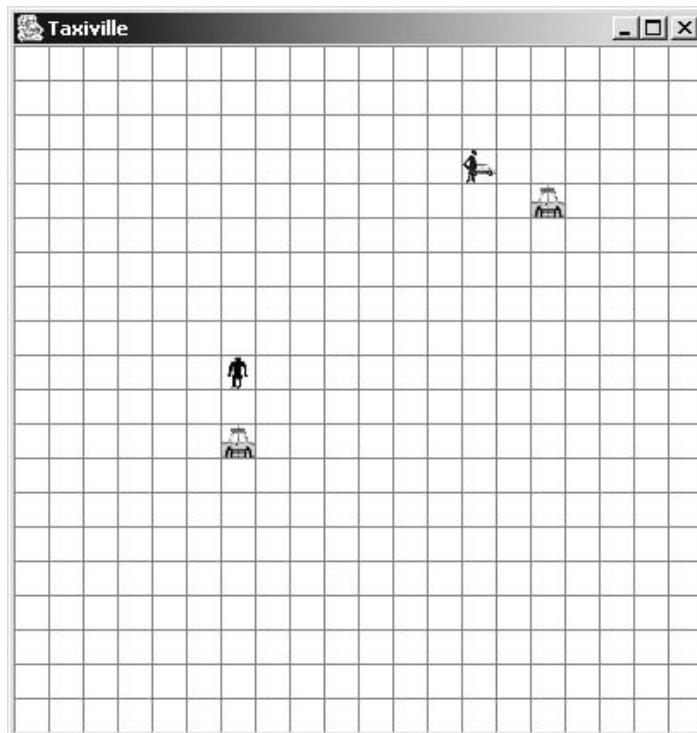
<sup>1</sup> Anticipamos que este método tendrá un extenso uso más adelante, en el desarrollo de la aplicación, para permitir que la compañía asigne los vehículos basándose en la cercanía de cada uno de ellos al punto de salida.

- La clase `Simulacion` maneja a los actores, tal como lo hicimos en el proyecto *zorros-y-conejos*. Los actores son los vehículos y el pasajero-fuente, y se proporciona una IGU mediante la clase `CiudadIGU`. Después de cada paso, la simulación hace una pausa breve de modo que la IGU no cambia demasiado rápidamente.
- La necesidad de una clase similar a `Ciudad` se identificó durante el desarrollo de la etapa uno. El objeto `Ciudad` define las dimensiones de la grilla que representa a la ciudad y contiene una colección de todos los elementos que nos interesan de la ciudad: los vehículos y los pasajeros.
- Los elementos de la ciudad podrían implementar opcionalmente la interfaz `Drawable` que permite que la IGU los muestre. Con este fin se proporcionan las imágenes de los vehículos y de las personas en la carpeta `images`, situada dentro de la carpeta del proyecto.
- La clase `Taxi` implementa la interfaz `Drawable` y devuelve imágenes alternativas a la IGU que dependen de si está ocupado o vacío. Los archivos de imagen que existen en la carpeta `images` sirven para que se haga lo mismo para un minibús.
- La clase `PasajeroFuente` ha sido rediseñada significativamente a partir de la versión anterior para mejorar su rol como actor. Además, mantiene la cantidad de viajes perdidos para un posterior análisis estadístico.
- La clase `CompaniaDeTaxis` es la responsable de crear los taxis que se usan en la simulación.

Cuando explore el código del proyecto *compania-de-taxis-etapa-avanzada* encontrará ilustraciones de varios de los tópicos que hemos cubierto en la segunda mitad de este libro: herencia, polimorfismo, clases abstractas, interfaces y manejo de errores.

**Figura 14.2**

Una visualización de la ciudad



**Ejercicio 14.26** Agregue controles de consistencia mediante aserciones y lanzamientos de excepciones en cada clase, para resguardarlas de usos inapropiados. Por ejemplo: asegúrese de que nunca se pueda crear un Pasajero con ubicaciones de salida y de destino idénticas; asegúrese de que no se solicite a un taxi que se dirija a una dirección de salida cuando ya está en ese lugar, etc.

**Ejercicio 14.27** Informe las estadísticas que se obtienen de los taxis y del pasajero-fuente: el tiempo ocioso de los taxis y los viajes que se han perdido. Experimente con diferentes cantidades de taxis para ver cómo varía el balance entre estos dos conjuntos de datos.

**Ejercicio 14.28** Adapte las clases de vehículos de modo que registren la cantidad de tiempo que emplean en viajar a las ubicaciones de salida y a los destinos de los pasajeros. ¿Puede ver la existencia de un posible conflicto con los minibuses?

#### 14.4.5 Más ideas para desarrollar

La versión de la aplicación provista en el proyecto *compania-de-taxis-etapa-avanzada* representa un punto significativo en el desarrollo, en vías de su implementación completa, sin embargo, aún existen un montón de cosas que se pueden agregar. Por ejemplo, todavía no hemos trabajado demasiado sobre la clase *Minibus* de modo que hay muchos desafíos que se pueden encontrar al completar su implementación. La principal diferencia entre los minibuses y los taxis es que un minibús está comprometido con múltiples pasajeros mientras que un taxi sólo con uno. El hecho de que un minibús todavía está trasladando a un solo pasajero no impide que se envíe a recoger a otro pasajero. De manera similar, si ya está en camino hacia una parada para recoger un nuevo pasajero, aún podría aceptar otro pedido más de viaje. Estas cuestiones generan preguntas sobre cómo deben organizarse las prioridades de un minibús. ¿Podría ocurrir que un pasajero termine siendo trasladado de un lado a otro mientras el minibús se ocupa de responder las demandas de distintos viajes, y de esta manera el pasajero no logra nunca llegar a su destino? ¿Qué significa no estar libre para un minibús? ¿Significa que tiene el máximo de pasajeros que puede trasladar (o sea, que está completo) o que tiene suficientes pedidos de viajes como para completarlo? Suponga que, como mínimo, uno de esos viajes alcanzara su destino antes de llegar a una nueva parada: ¿significa esto que podría aceptar más pedidos de viajes que su capacidad máxima?

Otra área a desarrollar es la asignación de vehículos. La compañía de taxis no opera, por el momento, de manera particularmente inteligente. ¿Cómo debiera decidir qué vehículo enviar cuando existe más de un vehículo disponible? No se hizo ningún intento para asignar vehículos en base a sus distancias respecto de la ubicación de salida. La compañía podría usar el método *distancia* de la clase *Ubicacion* para encontrar el vehículo libre que esté más cerca del punto de salida. Esta forma de asignación, ¿tendría una influencia significativa en el tiempo promedio de espera de los pasajeros? ¿Cómo se podría capturar información sobre el tiempo que tienen que esperar los pasajeros hasta ser recogidos? Con el objetivo de reducir los tiempos de espera, ¿qué pasaría si los taxis ociosos se dirigieran hacia una ubicación central, preparados para su próxima ubicación de salida? El tamaño de la ciudad, ¿tiene alguna influencia sobre la eficiencia de este abordaje? Por ejemplo, en una ciudad grande ¿sería mejor que los taxis ociosos se distribuyeran en distintos lugares en vez de que se centralicen en único lugar?

¿Podría usarse la simulación para modelar la competencia entre compañías de taxis que operan en la misma zona de la ciudad? En este caso, se debieran crear varios objetos CompaniaDeTaxis y el pasajero fuente podría ubicar pasajeros en ellas competitivamente en base al menor tiempo en que pueden ser recogidos. ¿Es este un cambio demasiado fundamental a partir de la aplicación existente?

#### 14.4.6 Reusabilidad

Nuestro objetivo real ha sido simular la operación de vehículos con el propósito de evaluar la factibilidad comercial de operar un negocio, aunque puede haber notado que las partes sustanciales de la aplicación también podrían ser útiles, una vez que el negocio esté efectivamente operando.

Si asumimos que desarrollamos un algoritmo de asignación inteligente para nuestra simulación con el fin de decidir qué vehículo debe responder a cada llamada, o que hemos armado un buen esquema para decidir hacia dónde dirigir los vehículos mientras están ociosos, podríamos decidir usar los mismos algoritmos cuando la compañía opere realmente. También podría ayudar la representación visual de cada ubicación del vehículo.

En otras palabras, existe potencial suficiente como para convertir la simulación de la compañía de taxis en un sistema de administración de taxis, que ayude a la compañía en sus operaciones reales. Por supuesto que la estructura de la aplicación cambiaría: el programa podría no controlar y mover a los taxis pero se podrían registrar sus ubicaciones mediante el uso de receptores GPS (*global position system*) en cada vehículo. Sin embargo, se podrían reutilizar varias de las clases desarrolladas para la simulación realizando pocos o ningún cambio. Esto ilustra el poder de reutilización que hemos obtenido a partir de una buena estructura de clases y de un buen diseño.

### 14.5

#### Otro ejemplo

Existen muchos otros proyectos que se podrían asumir siguiendo líneas similares a la aplicación de la compañía de taxis. Una alternativa popular es la cuestión de cómo asignar ascensores en un edificio grande. La coordinación entre los ascensores es particularmente significativa. Además, en un edificio cerrado, podría ser posible estimar el número de personas en cada piso y usarlo para anticipar la demanda. También existen comportamientos vinculados con el tiempo a tener en cuenta: las llegadas a la mañana, las salidas a la tarde, las actividades a la hora del almuerzo.

Emplee el abordaje que hemos delineado en este capítulo para implementar la simulación de un edificio en el que se desean instalar uno o más ascensores.

### 14.6

#### Para ir más lejos

Nosotros podemos conducirle un poco más lejos, sólo presentándole nuestras propias ideas de proyectos y mostrándole cómo los desarrollaríamos. Usted encontrará que puede ir mucho más lejos si desarrolla sus propias ideas y proyectos y las implementa a su manera. Seleccione un tema de su interés y trabájelo a través de las etapas que

hemos esquematizado: analizar el problema, armar varios escenarios, construir un diseño, planificar algunas etapas de implementación y luego hacerlo funcionar.

El diseño y la implementación de programas es una actividad excitante y creativa. Como toda actividad lleva tiempo y práctica volverse eficiente en ella, por lo tanto no se desaliente si sus primeros esfuerzos parecen eternos o si están llenos de errores; esto es normal y gradualmente mejorará con la experiencia. No sea demasiado ambicioso para comenzar y espere tener que revisar sus ideas a medida que camina, esto es parte del proceso natural de aprendizaje.

Y por sobre todo, ¡disfrútelo!

## APÉNDICE

# A

## Trabajar con un proyecto BlueJ

### A.1 Instalar BlueJ

Para trabajar con BlueJ se debe instalar el kit de desarrollo de Java 2 Standard Edition (J2SE SDK) y el entorno BlueJ.

Se puede encontrar el software J2SE SDK y las instrucciones detalladas para su instalación en el CD que acompaña este libro o bien en

<http://java.sun.com/j2se/>

Se puede encontrar el entorno BlueJ y las instrucciones para su instalación en el CD que acompaña este libro o bien en

<http://www.bluej.org/>

### A.2 Abrir un proyecto

Para usar cualquiera de los proyectos de ejemplo incluidos en el CD que acompaña a este libro, se deben copiar previamente a un disco en el que se pueda grabar (por ejemplo, al disco duro). Los proyectos BlueJ se pueden abrir directamente desde el CD pero no se pueden ejecutar desde él. Cuando BlueJ ejecuta un proyecto, necesita grabar información en la carpeta que lo contiene y este es el motivo por el que generalmente, no resulta adecuado utilizar los proyectos directamente desde el CD.

La manera más fácil de usar los proyectos es copiar al disco duro la carpeta que contiene todos los proyectos del libro (de nombre *projects*).

Después de instalar e iniciar BlueJ haciendo doble clic sobre su ícono, se selecciona la opción *Open...* del menú *Project*, se navega hasta la carpeta *projects* y se selecciona un proyecto. Se pueden abrir varios proyectos simultáneamente.

Se incluye más información sobre el uso de BlueJ en el Tutorial de BlueJ<sup>1</sup> que está en el CD del libro, al que también se puede acceder mediante la opción *BlueJ Tutorial* del menú *Help* de BlueJ.

### A.3 El depurador de BlueJ

Se puede encontrar información sobre el uso del depurador de BlueJ en el Apéndice G y en el Tutorial de BlueJ. El tutorial está incluido en el CD del libro y también se puede acceder a él mediante la opción *BlueJ Tutorial* del menú *Help* de BlueJ.

---

<sup>1</sup> N. del T. El Tutorial de BlueJ que se incluye en el CD está en idioma inglés. Si necesita una versión en español, puede encontrarla en el sitio <http://www.bluej.org/doc/tutorial.html>

## A.4 Contenido del CD

En el CD que se incluye en este libro se encuentran los siguientes archivos y directorios:

Carpeta	Comentario
acrobat/	<i>Acrobat Reader para varios sistemas operativos. Acrobat Reader es un programa que muestra e imprime archivos en formato PDF. Se necesita para leer o imprimir el Tutorial de Bluej. (Puede ocurrir que Acrobat Reader ya esté instalado; sólo se debe instalar si no se puede abrir el tutorial.)</i>
mac/	<i>Acrobat Reader para el S.O. Mac X.</i>
linux/	<i>Acrobat Reader para el S.O. Linux.</i>
solaris/	<i>Acrobat Reader para el S.O. Solaris.</i>
windows/	<i>Acrobat Reader para Microsoft Windows (todas las versiones).</i>
bluej/	<i>El sistema BlueJ y su documentación.</i>
bluejsetup-212.exe	<i>Instalador de BlueJ para Microsoft Windows (todas las versiones).</i>
bluej-212.zip	<i>BlueJ para S.O. Mac X.</i>
bluej-212.jar	<i>BlueJ para otros sistemas operativos.</i>
tutorial.pdf	<i>Tutorial de BlueJ.</i>
index.html	<i>Documentación del CD. Para leer este archivo, se debe abrir mediante un navegador. Contiene una visión global del CD, instrucciones de instalación y otras cuestiones útiles.</i>
j2sdk/	<i>Contiene el sistema Java 2 (Java 2 SDK) para varios sistemas operativos.</i>
linux/	<i>Instalador de Java 2 SDK para Linux.</i>
solaris/	<i>Instalador de Java 2 SDK para Solaris.</i>
windows/	<i>Instalador de Java 2 SDK para Microsoft Windows (todas las versiones.).</i>
j2sdk-doc/	<i>Contiene la documentación de la biblioteca de Java 2. Es un archivo de tipo zip. Para usar la documentación, se puede copiar este archivo al disco rígido y descomprimirlo.</i>
projects/	<i>Contiene todos los proyectos que se utilizan en este libro. Antes de usar los proyectos, se debe copiar esta carpeta completa al disco rígido. Contiene una subcarpeta para cada capítulo.</i>
runthis.exe	<i>Programa que utiliza la característica auto-abrir de Microsoft Windows (no es relevante para este libro).</i>
intro/	<i>Archivos de soporte para la documentación del CD. No es necesario usar directamente los archivos de esta carpeta, en su lugar se puede usar el archivo index.html.</i>

## APÉNDICE

# B

## Tipos de dato en Java

Java reconoce dos categorías de tipos: tipos primitivos y tipos objeto. Los tipos primitivos se almacenan directamente en las variables y tienen valores semánticos (se copian los valores cuando se asignan a otra variable). Los tipos objeto se almacenan mediante referencias al objeto (no se almacena el objeto propiamente dicho); cuando se asignan a otra variable sólo se copia la referencia, no el objeto.

### B.1 Tipos primitivos

En la siguiente tabla se listan todos los tipos primitivos del lenguaje Java:

Nombre del tipo	Descripción	Ejemplos de literales		
Números enteros				
byte	entero de 1 byte de tamaño (8 bit)	24	-2	
short	entero corto (16 bit)	137	-119	
int	entero (32 bit)	5409	-2003	
long	entero largo (64 bit)	423266353L	55L	
Números reales				
float	punto flotante de simple precisión	43.889F		
double	punto flotante de doble precisión	45.63	2.4e5	
Otros tipos				
char	un solo carácter (16 bit)	'm'	'?'	'\u00F6'
boolean	un valor lógico (verdadero o falso)	true	false	

Notas:

- Un número que no contiene un punto decimal se interpreta generalmente como un `int`, pero se convierte automáticamente a los tipos `short`, `byte` o `long` cuando se le asigna (si el valor encaja). Se puede declarar un literal como `long` añadiendo una ‘L’ al final del número (también se puede utilizar la letra ‘l’ (L minúscula) pero debería evitarse ya que se puede confundir fácilmente con el uno).
- Un número con un punto decimal se considera de tipo `double`. Se puede especificar un literal como un `float` añadiendo una ‘F’ o ‘f’ al final del número.
- Un carácter se puede escribir como un carácter Unicode encerrándolo entre comillas simples o como un valor Unicode de cuatro dígitos precedidos por ‘\u’.
- Los dos literales booleanos son `true` y `false`.

Debido a que las variables de tipos primitivos no hacen referencia a objetos, no existen métodos asociados con los tipos primitivos. Sin embargo, cuando se usa un tipo primitivo en un contexto que requiere un tipo objeto se puede usar el proceso de *auto-boxing* para convertir un valor primitivo en su correspondiente objeto. Para más detalles, recurra a la Sección B.3.

La siguiente tabla detalla los valores mínimo y máximo disponibles para los tipos numéricos.

Tipo	Mínimo	Máximo
byte	- 128	127
short	-32768	32767
int	-2147483648	2147483647
long	-9223372036854775808	9223372036854775807
	Mínimo positivo	Máximo positivo
float	1.4e-45	3.4028235e38
double	4.9e-324	1.7976931348623157e308

## B.2 Tipos objeto

Todos los tipos que no aparecen en la sección *Tipos primitivos* son tipos objeto. Esto incluye los tipos clase e interface de la biblioteca estándar de Java (como por ejemplo, `String`) y los tipos definidos por el usuario.

Una variable de tipo objeto contiene una referencia (o un «puntero») a un objeto. Las asignaciones y los pasajes de parámetros utilizan referencias semánticas (es decir, se copia la referencia, no el objeto). Después de asignar una variable a otra, ambas variables hacen referencia al mismo objeto. Se dice que las dos variables son alias del mismo objeto.

Las clases son las plantillas de los objetos: definen los campos y los métodos que poseerá cada instancia.

Los arreglos (*arrays*) se comportan como tipos objeto; también utilizan referencias semánticas.

## B.3 Clases «envoltorio»

En Java, cada tipo primitivo tiene su correspondiente clase «envoltorio» que representa el mismo tipo pero que en realidad, es un tipo objeto. Estas clases hacen posible que se usen valores de tipos primitivos en los lugares en que se requieren tipos objeto mediante un proceso conocido como *autoboxing*. La siguiente tabla enumera los tipos primitivos y sus correspondientes clases envoltorio del paquete `java.lang`. Excepto `Integer` y `Character`, los nombres de las clases envoltorio coinciden con los nombres de los tipos primitivos, pero con su primera letra en mayúscula.

Tipo primitivo	Tipo envoltorio
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Siempre que se use un valor de un tipo primitivo en un contexto que requiera un tipo objeto, el compilador utiliza la propiedad de *autoboxing* para encapsular automáticamente al valor de tipo primitivo en un objeto envoltorio equivalente. Esto quiere decir, por ejemplo, que los valores de tipos primitivos se pueden agregar directamente en una colección. La operación inversa (*autounboxing*) también se lleva a cabo automáticamente cuando se utiliza un objeto envoltorio en un contexto que requiere un valor del tipo primitivo correspondiente.



## APÉNDICE

# C

## Estructuras de control en Java

### C.1 Sentencias de selección

#### *If-else*

La sentencia *if-else* tiene dos formas:

```
if (expresión) {           if (expresión) {
    sentencias             sentencias
}                           }
                        else {
    sentencias
}
```

Ejemplos:

```
if (campo.size() == 0) {
    System.out.println("El campo está vacío");
}

if (numero < 0) {
    informarError();
}
else {
    procesarNumero(numero);
}

if (numero < 0) {
    procesarNegativo();
}
else if (numero == 0) {
    procesarCero();
}
else {
    procesarPositivo();
}
```

#### *switch*

La sentencia *switch* selecciona un único valor de un número arbitrario de casos. Existen dos esquemas posibles:

```

switch (expresión) {
    case valor: sentencias;
                break;
    case valor: sentencias;
                break;
    (se omiten los restantes
casos)
    default: sentencias;
                break;
}

```

```

switch (expresión) {
    case valor1:
    case valor2:
    case valor3:
        sentencias;
        break;
    case valor4:
    case valor5:
        sentencias;
        break;
    (se omiten los restantes
casos)
    default:
        sentencias;
        break;
}

```

Notas:

- Una sentencia *switch* puede tener cualquier número de etiquetas *case*.
- La instrucción *break* después de cada *case* es necesaria; en caso contrario la ejecución continúa pasando a través de las sentencias de la etiqueta siguiente. La segunda forma descrita anteriormente usa este esquema. En este caso, los tres primeros valores ejecutarán la primera sección de sentencias mientras que los valores cuatro y cinco ejecutarán la segunda sección de sentencias.
- El caso *default* es opcional. Si no se da ningún valor por defecto puede ocurrir que este caso no se ejecute nunca.
- No es necesaria la instrucción *break* al final del caso por *default* (o del último *case*, si es que no hay sección *default*) pero se considera de buen estilo incluirla.

Ejemplos:

```

switch (dia) {
    case 1: stringDia = "Lunes";
                break;
    case 2: stringDia = "Martes";
                break;
    case 3: stringDia = "Miércoles";
                break;
    case 4: stringDia = "Jueves";
                break;
    case 5: stringDia = "Viernes";
                break;
    case 6: stringDia = "Sábado";
                break;
    case 7: stringDia = "Domingo";
                break;
    default: stringDia = "Día no válido";
                break;
}

switch (mes) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:

```

```

        case 10:
        case 12:
            numeroDeDias = 31;
            break;
        case 4:
        case 6:
        case 9:
        case 11:
            numeroDeDias = 30;
            break;
        case 2:
            if (esAnioBisiesto())
                numeroDeDias = 29;
            else
                numeroDeDias = 28;
            break;
    }
}

```

## C.2 Ciclos

Java tiene tres tipos de ciclos: *while*, *do-while* y *for*.

### *while*

El *ciclo while* ejecuta un bloque de sentencias tantas veces como la evaluación de la expresión resulte verdadera. La expresión se evalúa antes de la ejecución del cuerpo del ciclo, por lo tanto, el cuerpo del ciclo podría ejecutarse cero veces (es decir, no ejecutarse).

```

while (expresión) {
    sentencias
}

```

#### Ejemplos:

```

int i = 0;
while (i < texto.size()) {
    System.out.println(texto.get(i));
    i++;
}

while (iter.hasNext()) {
    procesarObjeto(iter.next());
}

```

### *do-while*

El *ciclo do-while* ejecuta un bloque de sentencias tantas veces como la expresión resulte verdadera. La expresión es evaluada después de la ejecución del cuerpo del ciclo, por lo que el cuerpo de este ciclo se ejecuta siempre por lo menos una vez.

```

do {
    sentencias
} while (expresión);

```

#### Ejemplo:

```

do {
    entrada = leerEntrada();
    if (entrada == null) {
        System.out.println("Pruebe nuevamente");
    }
} while (entrada == null);

```

*for*

El *ciclo for* tiene dos formas diferentes. La primera se conoce también como *ciclo foreach* y se usa exclusivamente para recorrer los elementos de una colección. A la variable del ciclo se le asigna el valor de los sucesivos elementos de la colección en cada iteración del ciclo.

```

for (declaración-de-variable : colección) {
    sentencias
}
Ejemplo:
for (String nota : lista) {
    System.out.println(nota);
}

```

La segunda forma del *ciclo for* ejecuta un bloque de sentencias tantas veces como la condición se evalúe verdadera. Antes de iniciar el ciclo, se ejecuta exactamente una vez, una sentencia de *inicialización*. La condición es evaluada antes de cada ejecución del cuerpo del ciclo (por lo que el cuerpo del ciclo podría no ejecutarse). Se ejecuta una sentencia de *incremento* al finalizar cada ejecución del cuerpo del ciclo.

```

for (inicialización; condición; incremento) {
    sentencias
}

```

Ejemplo:

```

for(int i = 0; i < texto.size(); i++) {
    System.out.println(texto.get(i));
}

```

### C.3 Excepciones

El lanzamiento y la captura de excepciones proporciona otro par de construcciones que alteran el flujo del control.

```

try {
    sentencias
}
catch (tipo-de-excepción nombre) {
    sentencias
}
finally {
    sentencias
}

```

Ejemplo:

```
try {
```

```
    FileWriter writer = new FileWriter("foo.txt");
    writer.write(texto);
    writer.close();
}
catch (IOException e) {
    Debug.reportError("Falló la grabación del texto");
    Debug.reportError("La excepción es: " + e );
}
```

Una sentencia de excepción puede tener cualquier número de cláusulas *catch* que son evaluadas en el orden en que aparecen y se ejecuta sólo la primera cláusula que coincide. (Una cláusula coincide si el tipo dinámico del objeto excepción que ha sido lanzado es compatible en la asignación con el tipo de excepción declarado en la cláusula *catch*.) La cláusula *finally* es opcional.

## C.4 Aserciones

Hay dos formas de sentencias de aserción:

```
assert expresión-booleana;
assert expresión-booleana : expresión;
```

Ejemplos:

```
assert getDatos(clave) != null;

assert esperado = actual :
    " El valor actual: " + actual +
    " no coincide con el valor esperado: " + esperado;
```

Si la expresión de la aserción se evalúa falsa, se disparará un `AssertionError`.



## APÉNDICE

# D

## Operadores

### D.1 Expresiones aritméticas

Java dispone de una cantidad considerable de operadores para expresiones aritméticas y lógicas. La tabla D.1 muestra todo aquello que se clasifica como un operador, incluyendo la conversión de tipos (*casting*) y el pasaje de parámetros. Los principales operadores aritméticos son:

+	<i>suma</i>
-	<i>resta</i>
*	<i>multiplicación</i>
/	<i>división</i>
%	<i>módulo o resto de una división entera</i>

Tanto en la división como en el módulo, los resultados de las operaciones dependen de si sus operandos son enteros o si son valores de punto flotante. Entre dos valores enteros, la división retiene el resultado entero y descarta cualquier resto; pero entre dos valores de punto flotante, el resultado es un valor de punto flotante:

5 / 3 da por resultado 1  
5.0 / 3 da por resultado 1.6666666666666667

(Observe que es necesario que uno sólo de los operandos sea de punto flotante para que se produzca un resultado de punto flotante.)

Cuando en una operación aparecen más operadores, se deben usar las *reglas de precedencia* para indicar el orden de su aplicación. En la Tabla D.1, los operadores se presentan por nivel de precedencia, de mayor a menor (en la primera fila aparecen los operadores de nivel de precedencia más alto). Por ejemplo, podemos ver que la multiplicación, la división y el módulo preceden a la suma y a la resta y esta es la razón por la que los dos ejemplos siguientes dan por resultado 100:

51 \* 3 - 53  
154 - 2 \* 27

Los operadores que tienen el mismo nivel de precedencia se evalúan de izquierda a derecha.

Se pueden usar paréntesis cuando se necesite alterar el orden de evaluación. Es por este motivo que los dos ejemplos siguientes dan por resultado 100:

(205 - 5) / 2  
2 \* (47 + 3)

Observe que algunos operadores aparecen en las dos primeras filas de la Tabla D.1. Los que aparecen en la primera fila admiten un solo operando a su izquierda; los que están en la segunda fila admiten un solo operando a su derecha.

**Tabla D.1**

Operadores Java por nivel de precedencia  
(de mayor a menor)

[ ]	.	++	--	(parámetros)				
++	--	+	-	! "				
new	(cast)							
*	/	%						
+	-							
<<	>>	>>>						
<	>	>=	<=	instanceof				
==	!=							
&								
^								
&&								
?:								
=	+=	- =	* =	/ =	% =	>> =	>>> =	& =
								=
								^ =

## D.2. Expresiones lógicas

En las expresiones lógicas, se usan los operadores para combinar operandos y producir un único valor lógico, ya sea verdadero o falso (*true* o *false*). Las expresiones lógicas generalmente se encuentran en las condiciones de las sentencias *if-else* y en las de los ciclos.

Los operadores relacionales o de comparación combinan generalmente un par de operandos aritméticos, aunque también se utilizan para evaluar la igualdad y la desigualdad de referencias a objetos. Los operadores relacionales de Java son:

== igual	!= distinto
< menor	<= menor o igual
> mayor	>= mayor o igual

Los operadores lógicos binarios combinan dos expresiones lógicas para producir otro valor lógico. Los operadores son:

&& y (and)
o (or)
^ o excluyente

Y además,

! no (not)

que toma una expresión lógica y cambia su valor de verdadero a falso y viceversa.

La manera en que se aplican los operadores `&&` y `||` es un poco extraña. Si el operando izquierdo es falso entonces resulta irrelevante el valor del operando derecho y no será evaluado; de igual manera, si el operando izquierdo es verdadero, no será evaluado el operando derecho. Por este motivo, se conoce a estos operadores como operadores en «cortocircuito».



## APÉNDICE

# E

## Ejecutar Java fuera del entorno BlueJ

A lo largo de este libro hemos usado BlueJ para desarrollar y ejecutar nuestras aplicaciones Java. Hay una buena razón para esto: BlueJ nos ofrece algunas herramientas para que resulten más fáciles algunas de las tareas de desarrollo. En particular, nos permite ejecutar fácilmente métodos individuales de clases y de objetos, lo que resulta muy útil si queremos probar rápidamente un fragmento de código.

Dividimos la discusión sobre cómo trabajar fuera del entorno BlueJ en dos categorías: ejecutar una aplicación y desarrollarla fuera del entorno BlueJ.

### E.1 Ejecutar fuera del entorno BlueJ

Generalmente, cuando se entregan las aplicaciones a los usuarios finales, son ejecutadas de diferentes maneras. Las aplicaciones tienen un solo punto de comienzo que define el lugar en que empieza la ejecución cuando un usuario inicia la aplicación.

El mecanismo exacto que se usa para iniciar una aplicación depende del sistema operativo; generalmente, se hace doble clic sobre el ícono de la aplicación o se ingresa el nombre de la misma en una línea de comando. Luego, el sistema operativo necesita saber qué método o qué clase debe invocar para ejecutar el programa completo.

En Java, este problema se resuelve usando una convención: cuando se inicia un programa Java, el nombre de la clase se especifica como un parámetro del comando de inicio y el nombre del método es siempre el mismo, el nombre de este método es «main». Por ejemplo, considere el siguiente comando ingresado en una línea de comando, como si fuera un comando de Windows o de una terminal Unix:

```
java Juego
```

El comando `java` inicia la máquina virtual de Java, que forma parte del kit de desarrollo de Java (SDK) y que debe estar instalado en su sistema. `Juego` es el nombre de la clase que queremos iniciar.

Luego, el sistema Java buscará un método en la clase `Juego` cuya firma coincida exactamente con la siguiente:

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

El método debe ser público para que pueda ser invocado desde el exterior de la clase. Debe ser estático porque no existe ningún objeto cuando se inicia el programa; inicialmente, tenemos sólo clases, motivo por el cual sólo podemos invocar métodos estáticos. Este método estático crea el primer objeto. El tipo de retorno es `void` ya que este método no retorna ningún valor. Aunque el nombre «`main`» fue seleccionado arbitrariamente por los desarrolladores de Java, es fijo: el método debe tener siempre este

nombre. (La elección de «main» como nombre del método inicial en realidad proviene del lenguaje C, del que Java hereda gran parte de su sintaxis.)

El parámetro es un arreglo de `String`, que permite a los usuarios pasar argumentos adicionales. En nuestro ejemplo, el valor del parámetro `args` será un arreglo de longitud cero. Sin embargo, la línea de comandos que inicia el programa puede definir argumentos:

```
java Juego 2 Fred
```

En esta línea de comando, cada palabra ubicada a continuación del nombre de la clase será leído como un `String` independiente y pasado al método `main` como un elemento del arreglo de `String`. En este caso, el arreglo `args` contendrá dos elementos que son las cadenas «2» y «Fred». Los parámetros en la línea de comandos no son muy usados en Java.

En teoría, el cuerpo del método `main` puede contener el número de sentencias que se deseen. Sin embargo, un buen estilo indica que el método `main` debiera mantenerse lo más corto posible; específicamente, no debiera contener nada que forme parte de la lógica de la aplicación.

En general, el método `main` debe hacer exactamente lo que se hizo interactivamente para iniciar la misma aplicación en BlueJ. Por ejemplo, si para iniciar la aplicación en BlueJ se creó un objeto de la clase `Juego` y se invocó el método de nombre `start`, en el método `main` de la clase `Juego` deberían agregarse las siguientes sentencias:

```
public static void main (String[] args)
{
    Juego juego = new Juego();
    juego.start();
}
```

Ahora, al ejecutar el método `main` se imitará la invocación interactiva del juego.

Los proyectos Java se guardan generalmente en un directorio independiente para cada uno y todas las clases del proyecto se ubican dentro de este directorio. Cuando se ejecute el comando para iniciar Java y ejecutar su aplicación, se debe asegurar de que el directorio del proyecto sea el directorio activo en la terminal de comandos, lo que asegura que se encontrarán las clases que se usan.

Si no puede encontrar una clase específica, la máquina virtual de Java generará un mensaje de error similar a este:

```
Exception in thread "main" java.lang.NoClassDefFoundError: Juego
```

Si ve un mensaje como éste, asegúrese de que escribió correctamente el nombre de la clase y de que el directorio actual realmente contenga esta clase. La clase se guarda en un archivo de extensión «.class»: por ejemplo, el código de la clase `Juego` se almacena en un archivo de nombre `Juego.class`.

Si encuentra la clase pero ésta no contiene un método `main` (o el método `main` no posee la firma correcta) verá un mensaje similar a este:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

En este caso, asegúrese de que la clase que quiere ejecutar tenga el método `main` correcto.

## E.2 Crear archivos ejecutables .jar

Los proyectos Java se almacenan como una colección de archivos en un directorio (o carpeta). A continuación, hablaremos brevemente sobre los diferentes tipos de archivo.

Generalmente, para distribuir aplicaciones a otros usuarios es más fácil si toda la aplicación se guarda en un único archivo; el mecanismo de Java que realiza esto tiene el formato de archivo Java («.jar»). Todos los archivos de una aplicación se pueden reunir en un único archivo y aun así podrán ser ejecutados. (Si está familiarizado con el formato de compresión «zip», sería interesante saber que, de hecho, el formato es el mismo. Los archivos jar pueden abrirse mediante programas zip y viceversa.)

Para crear un archivo .jar ejecutable es necesario especificar la clase principal en algún lugar. (Recuerde: el método que se ejecuta siempre es el `main`, pero necesitamos especificar la clase que lo contiene.) Esta especificación se hace incluyendo un archivo de texto en el archivo .jar (el archivo explícito) con la información necesaria. Afortunadamente, BlueJ se ocupa por su propia cuenta de esta tarea.

Para crear un archivo ejecutable .jar en BlueJ use la función *Project – Export* y especifique la clase que contiene el método `main` en la caja de diálogo que aparece. (Debe escribir un método `main` exactamente igual al descrito anteriormente.)

Para ver detalles sobre esta función, lea el Tutorial de BlueJ al que puede acceder mediante el menú *Help-Tutorial* de BlueJ o bien visitando el sitio web de BlueJ.

Una vez que se creó el archivo ejecutable .jar, se puede ejecutar haciendo doble clic sobre él. La computadora que ejecuta este archivo .jar debe tener instalado el JDK (Java Development Kit) o el JRE (Java Runtime Environment) y asociado con archivos .jar.

## E.3 Desarrollar fuera del entorno BlueJ

Si no quiere solamente ejecutar programas, sino que también quiere desarrollarlos fuera del entorno BlueJ, necesitará editar y compilar las clases. El código de una clase se almacena en un archivo de extensión «.java»; por ejemplo, la clase Juego se almacena en un archivo de nombre `Juego.java`. Los archivos fuente pueden editarse con cualquier editor de textos. Existen muchos editores de textos libres o muy baratos. Algunos, como el *Notepad* o el *WordPad* se distribuyen con Windows, pero si en realidad quiere usar un editor para hacer algo más que una prueba rápida, querrá obtener uno mejor. Sin embargo, sea cuidadoso con los procesadores de texto: generalmente los procesadores de texto no graban en formato de texto plano y Java no podrá leerlos.

Los archivos fuente pueden compilarse desde una línea de comando usando el compilador Java que se incluye en el JDK y que se invoca mediante el comando `javac`. Para compilar un archivo fuente de nombre `Juego.java` use el comando

```
javac Juego.java
```

Este comando compilará la clase `Juego` y cualquier otra clase que dependa de ella; creará un archivo denominado `Juego.class` que contiene el código que puede ser ejecutado mediante la máquina virtual de Java. Para ejecutar este archivo use el comando

```
java Juego
```

Observe que este comando no incluye la extensión del archivo «.class».



## APÉNDICE

# F

## Configurar BlueJ

Se pueden configurar muchos de los valores de BlueJ para que se adapte mejor a su situación personal. Algunas opciones de configuración están disponibles mediante la caja de diálogo *Preferences* del sistema BlueJ, pero es posible acceder a muchas otras opciones editando el «archivo de definiciones de BlueJ» que está ubicado en `<bluej_home>/lib/bluej.defs`, donde `<bluej_home>` es la carpeta donde se encuentra instalado BlueJ.

Los detalles de configuración se explican en la sección «*Tips archive*» del sitio web de BlueJ a la que se puede acceder en la dirección

<http://www.bluej.org/help/archive.html>

A continuación presentamos las cosas más comunes que la gente suele cambiar. Puede encontrar muchas más opciones de configuración leyendo el archivo *bluej.defs*.

### F.1 Cambiar el idioma de la interfaz

Puede cambiar el idioma de la interfaz por cualquiera de los idiomas disponibles. Para hacerlo, abra el archivo *bluej.defs* y busque la línea que dice

`bluej.language=english`

y cámbiela por uno de los idiomas disponibles. Por ejemplo:

`bluej.language=spanish`

Los comentarios en el archivo de definición enumeran todos los lenguajes disponibles. Como mínimo se incluyen los idiomas afrikáans, chino, checo, inglés, francés, alemán, italiano, japonés, coreano, portugués, español y sueco.

### F.2 Usar la documentación API en forma local

Puede usar una copia local de la documentación de la biblioteca de clases de Java (API). De esta manera, el acceso a la documentación es más rápido y puede usar la documentación sin tener que estar conectado a Internet.

Para hacerlo, copie el archivo de documentación de Java desde el CD (un archivo zip) y descomprímalo en el lugar en que quiera guardar la documentación de Java; se creará una carpeta de nombre *docs*.

Luego abra un navegador y usando la función «*Abrir archivo...*» (o cualquier otra equivalente), abra el archivo *api/index.html* situado en la carpeta *docs*.

Una vez que se visualiza correctamente el API en el navegador, copie la URL (dirección web) del campo de dirección de su navegador, abra BlueJ, abra el diálogo *Prefrences*, seleccione la ficha *Miscellaneous* y pegue la URL copiada en el campo etiquetado como *JDK documentation URL*.

Ahora podrá abrir una copia local del API seleccionando la opción *Java Class Libraries* del menú *Help*.

### F.3 Cambiar las plantillas para las clases nuevas

Cuando crea una clase nueva, el código se presenta con un texto predeterminado que proviene de una plantilla. Se puede cambiar este texto de modo que se adapte a sus preferencias.

Las plantillas se almacenan en las carpetas

```
<bluej_home>/lib/<language>/templates/ y en  
<bluej_home>/lib/<language>/templates/newclass/
```

en donde *<bluej\_home>* es la carpeta de instalación de BlueJ y *<language>* es el idioma actualmente en uso (por ejemplo, *english*).

Las plantillas son archivos de texto y se pueden editar mediante cualquier editor de textos estándar.

## APÉNDICE

# G

## Usar el depurador

El depurador de BlueJ proporciona un conjunto de funcionalidades básicas de depuración intencionalmente simplificadas y que son genuinamente útiles tanto para la depuración de programas como para alcanzar mayor comprensión del comportamiento de la ejecución de un programa.

Se puede acceder a la ventana del depurador seleccionando el elemento *Show Debugger* del menú *View* o presionando el botón derecho del *ratón* sobre el indicador de trabajo y seleccionando *Show Debugger* desde el menú contextual. La Figura G.1 muestra la ventana del depurador.

**Figura G.1**

La ventana del depurador de BlueJ



La ventana del depurador tiene cinco zonas de visualización y cinco botones de control. Las zonas de visualización y los botones se activan solamente cuando un programa alcanza un punto de interrupción o se para por alguna otra razón. Las siguientes secciones describen cómo establecer puntos de interrupción para controlar la ejecución de un programa y el propósito de cada una de las zonas.

## G.1 Puntos de interrupción

Un punto de interrupción es un bandera que se asocia con una línea de código (Figura G.2). Cuando se alcanza un punto de interrupción durante la ejecución de un programa, se activan las zonas de visualización y los controles del depurador permitiendo inspeccionar el estado del programa y controlar la ejecución a partir de allí.

**Figura G.2**

Un punto de interrupción asociado con una línea de código

```

22  /**
23   * Imprime el siguiente mensaje (si es que hay alguno) para este
24   * usuario en la terminal de texto.
25   */
26 public void imprimirMensajeSiguiente()
27 {
28      Mensaje unMensaje = servidor.getMensajeSiguiente(usuario);
29     if(unMensaje == null) {
30         System.out.println("No hay ningún mensaje nuevo.");
31     }
32     else {
33         unMensaje.imprimir();
34     }
35 }
```

Los puntos de interrupción se establecen en la ventana del editor, ya sea presionando el botón izquierdo del ratón en la zona de puntos de interrupción situada a la izquierda del código o bien ubicando el cursor en la línea de código en la que debiera estar el punto de interrupción y seleccionando la opción *Set/Clear Breakpoint* del menú *Tools* del editor. Se pueden eliminar los puntos de interrupción mediante el proceso inverso. Sólo se pueden fijar puntos de interrupción en el código de las clases que hayan sido previamente compiladas.

## G.2 Los botones de control

La Figura G.3 muestra los botones de control que se activan ante un punto de interrupción.

**Figura G.3**

Botones de control activos ante un punto de interrupción



### G.2.1 Halt

El botón *Halt* está activo cuando el programa se está ejecutando, para permitir que la ejecución se pueda interrumpir, de ser necesario. Si la ejecución se detiene, el depurador mostrará el estado del programa como si hubiera alcanzado un punto de interrupción.

### G.2.2 Step

El botón *Step* ejecuta la sentencia actual. La ejecución se detendrá nuevamente cuando se complete dicha sentencia. Si la sentencia involucra una llamada a método, se completa la llamada al método antes de que la ejecución se detenga nuevamente (a menos que el método invocado tenga otro punto de interrupción explícito).

### G.2.3 Step Into

El botón *Step Into* ejecuta la sentencia actual. Si esta sentencia es una llamada a un método entonces la ejecución se introducirá en ese método y se detendrá nuevamente en la primer sentencia del mismo.

### G.2.4 Continue

El botón *Continue* continúa la ejecución del programa hasta que se alcance el siguiente punto de interrupción, se interrumpa la ejecución mediante el botón *Halt* o se complete la ejecución normalmente.

### G.2.5 Terminate

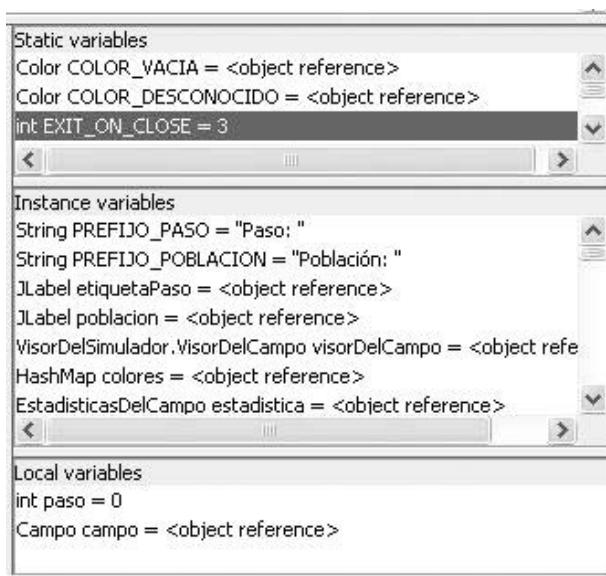
El botón *Terminate* finaliza agresivamente la ejecución del programa actual de manera tal que no puede ser detenida nuevamente. Si se desea simplemente interrumpir la ejecución para examinar el estado actual del programa es preferible utilizar la operación *Halt*.

## G.3 La zona de variables

La Figura G.4 muestra las tres zonas activas en las que se muestran las variables cuando se encuentra un punto de interrupción, en un ejemplo tomado de la simulación *predador-presa* trabajada en el Capítulo 10. Las variables estáticas se muestran en la zona superior, las variables de instancia en la del medio y las variables locales en la zona inferior.

**Figura G.4**

Zonas de variables activas



Cuando se alcanza un punto de interrupción, la ejecución se detendrá en una sentencia de un objeto arbitrario dentro del programa actual. La zona de variables estáticas (*Static variables*) muestra los valores de las variables estáticas definidas en la clase de dicho objeto. La zona de variables de instancia (*Instance variables*) muestra las variables de instancia de dicho objeto en particular. Ambas zonas también incluyen las variables heredadas de las superclases.

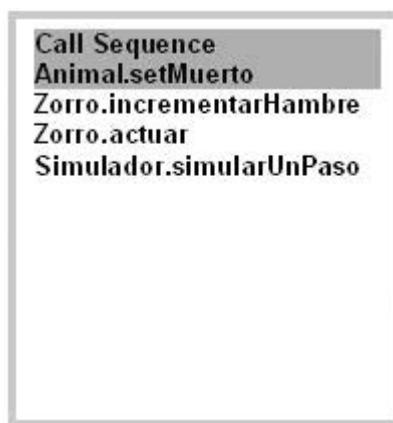
La zona de *variables locales* (*Local variables*) muestra los valores de las variables locales y de los parámetros del método o del constructor que se está ejecutando actualmente. Las variables locales aparecerán en esta zona sólo una vez que hayan sido inicializadas ya que solamente comienzan a existir en la máquina virtual de Java a partir de ese momento.

#### G.4 La zona de Secuencia de llamadas

La Figura G.5 muestra la zona *Call Sequence* que contiene una secuencia de cuatro métodos de profundidad. Los métodos aparecen en la secuencia en el formato Clase.método, independientemente de si son métodos estáticos o métodos de instancia. Los constructores aparecen en la secuencia como Clase.<init>.

**Figura G.5**

Una secuencia de llamadas



La secuencia de llamadas opera como una pila: el método que aparece en la parte superior de la secuencia es donde reside actualmente el flujo de la ejecución. Las zonas que muestran a las variables reflejan los detalles del método o del constructor que esté resaltado en ese momento en la secuencia de llamadas. Al seleccionar una línea diferente de la secuencia de llamadas se actualizarán los contenidos de las otras zonas.

#### G.5 La zona de Threads

Esta zona está fuera del alcance de este libro y no será tratada.

# Herramienta JUnit de pruebas unitarias

En este apéndice hacemos una breve revisión de las principales características que soporta BlueJ relacionadas con el estilo de pruebas unitarias creadas mediante JUnit. Se pueden encontrar más detalles sobre el tema en el tutorial que está disponible en el CD que acompaña este libro y en el sitio web de BlueJ.

## H.1 Habilitar la funcionalidad de pruebas unitarias

Para habilitar la funcionalidad de pruebas unitarias de BlueJ es necesario asegurarse de que esté marcada la opción *Show unit testing tools* en el menú *Tools-Preferences-Miscellaneous*. Una vez marcada, la ventana principal de BlueJ contendrá algunos botones adicionales que se activan cuando se abre un proyecto.

## H.2 Crear una clase de prueba

Se crea una clase de prueba haciendo clic con el botón derecho del ratón sobre una clase en el diagrama de clases y seleccionando la opción *Create Test Class*. El nombre de una clase de prueba se determina automáticamente agregando la palabra «Test» a modo de sufijo al nombre de la clase asociada. Alternativamente, puede crearse una clase de prueba seleccionando el botón *New Class...* y eligiendo *Unit Test* como el tipo de la clase. En este caso, la elección del nombre es totalmente libre.

Las clases de prueba se denotan con <>unit test></> en el diagrama de clases y tienen un color diferente del color de las clases ordinarias.

## H.3 Crear un método de prueba

Los métodos de prueba se pueden crear interactivamente. Se puede grabar la secuencia de interacciones del usuario con el diagrama de clases y con el banco de objetos y luego capturarla como una secuencia de sentencias y de declaraciones Java en un método de la clase de prueba. Se comienza la grabación seleccionando la opción *Create Test Method* del menú contextual asociado con una clase de prueba. BlueJ solicitará el nombre del nuevo método. Si el nombre no comienza con la palabra *test* entonces se agregarán como prefijo del nombre del método. El símbolo de grabación a la izquierda del diagrama de clase se pondrá de color rojo y se vuelven disponibles los botones *End* y *Cancel*.

Una vez que comenzó la grabación, cualquier creación de objeto o llamadas a métodos formarán parte del código del método que se está creando. Seleccione *End* para com-

pletar la grabación y capturar la prueba o *Cancel* para descartar la grabación y dejar sin cambios a la clase de prueba.

#### H.4 Pruebas con aserciones

Mientras se graba un método de prueba, cualquier llamada a método que retorne un resultado abrirá una ventana del tipo *Method Result* que ofrece la oportunidad de evaluar el valor del resultado marcando la opción *Assert that*. El menú desplegable que aparece contiene un conjunto de aserciones posibles para el valor del resultado. Si se estableció una aserción, será codificada como una llamada a método en el método de prueba que dará un *AssertionError* en el caso en que la prueba falle.

#### H.5 Ejecutar pruebas

Los métodos se pueden ejecutar individualmente seleccionándolos del menú contextual asociado a la clase de prueba. Si una prueba resulta exitosa, se indicará mediante un mensaje en la línea de estado de la ventana principal. Si una prueba fracasa aparecerá la ventana *Test Results*. Al seleccionar *Test All* del menú contextual de la clase de prueba se ejecutarán todas las pruebas de una sola clase. En la ventana *Test Results* se detallará el éxito o el fracaso de cada método.

#### H.6 Conjunto de Objetos de prueba (fixtures)

Se puede capturar el contenido del banco de objetos como un «juego de prueba» (*fixture*), seleccionando la opción *Object Bench to Test Fixture* del menú contextual asociado con la clase de prueba. El efecto de crear un juego de prueba es que se agrega una definición de campo para cada objeto en la clase de prueba y se agregan las sentencias en su respectivo método *setUp* que recreará el estado exacto de los objetos tal como estaban en el banco de objetos. Luego, los objetos son eliminados del banco.

El método *setUp* se ejecuta automáticamente antes de ejecutar cualquier método de prueba por lo que todos los objetos del juego de prueba estarán disponibles para todas las pruebas.

Los objetos del juego de prueba pueden ser creados nuevamente en el banco de objetos seleccionando la opción *Test Fixture to Object Bench* desde el menú de la clase de prueba.

# El documentador de Java: javadoc

La escritura de buena documentación de las definiciones de las clases y de las interfaces es un complemento importante para obtener código de buena calidad. La documentación le permite al programador comunicar sus intenciones a los lectores humanos en un lenguaje natural de alto nivel, en lugar de forzarlos a leer código de nivel relativamente bajo. La documentación de los elementos públicos de una clase o de una interfaz tienen un valor especial, pues los programadores pueden usarla sin tener que conocer los detalles de su implementación.

En todos los proyectos de ejemplo de este libro hemos usado un estilo particular de comentarios que es reconocido por la herramienta de documentación `javadoc` que se distribuye como parte del kit de desarrollo (SDK) de Java de Sun Microsystem. Esta herramienta automatiza la generación de documentación de clases en formato HTML con un estilo consistente. El API de Java ha sido documentado usando esta misma herramienta y se aprecia su valor cuando se usa la biblioteca de clases.

En este apéndice hacemos un breve resumen de los principales elementos de los comentarios de documentación que deberá introducir habitualmente en su propio código fuente.

### I.1 Comentarios de documentación

Los elementos de una clase que se documentarán son la definición de la clase, sus campos, constructores y métodos. Desde el punto de vista de un usuario, lo más importante de una clase es que tenga documentación sobre ella y sobre sus constructores y métodos públicos. Tendemos a no proporcionar comentarios del estilo de `javadoc` para los campos aunque recordamos que forman parte del detalle del nivel de implementación y no es algo que verán los usuarios.

Los comentarios de documentación comienzan siempre con los tres caracteres «`/**`» y terminan con el par de caracteres «`*/`». Entre estos símbolos, un comentario contendrá una **descripción principal** seguida por una sección de **etiqueta**, aunque ambas partes son opcionales.

#### I.1.1 *La descripción principal*

La descripción principal de una clase debiera consistir en una descripción del objetivo general de la clase. El Código I.1 muestra parte de una típica descripción principal, tomada de la clase `Juego` del proyecto *world-of-zuul*. Observe que la descripción incluye detalles sobre cómo usar esta clase para iniciar el juego.

**Código I.1**

La descripción principal de un comentario de clase

```
 /**
 * Esta es la clase principal de la aplicación "World of
Zuul"
 * "World of Zuul" es un juego de aventuras muy sencillo,
basado en texto.
 * Los usuarios pueden caminar por algún escenario, y eso
es todo lo
 * que hace el juego. ¡Podría ampliarse para que resulte
más interesante!
 * Para jugar, cree una instancia de esta clase e invoque
el método "jugar"
 */
```

La descripción principal de un método debiera ser bastante general, sin introducir demasiados detalles sobre su implementación. En realidad, la descripción principal de un método generalmente consiste en una sola oración, como por ejemplo

```
 /**
 * Crea un nuevo pasajero con distintas ubicaciones de
salida y de destino.
 */
```

Las ideas esenciales debieran presentarse en la primera sentencia de la descripción principal de una clase, de una interfaz o de un método ya que es lo que se usa a modo de resumen independiente en la parte superior de la documentación generada.

Javadoc también soporta el uso de etiquetas HTML en sus comentarios.

### I.1.2 La sección de etiquetas

A continuación de la descripción principal aparece la sección de etiquetas. Javadoc reconoce alrededor de 20 etiquetas pero sólo trataremos las más importantes (Tabla I.1). Las etiquetas pueden usarse de dos maneras: en bloques de etiquetas o como etiquetas de una sola línea. Sólo hablaremos de los bloques de etiquetas pues son los que se usan con mayor frecuencia. Para ver más detalles sobre las etiquetas de una sola línea y sobre las restantes etiquetas, puede recurrir a la sección *javadoc* de la documentación *Tools and Utilities* que forma parte del Java SDK.

**Tabla I.1**

Etiquetas más comunes de javadoc

Etiqueta	Texto asociado
@author	nombre(s) del autor(es)
@param	nombre de parámetro y descripción
@return	descripción del valor de retorno
@see	referencia cruzada
@throws	tipo de excepción que se lanza y las circunstancias en las que se hace
@version	descripción de la versión

Las etiquetas `@author` y `@version` se encuentran regularmente en los comentarios de una clase y de una interfaz y no pueden usarse en los comentarios de métodos, constructores o campos. Ambas etiquetas pueden estar seguidas de cualquier texto y no se requiere ningún formato especial para ninguna de ellas. Ejemplos:

```
@author Hakcer T. LargeBrain  
@version 2004.12.31
```

Las etiquetas `@param` y `@throws` se usan en métodos y en constructores, mientras que `@return` se usa sólo en métodos. Algunos ejemplos:

```
@param limite El valor máximo permitido.  
@return Un número aleatorio en el rango 1 a limite (inclusive)  
@throws IllegalLimitException Si el límite es menor que 1.
```

La etiqueta `@see` adopta varias formas diferentes y puede usarse en cualquier comentario de documentación. Proporciona un camino de referencia cruzada hacia un comentario de otra clase, método o cualquier otra forma de documentación. Se agrega una sección `See Also` al elemento que está siendo comentado. Algunos ejemplos típicos:

```
@see "The Java Language Specification, by Joy et al"  
@see <a href="http://www.bluej.org/>The BlueJ web site </a>  
@see #estaVivo  
@see java.util.ArrayList#add
```

La primera simplemente encierra un texto en forma de cadena sin un hipervínculo, la segunda es un hipervínculo hacia el documento especificado, la tercera es un vínculo a la documentación del método `estaVivo` de la misma clase, la cuarta vincula la documentación del método `add` con la clase `java.util.ArrayList`.

## I.2 Soporte de BlueJ para javadoc

Si un proyecto ha sido comentado usando el estilo de javadoc, BlueJ ofrece utilidades para generar la documentación HTML completa. En la ventana principal, seleccione el elemento *Tools/Project Documentation* del menú y se generará la documentación (si es necesario) y se mostrará en la ventana de un navegador.

Dentro del editor de BlueJ, se puede pasar de la vista del código fuente de una clase a la vista de su documentación cambiando la opción *Implementation* por la opción *Interface* en la parte superior derecha de la ventana (Figura I.1). Esta opción ofrece una vista previa y rápida de la documentación pero no contendrá referencias a la documentación de las superclases o de las clases que se usan.

**Figura I.1**

La opción de vistas  
*Implementation* e  
*Interface*



Puede encontrar más detalles sobre el documentador de java en la dirección

<http://java.sun.com/j2se/javadoc/writingdoccomments/index.html>



## APÉNDICE

# J

## Guía de estilo de programación

### J.1 Nombres

#### J.1.1 *Use nombres significativos*

Use nombres descriptivos para todos los identificadores (nombres de clases, de variables, de métodos). Evite ambigüedades. Evite abreviaturas. Los métodos de modificación debieran comenzar con el prefijo «*set*»: *setAlgo(...)*. Los métodos de acceso debieran comenzar con el prefijo «*get*»: *getAlgo(...)*. Los métodos de acceso con valores de retorno booleanos generalmente comienzan con el prefijo «*es*»: *esAlgo(...)*; por ejemplo, *esVacio()*.

#### J.1.2 *Los nombres de las clases comienzan con una letra mayúscula*

#### J.1.3 *Los nombres de las clases son sustantivos en singular*

#### J.1.4 *Los nombres de los métodos y de las variables comienzan con letras minúsculas*

Tanto los nombres de las clases, como los de los métodos y los de las variables, emplean letras mayúsculas entre medio para aumentar la legibilidad de los identificadores que lo componen; por ejemplo: *numeroDeElementos*.

#### J.1.5 *Las constantes se escriben en MAYÚSCULAS*

Ocasionalmente se utiliza el símbolo de subrayado en el nombre de una constante para diferenciar los identificadores que lo componen: *TAMANIO\_MAXIMO*.

### J.2 Esquema

#### J.2.1 *Un nivel de indentación es de cuatro espacios*

#### J.2.2 *Todas las sentencias de un bloque se indentan un nivel*

#### J.2.3 *Las llaves de las clases y de los métodos se ubican solas en una línea*

Las llaves que encierran el bloque de código de la clase y las de los bloques de código de los métodos se escriben en una sola línea y con el mismo nivel de indentación. Por ejemplo:

```
public int getEdad()
{
    sentencias
}
```

*J.2.4 Para los restantes bloques de código, las llaves se abren al final de una línea*

En todos los bloques de código restantes, la llave se abre al final de la línea que contiene la palabra clave que define al bloque. La llave se cierra en una línea independiente, alineada con la palabra clave que define dicho bloque. Por ejemplo:

```
while(condición) {
    sentencias
}

if(condición) {
    sentencias
}
else {
    sentencias
}
```

*J.2.5 Use siempre llaves en las estructuras de control*

Se usan llaves en las sentencias *if* y en los ciclos aun cuando el cuerpo esté compuesto por una única sentencia.

*J.2.6 Use un espacio antes de la llave de apertura de un bloque de una estructura de control*

*J.2.7 Use un espacio antes y después de un operador*

*J.2.8 Use una línea en blanco entre los métodos (y los constructores)*

Use líneas en blanco para separar bloques lógicos de código; es decir, use líneas en blanco por lo menos entre métodos, pero también entre las partes lógicas dentro de un mismo método.

### J.3 Documentación

*J.3.1 Cada clase tiene un comentario de clase en su parte superior*

El comentario de clase contiene como mínimo

- una descripción general de la clase

- el nombre del autor (o autores)
- un número de versión

Cada persona que ha contribuido en la clase debe ser nombrada como un autor o debe ser acreditada apropiadamente de otra manera.

Un número de versión puede ser simplemente un número o algún otro formato. Lo más importante es que el lector pueda reconocer si dos versiones no son iguales y determinar cuál es la más reciente.

#### *J.3.2 Cada método tiene un comentario*

#### *J.3.3 Los comentarios son legibles para javadoc*

Los comentarios de la clase y de los métodos deben ser reconocidos por javadoc; en otras palabras: deben comenzar con el símbolo de comentario «`/**`».

#### *J.3.4 Comente el código sólo donde sea necesario*

Se deben incluir comentarios en el código en los lugares en que no resulte obvio o sea difícil de comprender (y preferentemente, el código debe ser obvio o fácil de entender, siempre que sea posible) y donde ayude a la comprensión de un método. No comente sentencias obvias, ¡asuma que el lector comprende Java!

### **J.4 Restricciones de uso del lenguaje**

#### *J.4.1 Orden de las declaraciones: campos, constructores, métodos*

Los elementos de una definición de clase aparecen (si se presentan) en el siguiente orden: sentencias de paquete, sentencias de importación, comentario de clase, encabezado de la clase, definición de campos, constructores, métodos.

#### *J.4.2 Los campos no deben ser públicos (con excepción de los campos final)*

#### *J.4.3 Use siempre modificadores de acceso*

Especifique todos los campos y los métodos como privados, públicos o protegidos. Nunca use el acceso por defecto (*package private*).

#### *J.4.4 Importe las clases individualmente*

Es preferible que las sentencias de importación nombren explícitamente cada clase que se quiere importar y no al paquete completo. Por ejemplo:

```
import java.util.ArrayList;
import java.util.HashSet;
```

es mejor que

```
import java.util.*;
```

J.4.5 *Incluya siempre un constructor (aun cuando su cuerpo quede vacío)*

J.4.6 *Incluya siempre una llamada al constructor de una superclase*

En los constructores de las subclases no deje que se realice la inserción automática de una llamada a una superclase; incluya explícitamente la invocación `super(...)`, aun cuando funcione bien sin hacerlo.

J.4.7 *Inicialice todos los campos en el constructor*

## J.5 Modismos del código

J.5.1 *Use iteradores en las colecciones*

Para iterar o recorrer una colección, use un *ciclo for-each*. Cuando la colección debe ser modificada durante una iteración, use un `Iterator` en lugar de un índice entero.

# Clases importantes de la biblioteca de Java

La plataforma Java 2 incluye un rico conjunto de bibliotecas que sustentan una amplia variedad de tareas de programación.

En este apéndice resumiremos brevemente los detalles de algunas de las clases e interfaces de los paquetes más importantes del API de la plataforma Java 2. Un programador Java competente debe estar familiarizado con la mayoría de ellas. Este apéndice es sólo un resumen y debe leerse conjuntamente con toda la documentación del API de Java.

## K.1 El paquete `java.lang`

Las clases y las interfaces que contiene el paquete `java.lang` son fundamentales para el lenguaje Java; es por este motivo que este paquete se importa automática e implícitamente en cualquier definición de clase.

paquete <code>java.lang</code>	Síntesis de las clases más importantes
clase <code>Math</code>	<code>Math</code> es una clase que contiene sólo campos y métodos estáticos. En esta clase se definen los valores de las constantes matemáticas <code>e</code> y <code>π</code> , las funciones trigonométricas y otras funciones como <code>abs</code> , <code>min</code> , <code>max</code> y <code>sqrt</code> (raíz cuadrada).
clase <code>Object</code>	<code>Object</code> es la superclase de todas las clases, está en la raíz de todas las jerarquías de clases. Todos los objetos heredan de ella la implementación por defecto de métodos importantes como <code>equals</code> y <code>toString</code> . Otros métodos significativos definidos en esta clase son <code>clone</code> y <code>hashCode</code> .
clase <code>String</code>	Las cadenas constituyen una característica importante de muchas aplicaciones y reciben un tratamiento especial en Java. Los métodos más importantes de esta clase son <code>charAt</code> , <code>equals</code> , <code>indexOf</code> , <code>length</code> , <code>split</code> y <code>substring</code> . Las cadenas definidas a partir de esta clase son objetos inmutables, por lo tanto, métodos tales como <code>trim</code> , que parecieran ser métodos de modificación, en realidad devuelven un nuevo objeto <code>String</code> que representa el resultado de la operación.
clase <code>StringBuffer</code>	La clase <code>StringBuffer</code> aporta una alternativa eficiente a la clase <code>String</code> , en los casos en que se requiere construir una cadena a partir de un conjunto de componentes, como ocurre por ejemplo, en la concatenación. Sus métodos más importantes son <code>append</code> , <code>insert</code> y <code>toString</code> .

## K.2 El paquete `java.util`

El paquete `java.util` es una colección relativamente incoherente de clases e interfaces útiles.

paquete <code>java.util</code>	Síntesis de las clases más importantes
interfaz <code>Collection</code>	Esta interfaz proporciona el conjunto central de los métodos de la mayoría de las clases basadas en colecciones, que se definen en el paquete <code>java.util</code> tales como <code>ArrayList</code> , <code>HashSet</code> y <code>LinkedList</code> . Define la signatura de los métodos <code>add</code> , <code>clear</code> , <code>iterator</code> , <code>remove</code> y <code>size</code> .
interfaz <code>Iterator</code>	<code>Iterator</code> define una interfaz sencilla y consistente para recorrer el contenido de una colección. Sus tres métodos son <code>hasNext</code> , <code>next</code> y <code>remove</code> .
interfaz <code>List</code>	<code>List</code> es una extensión de la interfaz <code>Collection</code> y proporciona medios para tratar la colección como una secuencia; por este motivo muchos de sus métodos tienen un índice como parámetro, como por ejemplo: <code>add</code> , <code>get</code> , <code>remove</code> y <code>set</code> . Clases tales como <code>ArrayList</code> y <code>LinkedList</code> implementan la interfaz <code>List</code> .
interfaz <code>Map</code>	La interfaz <code>Map</code> ofrece una alternativa a las colecciones basadas en listas mediante la idea de asociar cada objeto de una colección con un valor clave. Los objetos se agregan y se acceden mediante sus métodos <code>put</code> y <code>get</code> . Observe que un <code>Map</code> no retorna un objeto <code>Iterator</code> sino que su método <code>keySet</code> devuelve un objeto <code>Set</code> de claves y su método <code>values</code> retorna un objeto <code>Collection</code> con los objetos del mapa.
interfaz <code>Set</code>	La interfaz <code>Set</code> es una extensión de la interfaz <code>Collection</code> que tiene la intención de asignar una colección que no contenga elementos duplicados. Dado que es una interfaz, merece la pena mencionar que <code>Set</code> no está implicada realmente en reforzar esta restricción. Esto quiere decir que <code>Set</code> es en realidad una interfaz indicativa, que permite a los implementadores de colecciones indicar que sus clases cumplen con esta particular restricción.
clase <code>ArrayList</code>	Es una implementación de la interfaz <code>List</code> que usa un arreglo con el fin de proporcionar acceso directo eficiente, mediante índices, a los objetos almacenados. Si se agregan o se eliminan objetos en cualquier lugar, excepto de la posición final de la lista, se deben desplazar los siguientes elementos para hacer espacio o para tapar los agujeros que quedan. Los métodos más importantes son <code>add</code> , <code>get</code> , <code>iterator</code> , <code>remove</code> y <code>size</code> .
clase <code>Collections</code>	Esta clase reúne los métodos estáticos que se usan para manipular las colecciones. Los métodos más importantes son <code>binarySearch</code> , <code>fill</code> y <code>sort</code> .
clase <code>HashMap</code>	<code>HashMap</code> es una implementación de la interfaz <code>Map</code> . Los métodos más importantes son <code>get</code> , <code>put</code> , <code>remove</code> y <code>size</code> . Para recorrer un <code>HashMap</code> generalmente se realiza un proceso en dos etapas: primero se obtiene el conjunto de claves mediante su método <code>keySet</code> y luego se recorre este conjunto de claves.
clase <code>HashSet</code>	<code>HashSet</code> es una implementación de la interfaz <code>Set</code> basada en la técnica de <i>hashing</i> . Su uso es más parecido al de una <code>Collection</code> que al de un <code>HashMap</code> . Sus métodos más importantes son <code>add</code> , <code>remove</code> y <code>size</code> .
clase <code>LinkedList</code>	<code>LinkedList</code> es una implementación de la interfaz <code>List</code> cuya estructura interna para almacenar objetos responde a una lista simplemente enlazada. El acceso directo a los extremos de la lista es eficiente, pero no es tan eficiente el acceso a objetos individuales mediante un índice como el de un <code>ArrayList</code> . Por otro lado, al agregar objetos o al eliminarlos de la lista no es necesario hacer ningún cambio en los objetos existentes. Los métodos más importantes son <code>add</code> , <code>getFirst</code> , <code>getLast</code> , <code>iterator</code> , <code>removeFirst</code> , <code>removeLast</code> y <code>size</code> .

paquete <code>java.util</code>	Síntesis de las clases más importantes
clase Random	La clase <code>Random</code> colabora en la generación de valores pseudo aleatorios, los típicos números aleatorios. La secuencia de números generada está determinada por un valor semilla que puede pasarse al constructor o asignarse mediante una llamada al método <code>setSeed</code> . Dos objetos <code>Random</code> que comienzan con la misma semilla devolverán la misma secuencia de valores ante llamadas idénticas. Los métodos más importantes son <code>nextBoolean</code> , <code>nextDouble</code> , <code>nextInt</code> y <code>setSeed</code> .
clase Scanner	La clase <code>Scanner</code> proporciona un medio para leer y analizar entradas. Se utiliza generalmente para leer entradas desde el teclado. Los métodos más importantes son <code>next</code> y <code>hasNext</code> .

### K.3 El paquete `java.io`

El paquete `java.io` contiene clases que permiten las operaciones de entrada y de salida (*input/output*). Muchas de las clases se diferencian porque se basan en flujos (*stream*) (es decir, operan sobre datos binarios), o porque operan con caracteres (*readers* y *writers*).

paquete <code>java.io</code>	Síntesis de las clases más importantes
interfaz <code>Serializable</code>	La interfaz <code>Serializable</code> es una interfaz vacía que no requiere que se escriba ningún código en la implementación de una clase. Las clases implementan esta interfaz con el fin de participar del proceso de serialización. Los objetos <code>Serializable</code> deben escribirse y leerse como un todo, desde y hacia fuentes de entrada/salida. Esto hace que el almacenamiento y la recuperación de datos persistentes sea un proceso relativamente simple en Java. Vea las clases <code>ObjectInputStream</code> y <code>ObjectOutputStream</code> para más información.
clase <code>BufferedReader</code>	Es una clase que proporciona acceso a un buffer de caracteres desde una fuente de entrada. El ingreso mediante un buffer generalmente es más eficiente que sin él, especialmente si la fuente de entrada es un archivo externo al sistema. Dado que el ingreso se hace mediante un buffer, ofrece un método <code>readLine</code> que no está disponible en la mayoría de las otras clases para procesar entradas. Los métodos más importantes son <code>close</code> , <code>read</code> y <code>readLine</code> .
clase <code>BufferedWriter</code>	Es una clase que proporciona salida de caracteres mediante un buffer. La salida mediante un buffer es más eficiente que sin él especialmente si el destino de la salida es un archivo externo al sistema. Los métodos más importantes son <code>close</code> , <code>flush</code> y <code>write</code> .
clase <code>File</code>	La clase <code>File</code> proporciona una representación en objeto de archivos y carpetas (directorios) de un sistema de archivos externo. Existen métodos para indicar si un archivo es de lectura y/o de escritura y si es un archivo o una carpeta. Se puede crear un objeto <code>File</code> mediante un archivo inexistente que será el primer paso en la creación de un archivo físico en el sistema de archivos. Los métodos más importantes son: <code>canRead</code> , <code>canWrite</code> , <code>createNewFile</code> , <code>createTempFile</code> , <code>getName</code> , <code>getParent</code> , <code>getPath</code> , <code>isDirectory</code> , <code>isFile</code> y <code>listFiles</code> .
clase <code>FileReader</code>	La clase <code>FileReader</code> se usa para abrir un archivo externo preparado para que su contenido se pueda leer mediante caracteres. Un objeto <code>FileReader</code> se pasa generalmente al constructor de otra clase lectora (tal como <code>BufferedReader</code> ) en lugar de usarlo directamente. Los métodos más importantes son <code>close</code> y <code>read</code> .

paquete java.io	Síntesis de las clases más importantes
clase <code>FileWriter</code>	La clase <code>FileWriter</code> se usa para abrir un archivo externo preparado para grabar datos mediante caracteres. Un par de constructores determinan si se agregara a un archivo existente o si el contenido existente se descarta. Un objeto <code>FileWriter</code> generalmente se pasa al constructor de otra clase escritora (tal como <code>BufferedWriter</code> ) en lugar de usarlo directamente. Los métodos más importantes son: <code>close</code> , <code>flush</code> y <code>write</code> .
clase <code>IOException</code>	Es una clase de excepción comprobada que es la raíz de la jerarquía de la mayoría de las excepciones de <i>input/output</i> .

## K.4 El paquete `java.net`

El paquete `java.net` contiene clases e interfaces que soportan aplicaciones para trabajar en red. La mayoría de ellas están fuera del alcance de este libro.

paquete java.net	Síntesis de las clases más importantes
clase <code>URL</code>	La clase <code>URL</code> representa un <i>Uniform Resource Locator</i> , en otras palabras, proporciona una manera de describir la ubicación de algo en Internet. De hecho, también se puede usar para describir la ubicación de algo en un sistema de archivos local. La hemos incluido porque las clases de <code>java.io</code> y de <code>javax.swing</code> usan con frecuencia objetos <code>URL</code> . Los métodos más importantes son: <code>getContent</code> , <code>getFile</code> , <code>getHost</code> , <code>getPath</code> y <code>openStream</code> .

## K.5 Otros paquetes importantes

Otros paquetes importantes son

```
java.awt
java.awt.event
javax.swing
javax.swing.event
```

Estas clases se usan extensamente cuando se escriben interfaces gráficas de usuario (IGU) y contienen muchas clases útiles con las que los programadores de IGU deberán familiarizarse.

## APÉNDICE

# L

## Tabla de conversión de términos que aparecen en el CD

Español	Inglés	Qué es	Proyecto	Capítulo
asignarAula	setRoom	método	lab-classes	1
asignarHorario	setTime	método	lab-classes	1
aula	room	campo	lab-classes	1
cadenaDiaHora	timeAndDayString	parámetro	lab-classes	1
cambiarNombre	changeName	método	lab-classes	1
capacidad	capacity	campo	lab-classes	1
creditos	credits	campo	lab-classes	1
CursoDeLaboratorio	LabClass	clase	lab-classes	1
diaYHora	timeAndDay	campo	lab-classes	1
Estudiante	Student	clase	lab-classes	1
estudiantes	students	campo	lab-classes	1
getCreditos	getCredits	método	lab-classes	1
getIdEstudiante	getStudentID	método	lab-classes	1
getNombreDeUsuario	getLoginName	método	lab-classes	1
iDEstudiante	studentID	parámetro	lab-classes	1
imprimir	print	método	lab-classes	1
imprimirLista	printList	método	lab-classes	1
inscribirEstudiante	enrollStudent	método	lab-classes	1
nombreCompleto	fullName	parámetro	lab-classes	1
nombreInstructor	instructorName	parámetro	lab-classes	1
nuevoEstudiante	newStudent	parámetro	lab-classes	1
nuevoNombre	replacementName	parámetro	lab-classes	1
numeroDeAula	roomNumber	parámetro	lab-classes	1
numeroDeEstudiantes	numberOfStudents	método	lab-classes	1
numeroMaximoDeEstudiantes	maxNumberOfStudents	parámetro	lab-classes	1
obtenerNombre	getName	método	lab-classes	1
puntosAdicionales	additionalPoints	parámetro	lab-classes	1
sumarCreditos	addCredits	método	lab-classes	1
cuadro	Picture	clase	picture	1
pared	wall	campo	picture	1
ponerBlancoYNegro	setBlackAndWhite	método	picture	1
ponerColor	setColor	método	picture	1
sol	sun	campo	picture	1
techo	roof	campo	picture	1
ventana	window	campo	picture	1

Español	Inglés	Qué es	Proyecto	Capítulo
alto	height	campo	shapes	1
ancho	width	campo	shapes	1
borrar	erase	método	shapes	1
cambiarColor	changeColor	método	shapes	1
cambiarTamanio	changeSize	método	shapes	1
Circulo	Circle	clase	shapes	1
Cuadrado	Square	clase	shapes	1
diametro	diameter	campo	shapes	1
dibujar	draw	método	shapes	1
distancia	distance	parámetro	shapes	1
esVisible	isVisible	campo	shapes	1
figuras	shapes	proyecto	shapes	1
lado	size	campo	shapes	1
moveAbajo	moveDown	método	shapes	1
moveArriba	moveUp	método	shapes	1
moveDerecha	moveRight	método	shapes	1
moveHorizontal	moveHorizontal	método	shapes	1
moveIzquierda	moveLeft	método	shapes	1
moveLentoHorizontal	slowMoveHorizontal	método	shapes	1
moveLentoVertical	slowMoveVertical	método	shapes	1
nuevoAlto	newHeight	parámetro	shapes	1
nuevoAncho	newWidth	parámetro	shapes	1
nuevoColor	newColor	parámetro	shapes	1
nuevoDiametro	newDiameter	parámetro	shapes	1
nuevoLado	newSize	parámetro	shapes	1
posicionX	xPosition	campo	shapes	1
posicionY	yPosition	campo	shapes	1
Triangulo	Triangle	clase	shapes	1
volverInvisible	makeInvisible	método	shapes	1
volverVisible	makeVisible	método	shapes	1
cantidadAReintegrar	amountToRefund	variable local	better-ticket-machine	2
maquina-de-boletos-mejorada	better-ticket-machine	proyecto	better-ticket-machine	2
reintegrarSaldo	refundBalance	método	better-ticket-machine	2
autor	author	campo	book-exercise	2
autorDelLibro	bookAuthor	parámetro	book-exercise	2
ejercicio-libro	book-exercise	proyecto	book-exercise	2
Libro	Book	clase	book-exercise	2
titulo	title	campo	book-exercise	2
tituloDelLibro	bookTitle	parámetro	book-exercise	2
cantidad	amount	parámetro	naive-ticket-machine	2
imprimirBoleto	printTicket	método	naive-ticket-machine	2
ingresarDinero	insertMoney	método	naive-ticket-machine	2
MaquinaDeBoletos	TicketMachine	clase	naive-ticket-machine	2
maquina-de-boletos-simple	naive-ticket-machine	proyecto	naive-ticket-machine	2
obtenerPrecio	getPrice	método	naive-ticket-machine	2

Español	Inglés	Qué es	Proyecto	Capítulo
obtenerSaldo	getBalance	método	naive-ticket-machine	2
obtenerTotal	getTotal	método	naive-ticket-machine	2
precio	price	campo	naive-ticket-machine	2
precioDelBoleto	ticketCost	parámetro	naive-ticket-machine	2
saldo	balance	campo	naive-ticket-machine	2
actualizarVisor	updateDisplay	método	clock-display	3
cadVisor	displayString	campo	clock-display	3
getHora	getTime	método	clock-display	3
getValor	getValue	método	clock-display	3
getValorDelVisor	getDisplayValue	método	clock-display	3
horas	hours	campo	clock-display	3
incrementar	increment	método	clock-display	3
límite	limit	campo	clock-display	3
límiteMaximo	rollOverLimit	parámetro	clock-display	3
minutos	minutes	campo	clock-display	3
nuevoValor	replacementValue	parámetro	clock-display	3
ponerEnHora	setTime	método	clock-display	3
setValor	setValue	método	clock-display	3
ticTac	timeTick	método	clock-display	3
valor	value	campo	clock-display	3
VisorDeNumeros	NumberDisplay	clase	clock-display	3
VisorDeReloj	ClockDisplay	clase	clock-display	3
visor-de-reloj	clock-display	proyecto	clock-display	3
agregarEnLista	post	método	mail-system	3
cantidad	count	variable local	mail-system	3
cantidadDeMensajes	howManyMailItems	método	mail-system	3
ClienteDeCorreo	MailClient	clase	mail-system	3
de	from	campo	mail-system	3
enviarMensaje	sendMailItem	método	mail-system	3
getDe	getFrom	método	mail-system	3
getMensajeSiguiente	getNextMailItem	método	mail-system	3
getMensajeSiguiente	getNextMailItem	método	mail-system	3
getPara	getTo	método	mail-system	3
getTexto	getMessage	método	mail-system	3
imprimirMensajeSiguiente	printNextMailItem	método	mail-system	3
mensaje	item	variable local	mail-system	3
Mensaje	MailItem	clase	mail-system	3
mensajes	items	campo	mail-system	3
para	to	campo	mail-system	3
quien	who	parámetro	mail-system	3
servidor	server	campo	mail-system	3
ServidorDeCorreo	MailServer	clase	mail-system	3
sistema-de-correo	mail-system	proyecto	mail-system	3
texto	message	campo	mail-system	3
usuario	user	campo	mail-system	3

Español	Inglés	Qué es	Proyecto	Capítulo
descripcion	description	campo	auction	4
detalles	details	variable local	auction	4
éxito	successful	variable local	auction	4
getDescripcion	getDescription	método	auction	4
getOfertaMaxima	getHighestBid	método	auction	4
getOfertante	getBidder	método	auction	4
ingresarLote	enterLot	método	auction	4
Lote	Lot	clase	auction	4
loteElegido	selectedLot	variable local	auction	4
lotes	lots	campo	auction	4
mostrarLotes	showLots	método	auction	4
numero	number	campo	auction	4
numeroDeLoteSiguiente	nextLotNumber	campo	auction	4
Oferta	Bid	clase	auction	4
ofertaMaxima	highestBid	campo	auction	4
ofertante	bidder	campo	auction	4
ofertaPara	bidFor	método	auction	4
Persona	Person	clase	auction	4
Subasta	Auction	clase	auction	4
subastas	auction	proyecto	auction	4
asociar	join	método	club	4
numeroDeSocios	numberOfMembers	método	club	4
Socio	Membership	clase	club	4
agenda1	notebook1	proyecto	notebook1	4
guardarNota	storeNote	método	notebook1	4
mostrarNota	showNote	método	notebook1	4
notas	notes	campo	notebook1	4
numeroDeNota	noteNumber	parámetro	notebook1	4
numeroDeNotas	numberOfNotes	campo	notebook1	4
agenda2	notebook2	proyecto	notebook2	4
eliminarNota	removeNote	método	notebook2	4
listarTodasLasNotas	listNotes	método	notebook2	4
administrador	manager	campo	products	4
AdministradorDeStock	StockManager	clase	products	4
agregarProducto	addProduct	método	products	4
aumentarCantidad	increaseQuantity	método	products	4
buscarProducto	findProduct	método	products	4
cantidad	quantity	campo	products	4
cantidadEnStock	numberInStock	método	products	4
getAdministrador	getManager	método	products	4
getProducto	getProduct	método	products	4
mostrarDetalles	showDetails	método	products	4
mostrarDetallesDeProductos	printProductDetails	método	products	4
Producto	Product	clase	products	4
productos	products	proyecto	products	4

Español	Inglés	Qué es	Proyecto	Capítulo
recibirProducto	delivery	método	products	4
venderProducto	sellProduct	método	products	4
venderUno	sellOne	método	products	4
AnalizadorLog	LogAnalyzer	clase	weblog-analyzer	4
analizador-weblog	weblog-analyzer	proyecto	weblog-analyzer	4
analizarPorHora	analyzeHourlyData	método	weblog-analyzer	4
ANIO	YEAR	constante	weblog-analyzer	4
archivoLog	logfile	variable local	weblog-analyzer	4
archivoURL	fileURL	variable local	weblog-analyzer	4
contadoresPorHora	hourCounts	campo	weblog-analyzer	4
crearDatosSimulados	createSimulateData	método	weblog-analyzer	4
DIA	DAY	constante	weblog-analyzer	4
entrada	entry	variable local	weblog-analyzer	4
EntradaLog	LogEntry	clase	weblog-analyzer	4
formato	format	campo	weblog-analyzer	4
getFormato	getFormat	método	weblog-analyzer	4
getHora	getHour	método	weblog-analyzer	4
getMinuto	getMinute	método	weblog-analyzer	4
hayMasDatos	hasMoreEntries	método	weblog-analyzer	4
hora	hour	variable local	weblog-analyzer	4
HORA	HOUR	constante	weblog-analyzer	4
imprimirContadoresPorHora	printHourlyCounts	método	weblog-analyzer	4
imprimirDatos	printData	método	weblog-analyzer	4
LectorDeArchivoLog	LogFileReader	clase	weblog-analyzer	4
LectorDeArchivoLog	LogFileReader	clase	weblog-analyzer	4
leeDatos	dataRead	campo	weblog-analyzer	4
lineaDeDatos	dataLine	parámetro	weblog-analyzer	4
lineaLog	lineLog	variable local	weblog-analyzer	4
lineaLog	logLine	parámetro	weblog-analyzer	4
MES	MONTH	constante	weblog-analyzer	4
minimo	lowest	variable local	weblog-analyzer	4
MINUTO	MINUTE	constante	weblog-analyzer	4
nombreDeArchivo	filename	parámetro	weblog-analyzer	4
otraEntrada	otherEntry	parámetro	weblog-analyzer	4
separador	tokenizer	variable local	weblog-analyzer	4
SeparadorDeLineaLog	LogLineTokenizer	clase	weblog-analyzer	4
separar	tokenize	método	weblog-analyzer	4
siguienteEntrada	nextEntry	variable local	weblog-analyzer	4
siguienteEntrada	nextEntry	método	weblog-analyzer	4
valores	dataValues	variable local	weblog-analyzer	4
valoresPorLinea	itemsPerLine	variable local	weblog-analyzer	4
borrarCirculo	eraseCircle	método	balls	5
borrarContorno	eraseOutline	método	balls	5
borrarRectangulo	eraseRectangle	método	balls	5
borrarTexto	eraseString	método	balls	5

Español	Inglés	Qué es	Proyecto	Capítulo
colorDeFondo	backgroundColor	campo	balls	5
colorPelota	ballColor	parámetro	balls	5
cuadroCanvas	drawingCancas	parámetro	balls	5
demoDibujar	drawDemo	método	balls	5
desaceleracionPelota	ballDegradatio	campo	balls	5
diametroPelota	ballDiameter	parámetro	balls	5
dibujarImagen	drawImage	método	balls	5
dibujarLinea	drawLine	método	balls	5
dibujarTexto	drawString	método	balls	5
espera	wait	método	balls	5
fdColor	bgColor	parámetro	balls	5
figura	shape	parámetro	balls	5
getColorDeFondo	getBackgroundColor	método	balls	5
getColorDeLapiz	getForegroundColor	método	balls	5
getTipoDeLetra	getFont	método	balls	5
grafico	graphic	campo	balls	5
GRAVEDAD	GRAVITY	constante	balls	5
imagenDelCanvas	canvasImage	campo	balls	5
imagenVieja	oldImage	variable local	balls	5
miCanvas	myCanvas	campo	balls	5
milisegundos	milliseconds	parámetro	balls	5
mover	move	método	balls	5
nuevoTipoDeLetra	newFont	parámetro	balls	5
pelotas	balls	proyecto	balls	5
PelotasDemo	BallDemo	clase	balls	5
piso	ground	variable local	balls	5
posicionDelPiso	groundPosition	campo	balls	5
posPiso	groundPos	parámetro	balls	5
rebotar	bounce	método	balls	5
ReboteDePelota	BouncingBall	clase	balls	5
rellenar	fill	método	balls	5
rellenarCirculo	fillCircle	método	balls	5
rellenarRectangulo	fillRectangle	método	balls	5
setColorDeFondo	setBackgroundColor	método	balls	5
setColorDeLapiz	setForegroundColor	método	balls	5
setTamanio	setSize	método	balls	5
setTipoDeLetra	setFont	método	balls	5
tamanio	size	variable local	balls	5
velocidadY	ySpeed	campo	balls	5
Contestador	Responder	clase	tech-support1	5
entrada	input	variable local	tech-support1	5
generarRespuesta	generateResponse	método	tech-support1	5
getEntrada	getInput	método	tech-support1	5
imprimirBienvenida	printWelcome	método	tech-support1	5
imprimirDespedida	printGoodBye	método	tech-support1	5

Español	Inglés	Qué es	Proyecto	Capítulo
iniciar	start	método	tech-support1	5
lector	reader	campo	tech-support1	5
LectorDeEntrada	InputReader	clase	tech-support1	5
linea	inputLine	variable local	tech-support1	5
respuesta	response	variable local	tech-support1	5
SistemaDeSoporte	SupportSystem	clase	tech-support1	5
soporte-tecnico1	tech-support1	proyecto	tech-support1	5
terminado	finished	variable local	tech-support1	5
generadorDeAzar	randomGenerator	campo	tech-support2	5
indice	index	variable local	tech-support2	5
rellenarRespuestas	fillResponses	método	tech-support2	5
respuestas	responses	campo	tech-support2	5
soporte-tecnico2	tech-support2	proyecto	tech-support2	5
arregloDePalabras	wordArray	variable local	tech-support-complete	5
contestador	responder	campo	tech-support-complete	5
imprimirDespedida	printGoodBye	método	tech-support-complete	5
lector	reader	campo	tech-support-complete	5
mapaDeRespuestas	responseMap	campo	tech-support-complete	5
rellenarMapaDeRespuestas	fillResponseMap	método	tech-support-complete	5
rellenarRespuestasPorDefecto	fillDefaultResponses	método	tech-support-complete	5
respuestasPorDefecto	defaultResponses	método	tech-support-complete	5
soporte-tecnico-completo	tech-support-complete	proyecto	tech-support-complete	5
tomarRespuestaPorDefecto	pickDefaultResponse	método	tech-support-complete	5
ALTO_BASE	BASE_HEIGHT	constante	bricks	6
cara1	side1	variable local	bricks	6
getPeso	getWeight	método	bricks	6
getSuperficieTotal	getSurfaceArea	método	bricks	6
getVolumen	getVolume	método	bricks	6
Ladrillo	Brick	clase	bricks	6
ladrillos	bricks	proyecto	bricks	6
ladrillosEnPlano	bricksInPlane	campo	bricks	6
numeroDeLadrillos	numberOfBricks	variable local	bricks	6
Pallete	Pallet	clase	bricks	6
PESO_BASE	BASE_WEIGHT	constante	bricks	6
PESO POR_CM3	WEIGHT_PER_CM3	constante	bricks	6
unLadrillo	aBrick	campo	bricks	6
aplicarOperadorPrevio	applyPreviousOperator	método	calculator-engine	6
calculadora-motor	calculator-engine	proyecto	calculator-engine	6
getAutor	getAuthor	método	calculator-engine	6
getTitulo	getTitle	método	calculator-engine	6
getValorEnVisor	getDisplayValue	método	calculator-engine	6
igual	equals	método	calculator-engine	6
limpiar	clear	método	calculator-engine	6
mas	plus	método	calculator-engine	6
menos	minus	método	calculator-engine	6

Español	Inglés	Qué es	Proyecto	Capítulo
motor	engine	campo	calculator-engine	6
MotorDeCalculadora	CalcEngine	clase	calculator-engine	6
MotorDeCalculadoraProbador	CalcEngineTester	clase	calculator-engine	6
numeroPresionado	numberPressed	método	calculator-engine	6
operadorPrevio	previousOperator	campo	calculator-engine	6
operandoIzquierdo	leftOperand	campo	calculator-engine	6
testMas	testPlus	método	calculator-engine	6
testMenos	testMinus	método	calculator-engine	6
valorEnVisor	displayValue	campo	calculator-engine	6
calculadora-motor-impresión	calculator-engine-print	proyecto	calculator-engine-print	6
dónde	where	parámetro	calculator-engine-print	6
informarEstado	reportState	método	calculator-engine-print	6
aplicarOperador	applyOperator	método	calculator-full-solution	6
calculadora-solucion-completa	calculator-full-solution	proyecto	calculator-full-solution	6
calcularResultado	calculateResult	método	calculator-full-solution	6
construyeValorEnVisor	buildingDisplayValue	campo	calculator-full-solution	6
errorEnSecuenciaDeTeclas	keySequenceError	método	calculator-full-solution	6
tieneOperandoIzquierdo	haveLeftOperand	campo	calculator-full-solution	6
ultimoOperador	lastOperator	campo	calculator-full-solution	6
actualizarVisor	redisplay	método	calculator-gui	6
agregarBotón	addButton	método	calculator-gui	6
armarFrame	makeFrame	método	calculator-gui	6
Calculadora	Calculator	clase	calculator-gui	6
calculadora-igu	calculator-gui	proyecto	calculator-gui	6
estado	status	campo	calculator-gui	6
InterfazDeUsuario	UserInterface	clase	calculator-gui	6
mostrarInformación	showInfo	método	calculator-gui	6
muestraAutor	showingAuthor	campo	calculator-gui	6
textoDelBotón	buttonText	parámetro	calculator-gui	6
visor	display	campo	calculator-gui	6
agenda-diaria-prototipo	diary-prototype	proyecto	diary-prototype	6
anotarCita	makeAppointment	método	diary-prototype	6
buscarEspacio	findSpace	método	diary-prototype	6
cantidad_filas_requeridas	further_slots_required	variable local	diary-prototype	6
Cita	Appointment	clase	diary-prototype	6
citas	appointments	campo	diary-prototype	6
Día	Day	clase	diary-prototype	6
diaEnAño	dayInYear	variable local	diary-prototype	6
diaEnSemana	dayInWeek	variable local	diary-prototype	6
diaNúmero	dayNumber	campo	diary-prototype	6
días	days	campo	diary-prototype	6
DIAS_AGENDABLES_ POR_SEMANA	BOOKABLE_DAYS_ PER_WEEK	constante	diary-prototype	6
duración	duration	campo	diary-prototype	6
fila	slot	variable local	diary-prototype	6

Español	Inglés	Qué es	Proyecto	Capítulo
filaSiguiente	nextSlot	variable local	diary-prototype	6
getCita	getAppointment	método	diary-prototype	6
getDia	getDay	método	diary-prototype	6
getDiaNumero	getDayNumber	método	diary-prototype	6
getDuracion	getDuration	método	diary-prototype	6
getSemanaNumero	getWeekNumber	método	diary-prototype	6
horaInicio	startTime	variable local	diary-prototype	6
horaValida	validTime	variable local	diary-prototype	6
MAX_CITAS_POR_DIA	MAX_APPOINTMENTS_PER_DAY	constante	diary-prototype	6
mostrarCitas	showAppointments	método	diary-prototype	6
PRIMER_HORA	START_OF_DAY	constante	diary-prototype	6
Semana	Week	clase	diary-prototype	6
semanaNumero	weekNumber	campo	diary-prototype	6
ULTIMA_HORA	FINAL_APPOINTMENT_TIME	constante	diary-prototype	6
agenda-diaria-prueba	diary-testing	proyecto	diary-testing	6
anotarTresCitas	makeThreeAppointments	método	diary-testing	6
citaMala	badAppointment	variable local	diary-testing	6
completarElDia	fillTheDay	método	diary-testing	6
primera	first	variable local	diary-testing	6
probarDobleCita	testDoubleBooking	método	diary-testing	6
PruebaUnaHora	OneHourTests	clase	diary-testing	6
segunda	second	variable local	diary-testing	6
tercera	third	variable local	diary-testing	6
agenda-diaria-prueba-junit-v1	diary-testing-junit-v1	proyecto	diary-testing-junit-v1	6
DiaTest	DayTest	clase	diary-testing-junit-v1	6
testAnotarTresCitas	testMakeThreeAppointments	método	diary-testing-junit-v1	6
testDobleCita	testDoubleBooking	método	diary-testing-junit-v1	6
Analizador	Parser	clase	zuul-bad	7
bar	pub	variable local	zuul-bad	7
Comando	Command	clase	zuul-bad	7
comandos	commands	campo	zuul-bad	7
comandosValidos	validCommands	método	zuul-bad	7
crearHabitaciones	createRooms	método	zuul-bad	7
esComando	isCommand	método	zuul-bad	7
esDesconocido	isUnknown	método	zuul-bad	7
establecerSalidas	setExits	método	zuul-bad	7
este	east	parámetro	zuul-bad	7
exterior	outside	variable local	zuul-bad	7
getComando	getCommand	método	zuul-bad	7
getPalabraComando	getCommandWord	método	zuul-bad	7
getSegundaPalabra	getSecondWord	método	zuul-bad	7
Habitacion	Room	clase	zuul-bad	7
habitacionActual	currentRoom	campo	zuul-bad	7

Español	Inglés	Qué es	Proyecto	Capítulo
imprimirAyuda	printHelp	método	zuul-bad	7
imprimirBienvenida	printWelcome	método	zuul-bad	7
irAHabitacion	goRoom	método	zuul-bad	7
Juego	Game	clase	zuul-bad	7
jugar	play	método	zuul-bad	7
laboratorio	lab	variable local	zuul-bad	7
lector	reader	campo	zuul-bad	7
lineaIngresada	inputLine	variable local	zuul-bad	7
norte	north	parámetro	zuul-bad	7
oeste	west	parámetro	zuul-bad	7
oficina	office	variable local	zuul-bad	7
palabraComando	commandWord	campo	zuul-bad	7
PalabrasComando	CommandWords	clase	zuul-bad	7
primerPalabra	firstWord	parámetro	zuul-bad	7
procesarComando	processCommand	método	zuul-bad	7
quiereSalir	wantToQuit	variable local	zuul-bad	7
salidaEste	eastExit	campo	zuul-bad	7
salidaNorte	northExit	campo	zuul-bad	7
salidaOeste	westExit	campo	zuul-bad	7
salidaSur	southExit	campo	zuul-bad	7
salir	quit	método	zuul-bad	7
segundaPalabra	secondWord	campo	zuul-bad	7
separador	tokenizer	variable local	zuul-bad	7
siguienteHabitacion	nextRoom	variable local	zuul-bad	7
sur	south	parámetro	zuul-bad	7
teatro	theatre	variable local	zuul-bad	7
tieneSegundaPalabra	hasSecondWord	método	zuul-bad	7
unaCadena	aString	parámetro	zuul-bad	7
zuul-malo	zull-bad	proyecto	zuul-bad	7
establecerSalida	setExit	método	zuul-better	7
getDescripcionCorta	getShortDescription	método	zuul-better	7
getDescripcionLarga	getLongDescription	método	zuul-better	7
getSalida	getExit	método	zuul-better	7
getStringDeSalidas	getExitString	método	zuul-better	7
llaves	keys	variable local	zuul-better	7
mostrarComandos	showCommands	método	zuul-better	7
mostrarTodos	showAll	método	zuul-better	7
salidas	exits	campo	zuul-better	7
stringADevolver	stringResult	variable local	zuul-better	7
vecina	neighbor	parámetro	zuul-better	7
zuul-mejorado	zuul-better	proyecto	zuul-better	7
AYUDA	HELP	valor	zuul-with-enums-v1	7
DESCONOCIDA	UNKNOWN	valor	zuul-with-enums-v1	7
getPalabraComando	getCommandWord	método	zuul-with-enums-v1	7
IR	GO	valor	zuul-with-enums-v1	7

Español	Inglés	Qué es	Proyecto	Capítulo
PalabraComando	CommandWord	clase	zuul-with-enums-v1	7
SALIR	QUIT	valor	zuul-with-enums-v1	7
zuul-con-enumeraciones-v1	zuul-with-enums-v1	proyecto	zuul-with-enums-v1	7
cadenaComando	commandString	campo	zuul-with-enums-v2	7
zuul-con-enumeraciones-v2	zuul-with-enums-v2	proyecto	zuul-with-enums-v2	7
agregarCD	addCD	método	dome-v1	8
agregarDVD	addDVD	método	dome-v1	8
BaseDeDatos	DataBase	clase	dome-v1	8
comentario	comment	campo	dome-v1	8
elArtista	theArtist	parámetro	dome-v1	8
elCD	theCD	parámetro	dome-v1	8
elDVD	theDVD	parámetro	dome-v1	8
elTitulo	theTitle	parámetro	dome-v1	8
getLoTengo	getOwn	método	dome-v1	8
interprete	artist	campo	dome-v1	8
listar	list	método	dome-v1	8
loTengo	gotIt	campo	dome-v1	8
mePertenece	ownIt	parámetro	dome-v1	8
numeroDeTemas	numberOfTracks	campo	dome-v1	8
setLoTengo	setOwn	método	dome-v1	8
temas	time	parámetro	dome-v1	8
tiempo	tracks	parámetro	dome-v1	8
tiempoDuracion	playingTime	campo	dome-v1	8
agregarElemento	addItem	método	dome-v2	8
elElemento	theItem	parámetro	dome-v2	8
Elemento	Item	clase	dome-v2	8
elementoSalir	items	campo	dome-v2	8
adyacente	adjacent	variable local	foxes-and-rabbits-v1	10
altoGrilla	gridHeight	variable local	foxes-and-rabbits-v1	10
ANCHO POR DEFECTO	DEFAULT_WIDTH	constante	foxes-and-rabbits-v1	10
anchoGrilla	gridWidth	variable local	foxes-and-rabbits-v1	10
buscarComida	findFood	método	foxes-and-rabbits-v1	10
Campo	Field	clase	foxes-and-rabbits-v1	10
campoActual	currentField	parámetro	foxes-and-rabbits-v1	10
campoActualizado	updatedField	parámetro	foxes-and-rabbits-v1	10
campoImage	fieldImage	variable local	foxes-and-rabbits-v1	10
cazar	hunt	método	foxes-and-rabbits-v1	10
claseDeAnimal	animalClass	parámetro	foxes-and-rabbits-v1	10
COLOR DESCONOCIDO	UNKNOWN_COLOR	constante	foxes-and-rabbits-v1	10
COLOR_VACIA	EMPTY_COLOR	constante	foxes-and-rabbits-v1	10
colores	colors	campo	foxes-and-rabbits-v1	10
colSiguiiente	nextCol	variable local	foxes-and-rabbits-v1	10
columnas	coffset	variable local	foxes-and-rabbits-v1	10
Conejo	Rabbit	clase	foxes-and-rabbits-v1	10
conejosNuevos	newRabbits	parámetro	foxes-and-rabbits-v1	10

Español	Inglés	Qué es	Proyecto	Capítulo
Contador	Counter	clase	foxes-and-rabbits-v1	10
contadores	counters	campo	foxes-and-rabbits-v1	10
correr	run	método	foxes-and-rabbits-v1	10
cuentasValidas	countsValid	campo	foxes-and-rabbits-v1	10
dibujarMarca	drawMark	método	foxes-and-rabbits-v1	10
direccionAdyacenteLibre	freeAdjacentLocation	método	foxes-and-rabbits-v1	10
direccionAdyacentePorAzar	randomAdjacentLocatio	método	foxes-and-rabbits-v1	10
direccionesAdyacentes	adjacentLocations	variable local	foxes-and-rabbits-v1	10
edad	age	campo	foxes-and-rabbits-v1	10
EDAD_DE_REPRODUCCION	BREEDING_AGE	constante	foxes-and-rabbits-v1	10
EDAD_MAX	MAX_AGE	constante	foxes-and-rabbits-v1	10
edadPorAzar	randomAge	parámetro	foxes-and-rabbits-v1	10
ejecutarSimulacionLarga	runLongSimulation	método	foxes-and-rabbits-v1	10
estadisticas	stats	campo	foxes-and-rabbits-v1	10
EstadisticasDelCampo	FieldStats	clase	foxes-and-rabbits-v1	10
estaVivo	isAlive	método	foxes-and-rabbits-v1	10
esViable	isViable	método	foxes-and-rabbits-v1	10
etiquetaPaso	stepLabel	campo	foxes-and-rabbits-v1	10
FACTOR_DE_ESCALA_ DEL_VISOR_DE_GRILLA	GRID_VIEW_SCALING_ FACTOR	constante	foxes-and-rabbits-v1	10
filas	roffset	variable local	foxes-and-rabbits-v1	10
filaSiguiiente	nextRow	variable local	foxes-and-rabbits-v1	10
generarCuentas	generateCounts	método	foxes-and-rabbits-v1	10
getCantidad	getCount	método	foxes-and-rabbits-v1	10
getDetallesDePoblacion	getPopulationDetails	método	foxes-and-rabbits-v1	10
getFila	getRow	método	foxes-and-rabbits-v1	10
incrementarContador	incrementCount	método	foxes-and-rabbits-v1	10
incrementarEdad	incrementAge	método	foxes-and-rabbits-v1	10
incrementarHambre	incrementHunger	método	foxes-and-rabbits-v1	10
LARGO_POR_DEFECTO	DEFAULT_DEPTH	constante	foxes-and-rabbits-v1	10
limpiar	clear	método	foxes-and-rabbits-v1	10
llave	key	variable local	foxes-and-rabbits-v1	10
lugar	where	variable local	foxes-and-rabbits-v1	10
MAX_TAMANIO_DE_ CAMADA	MAX_LITTER_SIZE	constante	foxes-and-rabbits-v1	10
mostrarEstado	showStatus	método	foxes-and-rabbits-v1	10
nacimientos	births	variable local	foxes-and-rabbits-v1	10
nivelDeComida	foodLevel	campo	foxes-and-rabbits-v1	10
nuevaUbicacion	newLocation	variable local	foxes-and-rabbits-v1	10
nuevoConejo	newRabbit	variable local	foxes-and-rabbits-v1	10
nuevoZorro	newFox	variable local	foxes-and-rabbits-v1	10
numeroDePasos	numSteps	parámetro	foxes-and-rabbits-v1	10
otra	other	variable local	foxes-and-rabbits-v1	10
poblacion	population	campo	foxes-and-rabbits-v1	10
poblar	populate	método	foxes-and-rabbits-v1	10

Español	Inglés	Qué es	Proyecto	Capítulo
PREFIJO_PASO	STEP_PREFIX	constante	foxes-and-rabbits-v1	10
PREFIJO_POBLACION	POPULATION_PREFIX	constante	foxes-and-rabbits-v1	10
PROBABILIDAD_DE_CREACION_DEL_CONEJO	RABBIT_CREATION_PROBABILITY	constante	foxes-and-rabbits-v1	10
PROBABILIDAD_DE_CREACION_DEL_ZORRO	FOX_CREATION_PROBABILITY	constante	foxes-and-rabbits-v1	10
PROBABILIDAD_DE_REPRODUCCION	PROBABILITY_BREEDING	constante	foxes-and-rabbits-v1	10
reinicializar	reset	método	foxes-and-rabbits-v1	10
reproducir	breed	método	foxes-and-rabbits-v1	10
sePuedeReproducir	canBreed	método	foxes-and-rabbits-v1	10
setComido	setEaten	método	foxes-and-rabbits-v1	10
setUbicacion	setLocation	método	foxes-and-rabbits-v1	10
Simulador	Simulator	clase	foxes-and-rabbits-v1	10
simular	simulate	método	foxes-and-rabbits-v1	10
simularUnPaso	simulateOneStep	método	foxes-and-rabbits-v1	10
terminoCuenta	countFinished	método	foxes-and-rabbits-v1	10
ubi	loc	variable local	foxes-and-rabbits-v1	10
Ubicacion	Location	clase	foxes-and-rabbits-v1	10
ubicación	Location	campo	foxes-and-rabbits-v1	10
ubicar	place	método	foxes-and-rabbits-v1	10
VALOR_COMIDA_CONEJO	RABBIT_FOOD_VALUE	constante	foxes-and-rabbits-v1	10
visor	view	campo	foxes-and-rabbits-v1	10
visorDelCampo	fieldView	campo	foxes-and-rabbits-v1	10
VisorDelCampo	fieldView	clase	foxes-and-rabbits-v1	10
VisorDelSimulador	SimulatorView	clase	foxes-and-rabbits-v1	10
vive	alive	campo	foxes-and-rabbits-v1	10
Zorro	Fox	clase	foxes-and-rabbits-v1	10
zorrosNuevos	newFoxes	parámetro	foxes-and-rabbits-v1	10
zorros-y-conejos-v1	foxes-and-rabbits-v1	proyecto	foxes-and-rabbits-v1	10
actuar	act	método	foxes-and-rabbits-v2	10
animalesNuevos	newAnimals	parámetro	foxes-and-rabbits-v2	10
getAnimalEn	getAnimalAt	método	foxes-and-rabbits-v2	10
setMuerto	setDead	método	foxes-and-rabbits-v2	10
zorros-y-conejos-v2	foxes-and-rabbits-v2	proyecto	foxes-and-rabbits-v2	10
construirVentana	makeFrame	método	imageviewer0-1	11
etiqueta	label	variable local	imageviewer0-1	11
panelContenedor	contentPane	variable local	imageviewer0-1	11
ventana	frame	campo	imageviewer0-1	11
VisorDeImagen	ImageViewer	clase	imageviewer0-1	11
visor-de-imagen-0-1	imageviewer0-1	proyecto	imageviewer0-1	11
barraDeMenu	menubar	variable local	imageviewer0-2	11
construirBarraDeMenu	makeMenuBar	método	imageviewer0-2	11
elementoAbrir	openItem	variable local	imageviewer0-2	11
elementoSalir	quitItem	variable local	imageviewer0-2	11

Español	Inglés	Qué es	Proyecto	Capítulo
evento	event	parámetro	imageviewer0-2	11
menuArchivo	fileMenu	variable local	imageviewer0-2	11
visor-de-imagen-0-2	imageviewer0-2	proyecto	imageviewer0-2	11
visor-de-imagen-0-3	imageviewer0-3	proyecto	imageviewer0-3	11
abrirArchivo	openFile	método	imageviewer0-4	11
AdministradorDeArchivos	ImageFileManager	clase	imageviewer0-4	11
alto	height	variable local	imageviewer0-4	11
ancho	width	variable local	imageviewer0-4	11
archivo	file	parámetro	imageviewer0-4	11
archivoDeImagen	imageFile	parámetro	imageviewer0-4	11
archivoSeleccionado	selectedFile	variable local	imageviewer0-4	11
cargarImagen	loadImage	método	imageviewer0-4	11
FORMATO_DE_IMAGEN	IMAGE_FORMAT	constante	imageviewer0-4	11
getImagen	getImage	método	imageviewer0-4	11
grabarImagen	saveImage	método	imageviewer0-4	11
graficoDeImagen	imageGraphics	variable local	imageviewer0-4	11
imagen	image	parámetro	imageviewer0-4	11
ImagenOF	OFImage	clase	imageviewer0-4	11
limpiarImagen	clearImage	método	imageviewer0-4	11
PanelDeImagen	ImagePanel	clase	imageviewer0-4	11
salir	quit	método	imageviewer0-4	11
selectorDeArchivos	fileChooser	campo	imageviewer0-4	11
setImagen	setImage	método	imageviewer0-4	11
tamanio	size	variable local	imageviewer0-4	11
valorDeRetorno	returnVal	variable local	imageviewer0-4	11
aplicarClaro	makeLighter	método	imageviewer1-0	11
aplicarOscuro	makeDarker	método	imageviewer1-0	11
aplicarUmbral	threshold	método	imageviewer1-0	11
brillo	brightness	variable local	imageviewer1-0	11
cerrar	close	método	imageviewer1-0	11
claro	lighter	método	imageviewer1-0	11
elemento	item	variable local	imageviewer1-0	11
etiquetaNombreDeArchivo	filenameLabel	campo	imageviewer1-0	11
mostrarAcercaDe	showAbout	método	imageviewer1-0	11
mostrarEstado	showStatus	método	imageviewer1-0	11
mostrarNombreDeArchivo	showFilename	método	imageviewer1-0	11
oscuro	darker	método	imageviewer1-0	11
texto	text	variable local	imageviewer1-0	11
umbral	threshold	método	imageviewer1-0	11
aplicar	apply	método	imageviewer2-0	11
aplicarFiltro	applyFilter	método	imageviewer2-0	11
crearFiltros	createFilters	método	imageviewer2-0	11
Filtro	Filter	clase	imageviewer2-0	11
FiltroClaro	LighterFilter	clase	imageviewer2-0	11
FiltroOscuro	DarkerFilter	clase	imageviewer2-0	11

Español	Inglés	Qué es	Proyecto	Capítulo
filtros	filters	campo	imageviewer2-0	11
FiltroUmbra	ThresholdFilter	clase	imageviewer2-0	11
getNombre	getName	método	imageviewer2-0	11
listaDeFiltros	filterList	variable local	imageviewer2-0	11
achicar	makeSmaller	método	imageviewer3-0	11
agrandar	makeLarger	método	imageviewer3-0	11
arregloCalculoDeX	computeXArray	método	imageviewer3-0	11
arregloCalculoDeY	computeYArray	método	imageviewer3-0	11
arregloX	xArray	variable local	imageviewer3-0	11
arregloY	yArray	variable local	imageviewer3-0	11
barraDeHerramientas	toolbar	variable local	imageviewer3-0	11
botonAchicar	smallerButton	campo	imageviewer3-0	11
botonAgrandar	largerButton	campo	imageviewer3-0	11
DOS_PI	TWO_PI	constante	imageviewer3-0	11
ESCALA	SCALE	constante	imageviewer3-0	11
FiltroOjoDePez	FishEyeFilter	clase	imageviewer3-0	11
grabarComo	saveAs	método	imageviewer3-0	11
habilitarBotones	setButtonsEnabled	método	imageviewer3-0	11
nuevaImagen	newImage	variable local	imageviewer3-0	11
azul	blue	variable local	imageviewer-final	11
borde	edge	método	imageviewer-final	11
difAzul	diffBlue	método	imageviewer-final	11
diferencia	difference	variable local	imageviewer-final	11
difRojo	diffRed	método	imageviewer-final	11
difVerde	diffGreen	método	imageviewer-final	11
FiltroBordes	EdgeFilter	clase	imageviewer-final	11
FiltroInvertir	InvertFilter	clase	imageviewer-final	11
FiltroSuavizar	SmoothFilter	clase	imageviewer-final	11
izquierda	left	variable local	imageviewer-final	11
prom	avg	variable local	imageviewer-final	11
promAzul	avgBlue	método	imageviewer-final	11
promRojo	avgRed	método	imageviewer-final	11
promVerde	avgGreen	método	imageviewer-final	11
rojo	red	variable local	imageviewer-final	11
suavizado	smooth	método	imageviewer-final	11
TAMANIO_DEL_PIXEL	PIXEL_SIZE	constante	imageviewer-final	11
verde	green	variable local	imageviewer-final	11
archivoDeSonido	soundFile	parámetro	simplesound	11
archivosDeAudio	audioFiles	parámetro	simplesound	11
archivosLista	fileList	campo	simplesound	11
buscar	seek	método	simplesound	11
buscarArchivos	findFiles	método	simplesound	11
cargarSonido	loadSound	método	simplesound	11
comenzar	start	método	simplesound	11
construirVentana	makeFrame	método	simplesound	11

Español	Inglés	Qué es	Proyecto	Capítulo
detener	stop	método	simplesound	11
duracionActualDelSonido	currentSoundDuration	campo	simplesound	11
ejecutar	play	método	simplesound	11
ejecutar	play	método	simplesound	11
etiquetaInfo	infoLabel	campo	simplesound	11
largoActualDelSonido	currentSoundFrameLength	variable local	simplesound	11
mostrarAcercaDe	showAbout	método	simplesound	11
mostrarInformacion	showInfo	método	simplesound	11
MotorDeSonido	SoundEngine	clase	simplesound	11
nombreDeArchivo	fileName	variable local	simplesound	11
nombreDir	nameDir	parámetro	simplesound	11
nombresDeArchivosDeAudio	audioFileNames	variable local	simplesound	11
pausar	pause	método	simplesound	11
posicionABuscar	seekPosition	variable local	simplesound	11
reproductor	player	campo	simplesound	11
ReproductorDeSonidoIGU	SoundPlayerGUI	clase	simplesound	11
resumir	resume	método	simplesound	11
seleccionado	selected	variable local	simplesound	11
setVolumen	setVolume	método	simplesound	11
sonidoActual	currentSoundClip	campo	simplesound	11
sonidossimples	simplesound	proyecto	simplesound	11
sufijo	suffix	parámetro	simplesound	11
todosLosArchivos	allFiles	variable local	simplesound	11
buscarNuevamente	research	método	address-book-v1g	12
cadenaDeBusqueda	searchString	variable local	address-book-v1g	12
campoBuscar	searchField	variable local	address-book-v1g	12
campoNombre	nameField	variable local	address-book-v1g	12
camposDeUnaLinea	singleLineFields	variable local	address-book-v1g	12
campoTelefono	phoneField	variable local	address-book-v1g	12
ejecutar	run	método	address-book-v1g	12
getTamanioPreferido	getPreferredSize	método	address-book-v1g	12
LibretaDeDireccionesIGU	AddressBookGUI	clase	address-book-v1g	12
listaDeResultados	resultList	variable local	address-book-v1g	12
mostrarVentana	showWindow	método	address-book-v1g	12
panelDeDatos	detailsPanel	variable local	address-book-v1g	12
panelTabulado	tabbedArea	variable local	address-book-v1g	12
prepararZonaDeIngreso	setupDataEntry	método	address-book-v1g	12
prepararZonaDeLista	setupListArea	método	address-book-v1g	12
zonaBuscar	searchArea	variable local	address-book-v1g	12
zonaDeBotones	buttonArea	variable local	address-book-v1g	12
zonaDeDesplazamiento	scrollArea	variable local	address-book-v1g	12
zonaDeEtiquetaBuscar	searchLabelArea	variable local	address-book-v1g	12
zonaDeEtiquetaDireccion	AddressLabelArea	variable local	address-book-v1g	12
zonaDeEtiquetaNombre	nameLabelArea	variable local	address-book-v1g	12
zonaDeEtiquetaTelefono	phoneLabelArea	variable local	address-book-v1g	12

Español	Inglés	Qué es	Proyecto	Capítulo
zonaDeLista	listArea	variable local	address-book-v1g	12
zonaDireccion	addressArea	variable local	address-book-v1g	12
zonaNombre	nameArea	variable local	address-book-v1g	12
zonaTelefono	phoneArea	variable local	address-book-v1g	12
agregar	add	método	address-book-v1t	12
agregarContacto	addDetails	método	address-book-v1t	12
Analizador	Parser	clase	address-book-v1t	12
ayuda	help	método	address-book-v1t	12
buscar	search	método	address-book-v1t	12
clave	key	parámetro	address-book-v1t	12
claveEnUso	keyInUse	método	address-book-v1t	12
claveVieja	oldKey	parámetro	address-book-v1t	12
codigo	code	variable local	address-book-v1t	12
coincidencias	matches	variable local	address-book-v1t	12
comando	command	variable local	address-book-v1t	12
comandos	commands	campo	address-book-v1t	12
comandosValidos	validCommands	campo	address-book-v1t	12
comparacion	comparison	variable local	address-book-v1t	12
contacto	details	parámetro	address-book-v1t	12
contactosDeEjemplo	sampleDetails	variable local	address-book-v1t	12
contactosOrdenados	sortedDetails	variable local	address-book-v1t	12
DatosDelContacto	ContactDetails	clase	address-book-v1t	12
direccion	address	campo	address-book-v1t	12
eliminarContacto	removeDetails	método	address-book-v1t	12
encontrar	find	método	address-book-v1t	12
esComando	isCommand	método	address-book-v1t	12
finDeBusqueda	endOfSearch	variable local	address-book-v1t	12
getComando	getCommand	método	address-book-v1t	12
getContacto	getDetails	método	address-book-v1t	12
getDireccion	getAddress	método	address-book-v1t	12
getLibreta	getBook	método	address-book-v1t	12
getNombre	getName	método	address-book-v1t	12
getNumeroDeEntradas	getNumberOfEntries	método	address-book-v1t	12
getTelefono	getPhone	método	address-book-v1t	12
interaccion	interaction	campo	address-book-v1t	12
lector	reader	campo	address-book-v1t	12
leerLinea	readLine	método	address-book-v1t	12
libreta	book	campo	address-book-v1t	12
LibretaDeDirecciones	AddressBook	clase	address-book-v1t	12
LibretaDeDireccionesDemo	AddressBookDemo	clase	address-book-v1t	12
LibretaDeDireccionesInterfazDeTexto	AddressBookTextInterface	clase	address-book-v1t	12
libreta-de-direcciones-v1t	address-book-v1t	proyecto	address-book-v1t	12
listar	list	método	address-book-v1t	12
listarContactos	listDetails	método	address-book-v1t	12
modificarContacto	changeDetails	método	address-book-v1t	12

Español	Inglés	Qué es	Proyecto	Capítulo
mostrarComandos	showCommands	método	address-book-v1t	12
mostrarInterfaz	showInterface	método	address-book-v1t	12
mostrarTodos	showAll	método	address-book-v1t	12
nombre	name	campo	address-book-v1t	12
numeroDeEntradas	numberOfEntries	campo	address-book-v1t	12
otro	other	parámetro	address-book-v1t	12
otroContacto	otherDetails	variable local	address-book-v1t	12
palabra	word	variable local	address-book-v1t	12
PalabrasComando	CommandWords	clase	address-book-v1t	12
prefijo	keyPrefix	parámetro	address-book-v1t	12
resultados	results	variable local	address-book-v1t	12
telefono	phone	campo	address-book-v1t	12
todasLasEntradas	allEntries	variable local	address-book-v1t	12
getClave	getKey	método	address-book-v3t	12
NoCoincideContactoException	NoMatchingDetailsException	clase	address-book-v3t	12
arriboADestino	arrivedAtDestination	método	taxi-company-outline	14
arriboASalida	arrivedAtPickup	método	taxi-company-outline	14
asignarVehiculo	scheduleVehicle	método	taxi-company-outline	14
compania	company	parámetro	taxi-company-outline	14
CompaniaDeTaxis	TaxiCompany	clase	taxi-company-outline	14
compania-de-taxis-esquema	taxi-company-outline	proyecto	taxi-company-outline	14
crearPasajero	createPassenger	método	taxi-company-outline	14
dejarPasajero	offloadPassenger	método	taxi-company-outline	14
destino	destination	campo	taxi-company-outline	14
destinos	destinations	campo	taxi-company-outline	14
elegirUbicacionDelDestino	chooseTargetLocation	método	taxi-company-outline	14
estaLibre	isFree	método	taxi-company-outline	14
getDestino	getDestination	método	taxi-company-outline	14
getUbicacion	getLocation	método	taxi-company-outline	14
getUbicacionDelDestino	getTargetLocation	método	taxi-company-outline	14
getUbicacionDeSalida	getPickupLocation	método	taxi-company-outline	14
limpiarUbicacionDelDestino	clearTargetLocation	método	taxi-company-outline	14
Minibus	Shuttle	clase	taxi-company-outline	14
notificarArriboASalida	notifyPickupArrival	método	taxi-company-outline	14
notificarArriboDePasajero	notifyPassengerArrival	método	taxi-company-outline	14
Pasajero	Passenger	clase	taxi-company-outline	14
PasajeroFuente	PassengerSource	clase	taxi-company-outline	14
pasajeros	passengers	campo	taxi-company-outline	14
recoger	pickup	método	taxi-company-outline	14
salida	pickup	campo	taxi-company-outline	14
setUbicacion	setLocation	método	taxi-company-outline	14
setUbicacionDelDestino	setTargetLocation	método	taxi-company-outline	14
setUbicacionDeSalida	setPickupLocation	método	taxi-company-outline	14
solicitarViaje	requestPickup	método	taxi-company-outline	14
Ubicación	Location	clase	taxi-company-outline	14

Español	Inglés	Qué es	Proyecto	Capítulo
ubicacionDelDestino	targetLocation	campo	taxi-company-outline	14
Vehiculo	Vehicle	clase	taxi-company-outline	14
vehiculos	vehicles	campo	taxi-company-outline	14
fuente	source	variable local	taxi-company-outline-test	14
PasajeroFuenteTest	PassengerSourceTest	clase	taxi-company-outline-test	14
testCreacion	testCreation	método	taxi-company-outline-test	14
ubicacionDelTaxi	taxiLocation	variable local	taxi-company-outline-test	14
compania-de-taxis-etapa-uno	taxi-company-stage-one	proyecto	taxi-company-stage-one	14
destino	target	variable local	taxi-company-stage-one	14



# Índice analítico

@author 147, 457–8  
@param 147, 457–8  
@return 147, 457–8  
@see 457, 458  
@throws 457–8  
@version 147, 457–8

## A

abrir un proyecto 433  
abstracción 53–4, 110  
    en software 54  
interacción de objetos 70, 77  
lectura de documentación de clase 126  
*ver también* técnicas de abstracción,  
herencia  
Abstract Window Toolkit *ver* AWT  
AbstractCollection 255  
AbstractList 255  
acceso protegido 271–3  
acoplamiento 149, 193, 199–203, 222, 398  
    alto 200, 207, 293  
    bajo 149, 193, 199, 207  
    directo 422  
    implícito 207–10, 218, 422  
    usar encapsulamiento para reducir el aco-  
        plamiento 200–3  
y responsabilidades 204–6  
ActionEvent 319, 31  
actionPerformed 321, 322, 323, 324, 325  
actores 302–3  
    concepto 425–6  
actores dibujables 302–3  
agregar componentes simples 317–18  
agregar menús 318–19  
agrupar objetos 81–118  
*agenda* proyecto 82, 97–8

biblioteca de clases 82–4  
clases genéricas 86–7  
eliminar un elemento de una colección 88–  
    9  
estructuras de objetos con colecciones 84–6  
numeración dentro de las colecciones 87–  
    8  
*ver también* proyecto subastas, colecciones  
de tamaño fijo,  
análisis y diseño de aplicaciones 393–9  
cooperación 402  
crecimiento del software 403–5  
descubrir clases 394–5  
documentación 401–2  
ejemplo reserva de entradas para el cine  
    394  
escenarios 396–9  
método verbo/sustantivo 394–5  
prototipos 402–3  
tarjetas CRC 395–6  
*ver también* usar patrones de diseño  
analizador de un archivo de registro 106–8,  
    111  
API 126, 136, 347, 385, 456, 463, 488–9  
aplicaciones  
prueba 159  
*ver también* análisis y diseño de apli-  
    caciones, diseñar aplicaciones  
Application Programming Interface *ver* API  
archivo de definiciones 448  
archivo explícito 447  
archivos .jar 446–7  
archivos binarios 385  
archivos de texto 385  
archivos ejecutables .jar 446–7  
argumento

- en un constructor 418
- en un método 418
- validar 360-1
- ArrayList** clase 82-9, 92, 95-7, 99, 102-5, 464
  - abstracción 307
  - comportamiento aleatorio 131, 133, 135, 136
  - conjuntos 142
  - diseñar aplicaciones 397, 408
  - documentación de clases de biblioteca 120
  - herencia 231, 251, 255
  - interfaces 152
  - mapas 138
  - modificadores de acceso 149
  - paquetes y la sentencia `import` 137
- arreglo** 436
  - creación de objetos arreglo 109-10
  - declaración de variables arreglo 108-9
  - objetos arreglo 110
  - ver también* colecciones de tamaño fijo
  - aserciones 172, 378-81, 442
    - controlar la consistencia interna 378-9
    - facilidad 379
    - pautas para usar aserciones 380-1
    - sentencia `assert` 378-80
    - y marco de trabajo de unidades de prueba de BlueJ 381
  - asignación 27-8
    - sentencia de asignación 27-8, 32, 73
    - y subtipos 249-50
- autoboxing 105, 254, 435, 436-7
- AWT** 313-14
- B**
  - barra de menú 316-17
  - barra de título 316-17
  - barras de desplazamiento 351
  - Base de Datos de Entretenimientos Multimediales** *ver DoME*
  - batería de prueba 166-8
  - Beck, Kent 169, 395n
  - BevelBorder** clase 346
  - bloque 29, 42, 324
    - `catch` 372, 373, 374, 376, 382
    - de etiquetas 457
  - try 372, 374, 376
  - bloque `catch` 372, 373, 374, 376, 382
  - bloque `try` 372, 374, 376
- BlueJ**
  - configurar 448-9
  - depurador 433
  - proyecto 433-4
  - pruebas de unidad 159-66, 381
  - soporte para javadoc 458
  - Tutorial 433
- boolean** 58, 254
- booleana/o**
  - bandera 274
  - campo 216
  - campo depuracion 185
  - expresión 9, 39, 92, 379, 380, 381
  - tipo 363
    - valores `true` o `false` 60
    - variable en sistema de *SoporteTecnico* 125
- BorderLayout** 329-33, 344, 345, 347
- bordes 346-7
- botón Halt 451
- botón Step 451
- botón Step Into 452
- botón Terminate 452
- botones 343-6
- botones de control 451-2
- BoxLayout** 329, 331
- Brooks, Frederick P. Jnr 404n
- búsqueda dinámica del método 264-7
- C**
  - cadena de comando 321
  - cadena del visor 68
  - caja de diálogo 7, 337-8
  - cambiar las plantillas para las clases nuevas 449
  - campos 9, 22, 23-4, 34-5, 42-3, 230n, 231
    - alcance 26
    - creditos 47
    - declaraciones 29
    - ID 47
    - interacción de objetos 5, 65, 73
    - nombre 47
    - privados 24
    - saldo 25

- total 25
- campos estáticos 284, 407
- campos públicos 149-50, 207
- capturar excepciones 368-9, 442
- característica de serialización 385
- caso de estudio entrada/salida de texto 384-91
  - entrada de texto mediante `FileReader` 389-90
  - lectores, escritores y flujos 385
  - proyecto *libreta-de-direcciones-io* 385-7
  - salida de texto mediante `FileWriter` 388-9
- scanner: leer entradas desde la terminal 390-1
- serialización de objetos 391
- casos de uso *ver* escenarios
- char 254
- ciclo do-while 441
- ciclo for 106, 111-14, 271, 441
- ciclo for mejorado 111
- ciclo for-each 90-3, 95-7, 99, 112-14, 271, 441, 462
- ciclo infinito 93
- ciclo simple 291
- ciclos 440-1, 460
  - cuerpo 90-2, 93
  - do-while 441
  - encabezado 90, 92, 112
  - for 106, 111-14, 271, 441
    - for-each 90-3, 95-7, 99, 112-14, 271, 441, 462
  - infinito 93
  - simple 291
  - variable 91
  - while 92-5, 96, 97, 112, 114, 440
- Círculo 4, 5, 6, 7, 8, 9, 10, 16
- clase 11-12, 17, 36, 42, 58, 436
  - cohesión 212-13
  - comentario 147, 460-1
  - constantes 155
  - define tipos 55
  - definiciones 17-51
    - asignación 27-8
    - campos 23-4, 42-3
    - constructores 25-6
    - examinar una definición de clase 19-21
  - imprimir desde métodos 32-4
  - métodos de acceso 28-31
  - métodos de modificación 31-2
  - parámetros 42-3
  - pasar datos mediante parámetros 26-7
  - proyecto *maquina-de-boletos* 17-18, 34-6, 43
  - sentencia condicional 36-40
  - variables locales 41-3
- diagrama de clases 232
- diagramas de clases y diagramas de objetos 56-8
- clase diferente 9
- descubrir clases 394-5, 414-15
- lectura de documentación de clases 126-30
  - comparar interfaz e implementación 127-8
  - comprobar la igualdad de cadenas 130
  - usar métodos de clases de biblioteca 128-30
- alcance 42
- DoME* 230-2
- escribir documentación de clases 146-8
- herencia 231
- interfaces 418
- jerarquía 245
- métodos 224, 225
  - ver también* métodos estáticos
- misma clase 9
- variables y constantes 153-5
  - ver también* clases abstractas, diseñar clases, objetos y clases, subclase, superclase
- clase Actor 300-1, 302, 303-5, 306
- clase Agenda 83-6, 87-8, 89
- clase Analizador 192, 208-10, 216, 407
- clase AnalizadorLog 106, 109, 111
- clase Animal 293-4, 295, 296, 297, 298-9, 300, 304-5
- clase ArchivoDeArchivos 326, 327, 328
- clase Asiento 398
- clase Auto 252, 261
- clase BaseDeDatos 232, 236-9, 247-8, 250, 261-2
  - métodos del objeto 270
- tipo dinámico y tipo estático 260

- clase *Bicicleta* 252
- clase *BufferedImage* 326
- clase *BufferedReader* 385, 389, 406, 465
- clase *BufferedWriter* 465
- clase *Button* 314
- clase *Calendar* 224
- clase *Campo* 281, 292, 300, 409
- clase *Canvas* 4, 150, 152
- clase *Cazador* 300, 305
- clase *CD* 230-4, 238, 239-44, 251, 253
  - búsqueda dinámica del método 264, 267
  - métodos de los objetos 271
  - tipo estático y tipo dinámico 260, 262
- clase *Cine* 395
- clase *Cita* 160, 164, 171, 172, 173
- clase *Ciudad* 429
- clase *CiudadIGU* 429
- clase *ClienteDeCorreo* 72, 74, 75, 78
- clase *Color* 9, 326, 351
- clase *Comando* 192, 216
- clase *CompaniaDeTaxis* 416-19, 422, 425, 429-31
- clase *Conejo* 280, 281, 282-5, 292, 293, 294, 295, 299, 300
- clase *Contador* 281
- clase *Contestador* 121-2, 124-5, 133-5, 141, 144-5
- clase *DatosDelContacto* 355, 360, 370, 371, 378, 385, 391
- clase *Demo* 425, 428
- clase *Dia* 160-3, 164, 171, 172, 173
- clase *DiaTest* 169, 170
- clase *Drawable* 302, 304, 305
- clase *DVD* 230-2, 234-6, 238-40, 242, 248, 250-1, 253
  - búsqueda dinámica del método 264, 265-6, 267
  - métodos de los objetos 269, 271
  - tipo estático y tipo dinámico 260, 262
- clase *Elemento* 212-13, 216, 239-45, 248, 250-1
  - búsqueda dinámica del método 264, 266
  - DoME* 259
  - métodos de los objetos 269-70, 264, 262
- clase *EntradaDeLog* 106, 111
- clase *envolvente* 323, 325
- clase *EstadisticasDelCampo* 281
- clase *Exception* 366, 374
- clase *FabricaDeActor* 408
- clase *FabricaDeConejo* 408
- clase *FabricaDeZorro* 408
- clase *Fila* 398
- clase *File* 465
- clase *FileReader* 385, 389-90, 466
- clase *FileWriter* 385, 388-9, 466
- clase *Frame* 314
- clase *Funcion* 397
- clase *Habitacion* 192, 198-9, 200-3, 204-6, 212-13, 216, 273-5
- clase *HashMap* 137, 138-9, 46
  - acoplamiento 199-201, 203
  - diseño dirigido por responsabilidades 205
  - interfaces 152
- clase *HashSet* 141-2, 144, 464
  - diseñar aplicaciones 408
  - escribir documentación de clases 146
  - interfaces 152
- clase *hijo* *ver* subclase
- clase *ImagenOF* 326, 327, 335-6, 338, 341
- clase *ImagenOF* 327
- clase *ImagePanel* 327
- clase *InterfazDeUsuario* 176
- clase *IOExcepction* 466
- clase *JButton* 314, 331
- clase *JComboBox* 350-1
- clase *JComponent* 327
- clase *JDialog* 337
- clase *JFrame* 314, 315, 316, 317, 318, 332, 337, 350
- clase *JLabel* 329, 350
- clase *JList* 351
- clase *JMenu* 314, 318, 334
- clase *JMenuBar* 318
- clase *JMenuItem* 318, 319, 334
- clase *JOptionPane* 337
- clase *JPanel* 331-2, 344, 345, 346
- clase *JScrollPane* 351
- clase *Juego* 192, 194-6, 198-201, 204, 206, 208-11, 216-18
  - herencia 245
  - herencia con sobrescritura 274-5
- clase *JuegoDeMesa* 245

- clase JuegoDeVideo 244-5  
clase Jugador 217  
clase Labrador 240  
clase LectorDeArchivoLog 106, 111  
clase LectorDeEntrada 121-2, 141, 142, 146  
clase LibretaDeDirecciones 355-8, 359-61, 385, 387, 391  
clase LibretaDeDireccionesDemo 355  
clase LibretaManejadorDeArchivos 385-7, 389, 391  
clase LinkedList 307, 464  
clase Lote 98-9, 102, 136  
clase Math 463  
clase Mensaje 72, 73, 77, 78  
clase Menu 314  
clase Minibus 417, 422  
clase MotorDeCalculadora 176, 178-80, 181, 183-4  
clase MotorDeCalculadoraProbador 177-8  
clase MotorDeSonido 350  
clase Object 253, 254, 266, 268-9, 463  
clase padre *ver* superclase  
clase PalabrasComando 192, 207-10, 216, 218, 219-22  
clase Palette 187  
clase Pasajero 416, 417, 418, 422  
clase PasajeroFuente 416, 418, 422, 428, 430  
clase PasajeroFuenteTest 425  
clase PasajeroPerdidoException 428  
clase PasajeroTest 425  
clase PelotasDemo 150-1  
clase Random 135, 137, 146, 465  
clase ReboteDePelota 150-1, 153, 155  
clase referencia 169  
clase ReproductorDeSonidoIGU 350  
clase Sala 398  
clase Scanner 465  
clase Semana 160  
clase SeparadorDeLineaLog 106, 391  
clase ServidorDeCorreo 72, 78  
clase Simulador 280-2, 293-5, 301, 306, 308, 428  
configuración 288-91  
diseñar aplicaciones 408  
un paso 291-2  
clase SistemaDeReserva 396, 397  
clase SistemaDeSoporte 121-5, 144, 146, 149  
clase String 84, 91, 94, 364, 463  
comportamiento aleatorio 135, 136  
dividir cadenas 143  
lectura de documentación de clases 126, 128, 129, 130  
sistema *SoporteTecnico* 126  
clase StringBuffer 463  
clase Taxi 417, 422, 425, 426-7, 429  
clase Throwable 366  
clase Ubicacion 281, 416, 417, 425, 431  
clase UbicacionTest 425, 427, 428  
clase Vehiculo 252, 261, 416, 417, 419-21, 422, 423, 425  
clase VisorDeImagen 315-16, 319-21, 323, 327-8, 335  
clase VisorDelCampo 323  
clase VisorDelSimulador 281, 323, 409  
clase VisorDeNumeros 55-8, 59-61, 63, 65, 67, 69-70  
clase Zorro 280, 281, 282, 285-8, 292, 293, 294, 295, 299, 300  
Clase/Responsabilidades/Colaboradores  
    *ver* tarjetas CRC  
clases abstractas 245, 292-8  
    métodos abstractos 294-6  
    superclase Animal 293-4  
clases concretas 296  
clases de biblioteca 82-4, 128-30, 463-6  
    estándar 313  
clases de la biblioteca estándar 313  
clases envoltorio 254, 436-7  
clases genéricas 84, 86-7  
clases internas 322-3  
    anónimas 323-5  
cláusula catch 442  
cláusula finally 376, 390  
cláusula implements 304  
cláusula throws 371, 375  
cláusulas  
    finally 376, 390  
    implements 304

- throws 371, 375
- código de *VisorDeReloj* 59-66, 67-70
  - clase *VisorDeNumeros* 59-61
  - concatenación de cadenas 61-2
  - operador módulo (%) 62-3
- código fuente 11-12, 17, 19, 23, 28
  - DoME* 232-8
  - herencia 238, 247
  - interfaces gráficas de usuario 349
  - interacción de objetos 63, 65, 70, 71, 72
  - relevancia 198-9
  - sistema *SoporteTecnico* 122-3
  - sobrescritura 263-4
  - legibilidad 74
- código
  - comentarios 461
  - duplicación 194-7, 238-9, 245-6, 251, 338-9, 342
  - estilo 159
  - fragmentos 349
  - lectura 70-1, 122-6
  - modismos 462
  - reutilizar 246
  - ver también* pseudocódigo
- cohesión 193, 211-14, 222
  - alta 207
  - de clases 212-13
  - de métodos 211-12
  - mala 196
  - para la legibilidad 213
  - para la reusabilidad 213-14
- colaboradores 418-19
- colección tipeada *ver* clases genéricas
- colección
  - flexible 81-2, 105
  - jerarquía 255
  - usar colecciones 103-5
    - ver también* colecciones de tamaño fijo,
    - procesar una colección completa
- colecciones de tamaño fijo 105-16
  - analizador de un archivo de registro 106-8, 111
  - ciclo *for* 111-14
  - creación de objetos arreglo 109-10
  - declaración de variables arreglo 108-9
  - objetos arreglo 110
- colecciones de tamaño flexible 81-2, 105
- comando Step Into 77, 78
- comentarios 7, 23-4, 29, 128, 159, 177-8
  - símbolo \*/ 456
  - símbolo /\*\* 456, 461
- compilación 12
- compiladores 158
- complejidad 53
- componentes 313
- componentes Swing 313-14, 317, 319, 326-7, 329, 331, 346-7
- comportamiento aleatorio 131-6
  - generación de respuestas por azar 133-5
  - lectura de documentación de clases parametrizadas 135-6
  - números aleatorios en un rango limitado 132-3
- CompoundBorder 346
- comprobar la igualdad de cadenas 130
- concatenación de cadenas 61-2, 69
- conjuntos 141-2
- constante cadena de VERSION 338
- constructores 22, 25-9, 27, 30, 34-5, 36, 221, 231
  - acceso protegido 272
  - espacio de 26, 27
  - herencia 232, 236, 242-4
  - interacción de objetos 59, 66-7, 72, 73
  - manejo de errores 370-1
  - múltiples 67-8
  - parámetros 345, 346
- constructores múltiples 67-8
- contenedores anidados 331-4, 345
- contenido del CD 433-4
- control de consistencia interna 380
- controlar la consistencia interna 378-9
- cooperación 402
- crear una ventana 315-17
- crecimiento del software 403-5
- Crowther, Will 190
- Cuadrado 4, 8, 10
- Cuadro 10, 11, 12
- cuadros combinados 350-1
- cuerpo 58, 254
- Cunningham, Ward 395n

**C**

- declaraciones 29, 41
- default 439
- depurador/depuración 70-7, 78, 180, 187, 271, 433, 450-3
  - activar o desactivar la información de depuración 185-6
  - entrar en los métodos 77
  - escenario de depuración 176-7
  - herencia 268
  - objetos con buen comportamiento 159
  - paso a paso 76-7
  - poner puntos de interrupción 74-6
  - proyecto *sistema-de-correo* 71-2
  - salida 184
  - sentencias de impresión 183
- desacoplamiento de la interfaz de comandos 220-2
  - desacoplamiento de la interfaz de comandos 220-2
- desarrollo iterativo 404-5, 424-31
  - más ideas para desarrollar 429-30
  - pasos del desarrollo 424-5
  - primer etapa 425-8
  - probar la primera etapa 428-9
  - reusabilidad 431
- descripción principal 456-7
- destino estándar de salida 390
- diálogo de confirmación 337
- diálogo de mensaje 337
- diálogo modal 337, 338
- diálogo no modal 337
- diálogo preferencias 448, 449
- diámetro 9
- diapositiva 351
- dibujo selectivo 302
- directorio telefónico (mapa) 138
- diseñar aplicaciones 393-412
  - diseño de clases 399-401
    - ver también análisis y diseño
  - diseñar clases 189-226, 399-401, 418-23
    - acoplamiento 193, 199-203
    - acoplamiento implícito 207-10
    - cohesión 193, 211-14
    - colaboradores 418-19
    - diseño dirigido por responsabilidades 204-6
- duplicación de código 194-7
- ejecutar un programa fuera de BlueJ 223-5
- esquema de implementación 419-23
- extensiones 197-9
- interfaces 400-1, 418
- interfaz de usuario 401
- localización de cambios 206-7
- pautas de diseño 222-3
- pensar a futuro 210-11
- prueba 423
- refactorización 214-18
- refactorización para independizarse del idioma 218-22
- world-of-zuul* juego 191-2
- diseño dirigido por responsabilidades 204-6, 209, 217, 335
- diseño
  - dirigido por responsabilidad 204-6, 209, 217, 335
  - malo 190-1, 194-6, 197, 199, 217
  - ver también diseño de clases, diseñar aplicaciones, diseñar clases, usar patrones de diseño
- disposición 313, 328-31, 459-60
- gestores 313, 329, 331-2, 333n, 345, 346
- distancia 6,7
- divide y reinarás 53
- división de cadenas 142-4
- documentación 401-2, 460-1
  - biblioteca estándar 126-7
  - comentarios 456-8
  - de clases de biblioteca 120
  - escribir documentación de clases 146-8
  - lectura 126-30
  - lectura de clases parametrizadas 135-6
- documentación de la biblioteca estándar 126-7
- documentación de la biblioteca estándar 126-7
- DoME* 229-38, 246, 266, 267, 291, 293
  - agregar otros tipos de elementos 244-6
  - clases y objetos 230-2
  - código fuente 232-8
  - discusión de la aplicación 238-9
  - método imprimir 258-9

- duplicación 86
- E**
- ejecutar pruebas 455
  - ejemplo reloj 52, 54-5
  - ejemplo reserva de entradas para el cine 394
  - elemento *Acerca del Visor de Imágenes* 328
  - eliminar un elemento de una colección 88-9
  - `EmptyBorder` 346
  - encabezado 29, 42
  - encapsulamiento 204
    - para reducir el acoplamiento 200-3
  - enmascaramiento de tipos 251-2
  - entero
    - arreglo de enteros 108
    - campos 55
    - valor 62
  - entrada
    - clases de entrada/salida 465
    - diálogo 337
    - variable 130
    - ver también* entrada/salida de texto
  - error
    - anular el error 383-4
    - aserciones 378-81
    - definir nuevas clases de excepción 377-8
    - informar errores del servidor 361-5
    - informar un error 355
    - manejo 354-92
    - programación defensiva 359-61
    - proyecto *libreta-de-direcciones* 355-8
    - recuperarse del error 382-3
    - sintáctico 158, 354
    - valor fuera de los límites válidos 364
    - ver también* manejar excepciones, lanzar excepciones, error lógico, error del servidor, caso de estudio entrada/salida de texto
    - error fuera de los límites válidos 364
    - errores lógicos 158, 354, 365, 367, 383, 384
    - errores sintácticos 158, 354
    - escenarios 396-9, 414, 416-18
    - escribir para el mantenimiento 159
    - escritores 385
    - espaciar 346-7
    - especialización 15
    - esquema de implementación 419-23
    - estado 8-9
    - `estaVisible` 9
    - estilo 177-8
    - estructuras de control 438-42
    - estudiante 13, 14
    - `EtchedBorder` 346, 347
    - etiquetas de una sola línea 457
    - exactitud de un programa 159
    - excepción 442
      - definir nuevas clases de excepción 377-8
      - ver también* capturar excepciones, excepciones comprobadas, manejo de excepciones
      - lanzar excepciones, excepciones no comprobadas
      - excepciones comprobadas 366-8, 372, 374, 384, 385
      - excepciones no comprobadas 366-7, 371, 379, 384
      - expresión 28
      - expresión entera 109
      - expresiones aritméticas 443-4
      - extensibilidad 247
      - extensión 197-9, 207

**F**

    - fallos 158
      - ver también* depurador/depuración
    - figuras 4, 9, 11, 15
    - filtro `OjoDePez` 347
    - filtros 334-6, 339-40, 343, 347
    - flexibilidad a través de la abstracción 302
    - `FlowLayout` 39-30, 332, 344, 345
    - fluxos 385
    - formato PNG 314, 326, 327, 350
    - función inspeccionar 8
    - funcionalidad 214, 215, 217, 342, 343, 347, 350, 454

**G**

    - `Gamma`, Erich 169, 406n
    - `GIF` 350
    - `GridLayout` 329-32, 345
    - guía de estilo de programación 459-62

**H**

Helm, Richard 406n  
herencia 151, 258-77, 309  
    acceso protegido 271-3  
    búsqueda dinámica del método 264-7  
    de JFrame 350  
    diseñar aplicaciones 406  
    *Dome* método imprimir 258-9  
    jerarquía 249, 266, 267, 271, 366, 416, 423  
    llamada a super en métodos 267-8  
    manejar errores 354  
    manejo del error 377  
    métodos de los objetos: `toString` 268-71  
    métodos polimórficos 268  
    sobrescritura 262-4, 273-5  
    técnicas de abstracción 291, 292, 294, 295,  
        306  
    tipo estático y tipo dinámico 260-2  
    *ver también* herencia múltiple, mejorar la  
        estructura mediante herencia  
herencia múltiple 300-3, 306, 309  
    actores dibujables 302-3  
    clase Actor 300-1  
    dibujo selectivo 302  
    flexibilidad a través de la abstracción 302  
    interfaces 305-6  
herencia y derechos de acceso 242  
herramienta JUnit de pruebas unitarias 454-5  
HTML 120, 456, 457, 458

**I**

imagen  
    agregar la imagen 327-8  
    archivos 385  
    clases para procesar imágenes 326-7  
    filtros 334-6  
implementación 148-9, 190, 199  
    comparar con interfaces 127-8  
importar las clases 461  
imprimir desde métodos 32-4  
indentación 459  
índice 87, 97, 110  
    comparar acceso mediante índices e iteradores 95-6  
    valores 88-9  
informe de errores del servidor 361-5

notificar al objeto cliente 362-5  
notificar al usuario 362  
inicialización  
    sentencia 441  
*ver también* constructores  
    y herencia 242-4  
inspectores 164-5  
instancia 3-4, 10  
    campos 225, 329, 400  
    constantes específicas 155  
    métodos 224  
    única 324-5  
    variables 76, 153-4, 274, 284, 316  
*ver también* campos  
*ver también* instancias múltiples  
instancia envolvente 323  
instancia única 324-5  
instancias múltiples 8  
instrucción break 439  
int 58, 108, 254  
Integer 254  
interacción cliente-servidor 359-60  
interacción con el cliente 404  
interfaces 148-9, 150-3, 199, 203, 204, 218,  
    303-9  
    agregar componentes 343-7  
    cambiar de idioma 448  
    clase 400-1  
    como especificaciones 306-7  
    como tipos 306  
    comparar con implementación 127-8  
    herencia 255  
    herencia múltiple 305-6  
    interacción de objetos 70  
    interfaz Actor 303-5  
    o clase abstracta 309  
    usuario 401  
    y modularización 174-6  
interfaces gráficas de usuario 312-53, 358,  
    429, 466  
    AWT y Swing 313-14  
componentes, gestores de disposición y  
    captura de eventos 313  
extensiones 347-9  
reproductor de sonido 349-52  
*ver también* ejemplo Visor de Imagen

- ejemplo
- interfaz `ActionListener` 319-21, 323, 324, 325
- interfaz `Collection` 82, 416, 417, 464
- interfaz `DrawableItem` 429
- interfaz `Iterator` 95, 96, 97, 205, 464
- interfaz `List` 307, 464
- interfaz `Map` 137, 138, 464
- interfaz `Observable` 409, 422
- interfaz `Observer` 409
- interfaz `Serializable` 465
- interfaz `Set` 205, 46
- invocación de métodos anidados 181
- invocar métodos 5-6
- iterador 462
  - comparar con acceso mediante índices 95-6
  - de colecciones 408
  - objeto 95
- J**
- Java
  - 2 plataforma 463
  - 2 Standard Edition (J2SE) Software Development Kit (SDK) 433
  - biblioteca 151
  - desarrollar fuera del entorno BlueJ 447
  - ejecutar fuera del entorno BlueJ 445-6
  - estructuras de control 438-42
  - jerarquía de colecciones 306-7
  - Kit de desarrollo *ver* JDK
  - Runtime Environment *ver* JRE
  - tipos de datos 435-7
- javac 447
- javadoc 456-8
  - documentación 150, 371
  - etiqueta `@throws` 366
- JDK 447
- Johnson, Ralph 406n
- JPEG formato 314, 326, 327, 350
- JRE 447
- L**
- lanzar una excepción 362, 365-71, 382, 383, 384, 388-9, 442
  - clases de excepción 366-8
- efecto de una excepción 368-9
- excepciones comprobadas 369-70
- impedir la creación de un objeto 370-1
- lanzar una excepción 366
- mecanismo 379
- lectores 385
- legibilidad y cohesión 213
- legible para javadoc 461
- letras mayúsculas 459
- letras minúsculas 459
- límites 164
  - exclusive 133
  - inclusive 133
- lista 142, 351
- llamada super en métodos 267-8
- llaves 459-60
- localización de cambios 206-7

## M

- MacOS 318n
- manejo de eventos 313, 319
- manejo de excepciones 371-6
  - capturar excepciones, sentencia try 372-3
  - cláusula finally 376
  - excepciones comprobadas: cláusula throws 371
  - lanzar y capturar varias excepciones 374-5
  - propagar una excepción 375
- mantenibilidad 274
- mantenimiento 246
- marco de trabajo JUnit para pruebas 169
- mejorar la estructura del programa 338-43
- mejorar la estructura mediante herencia 229-57
  - autoboxing y clases envoltorio 254
  - clase `Object` 253
  - herencia e inicialización 242-4
  - herencia y derechos de acceso 242
  - jerarquía de colecciones 255
  - jerarquías de herencia 240-1
  - subtipos 247-52
  - usar herencia 239-40
  - ventajas de la herencia 246-7
- ver* también *DoME*
- menú Ayuda 328, 334, 337
- menú Filtro 328, 339-40

- método 8, 9, 22, 26, 30, 34-5, 230n, 231  
 acceso protegido 272  
 búsqueda 260  
 cohesión 211-12  
 comentario 147, 161  
 despacho *ver* búsqueda dinámica del método  
 espacio 26  
 get 36  
 imprimir desde métodos 32-4  
 interacción de objetos 59, 65  
 invocado 5  
 invocar métodos 5-6, 13, 68-70, 78  
     abstracción 306  
 diseñar aplicaciones 400  
 llamadas a métodos externos 69-70  
 llamadas a métodos internos 68  
 ligadura *ver* búsqueda dinámica del método  
 polimorfismo 268  
 signatura 29, 203, 400, 418  
 sobreescritura 266  
 stub 400  
     *ver también* métodos de modificación  
 método get *ver* abstracción  
 método imprimir 258-9, 260, 261-2  
 método main 224-5, 445-6, 447  
 método set *ver* métodos de modificación  
 método split 143  
 método toString 268-71, 378  
 método verbos/sustantivos 394  
 métodos de acceso 28-31, 35, 47, 54, 59, 150, 200-1, 230  
 herencia 232, 242, 245  
 herencia con sobreescritura 274  
 técnicas de abstracción 293  
 tipo estático y tipo dinámico 260  
 métodos de consulta 389  
 métodos de impresión 185  
 métodos de modificación 31-2, 35, 47, 54, 59, 110, 150, 230  
 herencia 232, 242, 245  
 técnicas de abstracción 293  
 métodos estáticos 143n, 337, 407  
     *ver también* métodos de clase  
 métodos para informar estado 183  
 modelo de cascada 403-4  
 modificación 207  
 modificadores *ver* métodos de modificación  
 modificadores de acceso 148, 461  
 modularización 53-4  
     en el proyecto reloj 54-5  
     e interfaces 174-6  
*MouseEvent* 319
- ## N
- nombre 7, 16, 459  
 sobrecarga 73  
 notación de punto 69  
*NotePad* 447  
 numeración dentro de las colecciones 87-8
- ## O
- objeto llave 138  
 objeto valor 138  
 objeto  
     lenguaje orientado a objetos 239  
     banco de objetos 4-5, 8, 23, 160  
     impedir la creación de un objeto 370-1  
     diagrama de objetos 66, 84  
     comparación de diagramas de clases con diagramas de objetos 56-8  
*DoME* 230-2  
 inspector de objeto 8, 9  
 interacción 10-11, 52-80  
 abstracción en el software 54  
 abstracción y modularización 53-4  
 comparación de diagramas de clases con diagramas de objetos 56-8  
 constructores múltiples 67-8  
 depurador 70-7  
 invocar métodos 68-70, 78  
 modularización en el proyecto *reloj* 54-5  
 objetos que crean objetos 66-7  
 proyecto *reloj* 52, 55  
 tipos primitivos y tipos objeto 58  
*visorDeReloj* 70  
     *ver también* código fuente del *VisorDeReloj*  
 estructuras de objetos con colecciones 84-6  
 métodos: *toString* 268-71

- programación orientada a objetos 18
  - referencia 56
  - serialización 391
  - tipos 58, 435, 436
    - ver también* agrupar objetos, objetos y clases, objetos con buen comportamiento
  - objetos anónimos 102-3
  - objetos con buen comportamiento 158-88
    - comentarios y estilo 177-8
    - depuradores 187
    - escenario de depuración 176-7
    - modularización e interfaces 174-6
    - poner en práctica las técnicas 187
    - prueba y depuración 159
    - seguimiento manual 178-82
    - selección de estrategia de prueba 186
    - sentencias de impresión 183-6
    - ver también* automatización de pruebas, pruebas de unidad con BlueJ
  - objetos de prueba 173-4, 455
  - objetos inmutables 129
  - objetos y clases 3-16
    - código fuente 11-12
    - crear objetos 4-5
    - definición de un objeto 9-10
    - estado 8-9
    - instancias múltiples 8
    - interacción de objetos 10-11
    - invocar métodos 5-6
    - parámetros 6-7, 13-15
    - tipos de datos 7-8
    - valores de retorno 13
  - opciones de configuración 448
  - operaciones de manejo de archivo 385-6
  - operador (>) mayor 39, 444
  - operador (>=) mayor o igual 444
  - operador “o” excluyente 444
  - operador “y” (&&) 59, 60, 444
  - operador de asignación compuesto 32n
  - operador distinto de (!=) 444
  - operador igualdad (==) 130, 444
  - operador instanceof 294
  - operador mas (+) 34, 61, 62, 139, 181
  - operador menor (<) 113, 444
  - operador menor o igual (<=) 444
  - operador menos (-) 181
  - operador módulo (%) 62-3
  - operador no (!) 60, 444
  - operadores 443-4
    - ! 60, 125, 444
    - != 444
    - % 62, 443
    - && 60, 444
    - \* 443
    - / 62, 443
    - ^ 444
    - || 60, 444
    - + 31, 443
    - += 3
    - < 60, 444
    - <= 444
    - = 31
    - == 444
    - > 39, 60, 444
    - >= 39, 60, 444
    - 67, 443
    - entre cadenas 130
    - lógico 60
    - new 66-7
    - uso con cadenas 62
  - operadores en cortocircuito 444
  - operadores lógicos 60
  - orden de las declaraciones 461
  - origen estándar de entradas 390
  - oyentes de eventos 319, 325
- P**
- palabra clave null 99
  - palabra clave private 148-50
  - palabra clave protected 271
  - palabra clave public 148-50
  - palabra clave static 153-4
  - palabra clave this 398
  - panel contenedor 316-7, 329, 333n, 346
  - panel de desplazamiento 351
  - paquete java.awt 316, 326
  - paquete java.awt.event 316, 319
  - paquete java.awt.image 326
  - paquete java.io 371, 385, 387, 390, 391, 465-6
  - paquete java.lang 135, 137, 366, 369, 436, 463

- paquete `java.net` 466  
paquete `java.util` 135, 136-7, 390, 409n, 464-5  
paquete `javax.imageio` 327  
paquete `javax.swing` 316  
paquete `javax.swing.border` 347  
paquetes y la sentencia `import` 136-7  
*ver también bibliotecas, y paquetes específicos especialmente bajo java*  
par de objetos 138  
parámetros 6-7, 26-7, 35, 42-3  
    actual 26, 28, 33  
    alcance 26  
    clases 135-6  
    diseñar aplicaciones 400  
    formal 26, 28, 30, 31, 42  
    interacción de objetos 67  
    listas 68  
    objetos como parámetros 13-15  
    pasaje y subtipos 250-1  
    una cadena única 34  
    tipos 248  
    valores 47, 87, 221, 361  
paréntesis 443  
parte privada de una clase *ver* implementación  
parte pública de una clase *ver* interfaz  
paso a paso 76-7  
paso único 76-7  
pasos 281  
patrón de diseño fábrica 408  
patrón de diseño observador 408-10  
patrón de diseño singleton 407  
patrón decorador 406-7  
píxel 6n  
polimorfismo 229, 260, 268  
    invocar métodos 252  
    manejo de errores 374  
    técnicas de abstracción 292, 295, 297  
    variables 251, 253, 266, 306  
poner en práctica las técnicas 187  
posicionX 9  
posicionY 9  
principio de ocultamiento de la información 148, 149, 150  
principio necesidad de conocer 149  
principio no se permite conocer 149  
procesar una colección completa 89-96  
    ciclo `for-each` 90-2  
    ciclo `while` 92-5  
comparación de acceso mediante índices e iteradores 96  
recorrer una colección 95-6  
programación defensiva 359-61  
programación extrema 402  
programación por parejas 402  
programas ejecutables 385  
prototipos 402-3, 404  
proyecto *agenda* 82, 97-8  
proyecto *agenda-diaria-prueba* 160, 166, 169, 170-1, 173  
proyecto *analizador-weblog* 106, 391  
proyecto *calculadora-motor* 175-6, 178-9, 184  
proyecto *compania-de-taxis* 413-32  
    descripción del problema 413-14  
    descubrir clases 414-15  
    escenarios 416-18  
    esquema 419, 422, 424  
    etapa de desarrollo más avanzada 429, 430-1  
    primer etapa 425  
    tarjetas CRC 415-16  
    *ver también* diseño de clases, desarrollo iterativo  
proyecto *curso-de-laboratorio* 13, 14, 15, 17, 45  
proyecto *dome-v3* 264  
proyecto *ladrillos* 187  
proyecto *libreta-de-direcciones* 355-8, 370, 372, 380, 383  
proyecto *libreta-de-direcciones- v2t* 361  
proyecto *libreta-de-direcciones- v3t* 377  
proyecto *libreta-de-direcciones-assert* 379  
proyecto *libreta-de-direcciones-io* 385-7  
proyecto *libreta-de-direcciones-junit* 381  
proyecto *libreta-de-direcciones-v1g* 358  
proyecto *libreta-de-direcciones-v1t* 358  
proyecto *libreta-de-direcciones-v2g* 361  
proyecto *maquina-de-boletos* 17-26, 28-30, 34-6, 43  
proyecto *pelotas* 150-2  
proyecto *sistema-de-correo* 71-2  
proyecto *soporte-tecnico* 119, 126, 128, 137,

- 145, 149
  - aplicación 120-6
  - comportamiento aleatorio 131
  - lectura de código 122-6
  - proyecto *soporte-tecnico-completo* 390
  - proyecto *subastas* 98-105
    - clase *Lote* 98-9
    - clase *Subasta* 99-102
    - objetos anónimos 102-3
    - usar colecciones 103-5
  - proyecto *visor-de-imagen* 314-25
    - agregar componentes simples 317-18
    - agregar menús 318-19
    - clases internas 322-3
    - clases internas anónimas 323-5
    - crear una ventana 315-17
    - manejo de eventos 319
    - recepción centralizada de eventos 319-22
  - proyecto *visor-de-imagen-0-2* 321
  - proyecto *visor-de-imagen-0-3* 323
  - proyecto *visor-de-imagen-0-4* 326
  - proyecto *visor-de-imagen-1-0* 315, 316, 328, 336, 338
    - agregar la imagen 327-8
    - clases para procesar imágenes 326-7
    - contenedores anidados 331-4
    - diálogos 337-8
    - esquemas de disposición 328-31
    - filtros de imagen 334-6
    - primera versión completa 325-38
  - proyecto *visor-de-imagen-2-0* 325, 338-43
  - proyecto *visor-de-imagen-3-0* 343-7
  - proyecto *world-of-zuul* 190-2, 197, 212, 218-19, 273, 407, 456
  - proyecto *zorros-y-conejos* 279-92, 323, 425, 429
    - clase *Conejo* 282-5
    - clase *Simulador*: configuración 288-91
    - clase *Simulador*: un paso de simulación 291-2
    - clase *Zorro* 285-8
    - diseñar aplicaciones 408
    - mejorar la simulación 292
  - proyecto *zorros-y-conejos-v1* 293
  - proyecto *zuul-con-enumeraciones* 218, 220-2
  - proyecto *zuul-mejorado* 210
  - prueba 423
    - aserciones 455
    - automatización 166-74
    - grabar una prueba 171-3
    - objetos de prueba 173-4
    - prueba de regresión 166-8
    - resultados de las pruebas 169-71
    - clases 169-71, 177, 454
    - crear un método de prueba 454
    - ejecutar pruebas 455
    - interactiva 186
    - seleccionar estrategia de prueba 186
    - objetos con buen comportamiento 159
    - y refactorización 215
    - ver también* pruebas de regresión, pruebas de unidad
  - prueba de regresión 166-8, 169, 171, 215, 378
  - prueba interactiva 186
  - pruebas de unidad
    - clases 186
    - comparar pruebas positivas con pruebas negativas 165-6
    - con BlueJ 159-66
    - herramientas 454-5
    - inspectores 164-5
    - marco de trabajo 381
  - pseudoaleatorio 131
  - pseudocódigo 39, 90-1, 95
  - puntos de interrupción 74-6, 451
- ## R
- recepción centralizada de eventos 319-22
  - recorrer una colección 95-6
  - redefinición *ver* sobrescritura
  - redefinir un método 267
  - refactorización 214-18, 293, 339, 342
    - para independizarse del idioma 218-22
  - refactorización para independizarse del idioma 218-22
  - refactorizar métodos 217
  - reglas de precedencia 443-4
  - relación *es-un* *ver* herencia
  - ReproductorDeSonido 349-52
  - restrictiones de uso del lenguaje 461-2
  - reusabilidad 213-14, 245, 431

**S**

- scanner: leer entradas desde la terminal 390-1  
 SDK 445, 456  
 sección de etiquetas 457-8  
 secuencia de llamadas (pila) 187  
 seguimiento manual 178-82  
     controlar el estado 180-2  
     de alto nivel 178-80  
     verbal 182  
 seguimientos 186, 187  
     *ver también* seguimiento manual  
 sentencia 29, 33, 41, 439  
     asignación 27-8, 32  
     if 59, 69, 94, 363, 438, 460  
     if-else 438  
     imprimir 183-6  
     incremento 441  
      inicialización 441  
      protegida 372  
     return 30, 35  
     salida 93  
     selección 438-40  
     simple 32  
     switch 439-40  
     throw 366  
     try 372-3, 376, 382, 390  
         *ver también* sentencias condicionales  
     sentencia condicional 17, 36-40, 43, 77  
     sentencia de incremento 441  
     sentencia de salida 93  
     sentencia if 59, 69, 94, 363, 438, 460  
     sentencia if-else 438  
     sentencia import 84  
     sentencia return 30, 35  
     sentencia switch 439-40  
     sentencia throw 366  
     sentencia try 372-3, 376, 382, 390  
     sentencia única 32  
     sentencias de impresión 183-6  
     sentencias de selección 438-40  
     sentencias protegidas 372  
     separación modelo/vista 350  
     setLayout 345  
     signatura 7, 13, 33, 128  
     simulación 278-9  
     lógica 308  
     simulación predador-presa 452  
         *ver también* proyecto zorros y conejos  
     sinónimos 415  
     sintaxis 106, 108  
     sistema *SoporteTecnico* 139-41, 144-5  
     sobrecarga 68  
     sobrescritura 262-4, 267, 271, 273-5  
     Software Development Kit *ver* SDK  
     solución lista única 294  
     subclase 240, 242-3, 245  
         abstracción 306  
         acceso protegido 271, 272, 273  
         búsqueda dinámica del método 266, 267  
         constructor 244  
         herencia con sobrescritura 275  
         interfaces gráficas de usuario 324, 340  
         llamada super en métodos 268  
         manejo de errores 366  
         sobrescritura 262-3, 264  
         técnicas de abstracción 292, 293, 295, 297, 298, 299, 302  
         y subtipos 248  
     subtipo 247-52, 261  
         conversión de tipos 251-2  
         interfaces gráficas de usuario 325  
         manejo de errores 374  
         variables polimórficas 251  
         y asignación 249-50  
         y pasaje de parámetros 250-1  
         y subclases 248  
     Sun Microsystems 255, 347, 385, 456  
     superclase 240, 242, 243, 245, 249, 253  
         acceso protegido 271, 272, 273  
         búsqueda dinámica del método 265-6, 267  
         constructor 244, 462  
         *Dome* 259  
         interfaces gráficas de usuario 339, 340  
         llamada super en métodos 268  
         tipo 248  
         sobrescritura 262-3, 264  
         técnicas de abstracción 293, 294, 295, 297, 298, 300, 302, 303, 306  
         tipo estático y tipo dinámico 260  
         supertipos 251, 324, 325, 374  
         sustantivos 394-5, 414-15, 459

sustitución 249  
reglas 261

## T

tarjetas CRC 394, 395-6, 415-16, 418, 422  
*TaxiPrueba* 425, 428  
técnicas de abstracción 278-311  
herencia 309  
herencia múltiple 300-3  
simulaciones 278-9  
ver también clases abstractas, proyecto  
*zorros-y-conejos*, interfaces  
y flexibilidad 302  
tiempo de ejecución *ver* vista dinámica  
tiempo de vida de una variable 27  
tipo 7  
pérdida de tipo 252  
tipo base 108  
tipo de retorno 30, 31, 35, 363, 365  
tipo dinámico 5, 260-2, 266, 267, 271, 295  
tipo estático 260-2, 266, 271, 295  
tipo void 363  
tipos de dato 7-8  
Java 435-7  
tipos enumerados 218, 219-20  
*TitledBorder* 346  
*Triangulo* 4, 9, 10

## U

UML 230n  
unboxing 254, 437  
URL clase 466  
usar colecciones 103-5  
usar mapas para las asociaciones 137-41  
concepto de mapa 138  
*HashMap* 138-9  
sistema *SoporteTecnico* 139-41  
usar patrones de diseño 405-10  
decorador 406-7

estructura de un patrón 406  
método fábrica 408  
observador 408-10  
singleton 407

## V

valor del “medio” 354  
valores 10, 26  
falso 9  
verdadero 9  
valores de retorno 13, 362, 365, 400  
valores de tipos primitivos 58, 254, 364, 435-6  
variable 24, 249, 436  
local 41-3, 49  
polimórfica *ver* variables polimórficas  
tiempo de vida 27  
variables estáticas 76  
variables locales 41-3, 76  
ventana 317  
verbos 394-5, 415  
visor de imágenes estáticas 350  
*VisorDeImagen* 325, 333, 338-9  
vista estática 56  
*Vlissides*, John 406n

## W

while ciclo 92-5, 96, 97, 112, 114, 440  
*WindowEvent* 319  
Woods, Don 190  
*Wordpad* 447

## Z

zona de Secuencia de llamadas 453  
zona de variables 452-3  
*zonna Threads* 453