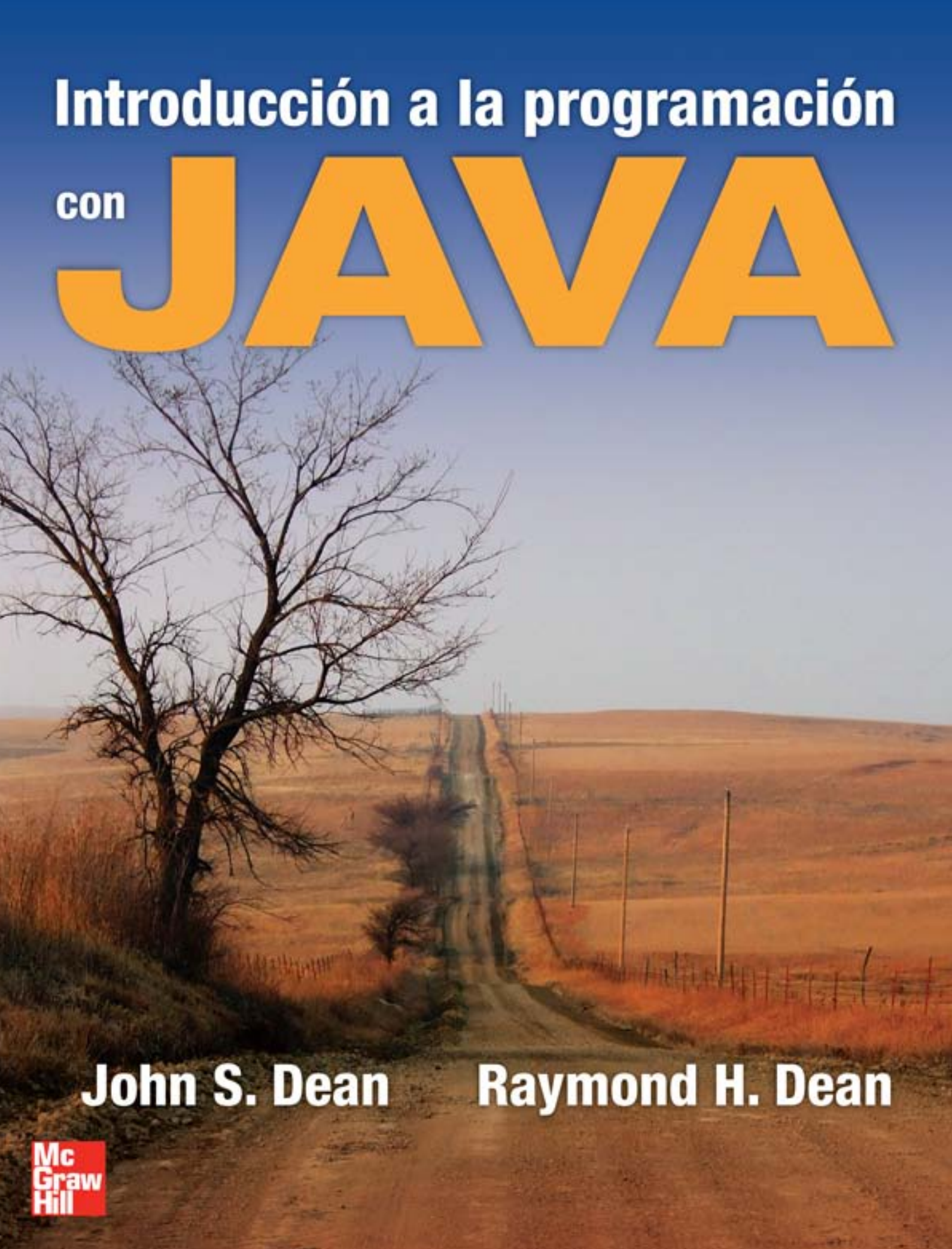
Introducción a la programación con JAVA



John S. Dean Raymond H. Dean

Mc Graw Hill

Introducción a la programación

CON

JAVA

Introducción a la programación CON JAVA

John S. Dean

Park University

Raymond H. Dean

University of Kansas

Revisión técnica:

Carlos Villegas Quezada

Universidad Iberoamericana,
Ciudad de México

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

Director Higher Education: Miguel Ángel Toledo Castellanos
Director editorial: Ricardo Alejandro del Bosque Alayón
Editor sponsor: Pablo Eduardo Roig Vázquez
Coordinadora editorial: Marcela Rocha Martínez
Editor de desarrollo: Edmundo Carlos Zúñiga Gutiérrez
Supervisor de producción: Zeferino García García
Traductor: Hugo Villagómez Velázquez

INTRODUCCIÓN A LA PROGRAMACIÓN CON JAVA
Primera edición

Prolongación Paseo de la Reforma 1015, Torre A
Pisos 16 y 17, Colonia Desarrollo Santa Fe,
Delegación Álvaro Obregón
C.P. 01376, México, D. F.
Miembro de la Cámara Nacional de la Industria Editorial Mexicana, Reg. Núm. 736

ISBN: 978-970-10-7278-3

Traducido de la primera edición de: INTRODUCTION PROGRAMMING WITH JAVA:
A PROBLEM SOLVING APPROACH

0-07-3047023

Impreso en México *Printed in Mexico*

0123456789 098765432109

Dedicatoria

—A Stacy y Sarah

Acerca de los autores

John Dean es catedrático del Departamento de Información y Ciencias de la Computación en la Universidad Park. Obtuvo un grado de maestro en ciencias de la computación en la Universidad de Kansas. Está certificado por Java Sun y ha trabajado en la industria como ingeniero de software y director de proyectos, con especialidad en Java y varias tecnologías en la Web: JavaScript, JavaServer Pages y servlets. Ha impartido un sinnúmero de cursos de ciencias de la computación, incluyendo programación con Java y programación con Java basada en la Web.

Raymond Dean es profesor emérito en ingeniería y ciencias de la computación en la Universidad de Kansas. Obtuvo su grado de maestro en ciencias en el MIT y un doctorado en la Universidad de Princeton, y es miembro decano de la IEEE. Ha publicado numerosos artículos científicos y posee 21 patentes estadounidenses. Actualmente es investigador del programa *Climate and Energy* del Land Institute, que reivindica la conservación detallada de la energía y el reemplazo del consumo de combustibles de origen fósil y nuclear por energía eólica y almacenamiento de energía eléctrica.

Contenido

Prólogo

En este libro haremos un viaje al interior del divertido y excitante mundo de la programación de computadoras. Durante este viaje, proporcionaremos diversas prácticas para la solución de problemas. Después de todo, los buenos programadores necesitan ser buenos resolviendo problemas. Mostraremos cómo implementar la solución de problemas con programas Java. Proporcionaremos una amplia gama de ejemplos, algunos cortos y enfocados a un simple concepto, algunos largos y más orientados al "mundo real". Presentamos el material en un formato convencional y fácil de seguir, con el objetivo de que nuestro viaje sea placentero. Cuando usted termine de leer este libro, será un eficiente programador en Java.

El libro se dirige a diversos tipos de lectores. En primer lugar, a los estudiantes de nivel universitario que realizan un curso de "Introducción a la programación" o un curso secuencia, en el que no se requiere experiencia previa en programación.

Además, también se dirige a personas con experiencia en la industria de la programación, a estudiantes de nivel universitario con algún tipo de experiencia en programación, y a todo aquel que necesite y desee aprender Java. Estos segundos lectores pueden omitir la lectura de los primeros capítulos concernientes a conceptos generales de programación, y dirigir su atención a las características de Java que difieren de otros lenguajes que conocen. En particular, debido a que C++ y Java son lenguajes de programación muy similares, los lectores con algo de conocimiento en C++ deben ser capaces de cubrir el libro con un simple curso de tres horas semanales; pero para aquellos sin experiencia previa en el campo de la programación, el texto, reiteramos, es adecuado, pues en él no se requiere de algún tipo de conocimiento previo.

Finalmente, el texto se dirige a estudiantes de nivel bachillerato y lectores fuera del ámbito académico sin experiencia en programación. A este tercer grupo de lectores recomendamos la lectura del libro completo, a su propio ritmo y en un nivel de caso por caso.

Objetivo general núm. 1: Resolución de problemas

La resolución de problemas es una habilidad básica que todo programador debe poseer. Se enseña la resolución de problemas haciendo énfasis en dos elementos: el desarrollo de algoritmos y el diseño de programas.

Énfasis en el desarrollo de algoritmos

En el capítulo 2, se introduce al lector en el desarrollo de algoritmos mediante el uso del seudocódigo, en lugar de usar la sintaxis del lenguaje Java. Con el uso del seudocódigo, el estudiante será capaz de trabajar con problemas no triviales aun sin conocer la sintaxis de Java (no tendrá que preocuparse de los encabezados de las clases, de los puntos y comas, de los corchetes, etc.).[1] Al trabajar en la resolución de problemas no triviales el estudiante obtiene una primera apreciación de la creatividad, la lógica y la organización. Sin esta apreciación, el estudiante tiende a aprender la sintaxis de Java mediante memorización por repetición; pero, si cuenta con ella, el estudiante aprenderá la sintaxis de Java de una manera más rápida, pues se sentirá motivado para hacerlo. Además, podrá cumplir con los ejercicios asignados de una manera bastante sencilla debido a que cuenta con experiencia previa en resolución de problemas mediante el uso del seudocódigo.

[1] Inevitablemente para el seudocódigo se usa un estilo particular, aunque de manera repetida se recalca que otros estilos de seudocódigo también son válidos en la medida en que contengan el significado idóneo. El estilo de seudocódigo que se utiliza aquí es una combinación de descripción libre para tareas de alto nivel y de comandos más específicos para tareas de bajo nivel. Para los comandos específicos, se usan palabras en inglés natural más que símbolos crípticos. Se ha escogido un estilo de seudocódigo intuitivo a fin de dar la bienvenida a nuevos programadores, y estructurado para permitir la lógica del programa.

A partir del capítulo 3, se hará uso de Java para la resolución de ejemplos de desarrollo de algoritmos; sin embargo, para la mayoría de los problemas involucrados, se hará uso de cuando en cuando del seudocódigo de alto nivel para describir soluciones de primera mano. La utilización del seudocódigo permite a los lectores olvidarse de los detalles de la sintaxis y enfocar su atención en la parte de la solución del algoritmo.

Énfasis en el diseño del programa

La solución del problema va más allá del simple desarrollo del algoritmo, implica también encontrar la mejor implementación del algoritmo. Esto es el diseño del programa. El diseño del programa es extremadamente importante, ésa es la razón de que se ocupe mucho tiempo en ello. En este libro no sólo se da una solución, también se explica el proceso para llegar a la misma. Por ejemplo, se explica cómo elegir entre las diferentes estructuras de control, cómo se divide un método en más de uno, cómo elegir la clase más apropiada, cómo elegir entre miembros de instancia y miembros de clase, y cómo determinar las relaciones haciendo uso de la herencia y la composición. Se reta al estudiante a que encuentre las implementaciones más elegantes de una tarea en particular.

Se dedica un capítulo completo al diseño de un programa (capítulo 8, Ingeniería de software). En este capítulo, se proporciona una amplia visión de las convenciones en el estilo de programación, modularización y encapsulamiento. También, se describen las diferentes estrategias alternativas de diseño (arriba-abajo, ascendente, basado en casos y mejora iterativa).

Secciones de resolución de problemas

A menudo se trata el tema de la resolución de problemas (desarrollo del algoritmo y diseño del programa) en el flujo natural de los conceptos explicados, pero también se explicará en secciones dedicadas específicamente a ello. En cada sección de resolución de problemas se presenta una situación que contiene un problema sin resolver, y en seguida la solución de éste, que intenta simular la resolución de problemas, tal como se haría en la vida real, mediante una estrategia de diseño iterativo. Se presenta una solución inmediata, se analiza ésta, y después se plantean posibles mejoras a la misma, para lo cual se utiliza el formato conversacional de prueba y error. Por ejemplo, "¿qué tipo de gestor de contenido deberíamos utilizar? Primero intentamos con el GridLayout manager; trabaja bien, pero no es el mejor. Intentemos con el BorderLayout manager". Este tono casual da confianza al estudiante al transmitirle el mensaje de una forma normal, como se esperaría en un caso en que un programador requiriera trabajar en la resolución de un problema, en diversas ocasiones, antes de encontrar la mejor solución.

Mecanismos adicionales en la resolución de problemas

Se incluyen ejemplos de resolución de problemas y consejos para la solución de los mismos en el texto (no únicamente en los capítulos 2 y 8 o en las secciones de solución de problemas). Es necesario destacar que se incluye un cuadro de resolución de problemas, con un icono y un breve tip al lado del mismo, un texto que contiene el ejemplo con la solución del problema y/o un consejo.

El libro contiene una multitud de ejemplos de programas pues se cree firmemente en la eficacia del aprendizaje mediante ejemplos. Se recomienda al lector utilizar los programas a manera de receta para resolver programas similares que se le pudieran presentar.

Objetivo general núm. 2: Fundamentos iniciales

Conceptos pospuestos que requieren sintaxis compleja

Muchos libros de texto introductorios no tratan la sintaxis compleja con la profundidad que requiere. Al utilizar una sintaxis compleja desde el principio, el estudiante adquiere el hábito de introducir el código sin entenderlo de manera adecuada o, peor aún, el de copiar y pegar el ejemplo sin entenderlo correctamente. Esto puede resultar en programas menos que ideales y en un estudiante que limita su habilidad de resolver una variedad de problemas. Por lo tanto, es preferible posponer el estudio de conceptos que requieran sintaxis compleja. Es mejor introducirlos más adelante o cuando el estudiante esté capacitado para comprenderlos en su totalidad.

Como primer ejemplo de esta filosofía, se cubren las formas simples de programación GUI (siglas en inglés de Graphical User Interface, interfaz gráfica del usuario) en los primeros capítulos, en una sección aparte, y en seguida se cubren formas de programación gráfica más complejas en capítulos más avanzados (específicamente, hasta el final del libro). Esto difiere de otros libros de texto de Java, que inician tratando ampliamente el tema de manejo de eventos con la programación GUI. En opinión de los autores, esa estrategia es un error, pues el manejo de eventos mediante programación gráfica requiere de madurez en el manejo de la programación. Como se incluye hasta el final del libro, los lectores tienen mayor capacidad para entenderla.

Ejemplos con trazado de resultado

Para escribir un código de manera eficaz, es imperativo entenderlo de manera adecuada. Se ha visto que la presentación paso a paso de los resultados de ejecución del código de un programa es una forma eficaz de asegurar su entendimiento. Así pues, en las primeras partes del libro, al introducir una nueva estructura de programación, a menudo se presenta una simulación de los resultados del mismo, de una manera meticulosa. La técnica de trazado que se utiliza en el libro, ilustra el proceso completo que los programadores emplean al llevar a cabo la depuración. Es una alternativa a la secuencia de pantallas que serían generadas por los depuradores en un software IDE (siglas en inglés de Integrated Development Environment, ambiente de desarrollo integrado).

Entrada y salida

En las secciones opcionales de tips en GUI y en los capítulos sobre el mismo tema al final del libro, se utilizan comandos GUI para entrada y salida (E/S), pero debido al énfasis en los fundamentos, se utilizan comandos de consola, para E/S en el resto del libro.[2] Para entrada por consola, se utiliza la clase Scanner. Para salida por consola, se utilizan los métodos estándar System.out.print y System.out.printf.

Objetivo general núm. 3: Mundo real

Muy a menudo los estudiantes en los salones de clase de hoy en día y los practicantes de la industria prefieren aprender mediante la práctica, enfocando su atención al mundo real. Para satisfacer esta necesidad, este libro incluye:

- Herramientas de compilación.
- Ejemplos de programas completos.
- Guía práctica en diseño de programa.
- Guía de estilo de codificación con base en los estándares de la industria.
- Notación UML para diagramas de relación de clases.
- Asignaciones de tareas de proyectos prácticos.

Herramientas de compilación

No es necesaria una herramienta de compilación específica, el lector puede escoger la herramienta de compilación que prefiera. Si no tiene preferencia por algún compilador en específico, podría utilizar alguno de éstos:

- Kit Java2 SDK de Sun.
- TextPad de Helios.
- Eclipse, de Eclipse Foundation.
- NetBeans, respaldado por Sun.
- BlueJ, de las Universidades de Kent y Decaen.

[2] El material relacionado con GUI de E/S se cubre pronto con la clase `JOptionPane`. De este modo se abre una puerta opcional para los aficionados a GUI. Si los lectores así lo desean, pueden usar `JOptionPane` para implementar todos sus programas con GUI de E/S en vez de hacerlo con la consola de E/S. Para llevar a cabo lo anterior, es necesario sustituir todas las llamadas por medio de la consola al método E/S con llamadas al método `JOptionPane`.

Para obtener los compiladores completos, el lector puede visitar el sitio Web del libro:[i] http://www.mhhe.com/dean, buscar los vínculos apropiados de compilador(es) y bajarlos gratuitamente.

Ejemplos de programas completos

Además de proporcionar fragmentos de código para ilustrar conceptos específicos, el libro contiene varios ejemplos de programas completos. Con programas completos, el estudiante será capaz: 1) de ver cómo el código analizado se enlaza con el resto del programa y 2) probar el código, corriéndolo.

Convenciones de estilo de codificación

En el libro también se incluyen tips de codificación. El estilo de los tips se basa en las convenciones de codificación de Sun (http://java.sun.com/docs/codeconv/) y en prácticas de la industria. En el apéndice 5, se proporciona una referencia completa a las convenciones de estilo de codificación del código y un ejemplo asociado de un programa que ilustra dichas convenciones.

Notación UML

El lenguaje de modelado universal (UML, por sus siglas en inglés) se ha convertido en el estándar para describir las entidades en grandes proyectos de software. En lugar de abrumar a los programadores principiantes con la sintaxis completa del UML completo (el cual es demasiado extenso), se presenta un subconjunto del UML. En el libro, se incluye notación UML para representar gráficamente clases y relaciones entre clases. Para aquellos interesados en más detalles, en el apéndice 7 se proporciona notación UML adicional.

Problemas de tarea

Se proporcionan problemas de tarea que son ilustrativos y prácticos, y se explican de manera muy clara. Los problemas tienen un rango de dificultad que va de sencillos a retadores, y están agrupados en tres categorías: preguntas de revisión, ejercicios, y proyectos. Se incluyen preguntas de revisión y ejercicios al final de cada capítulo, y se proporcionan proyectos en el sitio Web del libro.

Las preguntas de revisión por lo regular tienen respuestas cortas y respuestas que están en el libro. Los formatos de las preguntas son: de respuesta corta, opción múltiple, cierto/falso, llenar los espacios en blanco, transcripción de programa, depuración, escribir un fragmento de código. Cada pregunta de revisión se basa en una pequeña parte del capítulo.

Los ejercicios por lo regular tienen respuestas cortas a moderadamente largas, estas respuestas no se encuentran en el libro. En los ejercicios se utilizan estos formatos: respuesta corta, transcripción, depuración, y escritura de un fragmento de código. Los ejercicios son la clave del prerrequisito más importante de la sección numerada del capítulo, pero a menudo integran conceptos de varias partes del capítulo.

Los proyectos consisten en descripciones del problema cuyas soluciones son programas completos. Las soluciones a los proyectos no se incluyen en el libro. En estos proyectos se requiere que el estudiante emplee su creatividad y sus habilidades en la solución de problemas y aplique lo que ha aprendido en el capítulo. Estos proyectos a menudo incluyen partes opcionales, que significan retos para los estudiantes más talentosos. Los proyectos son la clave del prerrequisito más importante del número de sección en el capítulo, pero, por lo regular, integran conceptos provenientes de diversas partes del capítulo.

Una característica relevante y especial de esta obra es la forma en que se especifican los problemas del proyecto. Las "sesiones ejemplo" muestran la salida generada para un conjunto de valores de entrada. Estas sesiones de ejemplo incluyen entradas que representan situaciones típicas y a menudo también extremas en situaciones límite.

Proyectos del área académica

Para hacer más atractivos los proyectos y mostrar cómo las técnicas de programación del capítulo en curso se pudieran aplicar a áreas de interés diferente, se toman contenidos de diferentes áreas académicas:

[i] Es importante mencionar que el sitio Web del libro se encuentra totalmente en inglés: tutoriales, proyectos propuestos y tareas, presentaciones y otros temas, por lo cual en este libro, al hablar de alguna sección en el sitio Web, se presenta el título en inglés, tal y como lo encontrará el lector.

- Ciencias de la computación y métodos numéricos.
- Administración y contabilidad.
- Ciencias sociales y Estadística.
- Matemáticas y Física.
- Ingeniería y Arquitectura.
- Biología y Ecología.

Los proyectos de área académica no requieren un conocimiento previo del área en particular. Así, el instructor puede asignar libremente cualquiera de los proyectos a cualquiera de sus estudiantes. Para ofrecer al lector general un conocimiento suficientemente especializado con el que pueda trabajar en un problema de un área académica en particular, en ocasiones se amplía el planteamiento del problema para explicar algunos conceptos especiales en esa área académica.

En la mayoría de los proyectos de área académica no se requiere que los estudiantes hayan completado los proyectos de los capítulos previos; esto es, los proyectos no se construyen uno al otro. Así pues, el instructor puede asignar libremente los proyectos sin preocuparse por los prerrequisitos de los proyectos. En algunos casos, un proyecto repite un proyecto del capítulo previo con un enfoque diferente. El profesor puede usar esta repetición para ejemplificar la variedad de alternativas, pero no es necesario.

Las asignaciones de proyecto se pueden adaptar a las necesidades del lector. Por ejemplo:

- Para los lectores fuera de la academia
  (aquellos cuyos proyectos concuerdan con sus intereses).
- Cuando un curso tiene estudiantes de un área académica específica
  (el instructor puede asignar proyectos del área académica relevante).
- Cuando un curso tiene estudiantes con diferentes perfiles
  (el instructor puede solicitar a los estudiantes que elijan proyectos de su propia área académica, o bien puede ignorar los límites del área académica y simplemente asignar los proyectos que llamen más su atención).

Para ayudar al lector a decidir en qué proyectos trabajar, se incluye en este libro una sección de "Resumen de proyectos" después del prólogo; en éste se enlistan todos los proyectos por capítulo, y para cada uno se especifica:

- La sección asociada dentro del capítulo.
- El área académica.
- Su longitud y dificultad.
- Una breve descripción.

Después de utilizar la sección del "Resumen de proyectos" para obtener una idea acerca de en qué proyectos podría gustarle al lector trabajar, es conveniente consultar el sitio Web del libro para obtener descripciones completas del mismo.

Organización

En este libro, los autores guían a los lectores en tres importantes metodologías de programación: programación estructurada, programación orientada a objetos (POO), y programación orientada a eventos. Para cubrir la parte de programación estructurada, se introducen conceptos básicos tales como variables y operadores, sentencias condicionales, y sentencias de control. Para cubrir el tema de POO, primero se muestra a los lectores cómo llamar a los métodos precompilados de las API (siglas en inglés de Application Program Interface, bibliotecas de interfaces de programación aplicada) de Sun. Después, se introducen conceptos de POO, tales como clase, objeto, variable de instancia y métodos de instancia. Los capítulos sobre manejo de excepciones y archivos son una transición a la programación de la interfaz gráfica del usuario (GUI). Se cubre de manera importante el manejo de eventos en programación GUI en los dos últimos capítulos.

El contenido y secuencia que aquí se promueven hacen capaz al estudiante de desarrollar sus habilidades desde una base de programación fundamental. Para lograr este primer enfoque fundamental, el libro inicia con un mínimo de conceptos y detalles. Después, desarrolla los conceptos de manera gradual y

añade detalles posteriores. Se ha evitado sobrecargar los primeros capítulos difiriendo ciertos detalles menos importantes hasta los últimos capítulos.

Apartado GUI

Muchos programadores encuentran divertida la programación GUI. Como tal, la programación GUI puede ser una herramienta motivacional para mantener el interés y compromiso del lector. Ésa es la razón de que se incluyan secciones gráficas a lo largo de la obra, desde el primer capítulo. A estas secciones se les llama "Apartado GUI". Los lectores que no tengan tiempo para introducirse en el apartado GUI, no tienen ningún problema; cualquier sección del apartado GUI puede omitirse, ya que dichos temas son independientes del material que se trata más adelante.

Capítulo 1

El capítulo 1 inicia con una explicación de los términos básicos de computación: cuáles son los componentes del hardware, qué es el código fuente, qué es el código objeto, etc. También en este capítulo se describe el lenguaje de programación que se utilizará en el resto del libro: Java. Finalmente, se proporciona al estudiante una rápida visión del clásico programa "Hola mundo", fundamental en el inicio de un aprendizaje de cualquier lenguaje de programación. Se explica cómo crear y correr un programa utilizando software minimalista: el bloc de notas de Microsoft y las herramientas del kit de desarrollo de línea de comandos de Sun (SDK, por sus siglas en inglés).

Capítulo 2

En el capítulo 2 se presentan técnicas de desarrollo en la resolución de problemas con énfasis en el diseño de algoritmos. En la implementación de soluciones algorítmicas se utilizan herramientas genéricas (diagramas de flujo y seudocódigo), dando mayor peso al seudocódigo. Como parte de la explicación del diseño de algoritmos, se describen técnicas de programación estructurada. A fin de que el estudiante aprecie los detalles de la semántica, se muestra cómo trazar los algoritmos.

Capítulos 3-5

Se presentan técnicas de programación estructurada mediante el uso de Java. En el capítulo 3 se describen las bases de la programación secuencial: variables, entradas/salidas, sentencias de asignación y llamadas a métodos sencillos. En el capítulo 4 se describe el flujo de programación no secuencial: sentencia if, sentencia switch, y sentencias de control. En el capítulo 5 se explican los métodos con más detalle y se explica al lector cómo utilizar métodos preconstruidos en la biblioteca de API de Java. En los tres capítulos, se enseña diseño de algoritmos mediante la resolución de problemas y la escritura de programas con la nueva sintaxis de Java introducida.

Capítulos 6-8

En el capítulo 6 se introducen los elementos básicos de la POO en Java. Esto incluye la implementación de clases, y la implementación de métodos y variables dentro de estas clases. Se utilizan diagramas de clase de UML y técnicas de control de flujos en programación orientada a objetos para ilustrar dichos conceptos.

En el capítulo 7 se proporcionan detalles adicionales de la programación orientada a objetos. Se explica cómo las variables referenciadas son asignadas, cómo se comprueba su igualdad y cómo pasar argumentos a un método. Se cubre la sobrecarga de métodos y constructores.

Mientras que el arte del diseño de programas y la resolución de problemas de ciencias de la computación se desarrollan a lo largo del libro, en el capítulo 8 se enfocan estos aspectos en el contexto de la POO. El capítulo inicia con un tratamiento organizado al estilo de programación. Se describe la mayoría de los paradigmas de programación: diseño arriba-abajo, diseño ascendente, utilización de software preconstruido para módulos de bajo nivel, y uso de prototipos.

Capítulo 9

Algunos libros de Java enseñan cómo implementar miembros de clase antes de enseñar cómo implementar miembros de instancia. Con ese método, el estudiante aprende a escribir miembros de clase de ma-

nera inapropiada, y esa práctica es difícil de romper después, cuando finalmente el tema de los métodos de instancia se ha cubierto. La práctica de programación propiamente dicha, señala que los practicantes (incluidos los programadores novatos) deberían implementar miembros de instancia en mayor cantidad que miembros de clase. Así pues, se enseña cómo implementar miembros de instancia desde un principio y se pospone la implementación de miembros de clases hasta el capítulo 9.

Capítulo 10

En el capítulo 10 se describen las diferentes formas de almacenar datos relacionados. Se presentan los fundamentos de arreglos y muchas aplicaciones de importancia hechas con arreglos: búsqueda, ordenamiento, y construcción de histogramas. Se presentan conceptos más avanzados de arreglos, como la utilización de arreglos de dos dimensiones y arreglos de objetos. Finalmente, se habla de una forma más poderosa de arreglos: la lista de arreglos (ArrayList).

Capítulo 11

Desde un principio, el estudiante debe involucrarse en actividades relacionadas con la resolución de problemas. Cubrir con minuciosidad los detalles de la sintaxis puede distraernos de este objetivo. Por lo tanto, se pasarán por alto algunos detalles de importancia menor sobre la sintaxis y se regresará a ellos en el capítulo 11. En el capítulo 11 se proporcionan más detalles de temas tales como:

- Los tipos primitivos byte y short.
- El conjunto de caracteres Unicode.
- Conversión de tipos.
- Modo posfijo *vs*. prefijo para los operadores de incremento y decremento.
- El operador condicional.
- Evaluación en corto circuito.

Capítulo 12-13

En los capítulos 12 y 13 se describen las relaciones entre clases. Se dedican dos capítulos completos a las relaciones de clases, ya que este subtema es de gran importancia. Se toma el tiempo para explicar los detalles de la relación de clases a profundidad y se proporcionan numerosos ejemplos. En el capítulo 12 se habla de los temas de agregación, composición y herencia. En el capítulo 13 se tratan con más detalle aspectos de la herencia, como la clase de objeto, el polimorfismo, clases abstractas, e interfaces.

Capítulos 14-15

Se cubre el manejo de excepciones en el capítulo 14 y el de archivos en el 15. Se habla del manejo de excepciones antes que del de archivos debido a que el código de manejo de archivos hace uso del de manejo de excepciones; por ejemplo, el abrir un archivo requiere que se realice la verificación de alguna excepción.

Capítulos 16-17

Se trata el tema del manejo de eventos con programación GUI al final del libro en los capítulos 16 y 17. Al aprender sobre el manejo de eventos mediante la programación GUI en una etapa posterior, el estudiante es capaz de aprovechar mejor sus complejidades inherentes.

Apéndices

La mayoría de los apéndices cubren material de referencia, tales como la tabla de caracteres ASCII y la tabla de precedencia de operadores; sin embargo, en los últimos dos apéndices se cubre material avanzado de Java: recursión y multitareas.

Tema de dependencia y oportunidades de cambio en la secuencia

El material se ha dispuesto en un orden natural para alguien que desee introducirse en los fundamentos primero y que también quiera una pronta introducción a la POO. Se considera que el orden es el más ade-

cuado y eficaz para aprender cómo llegar a ser un eficiente programador en la POO. Sin embargo, se ha observado que los lectores tienen diferentes preferencias en cuanto al orden del contenido. Para ajustarse a estas diferencias, se proporciona algo de flexibilidad en la construcción. La figura 0.1 ilustra dicha flexibilidad al mostrar la dependencia entre capítulos y, aún más importante, la no dependencia de capítulos. Por ejemplo, la flecha entre los capítulos 3 y 4 significa que el capítulo 3 debe ser leído antes del capítulo 4. Y la ausencia de flecha entre los capítulos 1 y 2 significa que la lectura del capítulo 1 se puede omitir.

A continuación se presentan algunas oportunidades de cambio en la secuencia mostradas en la figura 0.1:

- El lector puede omitir el capítulo 1 (Introducción a las computadoras y programación).
- Para una pronta introducción a la POO, el lector puede leer la sección de introducción a la POO en el capítulo 6 después de haber leído el capítulo 1. Y puede aprender sobre sintaxis y semántica en la POO en el capítulo 7 después de haber finalizado con los fundamentos de Java en el capítulo 3.
- Para una práctica adicional de las secuencias de control, el lector puede aprender sobre arreglos en el capítulo 10 después de haber finalizado la lectura de las secuencias de control en el capítulo 4.
- El lector puede omitir el capítulo 15 (archivos).

Nota. En la figura 0.1, la flecha discontinua conecta el capítulo 3 con el 15. Se utiliza una flecha con líneas discontinuas para indicar que la conexión es parcial. Algunos lectores quizá quieran utilizar archivos desde una etapa temprana para entrada y salida (E/S). A estos lectores se les recomienda leer el capítulo 3 sobre los fundamentos de Java e inmediatamente pasar al capítulo 15, sección 15.3 y 15.4 para E/S en archivos de texto. Con un poco de esfuerzo, serán capaces de utilizar archivos para sus necesidades de E/S en el resto del libro. Se dice "con un poco de esfuerzo" debido que las secciones de E/S contienen algo de código que no será completamente entendible para alguien que viene del capítulo 3. Para utilizar el código de E/S de archivos de texto, deberán tratarlo como a una plantilla. En otras palabras, utilizarán el código a pesar de que sean incapaces de entenderlo.

Para apoyar la flexibilidad del orden de contenido, el libro contiene "hipervínculos". Un hipervínculo es un salto adelantado de un lugar del libro a otro distinto. Los saltos son válidos en términos de conocimiento de prerrequisito, lo que significa que el salto adelantado de material no es necesario para entender el material posterior. Se proporcionan los hipervínculos para cada una de las flechas no secuenciales en la figura 0.1. Por ejemplo, se proporcionan hipervínculos que van del capítulo 1 al capítulo 6 y del capítulo 3 al 11. Para cada hipervínculo hacia el final (en los primeros capítulos), se le indica al lector hacia dónde podría saltar. Para cada hipervínculo hacia el final (en los últimos capítulos), se proporciona un icono del lado del texto meta que ayuda al lector a encontrar el lugar donde comenzó a leer.

Pedagogía

Iconos

Elegancia de programa.
Indica que el texto asociado se relaciona con estilo de codificación del programa, legibilidad, mantenimiento, robustez y escalabilidad. La elegancia en un programa comprende estas cualidades.

Resolución de problema.
Indica que el texto asociado tiene que ver con temas de resolución de problemas. Los comentarios asociados con el icono intentan generalizar el material realzado en el texto adyacente.

Errores comunes.
Indica que el texto asociado se relaciona con errores comunes.

Hipervínculo meta.
Indica la meta final del hipervínculo.

Eficiencia de programa.
Indica que el texto asociado se refiere a problemas de eficiencia en el programa.

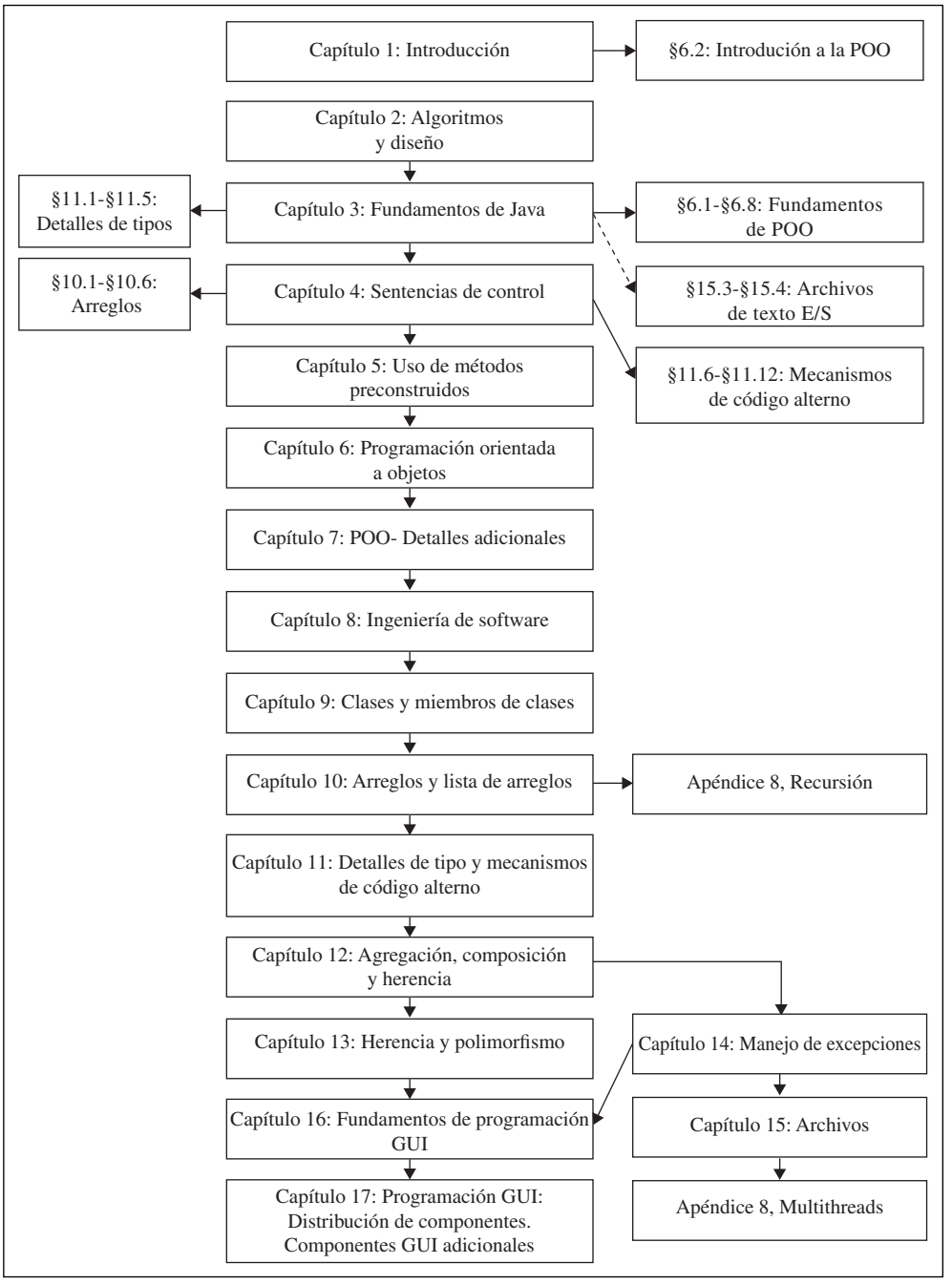


Figura 0.1 Dependencia de capítulos

Materiales de apoyo

Esta obra cuenta con valiosos complementos para fortalecer el proceso enseñanza-aprendizaje. Para mayor información y políticas al respecto, contacte a su representante McGraw-Hill.

Agradecimientos

Cualquiera que haya escrito un libro puede atestiguar el esfuerzo largo y bien orquestado que se requiere para hacerlo. Dicho libro nunca puede ser el trabajo de una sola persona o de unas pocas. Los autores es-

tamos en deuda con los integrantes del equipo de McGraw- Hill Higher Education, quienes siempre se mostraron confiados en su escritura, e invirtieron generosidad en ello.

Fue un placer haber trabajado con Alan Apt durante el periodo de revisión de dos años, quien proporcionó una guía excelente en el diseño en varios largos problemas de diseño. También estamos agradecidos por el esfuerzo incansable de Rebecca Olson. Rebecca realizó un tremendo trabajo de organización y de análisis del libro y de muchas revisiones del mismo. El auxilio durante las varias etapas de producción estuvo a cargo del administrador de proyectos Kay Brimeyer, y del diseñador Laurie Cansen. También deseamos expresar nuestro agradecimiento al resto del equipo editorial y de mercadotecnia, quienes ayudaron en las etapas finales: Raghu Srinivasan, publicista global; Kristine Tibbetts, director de desarrollo; Heidi Newsom, asistente editorial; y Michael Weitz, director ejecutivo de mercadotecnia.

Todos los profesionales hemos encontrado, de manera absoluta, que la organización McGraw-Hill ha sido maravillosa para trabajar y apreciamos sus esfuerzos.

Queremos reconocer con aprecio los numerosos y valiosos comentarios, sugerencias y críticas constructivas y los elogios de muchos instructores que han revisado el libro. En particular:

William Allen, *Instituto de Tecnología de Florida*.
Robert Burton, *Universidad Brigham Young.*
Priscila Dodds, *Colegio Georgia Perimeter.*
Jeanne M. Douglas, *Universidad de Vermont.*
Dr. H.E. Dunsmore, *Universidad Purdue*.
Deena Engel, *Universidad de Nueva York*.
Michael N. Huhns, *Universidad de Carolina del Sur.*
Ibrahim Imam, *Universidad de Louisville*.
Andree Jacobson, *Universidad de Nuevo México.*
Lawrence King, *Universidad de Texas, Dallas.*
Mark Llewellyn, *Universidad Central de Florida*.
Blayne E. Mayfield, *Universidad estatal de Oklahoma*.
Mary McCollam, *Universidad de Queens.*
Hugh McGuire, *Universidad estatal de Grand Valley.*
Jeanne Milostan, *Universidad Vanderbilt.*
Shyamal Mitra, *Universidad de Texas, Austin*.
Benjamin B. Nystuen, *Universidad de Colorado, Colorado Springs*.
Richard E. Pattis, *Universidad Carnegie Mellon*.
Tom Stokke, *Universidad de Dakota del Norte.*
Ronald Taylor, *Universidad estatal de Wright.*
Timothy A. Terrill, *Universidad de Buffalo, Universidad del estado de Nueva York*.
Ping Wu, *Dell Inc.*

También deseamos agradecer a nuestros colegas Wen Hsin, Kevin Burger, John Cigas, Bob Cotter, Alice Capson, y Mark Adams por su ayuda en los cuestionarios informales, y a Barbara Kushan, Ed Tankins, Mark Reith y Benny Phillips por sus pruebas en las clases. Y un agradecimiento muy especial al gramático sin igual Jeff Glauner, quien auxilió en los matices sutiles de la sintaxis en el idioma inglés.

Finalmente, agradecemos a los estudiantes, a aquellos que motivaron la escritura del libro, y a aquellos que nos brindaron retroalimentación y una búsqueda diligente de errores que impidieron a otros estudiantes ganar puntos buenos en las tareas. En particular, a ustedes Aris Czamaske, Malalai Zalmai, Paul John, Joby John, Matt Thebo, Josh McKinzie, Carol Liberty, Adeeb Jarrah, y Virginia Maikweki.

Sinceramente,
John y Ray

Resumen del proyecto

Una de las características de este libro son sus proyectos. Los temas de los proyectos abarcan seis amplias áreas académicas, como se resume a continuación:

Abreviatura	Descripción	Fácil	Moderado	Difícil	Total
CC	Ciencias de la computación y métodos numéricos	14	12	6	32
Administración	Administración y contabilidad	10	10	3	23
Sociología	Ciencias Sociales y Estadística	7	7	5	19
Mat. & Fís.	Matemáticas y Física	9	5	3	17
Ingeniería	Ingeniería y Arquitectura	3	7	5	15
Biol. & Ecol.	Biología y Ecología	0	2	4	6
	Total	**43**	**43**	**26**	**112**

La abreviatura en la primera columna será la que se utilice en una larga tabla que se presenta más adelante a manera de una breve identificación de un área académica en particular. Las cuatro columnas de la derecha en la tabla superior indican el número de proyectos en diversas categorías. Por supuesto, el número más alto de proyectos (32) ocurre en el área de Ciencias de la computación y métodos numéricos. Los 26 proyectos catalogados como fáciles y moderados en el área de CC son problemas de introducción a la programación. Los seis proyectos catalogados como difíciles proporcionan una discreta introducción a algunos temas avanzados, como operaciones con listas ligadas, operaciones con bases de datos y un templado simulado.

Además, existen 23 proyectos en el área de administración y contabilidad que incluyen cálculos financieros mezclados, sencillos problemas de contabilidad y aplicaciones de contabilidad de costos. Existen 19 proyectos en las áreas de Ciencias sociales, que incluyen aplicaciones en Sociología y Ciencias políticas, así como experiencia general. Hay 17 proyectos de Física y Matemáticas, que incluyen aplicaciones tanto en mecánica clásica como en caótica. 15 proyectos en Ingeniería y Arquitectura, que incluyen aplicaciones en calefacción y aire acondicionado (HVAC, por sus siglas en inglés), circuitos eléctricos y estructuras. Finalmente, seis proyectos en Biología y Ecología, que incluyen simulaciones realistas de crecimiento y presa-predador; aunque se ha asociado cada proyecto con un área académica, muchos de estos proyectos pueden pertenecer a otra distinta.

Debido a que muchos proyectos pertenecen a disciplinas fuera del área de Ciencias de la computación, no se espera que el lector promedio tenga conocimiento de estos "otros" temas. Por lo tanto, el planteamiento de los problemas se extenderá para dar explicación adicional sobre el tema y el problema; y a menudo se explicará cómo ir resolviendo el problema (en términos tutoriales). Por consiguiente, trabajar en muchos de estos proyectos será como implementar soluciones computacionales para los clientes que no son en sí programadores, pero que entienden su problema y saben lo que quieren que el programador haga por ellos. Se le explicará el problema y cómo debe resolverse, pero se esperará que sea el programador quien codifique el programa que lo resuelva.

Debido a que las explicaciones de los problemas frecuentemente toman mucho espacio de impresión, en lugar de presentarlos en el libro mismo, se pusieron en un sitio Web:

http://www.mhhe.com/dean

La siguiente tabla proporciona un resumen de lo que se tiene en el sitio Web. Esta tabla enlista todos los proyectos en una secuencia que concuerda con la secuencia del libro. La primera columna identifica el primer punto en el libro en el que el lector debe ser capaz de realizar el proyecto por capítulo y sección, en el formato: NúmerodeCapítulo.NúmerodeSección. La segunda columna es un número de proyecto único para el capítulo en cuestión. La tercera columna identifica el área académica primaria del proyecto con una abreviatura que se explica en la tabla que se presenta más arriba. La cuarta columna indica el número aproximado de páginas de código que contiene la solución del libro. La quinta columna indica la

dificultad con relación al nivel de estudio en el cual se encuentra el estudiante. Por ejemplo, se puede ver que lo que se denomina "fácil" implica más páginas de código mientras se avanza en la lectura del libro. Las últimas dos columnas proporcionan un título y una breve descripción del proyecto.[ii]

Resumen del proyecto

Cap./ Secc.	Proyecto	Área académica	Páginas soluc.	Dificultad	Título	Breve descripción
2.7	1	Administración	0.6	Fácil	Bono anual (diagrama de flujo)	Dibujar un diagrama de flujo que calcule un bono anual.
2.7	2	Administración	0.3	Fácil	Bono anual (seudocódigo)	Escribir en seudocódigo un algoritmo que calcule un bono anual.
2.7	3	Administración	0.6	Fácil	Número de timbres (diagrama de flujo)	Dibujar un diagrama de flujo para un algoritmo que calcule el número de timbres necesarios para un sobre. Utilizar un timbre por cada 5 hojas de papel.
2.7	4	Administración	0.4	Fácil	Número de timbres (seudocódigo)	Escribir en seudocódigo un algoritmo que calcule el número de timbres necesarios para un sobre. Utilizar un timbre por cada cinco hojas de papel.
2.7	5	Biol. & Ecol.	0.4	Fácil	Cinco reinos (seudocódigo)	Escribir en seudocódigo un algoritmo que identifique un reino biológico a partir de una serie de características.
2.7	6	Mat. & Fís.	0.4	Fácil	Velocidad del sonido (diagrama de flujo)	Dibujar un diagrama de flujo para un algoritmo que calcule la velocidad del sonido en un medio particular.
2.7	7	Mat. & Fís.	0.4	Fácil	Velocidad del sonido (seudocódigo)	Escribir en seudocódigo un algoritmo que calcule la velocidad del sonido en un medio particular.
2.7	8	Administración	0.6	Moderado	Rendimiento del mercado de valores (diagrama de flujo)	Dibujar un diagrama de flujo para un algoritmo que imprima el tipo de mercado y su probabilidad, dada una tasa de rendimiento en particular.
2.7	9	Administración	0.4	Moderado	Rendimiento del mercado de valores (seudocódigo)	Escribir en seudocódigo un algoritmo que imprima el tipo de mercado y su probabilidad dada una tasa de rendimiento en particular.
2.7	10	Administración	0.3	Moderado	Estado de cuenta bancario (seudocódigo)	Escribir en seudocódigo un algoritmo que determine el número de años que transcurren hasta que el balance en un estado de cuenta alcanza el millón de dólares.
2.7	11	Ingeniería	0.3	Moderado	Terminación de ciclo por petición del usuario (diagrama de flujo)	Escribir en seudocódigo un algoritmo que calcule el número de millas por galón para una serie de entradas de millas y galones por parte del usuario.

[ii] En la presente tabla se presentan las generalidades de los proyectos: título, descripción, etcétera. Sin embargo, al consultar el sitio Web, el lector encontrará los textos de dichos proyectos en idioma inglés.

Cap./ Secc.	Proyecto	Área académica	Páginas soluc.	Dificultad	Título	Breve descripción
2.9	12	Ingeniería	0.5	Fácil	Terminación de ciclo por petición del usuario (seudocódigo)	Escribir en seudocódigo un algoritmo que calcule el número de millas por galón para una serie de entradas de millas y galones por parte del usuario.
2.9	13	Ingeniería	0.4	Moderado	Terminación de ciclo por valor de un Sentinal (seudocódigo)	Escribir en seudocódigo un algoritmo que calcule la cantidad de millas por galón para una serie de entradas de millas y galones por parte del usuario.
2.9	14	Ingeniería	0.3	Fácil	Terminación de ciclo por conteo (seudocódigo)	Escribir en seudocódigo un algoritmo que calcule la cantidad de millas y galones introducidas por parte del usuario.
2.10	15	CC	0.4	Moderado	Peso promedio (seudocódigo)	Escribir en seudocódigo un algoritmo que determine el peso promedio de un conjunto de objetos.
3.2	1	CC	N.A.	Fácil	Experimentación Hola Mundo	Experimentar con el programa Hola Mundo para entender los significados de los típicos mensajes de error de compilación y de ejecución.
3.3	2	CC	N.A.	Fácil	Investigación	Estudiar las convenciones de código de Java.
3.3	3	CC	N.A.	Moderado	Investigación	Estudiar el apéndice 5 “Convenciones de estilo de Java”.
3.16 3.23	4	Ingeniería	2.5	Difícil	Análisis de armadura	Dada la carga en el centro de un puente y el peso de todos los miembros, calcular la fuerza de compresión o tensión en cada miembro.
3.17	5	CC	1.0	Fácil	Secuencia de comandos	Establecer la secuencia de comandos y escribir un programa que ejecute dichos comandos.
3.17 3.23	6	CC	1.7	Moderado	Cálculo de la velocidad	Dado un conjunto de características de hardware y software, escribir un programa que determine el número de veces que corre un programa de cómputo.
3.17 3.23	7	Ingeniería	2.7	Moderado	Carga HVAC	Calcular las cargas de calefacción y aire acondicionado para una residencia tradicional.
3.17 3.23	8	Sociología	3.25	Difícil	Planeación de campaña	Escribir un programa que ayude a organizar los estimados de votos, dinero y trabajo.
3.17 3.23	9	Ingeniería	2.7	Fácil	Procesamiento de cadena de texto	Trazar un conjunto de operaciones de procesamiento de cadenas de texto y escribir un programa que las implemente.
3.23	10	CC	1.2	Fácil	Traspaso	Desarrollar un algoritmo que traspase el valor en dos variables y escribir un programa que implemente dicho algoritmo.

Cap./ Secc.	Proyecto	Área académica	Páginas soluc.	Dificultad	Título	Breve descripción
3.23	11	Mat. & Fís.	1.0	Fácil	Parámetros de un círculo	Escribir un programa que genere e imprima los valores relacionados con un círculo.
3.23	12	Sociología	0.4	Fácil	Cumpleaños número cien	Escribir un programa que solicite al usuario introducir el día, mes y año de su nacimiento e imprima la fecha de su cumpleaños número 100.
4.3	1	Mat. & Fís.	1.7	Fácil	Distancia de detenido	Escribir un programa que determine si la distancia que lleva un vehículo es segura, dada la velocidad del mismo y una fórmula que proporcione la distancia requerida para que éste frene.
4.3 4.9	2	Ingeniería	1.9	Fácil	Seguridad de columnas.	Escribir un programa que determine si una columna estructural es suficientemente ancha para soportar la carga total esperada de la columna.
4.3	3	Administración	1.1	Fácil	Política económica	Escribir un programa que lea los valores de tasa de crecimiento e inflación y despliegue la política económica recomendada.
4.8	4	Administración	2.0	Moderado	Estado de cuenta bancario	Escribir un programa que determine el número de años que tienen que pasar hasta que el saldo en una cuenta bancaria alcance el millón de dólares.
4.9 4.12	5	CC	2.6	Difícil	Juego de NIM	Implementar el juego NIM. Iniciar el juego con un número de piedras en una pila especificado por el usuario. El usuario y la computadora tendrán su turno para quitar una o dos piedras de la pila. El jugador que tome la última piedra pierde.
4.12	6	Mat. & Fís.	1.0	Fácil	Triángulo	Escribir un programa que genere un triángulo isósceles hecho con asteriscos, dado el tamaño del triángulo introducido por el usuario.
4.12	7	Sociología	0.8	Fácil	Calendario maya	Implementar un algoritmo que determine el número de Tzolkins y de Haabs en un calendario redondo.
4.12	8	CC	0.9	Fácil	Validación de entradas	Implementar un algoritmo que repetidamente solicite valores que caigan dentro de un rango y calcule el promedio de entradas.

Cap./ Secc.	Proyecto	Área académica	Páginas soluc.	Dificultad	Título	Breve descripción
4.14	9	Administración	2.6	Moderado	Preparación de impuestos	Escribir un programa que calcule los impuestos sobre el ingreso, haciendo uso de las siguientes reglas: La cantidad de impuestos debidos iguala el ingreso gravable las veces de la tasa de impuestos. El ingreso gravable iguala al ingreso bruto menos $1 000 para cada exención. El impuesto sobre ingreso no puede ser menor a 0.
4.14	10	CC	1.7	Moderado	Análisis de texto	Escribir un programa que convierta palabras a Pig Latin.[1]
5.3	1	Mat. & Fís.	1.2	Fácil	Funciones trigonométricas	Escribir un programa de demostración que solicite al usuario seleccionar una de las tres funciones inversas posibles: arcsen, arccos, o arccan y que introduzca el radio trigonométrico. Debe generarse un resultado apropiado, con su diagnóstico.
5.3	2	Mat. & Fís.	0.7	Fácil	Decibeles combinados	Determinar el nivel de potencia acústica producido por la combinación de dos fuentes de sonido.
5.5	3	CC	1.5	Moderado	Verificador de nombre de variable	Escribir un programa que verifique que el nombre de variable introducido por el usuario sea correcto; por ejemplo, si se tiene: 1) ilegal, 2) legal, pero estilo pobre, o 3) buen estilo. Asumir que los nombres de variables de "buen estilo" utilizan letras y dígitos únicamente y que utilizan una letra minúscula como primer carácter.
5.6	4	CC	1.0	Moderado	Disección de número telefónico	Implementar un programa que lea números telefónicos, y que para cada número, despliegue los componentes derivados del número telefónico: código de país, código de área y número local.
5.6	5	CC	1.1	Difícil	Disección de número telefónico (versión robusta)	Implementar una versión más robusta del programa de número telefónico. Permitir números telefónicos acortados, es decir, aquellos que tengan un solo grupo de dígitos y nada más, y números telefónicos que tengan sólo grupos de dígitos locales, un código de área y nada más.

[1] Pig Latin es un juego de idioma usado principalmente en inglés. El Pig Latin lo usan los niños para divertirse o para conversar secretamente.

Cap./ Secc.	Proyecto	Área académica	Páginas soluc.	Dificultad	Título	Breve descripción
5.8	6	Administración	1.0	Moderado	Cálculo de valor presente neto	Escribir un programa que calcule el valor presente neto de una inversión propuesta, dada una tasa de descuento y un conjunto arbitrario de flujos de caja futuros.
6.4	1	Biol. & Ecol.	1.5	Moderado	Observación de la germinación de una planta.	Escribir un programa que: 1) cree un objeto llamado árbol a partir de una clase MappleTree; 2) llame a un método planta que grabe la plantación de la semilla; 3) llame al método germinar que grabe la primera observación de una semilla en crecimiento y el registro de su altura; 4) llame a un método depósitoDatos que despliegue los valores actuales de todas las variables de instancia.
6.4	2	Administración	0.5	Fácil	Cuenta bancaria	Dado el código de una clase CuentaBancaria, generar un controlador que verifique dicha clase instanciando un objeto y llamando a sus métodos: setCliente, setCuenta e ImprimeInfoCuenta.
6.8	3	Mat. & Fís.	1.5	Moderada	Ecuación logística	Ejercitar la ecuación logística: netX= PresentX + r x presentX x preentX x (1 – present x) / (máximo x), y r es un factor de crecimiento.
6.9	4	Mat. & Fís.	0.9	Fácil	Círculo	Dado el código para la clase DriverCírculo, escribir una clase Círculo que defina una variable de instancia radio, un método setRadio y un método imprimeYCalculaDatosCírculo que utilice el radio del círculo para calcular e imprimir el diámetro del círculo, circunferencia y área.
6.10	5	Ingeniería	2.0	Moderado	Filtro digital	Dada la fórmula para filtros “pasabajos Chebyshev” o “pasabajos Butterworth”, con valores de parámetros apropiados, escribir un programa que pida al usuario proporcionar una secuencia de valores sin procesar y genere la salida filtrada correspondiente.

Cap./ Secc.	Proyecto	Área académica	Páginas soluc.	Dificultad	Título	Breve descripción
6.10	6	Sociología	3.1	Difícil	Máquina vendedora	Escribir un programa que simule la operación de una máquina vendedora. El programa debe leer la cantidad de dinero insertada en ésta, debe solicitar al usuario elegir un artículo, y después imprimir el cambio devuelto al comprador.
6.12	7	Mat. & Fís.	1.1	Fácil	Rectángulo	Implementar una clase Rectángulo que defina un rectángulo con variables de instancia longitud y anchura, métodos de acceso y modificación, y un método tipo booleano es cuadrado.
6.13	8	Biol. & Ecol.	4.0	Difícil	Dinámica Presa-depredador	Escribir un programa que modele una especie que pudiera ser ya sea un depredador o una presa o ambos. Correr una simulación que incluya presa, depredador, y sustento renovable para la presa.
6.13	9	Mat. & Fís.	2.1	Moderado	Guitarra mecánica	Escribir un programa que simule el movimiento de las cuerdas de la guitarra.
7.5 7.9	1	CC	3.5	Difícil	Lista ligada	Dado el código para un controlador, implementar una clase Receta que cree y mantenga una lista ligada de recetas. La asignación de problema especifica todas las variables de instancia y métodos en diagramas de clase UML.
7.7	2	CC	2.5	Fácil	Descripción de automóvil	Utilizar una cadena de llamadas a métodos que despliegue las propiedades de automóviles.
7.7 7.9	3	Biol. & Ecol.	4.6	Difícil	Ciclo de carbón	Dado el código de un controlador, escribir un par de clases para un programa que modele el ciclo de carbón en un ecosistema. Utilizar dos clases genéricas. Una clase, Entidad, define dos cosas. La otra clase, Relación, define interacciones.
7.8	4	CC	1.4	Fácil	Dirección IP	Implementar una clase Direcciones IP con una cadena que contenga puntos decimales y un arreglo de cuatro octetos tipo int.

Cap./ Secc.	Proyecto	Área académica	Páginas soluc.	Dificultad	Título	Breve descripción
7.9	5	Mat. & Fís.	4.5	Moderado	Manejador de fracciones	Dado el método main de una clase driver, escribir una clase Fracción. Incluir los siguientes métodos de instancia: sumar, multiplicar, imprimir, imprimirComoDoble, y un método de asignación de valores para cada variable de instancia.
7.10	6	Ingeniería	2.8	Moderado	Circuito eléctrico	Escribir las clases ramal y nodo como elementos de un circuito eléctrico. Un ramal transmite energía a través de un resistor en serie con un inductor. Un nodo mantiene voltaje en un condensador conectado a tierra. El código del controlador se proporciona en la parte de asignación del problema.
7.10	7	Administración	5.1	Difícil	Contabilidad de costos	Escribir un programa orientado a objetos que realice la contabilidad de costos en una planta manufacturera.
7.10	8	Sociología	6.4	Difícil	Campaña política	Escribir un programa que organice el estimado de votos, dinero y esfuerzo. Es una versión orientada a objetos del proyecto 8 en el capítulo 3.
8.4	1	CC	1.6	Fácil	Validación de entrada	Implementar un algoritmo que de forma repetida solicite al usuario que introduzca valores hasta que se caiga en un rango aceptable y calcule el promedio de entradas válidas. Ésta es una versión orientada a objetos del proyecto 8 en el capítulo 4.
8.4	2	Ingeniería	4.0	Difícil	Carga HVAC	Calcular las cargas de calefacción y aire acondicionado para una residencia tradicional. Ésta es una versión orientada a objetos de la versión del proyecto 7 en el capítulo 3.
8.6	3	Sociología	2.6	Moderada	Control de elevador	Escribir un programa que simule la operación de un elevador. Este programa debe simular lo que sucede cuando el usuario elige ir a un piso particular y cuando presiona el botón de alarma.
8.9	4	CC	2.0	Fácil	Reestructuración de prototipo	Tomando como prototipo al programa CicloAnidadoRectángulo de la figura 4.17 en la sección 4.12 y haciendo uso de la metodología arriba-abajo, reestructurarlo en un formato de POO.

Cap./ Secc.	Proyecto	Área académica	Páginas soluc.	Dificultad	Título	Breve descripción
9.3	1	Sociología	2.7	Fácil	Clase Persona	Definir una clase que simule la creación y el despliegue de objetos tipo Persona.
9.4	2	Sociología	2.7	Moderado	Calificación de tarea	Escribir un programa que maneje las calificaciones escolares. Utilizar variables de instancia para los puntos máximos y mínimos en una tarea en particular, y utilizar variables de clase para los puntos totales actuales y máximos de todas las tareas combinadas.
9.3	3	Sociología	3.9	Difícil	Grado de aprobación política	Escribir un programa que determine la media y la desviación estándar de estadísticas sencillas.
9.4	4	Ingeniería	5.7	Difícil	Entrada solar para colector solar y HVAC	Escribir un programa que mantenga el registro de la posición del Sol y que determine cuánta energía solar penetra en una ventana de vidrio en cualquier orientación, lugar y momento.
9.6	5	Administración	2.7	Moderado	Cálculo del valor presente neto	Escribir un programa que calcule el valor presente neto de una inversión propuesta, dada una tasa de descuento y un conjunto arbitrario de flujos de caja futuros. Ésta es una versión orientada a objetos de la versión del proyecto 6 en el capítulo 5.
9.7	6	Mat. & Fís.	7.0	Difícil	Problema de los tres cuerpos	Escribir un programa que modele el problema de los tres cuerpos en los cuales dos igualan el tamaño del círculo lunar sobre la Tierra en diferentes órbitas. Esto ejemplifica un movimiento dinámico caótico.
10.4	1	Biol. & Ecol.	5.0	Difícil	Proyecciones demográficas	Escribir un programa que proyecte la población mundial futura y la riqueza individual en función de las tasas de fertilidad y tasas de extracción, y que incluya los efectos de impuestos y gasto gubernamentales.
10.6	2	CC	3.3	Moderado	Simulador de lanzamiento de dados	Escribir un programa que simule el lanzamiento de un par de dados y que imprima un histograma mostrando las frecuencias de los posibles resultados.

Cap./ Secc.	Proyecto	Área académica	Páginas soluc.	Dificultad	Título	Breve descripción
10.6	3	CC	5.1	Difícil	Escalamiento simulado (el problema del vendedor viajero)	Escribir un programa que utilice escalamiento simulado para resolver el muy complicado problema de encontrar el itinerario más corto para visitar todas las principales ciudades del mundo exactamente una vez.
10.7	4	Sociología	2.1	Fácil	Lista de invitados a una fiesta	Escribir un programa que cree un objeto Fiesta, agregue invitados a la fiesta, y que imprima información de la misma.
10.9	5	Sociología	2.7	Fácil	Contador de vocales	Escribir un programa que cuente el número de vocales mayúsculas y minúsculas en líneas de texto completas introducidas por el usuario y que imprima un resumen del número de vocales contadas.
10.9	6	Mat. & Fís.	7.6	Difícil	Solución de ecuaciones algebraicas simultáneas	Escribir un programa que cargue un conjunto de ecuaciones algebraicas simultáneas en arreglos de dos dimensiones y que resuelva las mismas por el método de descomposición de abajo a arriba.
10.9	7	Mat. & Fís.	2.5	Moderado	Regresión lineal	Escribir un programa que calcule la regresión linear, trazando una línea recta en una serie de datos aleatorios.
10.10	8	Administración	3.4	Moderado	Recibos de compra	Escribir un programa que registre los recibos de compra del negocio, despliegue información del recibo actual y registre los pagos a dichas compras.
10.11	9	Sociología	1.1	Fácil	Baraja	Escribir una clase que utilice una lista de arreglos para repartir barajas.
10.13	10	Administración	1.9	Fácil	Tienda de libros	Escribir un programa que modele el almacenamiento y recuperación de libros con base en el título.
11.3	1	Biol. & Ecol.	5.5	Difícil	Juego de la vida	Modelar un “juego” que simule la reproducción y crecimiento en una malla de celdas rectangulares. Una X indica vida. Una celda muerta vuelve a vivir cuando tiene exactamente tres celdas vecinas vivas. Una celda viva permanece viva sólo cuando quedan dos o tres celdas vecinas vivas.

Cap./ Secc.	Proyecto	Área académica	Páginas soluc.	Dificultad	Título	Breve descripción
11.3	2	CC	0.7	Fácil	Tabla ASCCI	Escribir un programa que imprima una tabla con los 128 caracteres del código ASCII. La tabla debe imprimirse en un formato de ocho columnas separadas.
11.7	3	CC	0.8	Fácil	Cola circular	Un programa determinado implementa una cola circular. Reescribir los métodos EstáLleno, borrar y mostrarCola, mediante el reemplazo de los operadores condicionales, asignaciones incrustadas y operadores incrementales incrustados con un código más simple y entendible.
11.7	4	Mat. & Fís.	4.1	Moderado	Interpolación polinómica	Ajustar un polinomio a los puntos en cualquier parte de un par de puntos en un arreglo de datos y utilizar esto para estimar el valor en una posición entre el par de puntos.
11.9	5	CC	1.4	Moderado	Operaciones lógicas a nivel de bit	Utilizar el corrimiento aritmético y lógico para desplegar los valores binarios de ciertos números.
11.11	6	CC	3.5	Moderado	Ordenamiento de pila	Utilizar el algoritmo de ordenamiento de pila para ordenar datos. (Éste es un algoritmo robusto y no de ordenamiento de lugares con una complejidad computacional de NLogN.)
12.2	1	Administración	1.7	Fácil	Cuentas de ahorro	Calcular y desplegar los balances de las cuentas de ahorro haciendo uso del interés compuesto.
12.4	2	Mat. & Fís.	13.4	Difícil	Funciones estadísticas	Escribir un programa que genere valores para las funciones Gama, Gama incompleta, Beta, Beta incompleta y estadística binomial.
12.5	3	Administración	3.3	Fácil	Programa de carro	Haciendo uso de herencia, escribir un programa que mantenga un rastreo de los autos nuevos y usados.
12.10	4	Sociología	16.4	Difícil	Juego de corazones	Escribir un programa que simule el juego de corazones básico con un número arbitrario de jugadores. Dar a todos los jugadores un conjunto igual de estrategias que optimicen sus oportunidades de ganar.

Cap./ Secc.	Proyecto	Área académica	Páginas soluc.	Dificultad	Título	Breve descripción
13.7	1	Administración	9.0	Difícil	Inventario en una tienda de abarrotes	Escribir un programa de inventarios que mantenga un registro de varios artículos comestibles. Utilizar diferentes métodos en la clase Inventario para procesar objetos heterogéneos y artículos alimenticios de marca. Almacenar todos los objetos juntos en una estructura ArrayList.
13.7	2	Ingeniería	8.7	Difícil	Análisis de circuitos eléctricos	Escribir un programa que calcule las corrientes en estado estacionario en un circuito eléctrico de dos mallas que tenga una combinación arbitraria de resistores discretos, condensadores de capacidad y fuentes en las terminales del circuito. Incluir métodos que ejecuten suma, resta, multiplicación y división de números complejos (números con partes reales e imaginarias).
13.8	3	Administración	5.4	Moderado	Nómina	Utilizar polimorfismo para escribir un programa de nómina que calcule e imprima la nómina semanal de una compañía. Asumir tres tipos empleados: por hora, asalariados y asalariados con comisión. Asumir que cada tipo de empleado obtiene su pago utilizando una fórmula diferente. Utilizar una clase abstracta como base.
13. 8	4	Administración	2.9	Moderado	Cuentas bancarias	Escribir un programa que maneje los saldos de las cuentas bancarias para un arreglo de cuentas bancarias. Utilizar dos tipos de cuentas bancarias: cheques y ahorros, derivadas de una clase abstracta llamada CuentaBancaria.
14.4	1	Sociología	4.0	Moderado	Índice de masa corporal	Escribir un programa que solicite al usuario los valores de su peso y altura y que despliegue el índice de masa corporal asociado.
14.5	2	Sociología	4.0	Moderado	Almacenamiento y recuperación de objetos en un arreglo	Buscar una igualdad de los valores clave en una tabla relacional, utilizando dos tipos diferentes de algoritmos de búsqueda: búsqueda secuencial y hashing.

Cap./ Secc.	Proyecto	Área académica	Páginas soluc.	Dificultad	Título	Breve descripción
14.9	3	CC	2.5	Moderado	Formato de fechas	Crear una clave denominada Fecha que almacene valores e imprima la fecha ya sea en un formato numérico o alfabético. Utilizar una clase separada para manejar todas las excepciones.
14.9	4	CC	5.5	Difícil	Utilidad de entrada de datos	Escribir una clase que lea las entradas del teclado y determine a qué tipo de dato se refiere: String, char, double, float, long e int. Debe hacerlo de manera similar a como lo haría un escáner.
15.4	1	Ingeniería	3.7	Moderada	Cuestionario sobre uso de carreteras	Modelar el tráfico de una carretera que atraviesa un lugar específico, recolectar observaciones y leer el archivo para un análisis posterior.
15.4	2	Sociología	2.9	Fácil	Fusión de correo	Escribir un programa que lea un formato de carta desde un archivo de texto y modifique campos personalizados.
15.5 15.9	3	CC	5.0	Moderada	Conversión de archivos	Escribir un programa que reemplace los espacios en blanco en un archivo de texto.
15.8	4	CC	1.5	Fácil	Agregar datos a un archivo objeto	Implementar el código necesario para agregar datos a un archivo objeto.
16.12	1	Ingeniería	4.1	Moderada	Puerta de cochera animada	Escribir un programa que simule la operación de una puerta automática de cochera y sus controles, y que despliegue visualmente su posición al momento de operar.
16.14	2	Sociología	3.0	Moderada	Memorización de colores	Escribir un programa que verifique la habilidad del usuario de memorizar una secuencia de colores.
16.14	3	Administración	8.7	Difícil	Inventario de abarrotes GUI	Diseñar una versión gráfica del proyecto del capítulo 13 (inventario de almacén), utilizando la GUI.
16.15	4	Sociología	4.2	Moderada	Juego de orden de palabras	Escribir un simple juego interactivo que ayude a los niños a practicar el alfabeto.
16.16	5	Administración	3.8	Moderada	Reservaciones en aerolínea	Escribir un programa con interfaz gráfica que permita asignar asientos en los vuelos de una aerolínea.
17.3	1	CC	1.7	Fácil	Cambiar color y alineación	Escribir un programa interactivo que modifique el color y la posición de los botones en una ventana gráfica.

Cap./ Secc.	Proyecto	Área académica	Páginas soluc.	Dificultad	Título	Breve descripción
17.6	2	CC	1.9	Fácil	Rastreo de clic	Escribir un programa interactivo que modifique los bordes y las etiquetas de los botones en una ventana gráfica.
17.7	3	Sociología	3.4	Moderada	Juego de gato	Crear un juego de gato interactivo.
17.10	4	Sociología	4.3	Moderada	Juego de palabras, de nueva cuenta.	Modificar el juego del orden de palabras del capítulo 16, de tal manera que utilice un administrador de distribución.
17.10	5	Ingeniería	7.5	Difícil	Difusión termal en una fuente de bomba de calor central	Escribir un programa que calcule temperaturas en la Tierra en torno a una fuente de bomba de calor central. Desplegar los resultados en una gráfica a color de temperaturas en función de la distancia del centro terrestre y la época del año.

CAPÍTULO 1

Introducción a las computadoras y la programación

Objetivos

- Describir los diferentes componentes de un equipo de cómputo.
- Enumerar los pasos necesarios para el desarrollo de un programa.
- Conocer lo que significa la escritura de algoritmos haciendo uso del seudocódigo.
- Conocer lo que significa la escritura de programas usando el código de un lenguaje de programación.
- Entender el código fuente, el código objeto y el proceso de compilación.
- Describir cómo el código a nivel de byte hace de Java un lenguaje portátil.
- Familiarizarse con la historia del lenguaje Java: por qué fue desarrollado inicialmente, cómo obtuvo su nombre, etcétera.
- Escribir, compilar y ejecutar un simple programa de Java.

Relación de temas

1.1 Introducción

Este libro trata acerca de la resolución de problemas. Específicamente, sobre la resolución de problemas a través de un conjunto de instrucciones establecidas con precisión. A un conjunto de instrucciones, que se introducen y ejecutan en un formato para computadora, se le denomina *programa*. Para entender lo que es un programa, pensemos la siguiente situación. Supongamos que administramos una tienda departamental y que no sabemos cuándo debemos volver a surtir los estantes porque tenemos problemas para mantener un inventario. La solución a este problema sería escribir un conjunto de instrucciones para mantener el control de los artículos al momento que entran y salen de la tienda. Si las instrucciones son correctas y están en un formato entendible por la computadora, las podemos introducir como un programa, ejecutarlo e introducir los datos de entrada y salida de los artículos al momento en que éstos se

presenten. Podemos traer información del inventario desde la computadora en el momento en que así lo requiramos. Este conocimiento sencillo y seguro nos permite volver a surtir nuestros estantes de manera eficiente, y con ello obtener mayores utilidades.

El primer paso para aprender a escribir programas es comprender los conceptos básicos de computación. En este capítulo revisaremos esos conceptos. En los capítulos subsiguientes se hará uso de esos conocimientos para explicar los elementos verdaderamente fundamentales de la programación.

Este capítulo inicia con una descripción de las diferentes partes de una computadora, y continúa con la explicación de los pasos a seguir para escribir y ejecutar un programa. En seguida, se centra exclusivamente en la descripción del lenguaje de programación que se utilizará en el resto del libro: el lenguaje Java. Se presentan instrucciones paso a paso sobre cómo capturar y ejecutar un programa real de Java, con esto el lector adquiere cierta experiencia práctica desde el principio. El capítulo termina con una sección opcional del GUI (siglas en inglés de Graphical User Interface, interfaz gráfica del usuario) en la que se describe cómo capturar y ejecutar un programa usando la interfaz gráfica (GUI).

1.2 Terminología de hardware

Un *sistema de cómputo* es el conjunto de todos los componentes necesarios para que una computadora pueda operar y las conexiones entre estos componentes. Existen dos categorías básicas de componentes: *hardware* y *software*. El hardware son los componentes físicos asociados con una computadora. El software son los programas que le indican a la computadora lo que debe hacer. Por ahora, nos centraremos en la parte del hardware.

La descripción del hardware de una computadora proporcionará al lector la información que requiere como programador principiante. (Un *programador* es la persona que escribe programas.) Una vez adentrado en el material presentado aquí, si el lector lo desea, puede consultar el sitio Web de Webopedia en http://www.webopedia.com/ y escribir `hardware` en el cuadro de búsqueda.

Panorama general

La figura 1.1 muestra los componentes de hardware básicos de un sistema de cómputo. Muestra los dispositivos de entrada a la izquierda (teclado, mouse y escáner), dispositivos de salida a la derecha (monitor e impresora), dispositivos de almacenamiento en la parte de abajo, y la CPU (siglas en inglés de Central Process Unit, unidad central de proceso) y la memoria principal al centro. Las flechas en la figura 1.1 representan las conexiones entre los diferentes componentes. Por ejemplo, la flecha del teclado a la memoria principal de la CPU representa un cable (una conexión alámbrica) que transmite información del teclado a la CPU y a la memoria principal. En esta sección se explicará la CPU, la memoria principal y los demás dispositivos presentes en la figura 1.1.

Dispositivos de entrada y salida

Existen diferentes definiciones de *dispositivo de entrada*, pero usualmente el término se refiere a un dispositivo que transfiere información a una computadora. Recuerde que la información que entra en una computadora es *entrada*. Por ejemplo, un teclado es un dispositivo de entrada porque cuando se presiona una tecla, el teclado envía información a la computadora (le indica qué tecla fue presionada).

Existen diferentes definiciones de *dispositivo de salida*, pero usualmente el término se refiere a un dispositivo que transfiere información fuera de una computadora. Recuerde que la información que sale de una computadora es *salida*. Por ejemplo, un monitor (también llamado *pantalla*) es un dispositivo de salida pues despliega información proveniente de la computadora.

Unidad central de proceso

La *unidad central de proceso* (CPU), también conocida como *procesador* o *microprocesador*, es el cerebro de la computadora. De la misma manera que un cerebro biológico, la CPU divide su tiempo en dos actividades básicas: pensar y administrar el resto de su sistema. Las actividades de "pensamiento" ocurren cuando la CPU lee las instrucciones de un programa y las ejecuta; las actividades del "sistema de administración" suceden cuando transfiere información a y desde los otros dispositivos del sistema de cómputo.

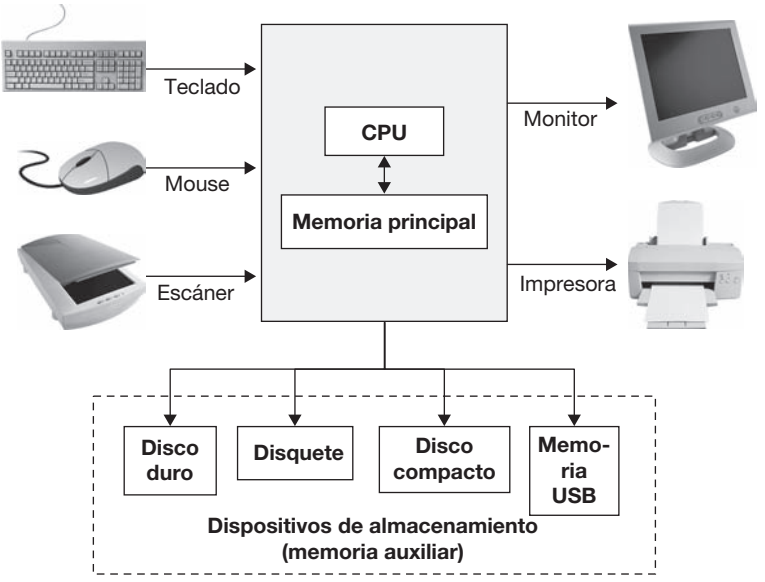


Figura 1.1 Una versión simplificada de una computadora.

El siguiente es un ejemplo de las actividades de pensamiento de una CPU. Supongamos que tenemos un programa que rastrea la posición de un satélite en su órbita alrededor de la Tierra. Dicho programa contiene bastantes cálculos matemáticos y la CPU los ejecuta.

El que sigue es un ejemplo de las actividades de administración del sistema de una CPU. Supongamos que usted tiene un programa de contratación de personal. El programa despliega los cuadros en que la persona introduce su nombre, teléfono, etc. Después de introducir la información, utiliza el mouse y da clic una vez que termina. Para este programa, la CPU administra su sistema de la siguiente manera: para desplegar el formato de solicitud, lee la información desde el mouse y el teclado.

Si el lector está pensando en adquirir un equipo de cómputo, es importante que considere la calidad de los componentes, para ello es necesario que conozca ciertas características de los mismos. Para la CPU, es necesario que conozca los procesadores populares y el rango de velocidad de los procesadores típicos. A continuación se presentan los siguientes procesadores y sus velocidades con cierta reserva, ya que tales cosas cambian en el mundo de las computadoras a una velocidad inusitada. Al presentar tales detalles, se está fechando nuestro libro a la suerte; sin embargo, no nos detenemos...

A partir de septiembre de 2007:

- CPU populares: Core 2 Duo (manufacturado por Intel), Athlon 64 (manufacturado por AMD).
- Velocidades actuales de CPU: de 2.5 a 3.8 GHz.

¿Qué son los *GHz*? GHz significa *gigahertz*. *Giga* significa 1 000 millones y *hertz* es una unidad de frecuencia igual a una vibración por segundo. Una CPU a 2.5 GHz utiliza un reloj que emite ciclos de 2 500 millones de veces por segundo. Eso significa rapidez, pero una CPU de 3.8 gigahertz es aun más rápida: utiliza un reloj que emite ciclos de 3 800 millones de veces por segundo. Un reloj proporciona sólo una vaga medida de qué tan rápido hace las cosas la CPU. Los ciclos de reloj son los iniciadores de las tareas de cómputo. Con más ciclos de reloj por segundo, hay más oportunidades de realizar mayor número de tareas.

Memoria principal

Cuando una computadora ejecuta instrucciones, usualmente requiere guardar los resultados intermedios; por ejemplo, en el cálculo del promedio de velocidad de 100 medidas de velocidad, la CPU calcula la suma mediante la creación de un área de almacenamiento usada exclusivamente para ello. Así, para cada valor de velocidad, la CPU agrega dicho valor al área de almacenamiento de la suma. Podemos imaginar a la memoria como una colección de cajas de almacenamiento, donde la suma sería guardada en uno de los cajones de almacenamiento de la memoria.

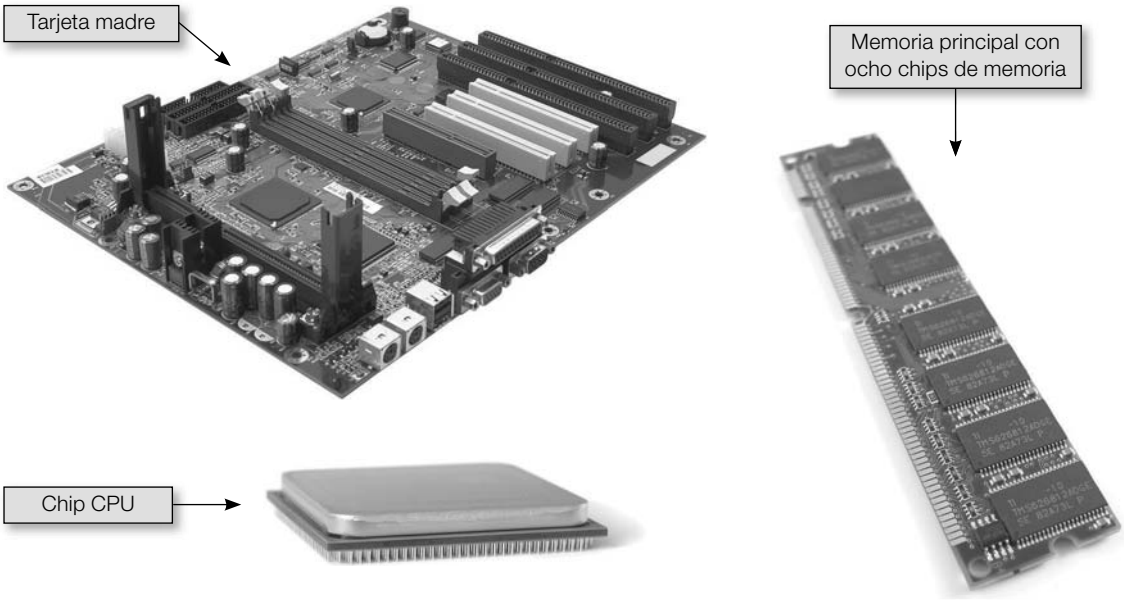


Figura 1.2 Tarjeta madre, chip CPU y chips de memoria principal.

Hay dos categorías de memoria: *memoria principal* y *memoria auxiliar o secundaria*. La CPU trabaja más con la memoria principal. Imaginémonos a la memoria principal como un cuarto de archivos al lado de la oficina del jefe. El jefe es la CPU, y él/ella guarda sus cosas en los cajones del cuarto de archivos cuando así lo requiere. Ahora, imaginémonos a la memoria auxiliar como una bodega que se encuentra al otro lado de la acera donde está la oficina del jefe. El jefe utiliza la bodega para almacenar objetos, pero no acude ahí muy seguido. Se entrará en más detalles sobre la memoria auxiliar en el siguiente apartado. Por ahora, será conveniente enfocarnos en los detalles de la memoria principal.

La CPU depende mucho de la memoria principal. Constantemente se encuentra almacenando datos en la memoria principal y leyendo datos de la misma. Con esta interacción constante, es necesario que la CPU y la memoria principal puedan comunicarse de manera muy rápida. Para asegurar una comunicación rápida, ambos elementos se encuentran físicamente colocados de manera muy cercana. Los dos están construidos con *chips*, y ambos conectados a la tarjeta de circuitos principal de la computadora, la *tarjeta madre*. En la figura 1.2 se muestran las fotos de una tarjeta madre, de un chip de la CPU y de los chips de la memoria principal.

La memoria principal contiene cajas de almacenamiento, y en cada una de estas cajas hay una pieza de información. Por ejemplo, si un programa almacena el apellido de uno de los autores de este libro, Dean, entonces utiliza ocho cajas de almacenamiento: una para la mitad de la letra D, una para la segunda mitad de la letra D, una para la primera mitad de la e, una para la segunda mitad de la e, etc. Después de almacenar las cuatro letras, el programa probablemente necesitará traerlas de vuelta en un punto posterior. Para poder recuperar la información, es necesaria una dirección. Una *dirección* es una localidad específica. Una dirección postal cuenta con valores de calle, ciudad y código postal para especificar una localidad. Una dirección de cómputo utiliza la posición de la información dentro de la memoria principal para especificar una localidad. La primera caja de almacenamiento de la memoria principal es la posición cero, por lo que decimos que se encuentra en la posición 0. La segunda caja de almacenamiento está en una posición que podemos decir es la dirección 1. En la figura 1.3 se muestra cómo el nombre Dean se almacena en la memoria que inicia en la dirección 50 000.

Es importante entender la terminología formal cuando hablamos del tamaño de la memoria. Supongamos que el lector está por adquirir una computadora y desea conocer qué tan grande es su memoria. Si le preguntara al vendedor cuántas "cajas de almacenamiento" contiene la computadora, probablemente éste lo miraría con perplejidad. Lo que tendría que preguntar el lector es acerca de su *capacidad*, ése es el término formal para referirse a su tamaño. Si le preguntara al vendedor acerca de la capacidad de memoria del equipo, la respuesta sería algo así como "es de un *gigabyte*". Ya se sabe que giga significa

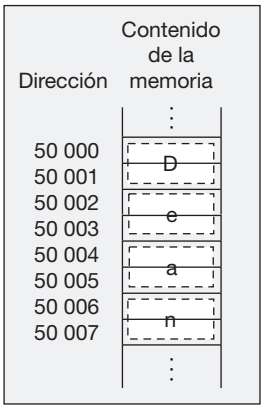


Figura 1.3 Los caracteres D, e, a, n son almacenados en la dirección 50 000.

1 000 millones. *Byte* se refiere al tamaño de una caja de almacenamiento. Así pues, una memoria principal con una capacidad de un gigabyte contiene 1 000 millones de cajas.

Continuando con la descripción de las cajas de almacenamiento con más detalle, sabemos que éstas pueden contener caracteres, tales como la letra D; pero las computadoras no son tan inteligentes como para entender el alfabeto, únicamente entienden los ceros y los unos, por tanto realizan un mapeo de cada carácter del alfabeto en una serie de 16 ceros y unos. Por ejemplo, la letra D sería 00000000 01000100. Así, para almacenar la letra D, la memoria principal almacena 00000000 01000100. Cada uno de los 0 y los 1 se denomina *bit*. Y cada uno los grupos de ocho bits es un byte.

La pregunta es: ¿por qué las computadoras usan 0 y 1? La respuesta es que sólo entienden señales de alta y de baja energía. Cuando una computadora genera una señal de baja energía, ésta es igual a 0. Cuando genera la señal de alta energía, ésta es igual a 1.

Se sabe que las computadoras almacenan caracteres como 0 y 1, pero ¿sabíamos que las computadoras también almacenan números como 0 y 1? Formalmente, se afirma que las computadoras utilizan el *sistema binario de números*. El sistema binario utiliza únicamente dos dígitos, el 0 y el 1, para representar todos los números. Por ejemplo, las computadoras almacenan el número 19 en un formato de 32 bits 00000000 00000000 00000000 00010011. La razón de que estos 32 bits representen el número 19 es que cada bit de valor 1 representa una potencia de 2. El lector debe observar que hay tres bits de valor 1 y que éstos se encuentran en las posiciones 0, 1 y 4, donde la posición 0 inicia en el lado derecho. El valor de un bit determinado se obtiene elevando el 2 al valor de la posición del bit. Así, el valor del bit más representativo a la derecha representa un 2 elevado a la potencia 0, lo cual da 1 ($2^0 = 1$). El bit en la posición 1 representa 2 elevado a la potencia 1, lo cual da como resultado el número 2 ($2^1 = 2$); el bit en la posición 4 representa 2 elevado a la potencia 4, lo cual da 16 ($2^4 = 16$). Al sumar las tres potencias se obtiene el número 19 ($1 + 2 + 16 = 19$). *Voilá!*

Es importante saber que la memoria principal se conoce también como *RAM*, que significa *memoria de acceso aleatorio* (*Random Access Memory*, en inglés). Se considera como de "acceso aleatorio" debido a que puede ser accedida en cualquier dirección (por ejemplo, en una dirección aleatoria). Lo anterior contrasta con los dispositivos de almacenamiento donde se tiene acceso a los datos desde el inicio, datos que se recorren hasta encontrar el que se busca.

Una vez más, si el usuario está por adquirir una computadora, es necesario que considere la calidad de sus componentes. Para lo referente a la memoria principal/RAM, requerirá saber si su capacidad es la adecuada. A partir de septiembre de 2007, la capacidad de las memorias principales está en los rangos de 512 MB hasta los 3 GB. MB significa megabyte, donde *mega* es 1 millón. *GB* significa gigabyte.

Memoria auxiliar

La memoria principal es *volátil*, esto significa que los datos se pierden una vez que la computadora se apaga. El usuario podría preguntarse, si los datos se pierden, entonces ¿cómo puedo guardarlos de forma permanente en la computadora, antes de apagarla? La respuesta es algo que el usuario hace (o debería hacer) frecuentemente. Cuando usted ejecuta un comando de salvar, la computadora realiza una copia de los datos de la memoria principal con la que usted está trabajando, y almacena la copia en la memoria

auxiliar. La memoria auxiliar es *no volátil*, lo que significa que los datos no se pierden al apagar la computadora.

Una ventaja de la memoria auxiliar sobre la memoria principal es que es no volátil. Otra, es que su costo por unidad de almacenamiento es mucho más bajo que el de la memoria principal. Una tercera ventaja es que es más *portátil* que la memoria principal (por ejemplo, puede transportarse de una computadora a otra de una manera más fácil).

La desventaja de la memoria auxiliar es que su *tiempo de acceso* es mucho más alto comparada con la memoria principal. El tiempo de acceso es el que se toma en localizar una simple porción de datos y hacerla disponible a la computadora para su procesamiento.

La memoria auxiliar tiene muchas formas diferentes; por ejemplo, los discos duros, disquetes, discos compactos y las memorias USB o unidades flash USB. Estos dispositivos se conocen como *medios de almacenamiento*, o simplemente *dispositivos de almacenamiento*. La figura 1.4 muestra una foto de los mismos.

Los tipos más populares de discos compactos se agrupan como sigue:

- CD-audio: para almacenar música grabada, conocido comúnmente como "CD" (o disco compacto).
- CD-ROM, CD-R, CD-RW: para almacenar datos de cómputo y música grabada.
- DVD, DVD-R, DVD-RW: para almacenar video, datos de cómputo y música grabada.

El término "ROM" en CD-ROM significa memoria de sólo lectura (*Read Only Memory*, en inglés), y se refiere a la memoria que se puede leer, pero en la que no se puede escribir. Así pues, se puede leer un CD-ROM, pero no se puede cambiar su contenido. En los CD-R, se puede escribir una vez y leer lo escrito cuantas veces se quiera. Con CD RW, puede escribir y leer tanto como lo desee. DVD significa "disco digital versátil" ("Digital Versatile Disc") o "disco de video digital" ("Digital Video Disc"). Los DVD se parecen a los CD-ROM porque pueden leerse, pero no se puede escribir sobre ellos. De la misma manera, los DVD-R y DVD-RW, son similares a los CD-R y CD-RW en términos de sus capacidades de lectura y escritura.

Las memorias USB son rápidas, tienen una capacidad grande de almacenamiento y son extraordinariamente portátiles porque son del tamaño del pulgar de una persona y porque pueden conectarse como un *dispositivo virtual* (*hot swapping*) en prácticamente cualquier computadora encendida. Las siglas USB en memoria USB, significan Bus Serial Universal (*Universal Serial Bus*, en inglés), y se refieren a un tipo particular de conexión. De manera más específica, se refieren a un tipo de conexión de cable y de enchufe. Una unidad flash utiliza ese tipo de conexión, por ello también se conocen como *unidades flash USB*. Por cierto, muchos dispositivos de cómputo utilizan conexiones USB, y todos son intercambiables.

Los diferentes dispositivos de almacenamiento tienen diferentes capacidades de almacenamiento. A septiembre de 2007 se tiene:

- Los discos duros típicos tienen un rango de almacenamiento de 80 GB a 1 TB (*TB* significa terabyte, donde *tera* representa 1 billón).

Figura 1.4 Disco duro, disquete, disco compacto y memoria USB.

- Los disquetes típicos tienen una capacidad de 1.44 MB.
- Los CD-ROM, CD-R y CD-RW típicos tienen una capacidad de 700 MB.
- Los DVD, DVD-R y DVD-RW típicos tienen un rango de capacidad de 4.7 GB hasta 8.5 GB.
- Las unidades flash USB o memorias USB tienen un rango de capacidad de 128 MB hasta 64 GB.

Una unidad o *drive* es un mecanismo que permite a un sistema de cómputo acceder (leer y escribir en) datos de un dispositivo de almacenamiento. Una *unidad de disco* es una unidad para disco duro, disquete o disco compacto. Una unidad de disco gira sus discos muy rápido, y una o más *cabezas* (sensores electrónicos) acceden a los datos del disco cuando éste ha dejado de girar.

Para especificar el medio de almacenamiento en el que residen los datos, es preciso utilizar el nombre del mismo mediante una letra seguida de dos puntos. En los equipos de cómputo que utilizan alguna versión de Microsoft Windows, las unidades de disquete son referidas como `A:`, los discos duros son referidos como `C:` o `D:`, los discos compactos son referidos como `D:` o `E:`, y las unidades de memoria USB son referidas como `E:` o `F:`.

Cuando usted copia datos, copia lo que se conoce como *archivo*, que es un conjunto de instrucciones o datos relacionados. Por ejemplo, un programa 1) es un archivo que almacena un conjunto de instrucciones, y 2) un documento Word es un archivo que almacena datos de texto que fueron creados en Microsoft Word.

Vocabulario común de hardware de cómputo

Al adquirir una computadora o cuando se habla sobre computadoras con los amigos, es importante asegurarse de que se entiende el lenguaje vernáculo (los términos que la gente utiliza día a día en su hablar y que es distinto a los vocablos que pueden encontrarse en los libros de texto), a fin de poder entender lo que están hablando. Cuando una persona con conocimiento se refiere a la memoria de la computadora en sí misma, normalmente se refiere a la memoria principal: la memoria RAM. Al referirse al *espacio en disco*, de lo que está hablado es de la capacidad del disco duro de la computadora. Cuando alguien se refiere a la computadora en sí misma, lo que quiere decir es la caja o gabinete que contiene el procesador, la memoria principal, la unidad de disco duro y los discos duros asociados, la unidad de disquete y los dispositivos de E/S; aunque éstos son parte del sistema de cómputo, usualmente no se consideran parte de la computadora. En lugar de ello, se consideran dispositivos periféricos porque se encuentran en la periferia del equipo. Cuando las personas dicen el *floppy,* el *disco flexible*, se refieren al disco removible.

¿Por qué utilizar el término "flexible" para referirse a un disquete? Si se abre el disquete y se le quita la cubierta de plástico, podrá observarse que el medio de almacenamiento es flexible. Es importante saber que al abrir la cubierta plástica del disco, se destruirá el disquete. Hay que asegurarse de que no tengamos en éste nuestra tarea. Los autores no desean que el alumno obtenga una mala nota en su calificación y que le comenten al profesor: "los autores me obligaron a hacerlo".

Avance en mejoras a las computadoras

Durante el tiempo en que la memoria y los componentes de la CPU han existido, los fabricantes de estos dispositivos han mejorado el desempeño de sus productos a un ritmo consistentemente alto. Por ejemplo, la memoria RAM y la capacidad de discos duros se duplican aproximadamente cada dos años. Las velocidades de las CPU también se duplican aproximadamente cada dos años.

Una *leyenda urbana* es una historia que se expande de manera espontánea en varias formas y popularmente se cree que es cierta. La siguiente es una clásica leyenda urbana en Internet que habla acerca del rápido avance en las mejoras de las computadoras.[1] Aunque los intercambios de los que se habla aquí nunca tuvieron lugar, los comentarios, en particular el primero de ellos, son relevantes.

En una reciente exposición de computadoras (COMDEX), Bill Gates, según se informó, comparó a la industria de cómputo con la automotriz y afirmó lo siguiente: "Si GM hubiera mantenido un ritmo de crecimiento similar al de la tecnología de la industria de cómputo, estaríamos manejando autos de 25 dólares estadounidenses con los que se obtuviera un rendimiento de 1 000 millas por galón" (aproximadamente 445 km por litro).

[1] Snopes.com, *Rumor Has it*, en el sitio de Internet: http://www.snopes.com/humor/jokes/autos.asp (consultado el 15 de marzo de 2007).

En respuesta a los comentarios de Bill Gates, General Motors emitió un comunicado de prensa asegurando lo siguiente:

Si GM hubiese desarrollado tecnología como Microsoft, estaríamos manejando autos con las siguientes características:

1. Sin razón alguna y por cualquier motivo, nuestro automóvil chocaría dos veces al día.
2. Cada vez que se repintaran las líneas en las carreteras, tendríamos que comprar un auto nuevo.
3. Ocasionalmente nuestro auto se detendría en la carretera sin motivo alguno. Tendríamos que quitarlo del camino, cerrar todas las ventanas, apagarlo, reiniciarlo y reabrir las ventanas antes de poder continuar. Por alguna razón, estaríamos resignados a aceptar esto.
4. Ocasionalmente, ejecutar una maniobra, tal como una vuelta a la izquierda, provocaría que nuestro automóvil se apagara y se negara a reiniciar, por lo que tendríamos que reinstalar el motor.
5. Macintosh fabricaría un auto que funcionara con energía solar, que fuera confiable, cinco veces más rápido y dos veces más fácil de manejar, pero se ejecutaría sólo en cinco por ciento de las carreteras.
6. El aceite, la temperatura del agua y las luces alternas serían reemplazadas por un simple mensaje: "este auto ha ejecutado una operación ilegal", además las luces intermitentes y el automóvil no funcionarían.
7. De manera ocasional y por cualquier motivo, nuestro automóvil se cerraría y se negaría a dejarnos entrar en él hasta que simultáneamente abriéramos la puerta, prendiéramos el automóvil y levantáramos la antena del radio.
8. El sistema de bolsas de aire nos preguntaría: "¿está usted seguro?", antes de inflar las bolsas.

1.3 Desarrollo del programa

Como se mencionó anteriormente, un programa es un conjunto de instrucciones que pueden utilizarse para resolver un problema. A menudo, el programa contiene muchas instrucciones, y por lo regular son complicadas. Por tanto, desarrollar un programa exitoso implica mucho esfuerzo. Se requiere planeación, una implementación cuidadosa y mantenimiento. He aquí una lista de los pasos del proceso de desarrollo:

- Análisis de requerimientos
- Diseño
- Implementación
- Prueba
- Documentación
- Mantenimiento

El *análisis de requerimientos* sirve para determinar las necesidades y las metas. El *diseño* es la elaboración del esquema del programa. La *implementación* es la escritura misma del programa. La *prueba* es la verificación de que el programa funciona. La *documentación* es la escritura de la descripción de lo que el programa hace. El *mantenimiento* es llevar a cabo las mejoras y prevenir los errores posteriores. Los pasos están ordenados en una secuencia lógica en la que el usuario ejecutaría, de forma normal, primero el análisis de requerimientos, en segundo lugar, el diseño, etc. Pero algunos de estos pasos deberían ejecutarse a través del proceso de desarrollo en lugar de hacerlo en un tiempo determinado. Por ejemplo, se debería trabajar en el paso de la documentación a través del proceso de desarrollo, y también en la etapa de pruebas durante y después de la implementación y después del mantenimiento. Es importante saber que con frecuencia será necesario repetir la secuencia de pasos conforme vayan surgiendo. Por ejemplo, si una de las metas del programa cambiara, sería necesario repetir todos los pasos en diferentes grados.

En esta sección se hablará del análisis de requerimientos y del diseño. En el capítulo 2 se hablará con más detalle sobre el diseño, y se mostrarán ejemplos a lo largo del libro. Se tratará el tema de la implementación en la sección "Código fuente" del presente capítulo y se presentarán ejemplos a lo largo del libro. El tema de pruebas será tratado en el capítulo 8. Lo relacionado con el paso de documentación iniciará en el capítulo 3 y se ilustrará con ejemplos a lo largo del libro. Finalmente, el tema de mantenimiento se estudiará en el capítulo 8 y se ilustrará con ejemplos a lo largo del libro.

Análisis de requerimientos

El primer paso en el proceso de desarrollo de un programa es el análisis de los requerimientos, en el que se determinan las necesidades y metas del programa. Es importante que el programador comprenda con exactitud lo que el cliente desea. Desafortunadamente, es muy común que los programadores produzcan programas sólo para darse cuenta después de que era otra cosa lo que el cliente quería. Esta circunstancia desafortunada quizá se deba a una mala comunicación entre el cliente y el programador al inicio del proyecto. Si el cliente y el programador confían únicamente en la descripción verbal de la solución, es muy fácil que se omitan detalles importantes. Más adelante, esos detalles omitidos pueden ser la causa del problema, cuando el cliente y el programador se dan cuenta de que habían asumido cosas distintas acerca de cómo se implementarían los detalles.

Para ayudar en el proceso de comunicación, el cliente y el programador deberían crear *capturas de pantallas* de entrada de datos y de reportes de salida. Una captura de pantalla es como un retrato de cómo aparecerá dicha pantalla. Para crear capturas de pantalla se pueden desarrollar pequeños programas que impriman reportes con resultados hipotéticos. Como una manera alternativa se pueden hacer capturas de pantalla con la ayuda de software de diseño o, si el usuario es un artista en potencia, con lápiz y papel.

Diseño de programa

Después del paso de análisis de requerimientos, el segundo es el diseño del programa, en el que se crea una versión del programa enfocado a la lógica básica, no a los detalles de formulación. De forma más específica, se escriben instrucciones coherentes y lógicamente correctas, pero no hay que preocuparse de los pasos menores o de las faltas de ortografía. Esta clase de programas se conoce como *algoritmo*. Por ejemplo, la receta de un pastel es un algoritmo; contiene instrucciones para resolver el problema de cómo hornearlo. Las instrucciones son coherentes y lógicamente correctas, pero en ellas no se indican los pasos sobre cómo debe cubrir sus manos el cocinero con protectores antes de sacar el pastel del horno.

Seudocódigo

Al escribir un algoritmo, hay que enfocar la atención en la organización del flujo de instrucciones, y evitar empantanarse con los detalles. Para facilitar este enfoque, los programadores a menudo escriben las instrucciones de un algoritmo haciendo uso del *seudocódigo*. El seudocódigo es un lenguaje informal que utiliza términos del idioma español común para describir los pasos de un programa. Con un seudocódigo preciso, la *sintaxis* de cómputo no es necesaria. La sintaxis se refiere a las palabras, gramática y puntuación que conforman al lenguaje Java. La sintaxis del seudocódigo es benévola. El seudocódigo debe ser suficientemente claro para que pueda entenderse, pero las palabras, gramática y puntuación no tienen que ser perfectas. Se menciona la benevolencia para contrastarla con la precisión que se requiere en la siguiente fase en el desarrollo de un programa. En la siguiente sección se cubrirá la fase posterior, y el usuario comprobará que se requiere de palabras, gramática y puntuación perfectas.

Ejemplo: utilización del seudocódigo para encontrar el promedio de millas por minuto

Supongamos que al usuario se le solicita escribir un algoritmo para encontrar el valor promedio de millas por hora para un determinado viaje en auto. A continuación se buscará la solución a dicho problema. Para determinar el promedio de millas por hora, se requiere dividir la distancia total viajada entre el tiempo total. Asumiendo que se tiene que calcular la distancia total para dos localidades dadas. Para determinar la distancia total, es necesario tomar los dos puntos de ubicación finales, llamados “distancias finales” y restar el punto de ubicación inicial, llamada para ello “distancia inicial”. Asumiendo que se tiene que calcular el tiempo total de la misma manera, es necesario restar el tiempo inicial del tiempo final. Al poner todo esto junto, el seudocódigo para calcular las millas promedio por hora sería algo como lo siguiente:

```
Calcular la resta de la distancia final menos la distancia inicial.
Poner el resultado en distancia total.
Al tiempo final restar el tiempo inicial.
```

Poner el resultado en el tiempo total.
Dividir la distancia total entre el tiempo total.

En este punto, algunos lectores querrán aprender una forma avanzada de desarrollo de programación: programación orientada a objetos, o POO, como se le conoce comúnmente. La POO implica la idea de que cuando se diseña un programa es necesario pensar primero en sus componentes (objeto), en lugar de sus tareas. Por el momento no se requiere aprender sobre POO, en este punto el lector no está preparado para aprender acerca de los detalles de la implementación de la POO, pero si está interesado puede echar un vistazo al capítulo 6, sección 2.

1.4 Código fuente

En las primeras etapas del desarrollo de un programa, se escribe un algoritmo usando el seudocódigo. Después, se traduce el seudocódigo a *código fuente.* El código fuente es un conjunto de instrucciones escritas en un lenguaje de programación.

Lenguajes de programación

Un *lenguaje de programación* es un lenguaje que utiliza palabras especialmente definidas, gramática y puntuación que una computadora entiende. Si se intentara ejecutar instrucciones en seudocódigo, la computadora sería incapaz de entenderlas. Pero, si se intentaran ejecutar instrucciones en un lenguaje de programación (en código fuente), la computadora sí las entendería.

Así como hay muchos lenguajes que se hablan en el mundo (inglés, chino, hindi, etc.), también existe una infinidad de lenguajes de programación. Algunos de los más populares son VisualBasic, C++ y Java. Cada lenguaje de programación define sus propias reglas de sintaxis. Este libro se enfoca en el lenguaje de programación Java. Si se escribe un programa en código fuente de Java, deben seguirse reglas precisas de sintaxis de Java en términos de palabras, gramática y puntuación. Si se escribe código fuente de Java utilizando una sintaxis errónea (por ejemplo, si se escribe una palabra incorrecta o se olvida introducir un punto y coma), y se intenta ejecutar dicho código fuente en una computadora, ésta será incapaz de entenderlo.

Ejemplo: utilización de Java para encontrar el promedio de millas por hora

Continuando con el ejemplo anterior donde se escribió seudocódigo para encontrar el valor promedio de millas por hora para un viaje en automóvil, traduzcamos el seudocódigo a código fuente en Java. En la tabla siguiente, el seudocódigo del lado izquierdo se traduce en código fuente Java a la derecha. Así, las primeras instrucciones de seudocódigo se traducen en simples instrucciones de código fuente Java a la derecha.

Seudocódigo	Código fuente Java
Calcular la resta de la distancia final menos la distancia inicial. Poner el resultado en distancia total.	`distanciaTotal= distanciaFinal – distanciaInicial;`
Al tiempo final restar el tiempo inicial. Poner el resultado en el tiempo total.	`tiempoTotal = tiempoFinal – tiempoInicial;`
Dividir la distancia total entre el tiempo total.	`PromedioMPH= distanciaTotal / tiempoTotal;`

Los programadores por lo regular se refieren a las instrucciones del código fuente en Java como *Sentencias Java*. Para que las sentencias Java funcionen, deben utilizar una sintaxis precisa. Por ejemplo, como se muestra a continuación, las sentencias Java deben: 1) utilizar el signo − para resta, 2) utilizar / para división, 3) tener un punto y coma al final de la sentencia, en el lado derecho. La precisión requerida para las sentencias Java contrasta con la flexibilidad del seudocódigo. El seudocódigo permite cualquier sintaxis, pues lo puede entender cualquier persona. Por ejemplo, en seudocódigo sería aceptable repre-

sentar la resta con un − o con la palabra “resta”. De la misma manera, sería aceptable representar la división con una / o un ÷ o con la palabra “divide”.

Omisión del paso de seudocódigo

Al inicio será más difícil entender el código del lenguaje de programación que el seudocódigo; pero después de obtener experiencia con un lenguaje de programación, el usuario se sentirá tan seguro de lo que puede hacer que podrá saltarse todo el paso del seudocódigo y pasar a la segunda etapa donde escribirá el programa, usando el código del lenguaje de programación.

Para programas más largos es recomendable no omitir el paso del seudocódigo. ¿Por qué? Porque en los programas más largos es importante centrarse primero en el cuadro completo y si eso no se hace bien, nada más importa. Y es más fácil centrarse en el cuadro completo si se utiliza seudocódigo, con el que no hay que preocuparse de los detalles de la sintaxis. Después de implementar una solución de seudocódigo, es relativamente sencillo convertir el seudocódigo en código fuente.

1.5 Compilación de código fuente en código objeto

Después de escribir el programa, el usuario querrá que la computadora ejecute las tareas que se especifican en el mismo. Lograrlo implica por lo regular un proceso de dos pasos: 1) Ejecutar un comando de compilación. 2) Ejecutar un comando de ejecución. Cuando se ejecuta un comando de *compilación*, se ordena a la computadora traducir el código del programa fuente en código que la computadora pueda ejecutar. Cuando se realiza un comando de *ejecución*, se ordena a la computadora ejecutar el código traducido y ejecutar las tareas especificadas en el código. En esta sección se explicará el proceso de traducción.

La computadora contiene un programa especial llamado *compilador* que se encarga del proceso de traducción. Si se envía código fuente al compilador, éste lo traduce a código que la computadora puede ejecutar. De manera más formal, el compilador compila el código fuente y produce el *código objeto* como resultado.[2] El código objeto es un conjunto de instrucciones en formato binario que puede ejecutarlo directamente la computadora para resolver un problema. Una instrucción en código objeto está conformada por 0 y 1 porque las computadoras sólo comprenden 0 y 1. He aquí una instrucción en código objeto:

```
0100001111101010
```

Esta instrucción particular en código objeto hace referencia a una *instrucción de 16 bits* porque cada uno de los 0 y 1 se denomina bit, y está conformada por 16 de éstos. Cada instrucción en código objeto es una simple tarea de cómputo. Por ejemplo, una instrucción en código objeto podría ser copiar un simple número de algún lugar en la memoria a algún lugar en la CPU. No es necesario que los programadores entiendan los detalles sobre cómo funciona el código objeto. Ése es trabajo de la computadora, no de los programadores.

Los programadores a menudo hacen referencia al código objeto como *código de máquina*. El código objeto se llama también código de máquina porque está escrito en binario, que es el código que entiende la “máquina”.

1.6 Portabilidad

En la subsección “Memoria auxiliar” de la sección 1.2, se señaló que la memoria auxiliar es más portátil que la memoria principal porque puede moverse de una computadora a otra de una manera sencilla. En ese contexto, la portabilidad se refiere al hardware. La portabilidad también puede referirse al software. Una pieza de software es *portátil* si se puede utilizar en muchos tipos de computadoras.

[2] La mayoría de los compiladores, pero no todos, producen código objeto. Como se verá en la siguiente sección, los compiladores Java producen una forma intermedia de instrucciones. En una etapa posterior, esa forma intermedia de instrucciones se traduce en código objeto.

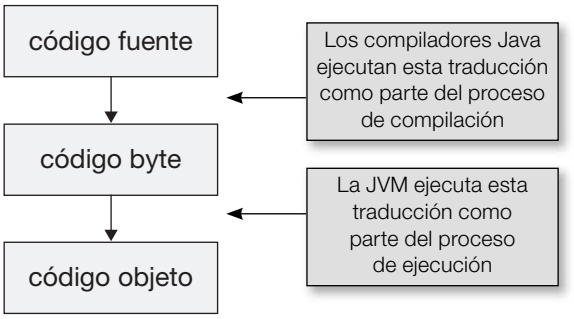


Figura 1.5 Manera en que un programa Java es convertido de código fuente en código objeto.

El problema de portabilidad con el código objeto

El código objeto no es muy portátil. Como se sabe, el código objeto está compuesto por instrucciones en formato de código binario. Estas instrucciones en formato binario están estrechamente unidas a un tipo de computadora. Si se tiene código objeto que fue creado en un tipo de computadora X, entonces dicho código objeto sólo puede ejecutarse en el tipo de computadoras X. De la misma forma, si se tiene código objeto que fue creado en un tipo de computadora Y, entonces dicho código objeto sólo puede ejecutarse en el tipo de computadoras Y.[3]

Entonces ¿por qué preocuparse por la portabilidad? ¿Quién se preocupa por el código objeto si no es muy portátil? La respuesta es los fabricantes de software. Si quieren vender un programa que se ejecute en diferentes tipos de computadoras, por lo regular tienen que compilar su programa en estos diferentes tipos de computadoras. Eso produce archivos con código objeto diferentes, y dichos archivos son comercializados. ¿No sería más sencillo que los fabricantes produjeran una forma en que los programas se ejecuten en cualquier tipo de computadora?

Java es la solución al problema de portabilidad

Los creadores de Java intentaron acabar con el problema inherente a la falta de portabilidad en el código objeto introduciendo el nivel de *código byte* entre los niveles de código fuente y código objeto. Los compiladores de Java no compilan en la forma de código objeto. En lugar de esto, lo hacen a nivel de código byte, que posee las mejores características, tanto de código objeto como de código fuente:

- Como el código objeto, el código byte utiliza un formato que funciona de forma muy cercana con el hardware de la computadora, por lo que se ejecuta de manera rápida.
- Como el código fuente, el código byte es genérico, por consiguiente puede ejecutarse en cualquier tipo de computadora.

¿Cómo puede el código byte ejecutarse en cualquier tipo de computadora? Cuando el código byte Java de un programa se ejecuta, éste se traduce en código objeto mediante el programa intérprete de código byte de la computadora. El intérprete de código byte se conoce también como *Java Virtual Machine* (máquina virtual de Java) o *JVM*, de forma abreviada. La figura 1.5 muestra cómo la JVM traduce el código byte a código objeto. También muestra cómo el compilador traduce el código fuente a código byte.

Para ejecutar el código byte Java, una computadora debe tener instalada la JVM. Afortunadamente, la instalación de una JVM es muy sencilla. Es un programa, y no requiere de mucho espacio en memoria; es fácil de obtener; cualquiera puede bajarlo de forma gratuita de Internet. En la sección 1.8, se explica cómo puede el usuario bajar e instalar la JVM en su computadora.

[3] Existen alrededor de 15 tipos de computadoras que son de uso común hoy en día. Estos 15 tipos de computadoras corresponden a 15 categorías de CPU. Cada CPU tiene su propio *conjunto de instrucciones* diferentes. Un conjunto de instrucciones define el formato y significado de todas las instrucciones de código objeto que funcionan con un particular tipo de CPU. Una mayor explicación acerca del conjunto de instrucciones está fuera del alcance de este libro. Si se desea aprender más, consultar el sitio de Wikipedia en http://es.wikipedia.org e introducir "conjunto de instrucciones" en la sección de búsqueda.

¿Por qué el programa intérprete de código byte se denomina "máquina virtual de Java"?

A continuación se explicará el origen del nombre "máquina virtual de Java". Para los programas escritos con la mayoría de los lenguajes de programación, la "máquina" CPU ejecuta el código compilado del programa. En el caso de los programas escritos con Java, el código intérprete de código byte ejecuta el código compilado del programa. Por tanto, con Java, el intérprete de código byte actúa como una máquina CPU; pero el intérprete de código byte es sólo una pieza de software, no una pieza de hardware como una CPU real. Esto es una máquina virtual, y se debe a que los diseñadores de Java decidieron llamar programa intérprete de código byte a una máquina virtual de Java.

1.7 Surgimiento de Java

Software de enseres domésticos

A principios de los años noventa, poner inteligencia en los aparatos domésticos se consideraba como el siguiente paso de la tecnología. Algunos ejemplos de enseres domésticos inteligentes incluyen las cafeteras controladas por computadoras, o televisores controlados por un dispositivo programable interactivo. Anticipándose a un fuerte mercado para tales dispositivos, en 1991, Sun Microsystems fundó un equipo de investigación para trabajar en un "Proyecto verde" secreto, cuya misión era desarrollar software para enseres domésticos inteligentes.

La inteligencia de un aparato doméstico proviene de los chips del procesador que contiene y del software que se ejecuta en los chips de ese procesador. Los chips del procesador del aparato cambian constantemente porque los ingenieros continuamente encuentran formas de hacerlos más pequeños, a menor costo y más poderosos. Para ajustar el nuevo tamaño de los chips, el software que se ejecuta en ellos debe ser extremadamente flexible.

Originalmente, Sun planeó la utilización de C++ para su software de enseres domésticos, pero pronto se dio cuenta de que C++ no era suficientemente portátil. En lugar de escribir software en C++ y lidiar con los problemas de portabilidad inherentes a C++, Sun decidió desarrollar un nuevo lenguaje de programación para su software de dispositivos del hogar.

El nuevo lenguaje de Sun fue originalmente nombrado Oak (Roble, debido al árbol que se veía desde la ventana del líder de proyecto, James Gosling), pero sucedió que el nombre de Oak ya existía y que era el nombre de otro lenguaje de programación. Lo que sigue en esta historia, es que mientras un grupo de empleados de Sun tomaban un descanso en una tienda de café local, se les ocurrió el nombre de "Java". Les gustó el nombre de "Java" porque la cafeína tiene un papel significativo en las vidas de los desarrolladores de software. ☺

World Wide Web

Cuando comprobó que el mercado de enseres domésticos inteligentes era menos fértil de lo que había previsto, Sun casi detiene la marcha de su proyecto Java durante la fase de preliberación. Para fortuna de Sun (y de todos los amantes de Java), la World Wide Web hizo popular este lenguaje. Sun se dio cuenta de que el crecimiento de la Web podría darle fuerza a la demanda de su lenguaje Java, por lo que decidió continuar con sus esfuerzos en el desarrollo de Java. Estos esfuerzos rindieron fruto cuando se presentó la primera versión en mayo de 1995 durante la conferencia SunWorld. Poco después de ello, Netscape, el fabricante del navegador más popular en ese tiempo, anunció su intención de utilizar Java en el software de su navegador. Con el apoyo de Netscape, Java inició con estrépito su crecimiento, el cual ha ido en aumento desde entonces.

La Web depende de las páginas que se bajan de ella y que se ejecutan en muchos tipos de computadoras. Para trabajar en distintos ambientes, el software de páginas Web debe ser extremadamente portátil. El usuario probablemente se imaginará a Java como el héroe, aunque eso es un poco exagerado. La Web no requiere de rescates (la Web ha mantenido su marcha bastante bien, aún antes de que Java saliera a cuadro, muchas gracias. Pero Java fue capaz de agregar mucha mayor funcionalidad a las páginas planas de la Web).

¿Páginas Web planas? Antes de Java, las páginas Web estaban limitadas a un tipo de comunicación unidireccional con sus usuarios. Las páginas Web enviaban información a los usuarios, pero éstos no lo hacían hacia la Web. De forma específica, las páginas Web desplegaban información para ser leída por los usuarios, pero los usuarios no podían introducir información para que fuera procesada por las páginas Web. De pronto, la comunidad Web se dio cuenta de cómo los programas Java contenidos en las páginas Web abrían la puerta a páginas Web más excitantes; las páginas Web con programas Java eran capaces de leer y procesar datos del usuario y le proporcionaban una experiencia interactiva más placentera.

Java hoy en día

Hoy en día, los programadores utilizan Java en muchos ambientes diferentes. Aún insertan programas Java en las páginas Web, que se denominan *applets*. La popularidad inicial de los applets ayudó a posicionar a Java como uno de los lenguajes de programación líderes en el mundo. A pesar de que los applets aún tienen un papel significativo en el éxito actual de Java, otros tipos de programa Java han venido a superarlos en términos de popularidad.

Para ayudar al lector con la pequeña charla que tendrá en su siguiente evento social de Java, hacemos una breve descripción de algunos de los usos más populares de Java. Un applet es un programa que está contenido en una página Web. Un *servlet* es un programa Java que auxilia a una página Web, pero se ejecuta en una computadora diferente a la de la página Web. Una *JavaServer Page* (JSP) es una página Web que tiene fragmentos de un programa Java (no un programa Java completo, tal como un applet) contenido en el mismo. Una ventaja de los servlets y los JSP sobre los applets es que los dos primeros producen páginas Web que se despliegan de forma más rápida. Una *aplicación Java Micro Edición* (ME) es un programa de Java que se ejecuta en un reducido número de dispositivos, por ejemplo, uno que tiene una cantidad limitada de memoria. Ejemplos de dispositivos con recursos limitados son los enseres de consumo, tales como teléfonos celulares y televisores con decodificador de señales digitales. Una *aplicación Java Edición Estándar* (SE) es un programa Java que se ejecuta en una computadora estándar (una de escritorio o una portátil). Este libro se enfoca a aplicaciones SE como distintas a otros tipos de programas de Java, porque las aplicaciones SE son las más extendidas y proporcionan el mejor ambiente para aprender los conceptos de programación.

1.8 Primer programa: Hola mundo

Anteriormente se aprendió lo que significa compilar y ejecutar un programa de Java, pero el aprender sólo mediante la lectura no conduce a mucho. Es ahora tiempo de aprender a hacerlo. En esta sección, se aprenderá a introducir un programa Java en una computadora, compilarlo y ejecutarlo. ¡Qué divertido!

Ambientes de desarrollo

Existen diferentes formas de introducir un programa Java en una computadora. Se puede utilizar un ambiente de desarrollo integrado o un editor de texto plano. Se describirán brevemente ambas opciones.

Un *ambiente de desarrollo integrado* (*IDE*, por sus siglas en inglés) es más que una larga pieza de software, que nos permite introducir, compilar y ejecutar programas. La introducción, compilación y ejecución son parte del desarrollo de un programa y están integradas juntas en un ambiente, de ahí el nombre "ambiente de desarrollo integrado". Algunos IDE son gratuitos y algunos bastante caros. En el sitio Web del libro se proporcionan algunos tutoriales de varios IDE populares.

Un *editor de texto plano* es una pieza de software que permite al usuario introducir y salvar texto a manera de archivos. Los editores de texto plano no saben acerca de compilación y ejecución de programas. Si se utiliza un editor de texto plano para introducir un programa, será necesario utilizar herramientas de software separadas para compilar y ejecutar el programa. Es importante mencionar que los *procesadores de palabra*, como Microsoft Word, pueden llamarse editores de texto, pero no lo son de texto "plano". Cuando un procesador de palabras almacena el texto en un archivo, agrega caracteres ocultos que proporcionan formatos para el texto, tales como la altura de la línea, color, etc. Estos caracteres ocultos crean problemas a los programas de Java. Si el usuario intenta introducir un programa en una

computadora usando un procesador de palabras, el programa no compilará con éxito y por consiguiente no se ejecutará.

Existen diferentes tipos de editores de texto para los diferentes tipos de computadoras. Por ejemplo, las computadoras que utilizan Windows tienen un editor de texto plano llamado bloc de notas; las que utilizan Unix o Linux, tienen el vi, y las del sistema operativo Mac OS X, el TextEdit. Nota: Windows, UNIX, Linux y Mac OS X son *sistemas operativos*. Un sistema operativo es una colección de programas cuyo propósito es ayudar a ejecutar un sistema de cómputo. Al ejecutar el sistema de cómputo, el sistema operativo maneja la transferencia de información entre los componentes de la computadora.

En el resto de esta sección se describirá cómo introducir, compilar y ejecutar un programa utilizando herramientas simples. Se utilizará un editor de texto plano para teclear el programa, y herramientas simples de Sun para compilar y ejecutarlo. Si el usuario no tiene interés en utilizar herramientas simples y prefiere hacer uso de un IDE, deberá entonces dirigirse a los tutoriales de IDE que se tienen en el sitio Web del libro y omitir libremente la lectura de esta sección. Si no está seguro sobre qué hacer, se le sugiere que intente el uso de herramientas simples, que son libres y no requieren tanta memoria como los IDE. Sirven como una base estándar que puede utilizarse en casi cualquier computadora.

Captura de un programa de cómputo

A continuación se describe cómo introducir un programa en una computadora utilizando el bloc de notas que viene con todas las versiones de Microsoft Windows.

Mover el cursor del mouse al botón Inicio. Dar clic con el mouse en el botón Inicio que aparece en la parte inferior izquierda de Windows (cuando se indica dar "clic" a un elemento, lo que se quiere decir es que hay que mover el mouse hacia la parte superior del elemento y presionar el botón izquierdo). Aparecerá otro menú. En éste, dar clic en la opción Programas. Aparecerá otro menú. En éste, dar "clic" en la opción Accesorios. Aparecerá otro menú. En éste, dar "clic" en la opción bloc de notas. En seguida aparecerá el editor de textos bloc de notas.

En el nuevo editor de textos bloc de notas, capturar el código para el primer programa. De manera más específica, dar clic en cualquier parte en medio de la ventana del bloc de notas e introducir las siete líneas de texto tal como se muestra en la figura 1.6. Cuando se introduzca el texto, es importante asegurarse de que el tipo de letras mayúsculas y minúsculas coincidan exactamente con las de la figura. Por ejemplo, introducir `Hola` con la `H` en mayúsculas y las `-o`, `-l` y `-a` en minúsculas. Utilizar espacios, no tabuladores para la sangría del texto. La captura del texto comprende el código fuente de lo que se conoce como programa Hola mundo, que es tradicionalmente el primer programa de cualquier estudiante de programación. Dicho programa simplemente imprime un mensaje de saludo. En el capítulo 3, se describirá el significado detrás de las palabras en el código fuente. En este capítulo el interés se centra en ofrecerle experiencia al lector y mostrarle cómo capturar, compilar y ejecutar el programa Hola mundo.

Después de introducir el código fuente en la ventana del bloc de notas, se requiere que el usuario guarde su trabajo, almacenándolo en un archivo. Para salvar el código fuente en un archivo, dar clic en el menú Archivo que se localiza en la esquina izquierda de la ventana. Aparecerá un menú. En este menú, dar clic en la opción Salvar como, entonces aparecerá el *cuadro de diálogo* Salvar como. Un cuadro de diálogo es una pequeña ventana que ejecuta una tarea. La tarea de este cuadro de diálogo es almacenar un archivo.

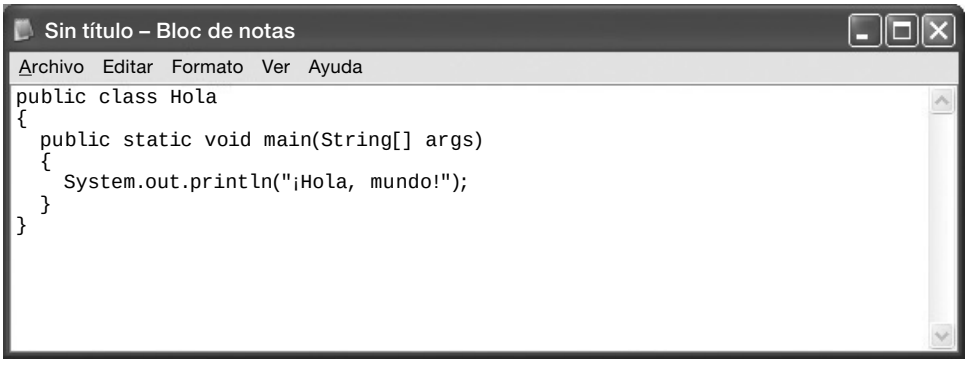


Figura 1.6 El editor de textos bloc de notas donde se captura el programa.

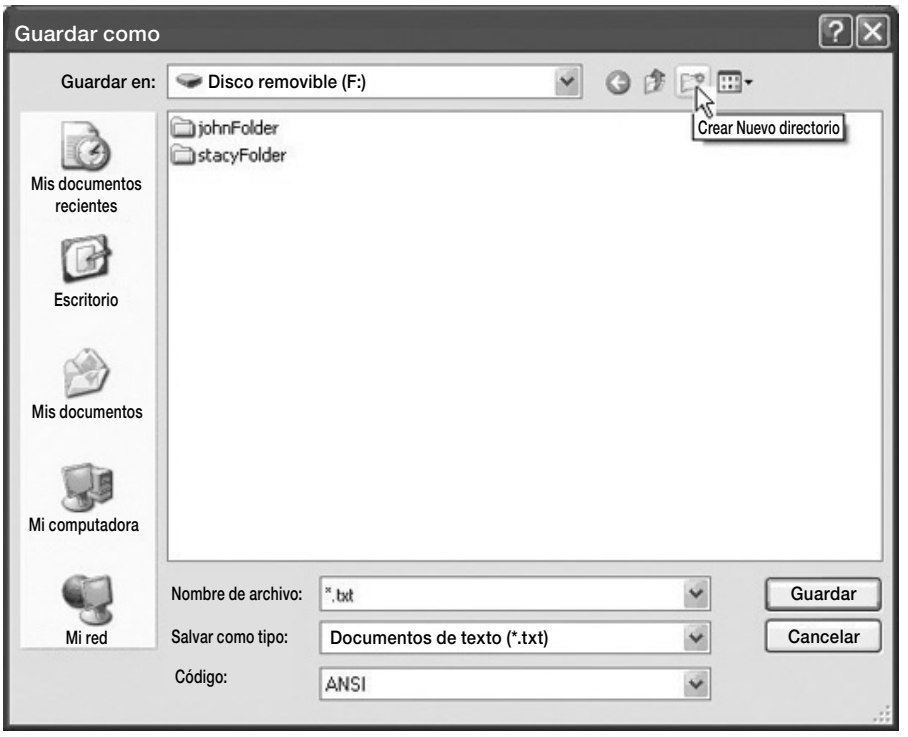
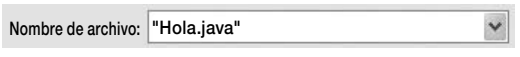


Figura 1.7 Bloc de notas. Cuadro de diálogo Salvar como con el usuario a punto de crear un nuevo directorio.

Observar el cuadro Nombre del archivo: en la parte inferior del cuadro de diálogo, ahí es donde se introduce el nombre del archivo; pero antes de eso es necesario crear un *directorio* para guardar el archivo. Un directorio, también llamado *fólder*, es una entidad organizacional que contiene un conjunto de archivos y otros directorios.[4] Mover el cursor del mouse sobre la flecha hacia abajo (▾) que se encuentra en la parte superior central del cuadro de diálogo Salvar como. Aparecerá un árbol de directorios bajo el cuadro con la flecha hacia abajo. En el árbol de directorios, mover el mouse hacia la parte de arriba del ícono C: (si es que se desea guardar el archivo en el disco duro), o hasta arriba de los íconos E: o F: si se prefiere guardar en la memoria USB. Al dar clic en el ícono de la letra adecuada, la letra del ícono seleccionado aparecerá en el cuadro Guardar en, al lado de la flecha hacia abajo. Verificar que el cuadro de diálogo Salvar como sea similar al que se presenta en la figura 1.7. En particular, observar la unidad F: de la figura 1.7. El cuadro Salvar en: puede ser un poco diferente, dependiendo de la unidad seleccionada.

Como se muestra en la figura 1.7, mover el cursor del mouse sobre el ícono Crear nuevo directorio al lado de la esquina superior derecha del cuadro de diálogo Salvar como. Dar clic en el ícono, en seguida aparecerá un nuevo directorio en el árbol. El nombre del nuevo directorio es `Nuevo Directorio` por omisión. El nombre del directorio `Nuevo Directorio` debe aparecer resaltado. Introducir `misPrgsJava`, con lo que dicho nombre reemplazará al de `Nuevo Directorio`. Dar clic al botón Abrir en la esquina inferior izquierda del cuadro de diálogo. Esto hará que el nuevo directorio `misPrgsJava` aparezca en el cuadro Salvar en:.

Teclear `"Hola.java"` en el cuadro Nombre de archivo: en la parte inferior del cuadro de diálogo. Debe introducirse `"Hola.java"` exactamente como se muestra a continuación:



[4] En los ambientes Windows y Macintosh, la gente tiende a utilizar el término "fólder", mientras que en Unix y Linux se emplea el vocablo "directorio". Como se verá en el capítulo 15, Sun utiliza el término "directorio" como parte del lenguaje de programación Java. En este libro se pretende seguir a Sun, y por tanto se utilizará el término "directorio" en lugar de "fólder".

No olvidar las comillas, la H en mayúscula y las siguientes letras en minúsculas. Dar clic en Guardar, en la esquina inferior derecha del cuadro de diálogo, hará que desaparezca el cuadro de diálogo Salvar como, y que la parte superior de la ventana del bloc de notas diga Hola.java. Cerrar el bloc de notas dando clic a la X que aparece en la esquina superior derecha de la ventana.

Instalación de un compilador Java y de la JVM

En la subsección anterior, se capturó el programa Hola mundo y se guardó en un archivo. ¡Excelente! Por lo regular, el siguiente paso sería compilar el archivo. Hay que recordar que la compilación es traducir un archivo de código fuente a un archivo de código byte. Para el programa Hola mundo el compilador traducirá un archivo de código fuente a archivo de código byte. Para el programa Hola mundo, el compilador traducirá el código fuente `Hola.java` a un archivo de código byte `Hello.Class`. Si se trabaja en el laboratorio de una escuela de cómputo, es probable que ya se tenga instalado el compilador de Java en los equipos. Si la computadora del lector no tiene instalado el compilador, se requerirá instalarlo ahora para completar la parte de preparación de esta sección.

Por lo regular, quien está interesado en la instalación del compilador (para compilar Java) también lo está en instalar la JVM (para ejecutar los programas). Para facilitar la instalación, Sun puso juntos el compilador de Java y la máquina virtual de Java. Sun denomina a este software en conjunto, JDK (*Java Development Kit*, siglas en inglés de kit de desarrollo de Java).

Para instalar el JDK en su equipo de cómputo, el usuario debe seguir las instrucciones de instalación del mismo en el sitio Web del libro: http://www.mhhe.com/dean y dar clic en las instrucciones del vínculo JDK Installation Instructions. Leer las instrucciones e instalar el JDK como se explica. En particular, seguir las instrucciones que describen cómo establecer el conjunto de la variable de ambiente PATH de manera permanente.

Compilación de un programa de Java

Se describirá cómo compilar un programa en la *ventana de línea de comandos* (también denominada *consola*). Una ventana de línea de comandos le permite al usuario introducir instrucciones del sistema operativo con palabras específicas. Las palabras se denominan *comandos*. Por ejemplo, en una computadora con el sistema operativo Windows, el comando para borrar un archivo es `del` (del inglés *delete*, suprimir). En una con sistema operativo Unix o Linux, el mismo comando sería `rm` (del inglés *remove*, borrar).

Para abrir una ventana de línea de comando en una computadora con el sistema operativo Windows, dar clic al botón Inicio en la esquina inferior izquierda del escritorio. En seguida aparecerá un menú. En el menú, seleccionar la opción Ejecutar, aparecerá el cuadro de diálogo Ejecutar; posicionarse en el cuadro Abrir: y teclear cmd (cmd significa "comando") y presionar el botón Aceptar. Una vez hecho esto, aparecerá la línea de comandos. La figura 1.8 muestra la nueva ventana de línea de comandos abierta.

En la figura 1.8 se observa la siguiente línea:

```
C:\Documents and Settings\John Dean>
```

Figura 1.8 Ventana de línea de comandos al abrirse.

El mensaje de compilación de error dice algo como lo siguiente:	Explicación:
`'javac' no es un comando reconocido`	Los tres mensajes de error indican que la computadora no entiende el comando `javac` porque no puede encontrar el programa de compilación javac. El error se debe probablemente a que la variable PATH se asignó de manera inapropiada. Revisar las instrucciones de instalación y modificar la variable PATH de manera adecuada.
`javac: comando no encontrado`	
`comando o nombre de archivo erróneo`	
`Hola.java:` *número: texto*	Existe un error de sintaxis en el código fuente. El *número* indica línea en el código donde se presenta el error. El *texto* proporciona una explicación al error. Revisar el contenido de archivo `Hola.java` y asegurarse de que cada carácter se haya tecleado correctamente y que se esté utilizando la tipografía adecuada (mayúsculas y minúsculas).

Figura 1.9 Errores de compilación y sus explicaciones.

Ésta es la línea de comandos. La línea de comandos nos solicita hacer algo. Para un símbolo del sistema de Windows, el símbolo indica introducir un comando. Más adelante, usted introducirá comandos en esta ventana. Pero antes hay que observar el símbolo >. El texto `C:\Documents and Settings\John Dean` conforma la *ruta* en el directorio actual. Una ruta especifica la ubicación de un directorio. De manera más específica, una ruta inicia con una letra correspondiente a la unidad de disco y contiene una serie de uno o más nombres de directorios separados por una diagonal. En el ejemplo que se presenta `C:` se refiere a la unidad de disco duro, `Documents and Settings` se refiere al directorio del mismo nombre que se encuentra en el disco duro, y `John Dean` es el directorio `John Dean` que se encuentra ubicado a su vez dentro del directorio `Documents and Settings`.

Para compilar el programa Hola mundo, es necesario primero ir a la unidad y directorio donde reside el mismo. Suponiendo que la ventana de la línea de comandos indica que la unidad actual es C: y que el lector guardó la clase `Hola.java` en F:, entonces tendrá que cambiarse a la unidad F:; al hacerlo así, aparecerá `f:` en la ventana de línea de comandos.

Para cambiarse al directorio del programa Hola mundo, introducir el siguiente comando `cd` (cd significa change directory, cambiar de directorio):

```
cd \misPrgsJava
```

Ahora que el lector está listo para compilar el programa, deberá introducir el siguiente comando de Java (`javac` significa java compile [compila java]).

```
javac Hola.java
```

Al introducir este comando, si la ventana de línea de comandos despliega un mensaje de error, consultar la tabla de la figura 1.9 para posibles soluciones. Si la línea de comandos no despliega mensajes de error, esto indica éxito. De forma más específica, indica que el compilador creó un archivo en código byte denominado `Hola.class`. Para ejecutar el archivo `Hola.class` introducir el siguiente comando de Java:

```
java Hola
```

La ventana de línea de comandos debe desplegar ahora la salida: `¡Hola, mundo!` La figura 1.10, muestra la ventana de línea de comandos después de completar los pasos descritos arriba.

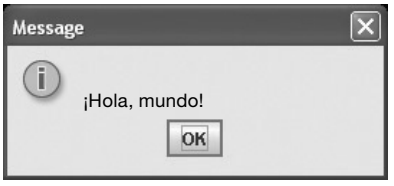
1.9 Apartado GUI: Hola mundo (opcional)

Esta sección es la primera entrega del apartado opcional de la interfaz gráfica del usuario (GUI), aquí se hará una breve introducción al concepto GUI. Por ejemplo, en esta sección se describe cómo desplegar un mensaje en una ventana GUI. En otra sección del apartado GUI, se describe cómo dibujar líneas y figuras. No hay problema para los lectores que no tengan tiempo de estudiar ese apartado. Cualquier sec-

Figura 1.10 Compilación y ejecución del programa Hola mundo.

ción relacionada con el mismo se puede omitir, ya que el material que se cubre es independiente de los demás temas. Se debe mencionar que la parte de GUI se cubre con más detalle al inicio y al final de los capítulos 16 y 17. El material de dichos capítulos es independiente del material que se presenta en el apartado GUI, por lo que una vez más se reitera, el lector puede omitir ese apartado. Pero se recomienda su lectura: ¡La programación GUI es muy divertida!

En esta sección se presenta una versión gráfica del programa Hola mundo. Se inicia mostrando la salida que genera.



Un *programa GUI* se caracteriza por utilizar herramientas gráficas para su interfaz. Se considera un programa porque utiliza estas herramientas gráficas para su interfaz: una barra de título (la barra en la parte superior de la ventana), un botón de cerrado del mismo (la "X" en la esquina superior derecha), un botón de Aceptar y un ícono i. Si se presiona el botón de cerrado de la ventana o el de Aceptar, la ventana se cierra. El icono i es un elemento visual que indica la naturaleza de la ventana: la i significa "información", pues la ventana simplemente muestra información.

Véase la figura 1.11. Las cajas punteadas indican que el código difiere de aquel que se presenta en el programa de la sección previa. Por ahora, la meta es solamente divertirse y obtener un poco de experiencia con este pequeño ejercicio.

```
import javax.swing.JOptionPane;
public class HolaGUI
{
  public static void main(String[] args)
  {
    JOptionPane.showMessageDialog(null, "¡Hola, mundo!");
  }
}
```

Las cajas punteadas indican código que difiere del código de la sección previa `Hola mundo`.

Figura 1.11 Versión GUI del programa Hola mundo.

El lector debe continuar y capturar el programa en un editor de textos, si necesita recordar cómo hacerlo, deberá consultar la sección anterior. En esta ocasión, el archivo del código fuente deberá salvarse como `HolaGUI.java`, asegurándose de utilizar la tipografía adecuada (mayúsculas) para las letras *H, G, U* e *I*, ya que `HolaGUI` es como se escribirá en el programa a partir de la tercera línea. A continuación, se deseará compilar y ejecutar el programa. Repetimos, si es necesario recordar estos pasos, conviene consultar la sección anterior.

Resumen

- Un sistema de cómputo es el conjunto de todos los componentes necesarios para que una computadora opere y las conexiones entre dichos componentes. De forma más específica, el sistema de cómputo lo integran la CPU, la memoria principal, la memoria auxiliar y los dispositivos de E/S.
- Los programadores escriben los algoritmos como un primer intento de solución a los problemas de programación.
- Los algoritmos son escritos con seudocódigo, que es similar a los lenguajes de programación, excepto que no se requiere de sintaxis precisa (palabras, gramática).
- El código fuente es el término formal para las instrucciones de programación.
- El código objeto es un conjunto de instrucciones en formato binario que pueden ser ejecutadas directamente por una computadora.
- La mayoría de los compiladores no-Java compilan del código fuente al objeto.
- Los compiladores Java compilan del código fuente al código byte.
- La forma en que un programa Java se ejecuta, es que la máquina virtual de Java (JVM) traduce el código byte del programa a código objeto.
- En un principio, Sun desarrolló Java para utilizarlo en aparatos electrodomésticos.
- Para agilizar el desarrollo, los programadores de Java a menudo utilizan ambientes integrados de desarrollo (IDE), pero se puede hacer uso de un simple editor de textos planos y de la línea de comando.

Preguntas de revisión

§1.2 Terminología de hardware

1. ¿Qué significan las siguientes abreviaturas?
- ***a***) E/S
- ***b***) CPU
- ***c***) RAM
- ***d***) GHz
- ***e***) MB

2. Nombre dos importantes dispositivos de cómputo de entrada.

3. Identifique dos importantes dispositivos de cómputo de salida.

4. Aseveraciones
- ***a***) La memoria principal es más rápida que la memoria auxiliar (C/F)
- ***b***) La memoria auxiliar es volátil (C/F)
- ***c***) La primera posición en la memoria principal es en la dirección 1 (C/F)
- ***d***) La CPU se considera un dispositivo periférico (C/F)
- ***e***) Conexión como dispositivo virtual (*Hot swapping*) es la conexión de un dispositivo en una computadora mientras ésta se encuentra prendida (C/F)

§1.3 Escritura de algoritmos mediante seudocódigo

5. ¿Qué es un algoritmo?

6. ¿Qué es el seudocódigo?

§1.4 Traducir seudocódigo a código de lenguaje de programación

7. ¿Las reglas de sintaxis son más estrictas para el seudocódigo o para el código de lenguaje?

§1.5 Compilación de código fuente en código objeto

8. ¿Qué sucede cuando se compila un programa?
9. ¿Qué es código objeto?

§1.6 Portabilidad

10. ¿Qué es la máquina virtual de Java?

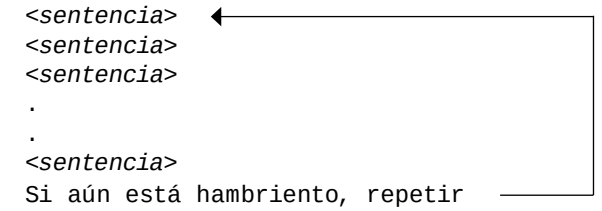
§1.7 Surgimiento de Java

11. Enumere cinco diferentes tipos de programas Java.

Ejercicios

1. [Después de §1.2] Para cada uno de los siguientes elementos, determinar si se encuentra asociado con la memoria principal o con la memoria auxiliar.
   a) Disco flexible ¿principal o auxiliar?
   b) RAM ¿principal o auxiliar?
   c) Disco duro ¿principal o auxiliar?
   d) CD-RW ¿principal o auxiliar?
2. [Después de §1.2] ¿Qué es un bit?
3. [Después de §1.2] ¿Qué es un byte?
4. [Después de §1.2] ¿A qué tipo de componente usualmente se refiere la unidad `C:`?
5. [Después de §1.2] Para cada uno de los siguientes componentes de un sistema de cómputo, identifique los componentes equivalentes en el sistema biológico de un oso.
   a) CPU
   b) Dispositivos de entrada
   c) Dispositivos de salida
6. [Después de §1.2] ¿Cuál es la "Ley de Moore"? La respuesta a esta pregunta no se encuentra en el libro, debe buscarse en Internet. (*Sugerencia:* Gordon Moore fue uno de los fundadores de Intel.)
7. [Después de §1.3] Esta pregunta no es muy específica, por lo que no hay necesidad de responder a manera de receta. Únicamente lo que parezca razonable al lector.

   A manera de sentencias cortas y mediante uso de seudocódigo, proporcione un algoritmo para un oso que intenta obtener miel. Si requiere repetir uno o más pasos, utilice sentencias condicionales y una flecha que muestre la repetición. Por ejemplo, el algoritmo podría contener algo como lo siguiente:



8. [Después de §1.5] Las personas por lo regular prefieren trabajar con código fuente en lugar de código objeto porque es más fácil de entender. Entonces ¿por qué se requiere del código objeto?
9. [Después de §1.6] La mayoría de los lenguajes compilan a nivel código objeto. Java compila a nivel de código byte. ¿Cuál es el beneficio primario del código byte sobre el código objeto?
10. [Después de §1.6] ¿Qué hace la máquina virtual de Java?
11. [Después de §1.7] ¿Cuál es el origen del nombre del lenguaje de programación Java?
12. [Después de §1.8] En una computadora cuyo sistema operativo es una versión reciente de *Hola.class* Microsoft Windows, invocar Inicio > Programas de aplicación > Accesorios > Línea de comandos. Navegar al directorio que contiene el código fuente del programa `Hola.java`. Introducir `dir Hello.*` para listar todos los archivos que inicien con "Hola". Si la lista incluye el archivo, borrarlo mediante el siguiente comando *del* `Hola.class`. Introducir la siguiente instrucción `javac Hola.java` para compilar el código fuente. Una vez más, introducir `dir Hola.*` y verificar que el archivo de código byte `Hola.class` ha sido creado. Ahora el usuario puede teclear `java Hola` para ejecutar el programa compilado. Introduzca `type Hola.java` y después `Hola.class` para verificar cómo difieren ambos tipos de código.
13. [Después de §1.8] Experimentar con el programa `Hola.java` para aprender el significado de una compilación típica y de mensajes de errores de ejecución:
   a) Omitir el último / del bloque de encabezado.

b) Omitir cualquier parte del argumento en el paréntesis después de `main`.
c) Omitir el punto y coma al final de cada sentencia de salida.
d) Uno a la vez, omitir los corchetes { y }.
e) Intentar utilizar minúsculas, `$, _,` o un número como primer carácter en el nombre de la clase.
f) Escribir un nombre diferente al del nombre de la clase para el nombre del archivo.
g) Cambiar `main` por `Main`.
h) Uno a la vez, intentar omitir las palabras `public, static` y `void` antes del `main`.

14. [Después de §1.8] Aprender a utilizar el bloc de notas trabajando a su ritmo con el tutorial "Getting Started" en el sitio Web del libro. Enviar una copia en papel del código fuente del programa CuentaRegresiva (imprimir el programa dentro del bloc de notas). Observar que no se requiere enviar el código fuente del programa Hola mundo o enviar la salida de ninguno de ellos.

Solución a las preguntas de revisión

1. ¿Qué significan las siguientes abreviaturas?
a) E/S: dispositivos de entrada/salida.
b) CPU: unidad central de proceso o procesador.
c) RAM: memoria de acceso aleatoria o memoria principal.
d) GHz: Gigahertz = mil millones de ciclos por segundo.
e) MB: Megabyte = millón de bytes, donde un byte es equivalente a ocho bits, y un bit es la respuesta a una simple pregunta sí/no.

2. El teclado y el mouse son dos de los ejemplos más obvios de dispositivos de entrada. Otro dispositivo de entrada es el módem telefónico.

3. El monitor y una impresora son dos de los ejemplos más obvios de dispositivos de salida. Otros ejemplos son un módem telefónico y las bocinas.

4. Aseveraciones
a) Cierto. La memoria principal está físicamente más cercana al procesador, y el bus que conecta la memoria principal con el procesador es más rápido que el bus que conecta la memoria auxiliar al procesador. La memoria principal es más cara, y por tanto, usualmente más pequeña.
b) Falso. Cuando la computadora se apaga, la memoria principal pierde su información, mientras que la memoria auxiliar no lo hace. Una falla inesperada en la energía eléctrica podría, sin embargo, corromper la información en la memoria auxiliar.
c) Falso. La primera posición en la memoria principal es en la dirección 0.
d) Falso. La CPU es parte de la computadora misma, no es un dispositivo periférico.
e) Cierto. La conexión como dispositivo virtual (*Hot swapping*) es cuando se conecta un dispositivo en una computadora mientras ésta se encuentra prendida.

5. Un algoritmo es un procedimiento paso a paso para resolver un problema.

6. El seudocódigo es un lenguaje informal que utiliza términos en inglés para describir los pasos de un programa.

7. Las reglas de sintaxis son menos estrictas para el seudocódigo (en comparación con el código de lenguaje de programación).

8. La mayoría de los compiladores convierten el código fuente en código objeto. Los compiladores Java convierten el código byte en código objeto.

9. El código objeto es el término formal para las instrucciones en formato binario que un procesador puede leer y entender.

10. Una máquina virtual de Java (JVM) es un intérprete que traduce el código byte en código objeto.

11. Cinco tipos diferentes de programas Java son: applets, servlets, páginas JSP, aplicaciones microedición y aplicaciones edición estándar.

CAPÍTULO 2

Algoritmos y diseño

Objetivos

- Aprender a escribir descripciones textuales informales de lo que se desea que un programa de cómputo haga.
- Entender cómo un diagrama de flujo describe lo que hace un programa.
- Familiarizarse con patrones de control estándar bien estructurados.
- Aprender cómo estructurar ejecuciones condicionales de estructura.
- Aprender cómo estructurar y terminar operaciones de bucle, incluyendo bucles anidados.
- Aprender cómo realizar el "trazado paso a paso" de la secuencia de operación de un programa.
- Entender cómo se puede describir la operación de un programa con diferentes niveles de detalle.

Relación de temas

2.1 Introducción

Como se indicó en el capítulo 1, la escritura de un programa de cómputo involucra dos actividades básicas: 1) determinar lo que se quiere hacer y 2) escribir el código que lo hará. El usuario estará tentado a saltarse el primer paso y avanzar de forma inmediata al segundo: la escritura del código. Hay que tratar de resistirse a esa tentación. El pasar inmediatamente a la codificación trae como consecuencia programas deficientes que trabajan pobremente y que son difíciles de arreglar, debido a que una pobre organización de los mismos complica su entendimiento. Por tanto, para los problemas más simples, vale más empezar por pensar en lo que se quiere hacer y organizar nuestros pensamientos.

Como parte del proceso de organización, se deseará escribir un *algoritmo*.[1] Un algoritmo es una secuencia de instrucciones para resolver un problema. Es una receta. Cuando se especifica un algoritmo, por lo común, se tienen dos tipos de formato:

1. El primer formato utiliza un código de lenguaje natural denominado *seudocódigo*, donde el prefijo "seudo" significa "ficticio o fingido", por lo que no es un código "real". El seudocódigo, al igual que el código real está compuesto de una o más *sentencias*. Una sentencia es el equivalente a las sentencias u oraciones que se emplean en el lenguaje natural. Si la sentencia es simple, por lo regular ocupará una línea, pero si es compleja, entonces se puede extender a más de una. Las sentencias pueden agruparse dentro de otra(s) como en una especie de jerarquía. Se utilizará mucho el término "sentencia", y el usuario tendrá un mejor aprecio del mismo conforme se avance en la lectura del libro.
2. El segundo formato es un arreglo de cajas y flechas que ayudan al usuario a avanzar visualmente a través del algoritmo. La forma más detallada de cuadros y flechas se denomina *diagrama de flujo.* Los cuadros en un típico diagrama de flujo contienen sentencias cortas y son muy similares a las que pueden observarse en las sentencias del seudocódigo.

Este capítulo explica cómo aplicar seudocódigo y diagramas de flujo a un conjunto estándar de problemas de programación: problemas que aparecen en la mayoría de los programas grandes. El capítulo muestra también cómo analizar paso a paso un algoritmo (de sentencia en sentencia) para entender lo que se está haciendo. El objetivo es proporcionar al lector un conjunto de herramientas informales que pueda utilizar para describir lo que desea que su programa realice. Las herramientas le ayudan a organizar su pensamiento antes de empezar a capturar el programa actual. Las ayudas le permiten entender cómo un algoritmo (o un programa completo) funciona. Le permiten verificar correcciones e identificar problemas cuando las cosas no marchan bien.

2.2 Salida

El primer problema a considerar al desplegar los resultados finales de un programa es su salida. Esto puede sonar como algo a tomar en cuenta hasta el final, ¿por qué al principio? La salida es lo que el *usuario final*: el cliente, que es la persona que finalmente utilizará el programa, quiere. Ésa es la meta. El pensar acerca de la salida en un principio evita perder tiempo en la resolución del problema equivocado.

Hay que ponerse en el lugar del usuario.

Algoritmo Hola mundo

En el capítulo 1 se mostró un programa Java que generaba la impresión en pantalla de la cadena "¡Hola, mundo!" Ahora se volverá al programa, pero enfocándose en el algoritmo y no en el programa. Como se recordará, el algoritmo Hola mundo contiene siete líneas. La figura 2.1 muestra el algoritmo Hola mundo y su longitud es de sólo una línea de seudocódigo. El objetivo de un algoritmo es mostrar los pasos necesarios para resolver un problema sin empantanarse con los detalles de sintaxis. El algoritmo Hola mundo hace simplemente eso. Muestra una sola sentencia de impresión, la cual resuelve el único paso necesario para resolver el problema Hola mundo.

El mensaje de la figura 2.1 es una cadena literal. Una *cadena* es un término genérico que se refiere a una secuencia de caracteres. Una *cadena literal* es una cadena, cuyos caracteres están escritos de manera explícita y encerrados entre comillas. Cuando se escribe una cadena literal, se imprimen fielmente los caracteres, tal y como aparecen en el comando. Así, en la figura 2.1 el algoritmo imprime los caracteres ¡, H, o, l, a, coma, espacio, m, u, n, d, o y !

```
imprimir "¡Hola, mundo!"
```

Figura 2.1 Algoritmo Hola mundo que imprime el mensaje "¡Hola, mundo!"

[1] El término *algoritmo* proviene de Algoritmi, que es la forma latinizada del nombre acortado al-Khwarizmi, correspondiente al matemático persa del siglo IX, Muhammad ibn Musa al-Khwarizmi, a quien se considera el padre del álgebra.

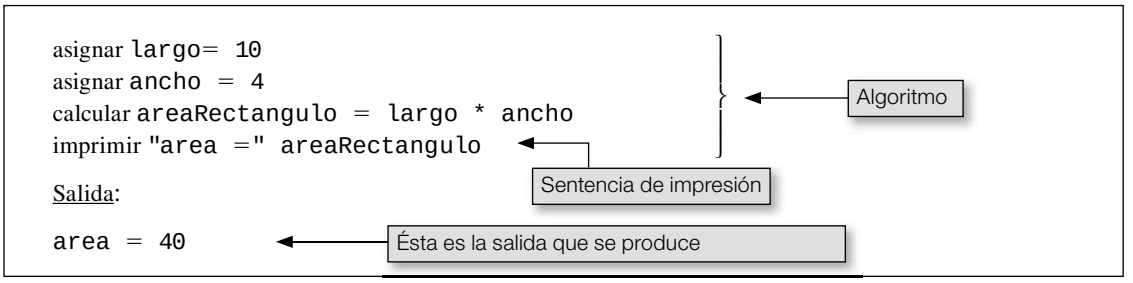


Figura 2.2 Algoritmo del rectángulo que imprime el área de dicha figura.

Algoritmo del rectángulo

Para el siguiente ejemplo hay que imaginar que se quiere desplegar el área de un rectángulo en particular. Primero hay que considerar lo que se desea que haga el programa. En la figura 2.2 se debe observar `area = 40` la línea bajo la leyenda: Salida, la cual muestra lo que el usuario desea que se muestre en la pantalla.

La parte superior de la figura 2.2 es el algoritmo para calcular el área de un rectángulo. Observar que algunas palabras tales como `largo` y `ancho` aparecen como letra monoespacio. La *letra monoespacio* es la que contiene caracteres con anchura uniforme. Se utiliza la letra monoespacio para indicar que algo es una variable. Una *variable* es un contenedor que almacena un valor. Las primeras dos líneas del algoritmo asignan 10 y 4 a las variables `largo` y `ancho` respectivamente. Lo anterior significa que la variable `largo` contiene el valor 10 y la variable `ancho` contiene el valor 4. La tercera línea describe dos operaciones: primero calcula el área multiplicando el `largo` por el `ancho` (el * es el signo de multiplicación). Después asigna el resultado (el producto) a la variable `areaRectangulo`. La cuarta línea imprime dos elementos: la cadena literal "área=" y el valor de la variable `areaRectangulo`. Cuando aparece una variable en la sentencia de impresión, se imprime el valor almacenado en la misma. La variable `areaRectangulo` contiene el valor 40, por lo que la sentencia de impresión imprime el número 40. La salida de la figura 2.2 muestra el despliegue deseado.

2.3 Variables

Ahora se estudiarán las variables con más detalle. En la figura 2.2, el algoritmo del rectángulo, aparecen tres variables: `largo`, `ancho` y `areaRectangulo`. En `areaRectangulo` verificar cómo se unen las dos palabras: "área" y "rectángulo", y observar cómo la segunda palabra se inicia con una letra en mayúscula. Se realiza esto para ayudar al usuario a desarrollar sus buenos hábitos para la codificación posterior en Java, la cual no permite ningún tipo de espacio dentro del nombre de una variable. Aunque no se necesita en seudocódigo, en Java, representa un buen estilo el iniciar el nombre de una variable con una letra minúscula, como en `areaRectangulo`. Si el nombre es una combinación de varias palabras, en Java deben omitirse el espacio(s) entre las diferentes palabras en un simple nombre, y deben iniciarse todas las palabras después de la primera de ellas con una letra mayúscula para hacer la combinación legible. Esta práctica de programación se denomina *joroba de camello* (*camelCase,* en inglés) por el bulto de en medio. Una vez más se reitera, no es necesario para el seudocódigo, pero representa un buen hábito a desarrollar. A continuación se presentan dos ejemplos que muestran cómo nombrar a las variables con joroba de camello:

Descripción	Un buen nombre de variable
nombre de un equipo de deportes	`nombreEquipo`
peso en gramos	`pesoEnGramos`

Las variables pueden almacenar diferentes *tipos* de datos. ¿Qué tipo de dato podría la variable `nombreEquipo` almacenar: numérico o de cadena de caracteres? Lo más probable es que sea utilizada para almacenar caracteres; por ejemplo, "Halcones", o "Piratas". ¿Qué tipo de dato podría la variable `pesoEnGramos` almacenar: numérico o de cadena de caracteres? Lo más probable es que se utilice para

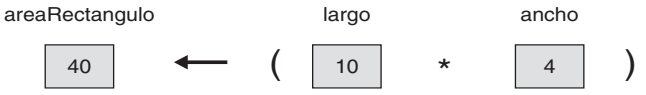


Figura 2.3 Operación de asignación representada por la flecha del lado izquierdo.

almacenar valores tales como 12.5. Es relativamente fácil para las personas determinar el tipo de datos de una determinada variable, tan sólo pensando en el nombre, pero este tipo de pensamiento es muy diferente para una computadora. Por tanto, en un programa real de Java, se debe especificar a la computadora el tipo de dato para cada elemento.

Sin embargo, ya que el seudocódigo es diseñado estrictamente para seres humanos y no para computadoras, en seudocódigo no es necesario molestarse en tales especificaciones. Observar que la representación del seudocódigo presentado en la figura 2.2 no contiene ninguna mención al tipo de datos. El seudocódigo ignora el tipo de datos, por lo que se enfoca en la esencia del algoritmo.

2.4 Operadores y sentencias de asignación

La sección anterior describió las variables en sí misma. Ahora se considerarán las relaciones mediante operadores y asignaciones.

Volviendo a la tercera sentencia del algoritmo del rectángulo de la figura 2.2:

```
calcular areaRectangulo = largo * ancho
```

Como se señaló anteriormente, el símbolo * se utiliza como operador de multiplicación. Los otros operadores aritméticos de uso común son: + para la suma, – para la sustracción y / para la división. Se asume que lo anterior es familiar para cualquier persona. Las variables `largo` y `ancho` son *operandos*. En matemáticas y, una vez más, en programación, un operando es una entidad (por ejemplo, una variable o un valor) que es operado por un operador (valga la redundancia). Las variables `largo` y `ancho` son operandos porque son operados por el operador *.

Cuando se dice "*Asignar* `variableA = x`" lo que se quiere decir es "asigna el valor de `x` a la variable `variableA`. Así, la sentencia "*Calcular* `areaRectangulo = largo * ancho`" asigna en la variable `areaRectangulo` el producto de las variables `largo * ancho`. Una imagen vale más que mil palabras. Observar la figura 2.3, la cual describe visualmente lo que la sentencia hace.

La figura 2.3 incluye un par de paréntesis no mostrados en las sentencias de seudocódigo, mismos que pueden ser colocados si así se desea; pero la mayoría de la gente espera que la operación de la multiplicación tenga una *precedencia* (que ocurra antes) que la operación de la asignación, por lo que se decidió no tomarse la molestia de incluir paréntesis en esta sentencia particular.

La figura 2.3 muestra que cada una de las tres variables es un contenedor de valor. La figura 2.3 también sugiere visualmente que la asignación va en dirección de derecha a izquierda. La asignación en el seudocódigo no tiene dirección, pero en el capítulo 3 se verá que la asignación en código Java va de derecha a izquierda. Por lo que si el lector es una persona que gusta de las cosas visuales, conviene que visualice cómo las asignaciones van de derecha a izquierda como se sugiere en la figura 2.3.

2.5 Entrada

El algoritmo precedente, del rectángulo, proporcionó los valores de las variables `largo` y `ancho`. Se hizo de esa manera para hacer la introducción tan simple como fuera posible. A veces esto resulta una estrategia apropiada, pero en este caso en particular, no tiene sentido porque el algoritmo resuelve el problema sólo para un conjunto de valores. Para hacer al algoritmo más general, en lugar de tener otro algoritmo que proporcionen los valores de `largo` y `ancho`, se debería permitir que el *usuario* (la persona que ejecuta el programa) proporcione dichos valores. El valor que un usuario proporciona para un programa se denomina *valor de entrada*, o simplemente *entrada*. La figura 2.4 presenta un algoritmo mejorado del algoritmo Rectángulo, donde las entradas `largo` y `ancho` ejecutan operaciones de captura por parte del usuario.

imprimir "Introduzca una longitud en metros:"
leer `longitud`
imprimir "Introduzca una anchura en metros:"
leer `ancho`
asignar a `areaRectangulo = largo * ancho`
imprimir "El área es de" `areaRectangulo` "metros cuadrados"

Simple sesión:

```
Introduzca una longitud en metros: 10
Introduzca una anchura en metros: 4
El área es de 40 metros cuadrados.
```

Las entradas del usuario se presentan en cursivas.

Figura 2.4 Algoritmo del rectángulo que obtiene los valores de largo y ancho por parte del usuario.

Observar que las dos primeras sentencias de `impresión` en la figura 2.4 se denominan *indicativos* porque le dicen (o indican) al usuario lo que debe introducir. Sin los *indicativos*, la mayoría de los usuarios se quedarían con una sensación no placentera y con la duda sobre qué hacer.

A través de este libro, se presentan *sesiones ejemplo* como un medio para mostrar lo que sucede cuando un algoritmo o programa se ejecuta con un conjunto típico de entradas. Cuando es posible, se presenta esta sesión ejemplo en la figura con el algoritmo o programa que lo genera. ¿Puede identificar el lector los valores de entrada en las sesiones ejemplo de la figura 2.4? La convención que se sigue en la obra es la de poner en cursivas los valores ejemplo de entrada para distinguirlos de los de salida. Así, *10* y *4* son los valores de entrada del usuario.

Escribir lo que hará y cómo lo hará.

La combinación de seudocódigo del algoritmo y una sesión ejemplo representan una conveniente y eficiente forma de especificar un simple algoritmo o programa. La sesión ejemplo muestra el formato o valores deseados de entrada y salida. También se muestran valores numéricos representativos de entrada y salida, los cuales le permiten al programador verificar que su programa recién terminado se comporta como se planeó. En muchos proyectos del libro (los que se presentan en el sitio Web), se proporciona alguna combinación de seudocódigo y sesiones ejemplo para especificar el problema que se está solicitando resolver.

2.6 Flujo de control y diagrama de flujo

En las sesiones anteriores se describieron varias sentencias: de impresión, de asignación y de entrada; enfocándose en la mecánica de funcionamiento de la sentencia. Es tiempo de enfocarse en la relación entre sentencias. De manera más específica, se efectuará un enfoque en el *flujo de control*. El flujo de control es el orden en el que las sentencias son ejecutadas. En el análisis sobre el flujo de control, se hará referencia tanto a los algoritmos como a los programas. Los conceptos de flujo de control se aplican de manera igual en ambos casos.

El flujo de control es mejor explicado con la ayuda de diagrama de flujo. Los diagramas de flujo son útiles porque son como imágenes. Como tal, ayudan a visualizar la lógica de un algoritmo. Un diagrama de flujo utiliza dos símbolos básicos: 1) rectángulos, los cuales contienen comandos como los de impresión, asignación y lectura, y 2) diamantes, los cuales contienen preguntas de afirmación/negación. En cada diamante, el flujo de control se divide. Si la respuesta es "sí" el flujo se va para un lado y si es "no" se va para el otro.

Las cajas con líneas discontinuas en la figura 2.5 muestran las tres estructuras estándar para el flujo de control: la estructura secuencial, una estructura condicional y una estructura de ciclo o bucle. El diagrama de flujo a la izquierda (la estructura secuencial) es un retrato del algoritmo del rectángulo descrito en la figura 2.2. Las *estructuras secuenciales* contienen sentencias que se ejecutan en el orden/secuencia en la cual están escritos; por ejemplo, después de ejecutar una sentencia, la computadora ejecuta otra más que aparece inmediatamente después. Las *estructuras condicionales* contienen una pregunta de afirmación/negación. Y la respuesta a la pregunta determina si se ejecutan los subsiguientes bloques

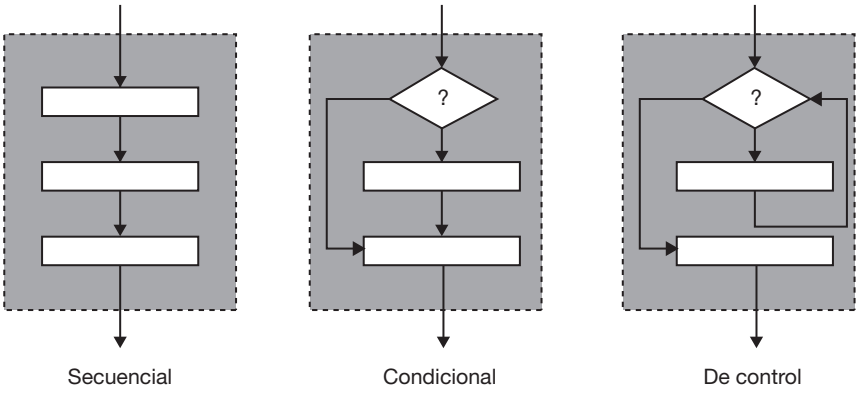


Figura 2.5 Flujo de control bien estructurado.

de sentencias o se saltan. Las *estructuras de control* también contienen respuestas a preguntas de afirmación/negación, y la respuesta a la pregunta determina si se repite el bloque de sentencias de control o si se mueve a las sentencias bajo el bucle.

La *programación estructurada* es una disciplina que requiere que los programas limiten su flujo de control a estructuras secuenciales, condicionales y de control. Un programa se considera bien estructurado si puede ser descompuesto en patrones como los de la figura 2.5. El lector debe esforzarse en los programas bien estructurados porque tienden a ser más fáciles de entender y trabajar con ellos. Para dar una idea clara sobre lo que no debe hacerse, observar la figura 2.6. Su flujo de control es malo porque hay dos puntos de entrada dentro del bucle, y cuando se está dentro del bucle, es difícil saber lo que sucedió anteriormente. Cuando un programa es difícil de entender, es propenso a errores y difícil de solucionar, al código que implementa un algoritmo como éste, a menudo se le denomina *código espagueti* porque cuando se dibuja un diagrama de flujo del código, el diagrama de flujo parece espagueti. Cuando usted vea un espagueti, ¡desenrédelo!

Además de las estructuras estandarizadas secuenciales, condicionales y de control también los problemas largos se pueden descomponer en pequeños subproblemas. En Java, se da una solución a cada subproblema mediante bloques de control denominados *métodos*. Sobre dichos métodos se hablará en el capítulo 5, pero por ahora, habrá que enfocarse en las estructuras de control que aparecen en la figura 2.5.

2.7 Sentencias if

En las secciones previas se describieron las sentencias de impresión, asignación y de entrada, se vieron ejemplos de estructuras de control secuencial en la parte izquierda de la figura 2.5. Al avanzar a través de

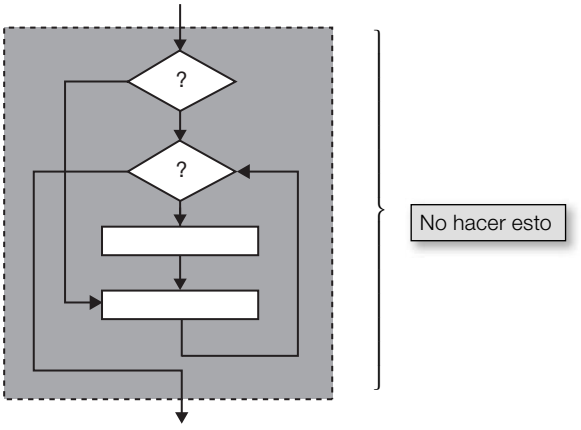


Figura 2.6 Flujo de control pobremente estructurado.

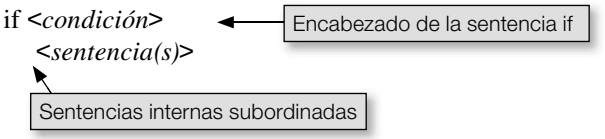
una secuencia de pasos, a veces nos encontramos con "piedras en el camino", por lo cual se debe señalar que se debe elegir el camino a seguir. Cuando un programa se topa con una piedra en el camino, los programadores utilizan una *sentencia if* para evitarla. La sentencia if (si) hace una pregunta y la respuesta a dicha pregunta indica el camino a seguir. De manera más informal, la sentencia if contiene una *condición*. Una condición es una pregunta cuya respuesta es ya sea un sí o un no. La respuesta a dicha pregunta determina la secuencia a seguir inmediatamente después. A continuación se presentan tres formas para las sentencias if:

"if"
"if, else"
"if, else if"

Por ahora habrá que enfocarse en cada una de ellas de manera separada.

"if"

Primero, suponiendo un caso en que se quiere hacer algo o nada en lo absoluto. En ese caso, se debería utilizar la forma simple "if" de la sentencia if. He aquí su formato:



Observar los paréntesis en ángulo "< >" que rodean "condiciones" y "sentencias". A lo largo del libro, se utilizan las cursivas y la notación de los paréntesis en ángulo para elementos que requieren de una descripción. Así, cuando aparezca en el texto la cadena "<condición>" quiere decir que una condición como tal y no la palabra "condición" va a aparecer después de la palabra "if". De la misma manera, cuando se lea "<*sentencia(s)*>", lo que indica es que una o más sentencias y no la palabra "sentencia" van a aparecer abajo del encabezado if.

En la ilustración anterior, observar cómo las <*sentencia(s)*> tienen sangría. El seudocódigo emula el lenguaje natural mediante la sangría para mostrar encapsulamiento o subordinación. Las sentencias bajo cualquier encabezado if están subordinadas a la sentencia if porque se consideran parte de otra sentencia if más larga que las rodea. Debido a que están subordinadas, deben mantener sangría.

He aquí cómo funciona la forma más simple del "if":

- Si la condición es verdadera, ejecutar todas las sentencias subordinadas, esto es, ejecutar todas las sentencias con sangría que aparecen bajo el "if".
- Si la condición es falsa, saltar a la línea debajo de la última subordinada, esto es, saltar a la primera línea sin sangría abajo del "if".

A continuación se pondrán en práctica estos conceptos, mostrando una sentencia if en el contexto de un algoritmo completo. La figura 2.7, el algoritmo de la figura, solicita al usuario introducir el nombre una figura. Si el usuario introduce "círculo", el algoritmo solicita el radio de la misma, calcula el área de la figura utilizando dicho radio, e imprime el valor resultante. Finalmente, ya sea que el usuario haya tecleado "círculo" o no, el algoritmo imprime un mensaje amigable.*

Es importante considerar varios puntos en el algoritmo de la figura. La sentencia if verifica la condición "si la `figura` es un círculo", y controla si las sentencias subordinadas se ejecutan. Observar cómo el comando calcula área y las sentencias de impresión subsecuentes se imprimen como sentencias separadas. Eso es perfectamente aceptable y bastante común, pero es importante asegurarse de una implementación alternativa donde los dos comandos se fusionen en una sola:

imprimir "El área es " (3.1416 `*` `radio` `*` `radio`)

* A lo largo de este y otros capítulos se habla de las sentencias "if", "else"; sin embargo, el texto que se está incluyendo es seudocódigo, y como se señaló en el capítulo 1, el seudocódigo hace uso del lenguaje natural (en este caso del castellano), por lo que en lugar de utilizar un if como tal se utiliza la palabra "si", y en lugar de utilizar un "else" se presenta un "de lo contrario", y muchas otras palabras por el estilo, ya que el seudocódigo es para uso exclusivo del usuario, y no para la computadora, y se considera que referirnos en las explicaciones a dichas estructuras por su nombre en inglés es importante, porque cuando se inicie con el código Java propiamente dicho, así es como deberán de teclearse. (*Nota del traductor.*)

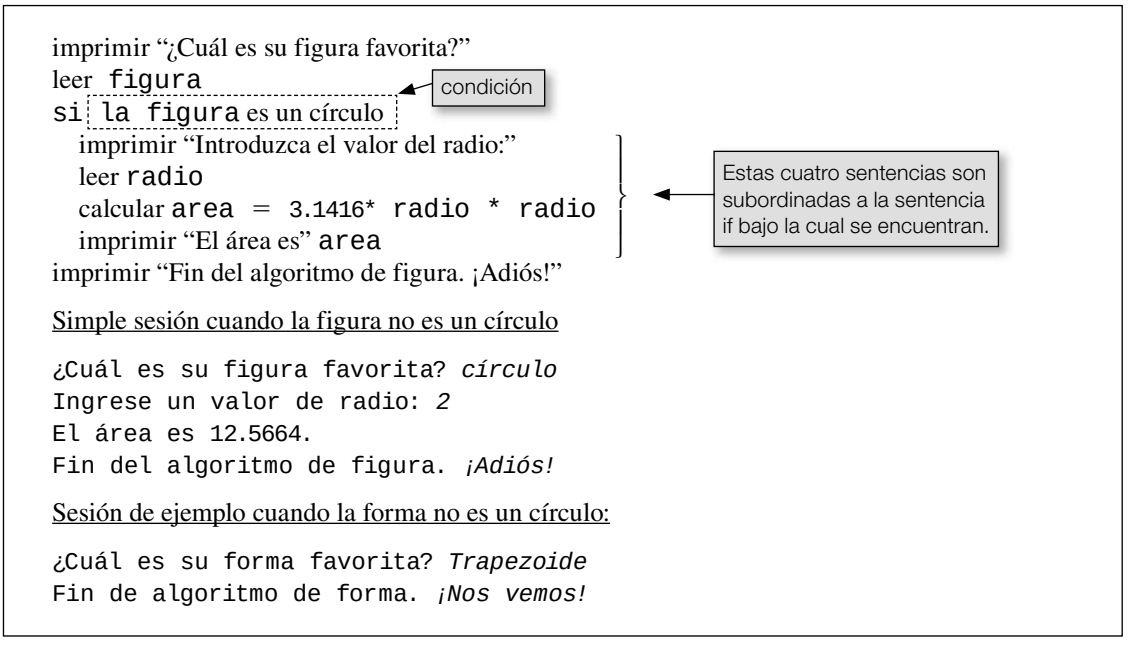


Figura 2.7 Algoritmo de la figura que calcula el área de un círculo (si la figura favorita es un círculo).

En este caso, se ponen paréntesis alrededor del cálculo matemático para enfatizar que se desea que la computadora imprima el resultado del mismo, en lugar de los valores individuales de las variables. Se pueden utilizar paréntesis todo el tiempo para especificar que las operaciones dentro del paréntesis deben ser hechas antes que aquéllas fuera de los mismos.

"if, else"

Ahora se pasará al segundo tipo de sentencia if: la forma "if else", en donde se desea hacer ya sea una u otra cosa. A continuación se presenta el formato:

if <*condición*>
    <*sentencia(s)*>
else
    <*sentencia(s)*>

Y a continuación, la forma en que el "if, else" funciona:

- Si la condición es verdadera, ejecutar las sentencias subordinadas al "if", y saltarse las sentencias subordinadas al "else".
- Si la condición es falsa, saltarse la sentencia(s) subordinada(s) al "if", y ejecutar todas las sentencias subordinadas al "else".

A continuación se ofrece un ejemplo que utiliza la sentencia con el formato "if, else":

si `calificación` es mayor o igual a 60
    imprimir "Aprueba"
de lo contrario
    imprimir "Reprueba"

Observar cómo se agrega sangría a la sentencia imprimir "Aprueba", ya que está subordinada a la condición if. Observar cómo se puede agregar sangría a la sentencia imprimir "Reprueba", ya que está subordinada al "else".

"if, else if"

La forma de sentencia "if, else" dirige las situaciones en las cuales hay exactamente dos posibilidades. Pero ¿qué tal si existen más de dos posibilidades? Por ejemplo, suponiendo que se desea imprimir una de

las cinco posibles letras empleadas para asignar una calificación de acuerdo con cierta escala numérica, se podría hacer mediante el uso de la forma "if, else if" de la sentencia if para establecer rutas paralelas.

si `calificación` es mayor o igual a 60
  imprimir "A"
de lo contrario, si `calificación` es mayor o igual a 80
  imprimir "B"
de lo contrario, si `calificación` es mayor o igual a 70
  imprimir "C"
de lo contrario, si `calificación` es mayor o igual a 60
  imprimir "D"
de lo contrario
  imprimir "F"

¿Qué sucede si la calificación es 85? La sentencia que despliega "A" es saltada, y entonces se ejecuta la sentencia que imprime "B". Una vez que una de las condiciones parece ser cierta, entonces el resto de las sentencias if completas son saltadas. Así, las sentencia tres, cuatro y cinco no son ejecutadas.

¿Qué sucede si todas las condiciones son falsas? Si todas las condiciones son falsas, entonces la sentencia subordinada bajo el "else" (de lo contrario) es ejecutada. Así, si la calificación es 55, la condición imprime "F" es ejecutada. Observar que no se requiere tener un "else" con la sentencia "if, else if". Si no se cuenta con un "else" y todas las condiciones son falsas, entonces ninguna sentencia es ejecutada.

Resumen de if

Utilizar la forma que se adecue más.

Utilizar la primera forma ("if") para problemas donde se desee hacer una cosa o nada. Utilizar la segunda forma ("if, else") para problemas donde se desee hacer ya sea una u otra cosa. Utilizar la tercer forma ("if, else if") para problemas donde haya tres o más posibilidades.

Problemas de práctica con diagrama de flujo y seudocódigo

Corresponde ahora practicar lo que se ha aprendido sobre las sentencias if mediante la presentación de un diagrama de flujo y escribiendo el seudocódigo correspondiente para un algoritmo que realiza un recorte del excesivo salario de un director general de una empresa a tan sólo la mitad. La figura 2.8 presenta el diagrama de flujo.

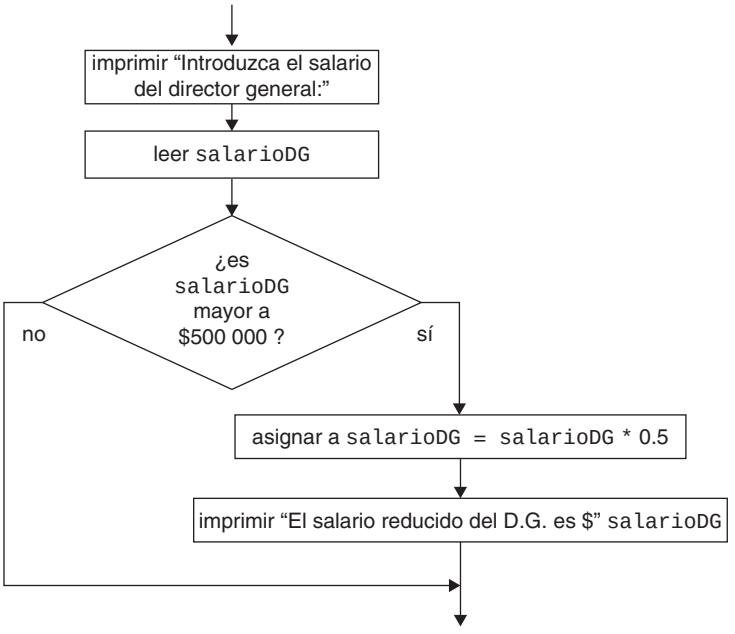


Figura 2.8 Diagrama de flujo para reducir el salario de un director general.

En los diagramas de flujo se omite la palabra "si" de las condiciones en diagramas y se agregan signos de interrogación para convertir una condición en pregunta. El formato de pregunta se ajusta al "sí" y "no" de las flechas de salida. Si la condición es verdadera, la respuesta a la pregunta es "sí". Si la condición es falsa, entonces la respuesta a la pregunta es "no". A partir del diagrama de flujo en la figura 2.8, tratar escribir una versión en seudocódigo del algoritmo recortar-a-la-mitad-sueldo-de-DG. Una vez hecho esto, comparar la respuesta con la del libro:

Practicar escribiendo el algoritmo en seudocódigo.

```
imprimir "Introduzca el sueldo del DG:"
leer salarioDG
si salarioDG es mayor que 500000
   asignar a salarioDG = salarioDG * 0.5
   imprimir "El salario reducido de DG es " salarioDG
```

Práctica de problemas sólo con seudocódigo

Todo mundo conoce el dicho de que una imagen vale más que mil palabras. Esto puede ser cierto, pero hay que comparar el espacio consumido y el esfuerzo para construir el diagrama de flujo de la figura 2.8 con el espacio consumido y el esfuerzo para escribir el seudocódigo correspondiente. Las imágenes ayudan al principio, pero el texto es más efectivo una vez que se sabe qué hacer. Así que ahora, se omitirán los diagramas de flujo y se pasará directamente al código.

Primero, hay que escribir un algoritmo que imprima "¡No hay clases!", si la temperatura está por debajo de los 0 grados. ¿Qué tipo de sentencia if utilizaría el lector para este problema? Ya que la descripción del problema dice que una cosa u otra, entonces es necesario utilizar la forma simple del "if":

```
imprimir "Introducir la temperatura:"
leer temperatura
si temperatura es menor a 0
   imprimir "¡No hay clases!"
```

Ahora se escribirá un algoritmo que imprima "cálido" si la temperatura está por arriba de los 10 grados y que imprima "frío" en caso contrario. ¿Qué tipo de sentencia debería utilizarse? Ya que la descripción señala hacer una u otra cosa, la forma adecuada es la del "if, else":

```
imprimir "Introduzca una temperatura:"
leer temperatura
si temperatura  es mayor a 10
   imprimir "cálido"
de lo contrario
   imprimir "frío"
```

Finalmente, se solicita escribir "cálido" si la temperatura está arriba de los 27 grados, imprimir "adecuado", si está entre los 10 y los 27, e imprimir "frío" si está debajo de los 10 grados. Para este problema, es apropiado utilizar la forma "if, else if", como a continuación:

```
imprimir "Introducir una temperatura:"
leer temperatura
si temperatura es mayor a 27
   imprimir "cálido"
por el contrario si temperatura  mayor o igual a 10
   imprimir "adecuado"
de lo contrario
   imprimir "frío"
```

2.8 Bucles

Se han tratado hasta el momento dos de las tres estructuras que aparecen en la figura 2.5: estructuras secuenciales y estructuras condicionales. Ahora se hablará del tercer tipo de estructura: *de control*. Las es-

tructuras de control repiten la ejecución de una secuencia particular de sentencias. Si se requiere ejecutar un bloque de código muchas veces, por supuesto que se podría escribir de manera repetitiva el código como se necesitara. Sin embargo, esto conduce a la redundancia, lo cual es algo que se quiere evitar en un programa de cómputo, porque abre la puerta a la inconsistencia. Es mejor escribir el código una vez y reutilizarlo. La forma más simple de reutilizar un bloque de código es ir hacia atrás en donde inicia dicho bloque, y ejecutarlo una vez más. Eso se denomina *un bucle* o *ciclo*. Cada ciclo tiene una condición que determina cuántas veces se repetirá el mismo. Al imaginar que se va manejando hacia el oeste de Kansas y que se ve un letrero que dice "Prairie Dog Town" nuestros hijos pedirían que tomáramos el camino de prairie dog. La decisión acerca de cuántas veces repetir el tour se asemeja a una sentencia de control.

Un ejemplo sencillo

Suponiendo que se desea imprimir "¡feliz cumpleaños!" 100 veces. En lugar de escribir 100 sentencias imprimir "¡feliz cumpleaños!", ¿no sería más conveniente utilizar un ciclo? La figura 2.9 presenta una solución al algoritmo de feliz cumpleaños en forma de un diagrama de flujo con un ciclo. El diagrama de flujo implementa la lógica del algoritmo con una flecha que va a "asigna `cuenta = cuenta + 1`" y vuelve a la condición "¿es `cuenta` menor igual a 100?"

En un bucle a menudo se utiliza una variable *cuenta* que mantiene el rastreo del número de veces que el ciclo es repetido. Se puede ya sea contar hacia arriba o hacia abajo. El diagrama de flujo feliz cumpleaños cuenta hacia arriba.

En la última operación, en lugar de decir "asignar a `cuenta = cuenta + 1`" se podría decir algo como "incrementar `cuenta` en 1". Se eligió la de la palabra *"asignar"* para reforzar una forma de pensamiento que corresponde a la manera en que la computadora actualiza el valor de una variable. Regresando a la revisión del pensamiento asociado con la figura 2.3. Primero la computadora ejecuta un cálculo matemático haciendo uso de los valores de variable existentes. En la figura 2.3, el cálculo involucra dos variables: `largo` y `ancho`, que eran diferentes para la variable a ser actualizada, `areaRectangulo`. En la figura 2.9 el cálculo involucra a la variable a ser actualizada, `cuenta`. Después de completado el cálculo, asigna el resultado del cálculo a la variable a ser actualizada. Esta asignación sobrescribe el anterior valor y lo reemplaza con el nuevo. Así, cuando efectúa el cálculo `cuenta + 1`, la computadora utiliza el anterior valor de `cuenta`. De esta manera (en la subsiguiente asignación) cambia el valor de `cuenta` con el nuevo.

En la práctica, todos los ciclos deberían tener un tipo de terminación. Esto es, deberían detener su ejecución en algún punto. Un ciclo que cuenta hacia arriba normalmente utiliza un valor máximo como condición de terminación. Por ejemplo, el ciclo de la figura 2.9 continúa mientras `cuenta` es menor o igual a 100, y termina (deja de efectuar ciclos) cuando `cuenta` alcanza el valor 101. Un ciclo que cuenta

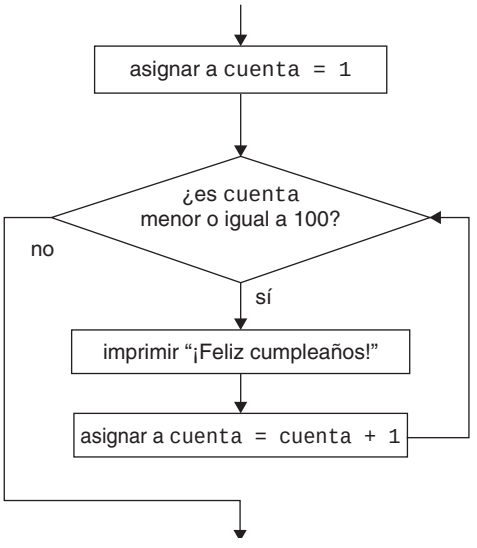


Figura 2.9 Diagrama de flujo para el algoritmo Feliz cumpleaños.

```
asignar cuenta = 1
mientras cuenta sea menor o igual a 100
   imprimir "Feliz cumpleaños"
   asignar cuenta = cuenta + 1
```

Figura 2.10 Seudocódigo para otro algoritmo Feliz cumpleaños.

hacia abajo utiliza por lo regular una condición de valor mínimo de terminación. Por ejemplo, un ciclo podría empezar con un valor para `cuenta` igual a 100 y continuar mientras dicha variable sea mayor que cero. Posteriormente, el ciclo terminaría cuando `cuenta` alcanzara el valor de cero.

Cuando una condición de ciclo compara una variable de conteo con un valor máximo, la pregunta que surge a menudo es si se debe utilizar la condición "menor que o igual a" o simplemente "menor que". De la misma manera, una condición de ciclo que compara una variable de conteo con un valor mínimo, la pregunta que surge a menudo es si se debe utilizar "mayor o igual que" o simplemente "mayor que". No existen respuestas absolutas para estas preguntas. A veces se requerirá hacerlo de una manera y en ocasiones, de otra, dependiendo de la situación. Por ejemplo, al observar otra vez la decisión que se presenta en la figura 2.9 del algoritmo feliz cumpleaños. Suponiendo que se utilizó el "menor que"; entonces en ese caso cuando `cuenta` iguale a 100, se saldría del ciclo antes de que se imprimiera el valor del último centésimo "¡Feliz cumpleaños!" Por tanto, en este caso se debería haber utilizado el "menor o igual que". Si por error se utilizó el "menor que", eso habría significado un *error fuera por una unidad*. Dichos errores se denominan "fuera por una unidad" debido a que ocurren cuando se ejecuta un bucle una vez más o una vez menos de lo que debería. Para evitar este tipo de errores, siempre es importante volver a verificar los límites para las iteraciones del algoritmo.

El bucle while

La mayoría de los lenguajes de programación tienen diferentes tipos de bucles. Aunque podría sonar absurdo, teóricamente hay maneras de convertir un tipo de bucle en otro diferente. Así, por simplicidad y brevedad, en esta plática sobre algoritmos se considerará sólo una clase de éstos y se estudiarán brevemente los otros tipos cuando se entre en más detalles al estudio del lenguaje Java. El tipo de bucle a considerar por el momento es uno bastante popular, el cual contiene el siguiente formato:

while <*condición*>
  <*sentencia(s)*>

Este formato debe sonar familiar al usuario ya que es parecido al formato de la sentencia if. La condición aparece en la parte superior, y las sentencias subordinadas tienen sangría. Las sentencias subordinadas, las cuales están dentro del bucle, se denominan *cuerpo del ciclo o del bucle*. El número de veces que se repite el ciclo se denomina el número de *iteraciones*. Es posible que un bucle se repita por siempre, lo que se denomina *bucle infinito*. También es posible que un ciclo se repita cero veces. No hay un nombre especial para el tipo de iteración con cero ocurrencias, pero es importante señalar que esto en ocasiones puede suceder. Por ejemplo, observar cómo aparece el diagrama de flujo del algoritmo feliz cumpleaños correspondiente a la figura 2.9 cuando se presenta como seudocódigo con un bucle while (figura 2.10).

Ésta es la explicación de cómo funciona el ciclo:

- Si la condición es verdadera, se ejecutan todas las sentencias subordinadas al bucle, y después regresa a la condición del bucle.
- Cuando la condición del bucle llega a ser finalmente falsa, entonces pasa la primera sentencia, después de la última sentencia subordinada dentro del ciclo, y continúa la ejecución a partir de ahí.

2.9 Técnicas de terminación de un ciclo

En esta sección se describirán tres maneras comunes de terminar un ciclo:

- Contador

  Utilizar una variable que sirva como contador y que lleve el registro del número de iteraciones realizadas.

- Petición del usuario

  Preguntar al usuario si desea continuar. Si éste responde que sí, entonces ejecutar el cuerpo del bucle. Después de cada paso a través de las sentencias subordinadas en el bucle, cuestionar al usuario sobre esto.

- Valor centinela

  Cuando un bucle incluye sentencias de entrada, identificar un tipo de valor especial (*centinela*, o también llamado en español, bandera) que se encuentre fuera del rango normal de entrada, y que se utilice para indicar que el ciclo debe terminar. Por ejemplo, si el rango normal de entrada son números positivos, la bandera podría ser un número negativo, por ejemplo, −21. He aquí cómo hacerlo: continuar leyendo los valores y ejecutar el ciclo hasta que el valor que se capture sea igual al de la bandera, y entonces detener el ciclo. En la práctica, un centinela es una especie de guardián que permite pasar a la gente hasta que el enemigo llega. Así, el valor centinela de un programa es como un centinela humano que permite o detiene la continuación de un ciclo.

Terminación de conteo

La figura 2.10 del algoritmo Feliz cumpleaños es un buen ejemplo de la utilización de un contador para finalizar la operación de un bucle. Se debe señalar, sin embargo, que el valor para que una computadora inicie su conteo es el 0, en lugar del uno. Si se utiliza una convención estándar de inicio en cero, el seudocódigo de la figura 2.10 cambiaría de la siguiente manera:

```
asignar a cuenta = 0
mientras cuenta sea menor a 100
   imprimir "¡Feliz cumpleaños!"
   asignar a cuenta = cuenta + 1
```

Observar que se ha cambiado el valor de conteo inicial de 1 a 0, con lo que también se cambió la comparación en la condición de a "menos de". Esto producirá las mismas 100 iteraciones, pero esta vez, los valores de cuenta serán 0, 1, 2,… 98, 99. Cada vez que se crea un bucle de conteo, es importante asegurarse por uno mismo de que el número de iteraciones será exactamente igual a las que se desean. Porque se puede iniciar con números diferentes a uno, y porque la condición de terminación puede emplear diferentes operadores de comparación, en ocasiones es difícil asegurarse sobre el número de iteraciones que se obtendrán. He aquí un pequeño truco que nos proporciona más seguridad:

Simplificar el problema para verificar su esencia.

Para verificar la condición terminal de un bucle, cambiar temporalmente la condición terminal para producir lo que se considera que generará una iteración. Por ejemplo, en la más reciente versión del código del seudocódigo del algoritmo Feliz cumpleaños (donde el conteo inicial es cero), cambiar la cuenta final de 100 a 1. Posteriormente, preguntarse: "¿cuántas operaciones de impresión ocurrirán?" En este caso, el conteo inició en 0. La primera vez que la condición sea probada, la condición es "0" y es menor que "1", lo cual es verdadero. Así la condición se satisface, y la sentencia subordinada y las sentencias subordinadas se ejecutan. Ya que la condición final se satisface, el bucle incrementa el contador en 1 la siguiente vez que la condición sea probada, ésta será así: ¿es "1" menor que "1"? lo cual es falso. Por tanto, la condición no se satisface, y el bucle termina. Ya que la utilización del 1 en la condición del bucle produce una iteración, se puede tener la seguridad de que la utilización del 100 producirá 100 iteraciones.

Terminación por petición del usuario

Para entender la terminación por petición del usuario, se brinda como ejemplo un algoritmo en el cual repetidamente se le solicita al usuario que introduzca un número y después calcula e imprime los cuadra-

```
asignar continuar = "s"
mientras continuar sea igual a "s"
   imprimir "Introduzca número:"
   leer num
   obtener cuadrado = num * num
   imprimir num "al cuadrado es" cuadrado
   imprimir "¿Desea continuar? (s/n):"
   leer continuar
```

Figura 2.11 Algoritmo de impresiones de cuadrados de números que utiliza un ciclo por petición del usuario.

dos de los números introducidos. Esta actividad debe continuar siempre y cuando el usuario responda "s" a la pregunta que se le presenta.

La figura 2.11 despliega el algoritmo a manera de seudocódigo. Dentro del cuerpo de la sentencia while la primera sentencia solicita al usuario introducir un número, la tercera hace los cálculos, y la cuarta imprime el resultado. La pregunta ¿Desea continuar (s/n)? y la respuesta correspondiente aparecen después del final del cuerpo. Este bucle se ejecuta siempre, al menos una vez, debido a que asigna el valor de "s" a la variable continuar antes de que inicie el bucle.

Suponiendo que queremos dar al usuario la oportunidad de salir antes de que introduzca aunque sea un número para obtener su cuadrado. Esto se podría hacer, mediante el reemplazo de la primera sentencia:

```
asignar a continuar = "s"
```

Con las siguientes dos sentencias:

```
imprimir "¿Desea imprimir el cuadrado? (s/n):"
leer continuar
```

Esto brinda al usuario la opción de responder "n" para que no se calcule ninguna operación.

Terminación mediante valor centinela

Para entender el valor centinela o bandera, es necesario considerar un algoritmo que lee los marcadores en el boliche de manera repetitiva hasta que el valor centinela (o la bandera) que se introduce es −1. Entonces, el algoritmo imprime el marcador promedio:

Medítelo. A menudo, se requiere emplear tiempo para pensar acerca de la solución antes de escribirla. Y se debe pensar en la solución en un nivel alto, sin preocuparse acerca de los detalles. Dicho esto, se alienta a continuar con el libro y pensar acerca de los pasos necesarios.

¿Ha pensado en él? De ser así se sugiere comparar sus ideas con esta descripción del alto nivel:

```
Leer los marcadores de manera repetitiva y encontrar la suma de los mismos.
Posteriormente cuando se introduzca −1, dividir la suma de todos los marcadores introducidos.
```

Hay dos detalles en la descripción a alto nivel a la cual es necesario dirigirse. Primero, se requiere pensar acerca de cómo encontrar la suma de los marcadores. Antes de solicitar cualquier entrada de datos, y antes de que se asigne un valor inicial a la variable `marcadorTotal`. En otras palabras, se debe *inicializar* en cero. Por tanto, en el mismo bucle, el cual solicita de manera repetitiva al usuario el siguiente marcador, después de leer el mismo, y agregarlo a la variable `marcadorTotal` para sumar los marcadores que se han ingresado. De esta manera, todos los marcadores que se encuentran en la variable `marcadorTotal` ya contendrán la suma total de todos los marcadores anteriores.

La suma de todos los marcadores es útil porque su objetivo es determinar el promedio de éstos, y calcular el promedio que se requerirá para la suma. Pero para el cálculo de un promedio también se toma el número total de elementos y eso es lo que se verá a continuación. ¿Cómo se puede mantener un registro del número de marcadores introducidos hasta este momento? Inicializando y agregando valor a una variable `cuenta` mientras se inicializa la variable `marcadorTotal`. Es importante observar que se requiere un solo valor centinela o bandera para las tres actividades (entrada, actualización de la variable

```
asignar marcadorTotal = 0
asignar contador = 0
imprimir "Introduzca marcador (teclee −1 para salir):"
leer marcador
mientras marcador no sea igual a −1
   asignar marcadorTotal a marcadorTotal + marcador
   asignar cuenta = cuenta + 1
   imprimir "Introduzca marcador (teclee −1 para salir):"
   leer marcador
calcular promedio = marcadorTotal / cuenta
imprimir "El marcador promedio es " promedio
```

Figura 2.12 Algoritmo del marcador de boliche que utiliza un valor centinela dentro de un bucle.

`marcadorTotal` y actualización de `cuenta`). Se seleccionó el valor −1 como valor centinela para el algoritmo marcador de boliche debido a que es un valor que nunca se introduciría en un marcador de boliche; pero cualquier otro número negativo funcionaría como valor centinela.

La figura 2.12 ilustra la solución algorítmica para este problema. Observar cómo el mensaje de solicitud al usuario especifica "(−1 para salir)". Esto es necesario, ya que sin esta indicación, el usuario no sabría cómo salir. En general, siempre se debe proporcionar suficiente información al dar las indicaciones para que el usuario sepa lo que debe hacer y cómo salir.

¿Qué se esperaría que sucediera si el usuario introdujera −1 en la primera entrada? Eso provocaría que el cuerpo del bucle se saltara y que la variable `cuenta` nunca se actualizara de su valor originalmente inicializado a cero. Cuando la sentencia del `promedio` intenta calcular el promedio del marcador, divide el valor de la variable `marcadorTotal` entre `cuenta`. Debido a que `cuenta` es igual a cero, entonces divide entre cero. Como el usuario podrá recordar de sus cursos de matemáticas, la división entre cero genera problemas. Si un algoritmo divide entre cero, el resultado es indefinido. Si un programa de Java divide entre cero, la computadora imprime un error indescifrable e inmediatamente después abandona el programa. Debido a que el algoritmo de marcador de boliche permite la posibilidad entre cero, no es tan *robusto.* El ser robusto involucra comportarse en una forma que a un usuario común y corriente se le consideraría ser *sensible* y *cortés*, a pesar de que su captura fuera irrazonable. Para hacerlo más robusto, es necesario reemplazar la dos últimas sentencias en el algoritmo de la figura 2.12 con una sentencia `if` como la siguiente:

```
si cuenta no es igual a 0
   asignar promedio = marcadorTotal / cuenta
   imprimir "El marcador promedio es " promedio
de lo contrario
   imprimir "No se obtuvieron entradas".
```

El utilizar la sentencia `if` permite al programa explicarle al usuario por qué no se produjo ninguna salida, y evita problemas inherentes a la división entre cero.

2.10 Bucles anidados

En las secciones precedentes, se presentaron algoritmos que contenían un simple bucle dentro de ellos, al avanzar en el libro y progresar en su carrera de programador, el usuario encontrará que la mayoría de los programas contienen más de un bucle. Si un programa tiene bucles que son independientes, es decir que el primer bucle termina antes de que el segundo inicie, entonces el flujo del programa debe ser razonablemente sencillo. Por otro lado, si un programa tiene un bucle dentro de otro bucle, entonces el flujo del programa debe ser difícil de entender. En esta sección, se buscará que el usuario se sienta seguro con los *bucles anidados*, lo cual es un término formal para los bucles internos que están dentro de los externos.

Suponiendo que se nos solicita escribir un algoritmo que ejecute varios juegos de "Encontrar el número más grande". En cada juego, el usuario introduce una serie de números no negativos. Cuando llega a introducir uno negativo, el algoritmo imprime el número más grande de la serie y pregunta al usuario(a) si desea volver a jugar.

Piense en el número de bucles que deberían utilizarse.

Antes de escribir cualquier cosa, es importante plantearse la siguiente pregunta: ¿Qué tipos de bucles deberían utilizarse? Se requerirá un bucle externo que continúe mientras el usuario señale que desea seguir jugando. ¿Qué tipo de bucle ha de emplearse: de conteo, de requerimiento del usuario, o de valor centinela? Se necesitará de un bucle interno que efectúe el juego mediante la lectura de los números hasta encontrarse con la captura de un número negativo. ¿Qué tipo de bucle ha de emplearse: de conteo, de requerimiento del usuario o de valor centinela? ¿Ha intentado el lector responder a la pregunta? Si es así, lea lo que viene a continuación. De lo contrario, reflexiónelo.

El bucle externo debe ser uno de tipo requerido por el usuario. El interno debe tener un bucle de valor centinela, donde el valor centinela sea un valor negativo. Ahora al analizar el algoritmo del problema en la figura 2.13, se observa que el algoritmo externo en efecto utiliza uno de tipo requerimiento por el usuario: en la parte final del bucle, se le pregunta al usuario si desea continuar, y al principio se verifica la respuesta. Observar que el bucle interno hace uso de un algoritmo de valor centinela: el bucle termina cuando el usuario introduce un número negativo.

La lógica del bucle interno no es trivial y merece atención especial. Antes de examinar el código mismo, se requiere pensar acerca de la meta y la solución en un alto nivel. La meta es leer una serie de números, donde el último número sea negativo y que posteriormente imprima el número más grande. Supo-

¿Cómo lo haría un ser humano?

niendo que la secuencia de números que se introducen son 7, 6, 8, 3, 4, −99. Después de que se introduce cada uno de los números, el algoritmo debe plantearse la siguiente pregunta: ¿es el nuevo número mayor que el previo número mayor? Si es así, entonces es el nuevo "campeón". Observar que la pregunta anterior inicia con la palabra "si". Esto es un buen indicativo de que se puede implementar esta lógica con una sentencia if. Localizar la sentencia if en el bucle de la figura 2.13 y verificar que implemente la lógica antes mencionada. Se verá que la sentencia if verifica el nuevo número para ver si es mayor que el previo número mayor, y si es más alto, entonces el nuevo número es asignado a la variable `mayor`. Dicha asignación corona al nuevo número como el nuevo campeón.

Utilizar un caso extremo.

Observar que la inicialización Asignar `mayor = −1` al inicio del bucle externo. ¿Cuál es el punto de inicializarlo a −1? Inicializar la variable campeona (la mayor) con una variable inicial que perderá automáticamente en cuanto se compare con el primer valor en el proceso de encontrar el número mayor porque las competencias están limitadas a números no negativos, y estos últimos siempre son mayores a −1. Después de que la primera entrada reemplaza al −1 como `mayor`, las subsiguientes entradas de datos podrían o no reemplazar el valor `mayor`, dependiendo del tamaño del número que se introduzca y del valor del `mayor` actual.

```
asignar continuar = "s"
mientras continuar = "s"
   asignar mayor = −1
   imprimir "Introduzca un número (uno negativo para salir):"
   leer num
   mientras num sea mayor que o igual a 0
      si num es mayor que o igual a 0
         asignar mayor = num
      imprimir "Introduzca un número (uno negativo para salir):"
      leer num
   si mayor no es igual a −1
      imprimir "El número mayor que se introdujo es" mayor
   imprimir "¿Desea jugar otra vez (s/n):?"
   leer continuar
```

bucle interno

bucle externo

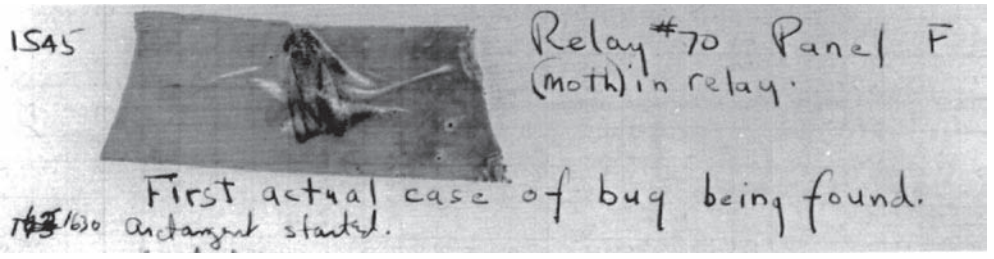
Figura 2.13 Algoritmo que efectúa multitud de juegos y encuentra el número mayor.

2.11 Trazado

Adentrarse en los detalles.

Hasta ahora el enfoque de la obra ha sido exclusivamente en el diseño. Ahora veremos lo tocante al *análisis*: descomponiendo el todo en sus partes. En el contexto actual, eso significa adentrarse en los detalles de un algoritmo ya existente. La técnica de análisis que se utilizará se denomina *trazado*, donde el usuario actuará como si fuese la computadora. Avanzará a través del algoritmo (o del programa) línea por línea y grabará cuidadosamente todo lo que suceda. El trazado proporciona una manera de asegurarse de que realmente se están entendiendo los nuevos mecanismos recientemente aprendidos. El trazado también proporciona una forma de verificar si un algoritmo o código de Java existentes son correctos o si tienen *bugs*.

¿Qué son los bugs?* Una de las primeras computadoras digitales, la Harvard Mark II, utilizaba rieles electromagnéticos en lugar de transistores, y los técnicos programaban mediante cambios en las conexiones. Conforme el tiempo pasó,[2] aunque todas las conexiones eléctricas estuvieran bien, la computadora continuaba cometiendo errores. Finalmente, el programador descubrió una palomilla apachurrada entre los contactos de uno de los rieles. Aparentemente, la palomilla se había triturado cuando los contactos se habían cerrado, y el cuerpo de ésta se encontraba interrumpiendo el flujo adecuado de la electricidad entre dichos contactos. Después de que el programador extrajo la palomilla ("debugueó" el programa) la computadora arrojó la respuesta correcta. Cuando alguien realiza el rastreo de un algoritmo o programa para encontrar bugs de software, se puede sentir en ocasiones como en aquellos tiempos adentrándose en el interior del CPU para encontrar palomillas.



Trazado en la forma corta

Se presentan dos formas de trazado: la corta, descrita en esta subsección, y la forma larga que se presenta en la siguiente subsección. El procedimiento de trazado de la forma corta es utilizado de manera común en la industria y en los salones de clase. Trabaja bien en ambientes dinámicos, donde se puede uno regresar y avanzar entre seudocódigo (o código de Java, posteriormente) y realizar un listado del flujo, y llenar información conforme se avanza. El lector puede consultar al instructor para avanzar en esta operación dinámica en un pizarrón en blanco. Por ejemplo, a continuación se presenta un algoritmo que imprime la canción "Feliz cumpleaños a ti".

```
imprimir "¿Cuál es tu nombre?"
leer nombre
asignar cuenta = 0
mientras cuenta  sea menor que 2
   imprimir "Feliz cumpleaños a ti."
   asignar cuenta = cuenta  + 1
imprimir "Feliz cumpleaños, querido" nombre "."
imprimir "Feliz cumpleaños a ti."
```

A continuación se presenta el trazado de la forma corta que muestra cómo se vería una vez que se complete:

---

* El término *bug* se traduce como insecto o bicho, y en realidad hace referencia a un error en el programa; no obstante, en esta sección se decidió escribirlo con su término en inglés a fin de que el lector comprenda la parte explicativa del término: insecto, ya que los autores comentan un poco sobre el tema. Más adelante, sin embargo, no se requiere hacer uso del término y simplemente se presenta como "error". (*Nota del traductor.*)

[2] http://www.faqs.org/docs/jargon/B/bug.html

entrada	nombre	cuenta	salida
~~Arjun~~	Arjun	0	¿Cuál es tu nombre?
		~~1~~	Feliz cumpleaños a ti.
		2	Feliz cumpleaños a ti.
			Feliz cumpleaños, querido Arjun.
			Feliz cumpleaños a ti.

La lista de trazado anterior tiene cuatro columnas: entrada, nombre, cuenta y salida. La columna de entrada muestra una captura hipotética para el algoritmo. La columna de salida muestra lo que produce el algoritmo cuando éste se ejecuta con la entrada específica. En este ejemplo, se inició con el valor de entrada "Arjun". Posteriormente se avanzó en el código, una línea a la vez. Al avanzar en el código, se agregaron valores bajo las columnas `nombre`, `cuenta` y `salida`, y se tachó a los valores anteriores de `cuenta` como si se sobrescribieran con los nuevos valores de `cuenta`. La figura 2.14 describe el procedimiento general.

El trazado de la forma corta funciona bien en un contexto interactivo que se realiza al momento, pero no así en un contexto estadístico como el que se presenta en las páginas impresas de un libro. Esto es porque en un libro, la forma corta de rastreo no describe la dinámica de actualización de manera adecuada. Con el sencillo algoritmo de Feliz cumpleaños, el usuario fue capaz de visualizar la dinámica; pero en algoritmos más grandes, un listado de la forma corta en la página de un libro solamente enmaraña los detalles que se requiere resaltar. Por tanto, en este libro, se utilizará el procedimiento de la forma larga de trazado que realiza un mejor rastreo de cada paso conforme avanza el proceso.

Trazado en la forma larga

Con el procedimiento de trazado en la forma larga, se pone mayor énfasis en mantener un rastreo del lugar en el que se encuentra el usuario en el algoritmo. Para implementar dicho énfasis: 1) se requiere de un renglón separado en la tabla de trazado para cada paso que se ejecute del algoritmo, y 2) para cada renglón en la tabla de trazado, se requiere proporcionar un número de línea que indique el renglón asociado con la línea del algoritmo. Para muestra, observar el trazado de la forma larga del trazado del algoritmo feliz cumpleaños en la figura 2.15.

El trazado de la forma larga en la figura 2.15 se asemeja en cierta manera al de la forma corta, aunque con sus excepciones. La columna de entrada ha sido movida arriba de la parte principal de la tabla de trazado. En su lugar se presenta la columna línea #, la cual señala el número de línea en el algoritmo que corresponde a los renglones en la tabla de trazado. Observar las dos secuencias de números de línea 5, 6. Esto demuestra cómo el trazado "desenrolla" el ciclo y repite la secuencia de sentencias dentro del mismo bucle para cada iteración.

Preparación de trazado:
- Si hay captura, agregar una columna con la etiqueta entrada.
- Agregar una columna con un encabezado para cada variable.
- Agregar un encabezado de columna con la etiqueta salida.

Trazar el programa mediante la ejecución del algoritmo una línea a la vez, y para cada línea, hacer lo siguiente:
- Para cada sentencia de `entrada`, tachar el siguiente valor de entrada, bajo la columna con el encabezado entrada.
- Para cada sentencia de asignación, actualizar el valor de variable mediante la escritura del nuevo valor, en la casilla bajo la que contiene el encabezado de la columna con el nombre de variable. Si ya hay valores en dicha casilla, insertar el nuevo valor bajo la casilla con el último valor que se tenga y tachar el valor anterior.
- Para cada sentencia de `impresión`, escribir el valor de impresión bajo la columna con el encabezado. Si ya se tienen valores en la casilla localizada debajo de la columna con el encabezado, imprimir el nuevo valor en la casilla disponible en la misma columna.

Figura 2.14 Procedimiento de trazado en la forma corta.

```
1 imprimir "¿Cuál es tu nombre?"
2 leer nombre
3 asignar cuenta = 0
4 mientras cuenta sea menor que 2
5     imprimir "Feliz cumpleaños a ti."
6     asignar cuenta = cuenta + 1
7 imprimir "Feliz cumpleaños, querido " nombre "."
8 imprimir "Feliz cumpleaños a ti."
```

Entrada
Arjun

línea#	nombre	cuenta	Salida
1			¿Cuál es tu nombre?
2	Arjun		
3		0	
5			Feliz cumpleaños a ti.
6		1	
5			Feliz cumpleaños a ti.
6		2	
7			Feliz cumpleaños, querido Arjun.
8			Feliz cumpleaños a ti.

Figura 2.15 Trazado en la forma corta del algoritmo Feliz cumpleaños.

Utilización del trazado para encontrar un error

Verificar cada paso.

Es momento de que el lector agregue valor de toda esta explicación de trazado. Se proporcionará un algoritmo y dependerá de él determinar si funciona adecuadamente. De forma más específica, realizar el trazado del algoritmo para determinar si cada paso genera una salida razonable. Si se produce una salida errónea, encontrar el error del algoritmo y solucionarlo.

Suponiendo que la oficina de alojamiento de la Universidad de South Park diseñó los algoritmos que aparecen en la figura 2.16. El algoritmo se supone que lee los nombres de los nuevos miembros y que asigna en cada dormitorio a dos de ellos. Los nuevos miembros cuyos nombres inicien de la letra "A" a la "M" son asignados al área de Chestnut Hall y aquellos cuyos nombres inicien de las letras "N" a la "Z" son asignados a la sección Herr House. Haciendo uso del trazado proporcionado en la figura 2.16, intentar ya sea completar el trazado u obtener un punto en el mismo, en el que se haya identificado un problema.

¿Ha finalizado el lector con el trabajo de trazado? De ser así, comparar su respuesta con la siguiente:

línea#	Apellido	Salida
1		Introduzca apellido (s para salir):"
2	Ponce	
7		Ponce es asignado en sección Herr House.
7		Ponce es asignado en sección Herr House.
7		Ponce es asignado en sección Herr House.
⋮		⋮

El trazado delata un problema: el algoritmo de manera repetitiva imprime la asignación del dormitorio para Ponce, pero no el del resto de los nombres introducidos. Parece haber un bucle infinito. ¿Puede el lector descubrir el problema? El trazado muestra que la variable `apellido` obtiene el primer valor, Ponce, pero nunca el del resto de las personas. Volviendo a la figura 2.16 se puede observar que el algoritmo solicita se introduzca el apellido antes de que inicie el bucle, y no adentro del mismo. Por tanto, el

```
1 imprimir "Introduzca apellido (s para salir):"
2 leer apellido
3 mientras apellido no sea igual a s
4     si primer carácter de apellido está entre A y M
5         imprimir apellido "es asignado en sección Chestnult Hall."
6     de lo contrario
7         imprimir apellido "es asignado en sección Herr House."
```

Entrada
Ponce
Galato
Aidoo
Nguyen
s

línea#	Apellido	Salida

Figura 2.16 Algoritmo de asignación de dormitorios a los nuevos miembros y organización del trazado.

primer valor de entrada es leído, y los otros no. La solución es agregar otra petición del apellido dentro del bucle while; a continuación se presenta el algoritmo corregido:

```
imprimir "Introduzca apellido (s para salir):"
leer apellido
mientras apellido no sea igual a s
    si primer carácter de apellido está entre A y M
        imprimir apellido "es asignado en sección Chestnult Hall."
    de lo contrario
        imprimir apellido "es asignado en sección Herr House."
    imprimir "Introduzca apellido (s para salir):"
    leer apellido
```

Se recomienda trazar correctamente el algoritmo por cuenta propia del lector, y se encontrará que los cuatro nuevos miembros son asignados en los dormitorios que les corresponden. ¡Claro!

Herramientas de desarrollo de software

La mayoría de las herramientas de desarrollo de software, de manera temporal, asignan un número a cada línea de código para ayudar en la identificación de la localización de los errores. Dichos números de línea no son de hecho parte del código, pero cuando aparecen, se pueden utilizar como identificadores en la columna *línea#* de un trazado de largo formato. Muchas herramientas de desarrollo también incluyen un *depurador* (*debugger* en inglés) que permite al desarrollador avanzar a través del programa una línea a la vez como cuando se ejecuta. El depurador permite monitorear el valor de una variable conforme se avanza. El procedimiento de trazado simula un tipo de evaluación del depurador paso a paso. La experiencia con el trazado utilizada en el presente libro facilitará el entendimiento de lo que un depurador automático está indicando.

2.12 Otros formatos de seudocódigo y aplicaciones

El seudocódigo aparece en un gran número de variedades. En esta sección se describen muchas variaciones de seudocódigo y las inherentes a las mismas.

Seudocódigo formal

El siguiente algoritmo de marcadores de boliche utiliza un seudocódigo más formal:

```
marcadorTotal ← 0
cuenta ← 0
imprimir "Introduzca el marcador (−1 para salir): "
leer marcador
mientras (marcador ≠ −1)
{
  marcadorTotal ← marcadorTotal + marcador
  cuenta ← cuenta + 1
  imprimir "Introduzca marcador (−1 para salir): "
  leer marcador
}
promedio ← marcadorTotal / cuenta
imprimir "El promedio del marcador es " + promedio
```

Esta variación final de seudocódigo utiliza símbolos especiales para destacar a las operaciones de una manera más exagerada. Las flechas que apuntan a la izquierda (←) representan la asignación de derecha a izquierda de la cual se habló previamente en la figura 2.3. El símbolo ≠ significa "no es igual a", palabras más sucintas no podría haber. Los corchetes enfatizan la naturaleza subordinada de las sentencias en el cuerpo del bucle `while`. Más adelante, se verá que Java requiere de dichos corchetes en los casos en que el cuerpo de una sentencia if o de un bucle incluyan más de una sentencia subordinada. El símbolo + en la última línea indica que los dos elementos impresos son de tipos diferentes ("El promedio del marcador es " se refiere a una cadena literal, mientras que `promedio` es una variable).

Hasta ahora se ha utilizado seudocódigo, diagrama de flujo y trazadores para describir la lógica algorítmica de una manera bastante precisa. Dicha precisión corresponde de manera cercana a la encontrada en las sentencias individuales del código Java. Estas descripciones algorítmicas han brindado una *visión formal de implementación* del programa. La gente que se preocupa más acerca de esto y que verifica la implementación de los programas son los programadores quienes escriben dichos programas.

Seudocódigo de alto nivel

Debido a que el seudocódigo es tan flexible, se puede también utilizar para describir algoritmos en un nivel más alto, a un nivel más macroscópico: con mayor abstracción. El truco es ignorar los detalles de las operaciones subordinadas y únicamente describir y mantener el rastro de las entradas y salidas de aquellas operaciones subordinadas. Esta estrategia presenta la "situación completa", tal como se observa fuera del mundo. Observa a los "bosques" en lugar de los "árboles". Ayuda a mantenerse en el carril derecho: ¡así, no hay necesidad de resolver el problema erróneo!

Por ejemplo, el siguiente algoritmo de marcadores de boliche utiliza un seudocódigo de nivel más alto que los que se han estudiado anteriormente:

```
Lee todos los marcadores.
Calcula el promedio de marcadores.
Imprime el promedio de marcadores.
```

La descripción a nivel alto presenta sólo las características principales, no los detalles. Indica lo que se supone que el programa hace, pero no cómo lo realiza.

Describir el programa al cliente.

En ocasiones es apropiado pensar en los programas de manera diferente a como lo hace un programador. Suponiendo que lo único que se desea hacer es utilizar el programa de alguien más y que no hay preocupación sobre cómo está escrito. En ese caso, seríamos un usuario o un *cliente*, y lo que necesitaríamos sería una visión de cliente sobre el programa. El seudocódigo de alto nivel que se acaba de presentar es un ejemplo de una *visión* informal *del cliente* de un programa deseado. Una visión formal del cliente de ese programa incluiría de manera típica una descripción sobre cómo utilizar el programa y ejemplos de entradas y salidas. Más

adelante, se verán muchas "visiones del cliente" en código Java que ya han sido escritos y que son de libre uso como parte de cualquier programa que se escriba.

Es útil tener en mente estas dos visiones alternativas de un típico programa de cómputo. Se deseará poder intercambiar entre la visión de un cliente (cuando se esté actuando como tal o cuando se esté comunicando con el usuario final de un programa), y una visión de implementación (cuando se esté diseñando y escribiendo un programa).

2.13 Resolución de problema: administración de activos (opcional)

En esta sección, solicitamos al lector pensar en un problema administrativo de la vida real en un nivel bastante abstracto. Imaginando que él mismo es un especialista en Tecnología de la Información (TI) y trabaja en el gobierno de una ciudad pequeña. El director de aguas del departamento de esa ciudad respeta sus habilidades organizacionales y le ha pedido venir a una junta al concilio de la ciudad y presentar una plática sobre cómo se podría generar un programa de cómputo para ayudar al concilio a administrar los activos del sistema de aguas de la ciudad.

En primer lugar, se sugiere a los miembros del concilio a ayudarlo a establecer una secuencia ordenada de pasos. En un pizarrón se escribirá seudocódigo de alto nivel para el "programa". Para evitar hacer uso de la jerga computacional, se denominará a este seudocódigo "lista de cosas por hacer".

Después de debatir, los miembros del concilio están de acuerdo (y el lector presenta) los siguientes pasos generales:[3]

1. Realizar un inventario de todos los activos del sistema de aguas.
2. Establecer prioridades entre esos activos.
3. Programar futuros cambios, reemplazos y adiciones a dichos activos.
4. Preparar un presupuesto de amplio alcance.

Este seudocódigo de alto nivel involucra sólo cuatro pasos secuenciales, como los pasos secuenciales en la parte derecha de la figura 2.5.

El concilio le agradece por su ayuda, y para la siguiente reunión le solicitarán complementar esta lista con suficientes detalles que muestren cómo planea implementar cada uno de los cuatro pasos. Ellos no desean ver un código de cómputo kilométrico. Simplemente desean saber cómo procederá: darse cuenta de lo complicado del proyecto.

Traducir la visión del cliente en fisión del servidor.

Al volver a su oficina, el lector crea un plan informal de la implementación del problema. Este plan a menudo se denomina *plan del programador* o *plan del servidor* porque la implementación del programador proporciona un servicio al cliente. Para el paso 1, se identifican siete variables: `nombreActivo`, `vidaEsperada`, `condicion`, `historiaServicio`, `vidaAjustada`, `edad` y `vidaRemanente`. Para cada activo, se tendrá que pedir a alguien en el departamento de aguas que proporcione el valor adecuado para cada una de las seis variables. Posteriormente, el programa calculará un valor para la última variable. Se tendrá que repetir esto para cada activo significante. A continuación se presenta una versión de la descripción abreviada del seudocódigo de la implementación del paso 1:

```
asignar a mas = 's'
mientras mas sea igual a 's'
   leer nombreActivo
   leer vidaEsperada
   leer condicion
   leer historiaServicio
   leer vidaAjustada
   leer edad
   asignar vidaRemanente = vidaAjustada – edad
```

[3] Estos cuatro pasos y su elaboración subsiguiente están basados en recomendaciones de la obra *Administración de activos: un manual para pequeños sistemas acuíferos (Asset Management: a Handbook for Small Water Systems).* Oficina de aguas (4606M) EPA 816-R-03-016, www.epa.gov/safewater, septiembre de 2003.

```
imprimir "¿Otro activo (s/n):?"
leer mas
```

Este algoritmo no incluye peticiones para valores de variables individuales. Algunas de estas variables podrían tener múltiples componentes, y el lector podría desear establecer y reforzar ciertas convenciones para que los valores de entrada sean aceptables. Por ejemplo, `condicion` e `historiaServicio` podría cada una tener componentes subordinados. Esto se verá con más detalle, posteriormente.

Para el paso 2, se tienen cinco variables: `nombreActivo`, `vidaRemanente`, `importancia`, `redundancia` y `prioridad`. Las variables `nombreActivo` y `vidaRemanente` son las mismas de las variables utilizadas para el paso 1, por lo que no se necesitará pedir su valor de nueva cuenta. Pero ¡espere! Si es un bucle separado, se tendrá que identificar también cada activo para asegurarse de que los nuevos valores están siendo asociados con el activo adecuado. Se podría hacer esto solicitando al usuario que vuelva a introducir el valor de `nombreActivo`, o se podría hacer un recorrido total de los activos existentes e imprimir cada nombre después de solicitar información adicional sobre el mismo. La segunda estrategia es más fácil para el usuario, así que se recomienda utilizarla. He aquí una descripción de seudocódigo abreviado para la implementación del paso 2:

```
mientras exista otro activo
   imprimir nombreActivo
   leer importancia
   leer redundancia
   leer prioridad
```

Una vez más, el algoritmo no incluye peticiones de entrada, y no establece ni refuerza las convenciones de entrada. Se tratará esto con más detalle más adelante.

Para el paso 3, se identifican cinco variables: `nombreActivo, actividad, añosPorVenir, costoEnDolrs` y `reservaAnual`. Una vez más el valor de la variable `nombreActivo` ya se encuentra en el sistema, así que una vez más, se puede identificar imprimiéndola. Pero al programar las cosas, se querrá que el programa las ordene por `prioridad`. La operación de ordenamiento podría ser un poco truculenta, pero si se tiene suerte, alguien más habrá escrito el código para esa popular tarea de cómputo, y el usuario no tendrá que "reinventar la rueda".

Las variables `actividad`, `añosRestantes` y `costoEnDlrs` son entradas, y se deseará que el programa calcule `reservaAnual` como `costoEnDlrs/añosRestantes`. Después de calcular la reserva anual para cada activo por separado, se querrá que el programa lo agregue a la variable `reservaTotalAnual` y que después del bucle se deseará que se imprima el valor final de `reservaTotalAnual`. A continuación se presenta una descripción abreviada del seudocódigo de la implementación del paso 3:

```
ordenar activos por prioridad
asignar a reservaTotalAnual 0
mientras exista otro activo
   imprimir nombreActivo
   leer actividad
   leer añosRestantes
   leer costoEnDlrs
   asignar a reservaAnual = costoDlrs / añosRestantes
   asignar a reservaTotalAnual = reservaTotalAnual + reservaAnual
imprimir reservaTotalAnual
```

Una vez más, el algoritmo no incluye peticiones. Se tratarán los detalles más adelante.

Para el paso 4, se identificarán las tres variables `reservaTotalAnual`, `ingresoNetoAnual` e `ingresoAdicional`. Para esto se requiere que alguien en el departamento de contabilidad proporcione un valor para `ingresoNetoAnual`. Después de ello, se necesita que el programa reste el valor de la variable `reservaTotalAnual` calculada en el paso 3 para obtener el `ingresoAdicional` requerido para hacer que el plan funcione. ¡Claro! Si la respuesta se vuelve negativa, se deseará que sólo imprima cero para indicar que la ciudad no tendrá que emitir un nuevo impuesto. A continuación la descripción del seudocódigo de la implementación del paso 4:

```
leer ingresoNetoAnual
asignar a ingresoAdicional = ingresoNetoAnual – reservaAnualTotal
si ingresoAdicional es menor a 0
   asignar a ingresoAdicional = 0
imprimir "Ingreso adicional necesario =" + ingresoAdicional
```

Bien, esta preparación es suficiente para la siguiente junta de concilio de la siguiente semana. Al menos se podrá brindar a los miembros del concilio una precisión razonable sobre la cantidad de trabajo requerida.

Resumen

- Utilizar seudocódigo para escribir descripciones informales de algoritmos. Utilizar nombres de variables entendibles. Agregar sangría a las sentencias subordinadas.
- Cuando el programa requiera de entradas, proporcionar una petición informativa para indicarle al usuario qué tipo de información se debe proporcionar.
- Un diagrama de flujo proporciona un diagrama visual de cómo se relacionan los elementos de un programa y de cómo se fluye a través de dichos elementos mientras se ejecuta el programa.
- Hay tres patrones básicos bien estructurados de flujo de control: secuencial, condicional y de bucle.
- Se puede implementar la ejecución condicional mediante el uso de las tres formas de la sentencia if: "if", "if, else" y "if, else if".
- Proporcionar todos los bucles con alguna clase de condición terminal, tal como un contador, petición de usuario o valor centinela.
- Utilizar un bucle anidado si se requiere repetir algo durante cada iteración de un bucle externo.
- Utilizar el rastreo para: 1) obtener un entendimiento íntimo de lo que hace un algoritmo y 2) depurar los programas que tienen errores lógicos.
- Utilizar un lenguaje más abstracto para describir operaciones de programación más largas y más complejas de manera sucinta.

Preguntas de revisión

§2.2 Salida

1. Describir lo que la siguiente sentencia realiza:

```
Imprimir "nombre del usuario =" nombreUsuario
```

§2.3 Variables

2. Proporcionar un nombre de variable adecuado para una variable que mantenga el número total de estudiantes.

§2.4 Operadores y sentencias de asignación

3. Escribir una línea de seudocódigo que le ordene a la computadora que asigne a la variable `velocidad` el resultado de dividir `distancia` entre `tiempo`.

§2.5 Operadores y sentencias de asignación

4. Escribir una línea de seudocódigo que le indique a la computadora que asigne el valor introducido por el usuario en una variable denominada `altura`.

§2.6 Flujo de control y diagramas de flujo

5. ¿Cuáles son los tres tipos de flujo de control descritos en el capítulo?

6. Introducir un ciclo es apropiado siempre que la siguiente acción a realizar sea algo previamente ya hecho. (C/F).

§2.7 Sentencias if

7. Considere el siguiente seudocódigo:

```
si es de noche, asignar a limiteVelocidad = 55;
de lo contrario, asignar limiteVelocidad = 65.
```

Suponiendo que el valor de la variable `noche` sea "falso". Después de que este código se ejecute, ¿cuál debería ser el valor de la variable `limiteVelocidad`?

8. El seudocódigo anterior ¿tiene la forma exacta sugerida en el texto?

9. Dibujar un diagrama de flujo que implemente esta lógica:
Si la temperatura es mayor a 10°C y no está lloviendo, imprimir "caminar". De lo contrario, imprimir "manejar".

10. Proporcionar una solución al problema anterior mediante seudocódigo.

§2.8 Bucles

11. ¿Dónde se realiza la decisión de terminación de un bucle `while`?

12. ¿Cuándo finaliza un bucle `while`, qué se ejecuta después?

13. ¿Es posible para un bucle `while` tener un número infinito de iteraciones?

14. ¿Es posible para un bucle `while` tener cero iteraciones?

§2.9 Técnica de terminación de bucle

15. ¿Cuáles son las tres técnicas de terminación de un bucle descritas en este capítulo?

16. Un *valor centinela* es utilizado para hacer qué de lo siguiente?

a) Especificar el primer valor impreso.

b) Imprimir un mensaje de error.

c) Señale el final de la entrada.

§2.10 Bucle anidado

17. ¿Con la forma de seudocódigo que se utiliza en este capítulo, cómo se diferencia un bucle interno de uno externo?

§2.11 Trazado

18. De las siguientes oraciones, ¿cuáles son verdadero?

a) El trazado muestra la secuencia de ejecución.

b) El trazado ayuda a depurar un programa.

c) El trazado resalta los errores en la inicialización y terminación de un bucle.

d) Todas las anteriores.

19. Trazar el siguiente algoritmo de marcador de boliche (tomado de la sección 2.9). Utilizar los encabezados de la tabla que aparece abajo del algoritmo.

```
1  asignar a marcadorTotal = 0
2  asignar a cuenta = 0
3  imprimir "Introduzca marcador (−1 para salir):"
4  leer marcador
5  mientras marcador no sea igual a −1
6    asignar a marcadorTotal = marcadorTotal + marcador
7    asignar a cuenta = cuenta + 1
8    imprimir "Introduzca marcador (−1 para salir):"
9    leer marcador
10 asignar a promedio = marcadorTotal / cuenta
11 imprimir "El marcador promedio es " promedio
```

Organización del trazado:

Entrada
94
104
114
−1

línea#	Marcador	marcadorTotal	cuenta	promedio	Salida

Ejercicios

1. [Después de §2.5] Escribir seudocódigo para un algoritmo que 1) solicite al usuario que introduzca el largo del lado de un cuadrado, 2) calcule el área del cuadrado y 3) imprima el área del cuadrado. Utilizar las siguientes sesiones de ejemplo.

Sesión ejemplo:

```
Introduzca el largo del lado de un cuadrado en metros: 15
El área del cuadrado es de 225 metros cuadrados
```

Las itálicas significan captura por parte del usuario.

2. [Después de §2.8] ¿Qué es un bucle infinito?

3. [Después de §2.8] Dado el siguiente seudocódigo, encierre en un círculo las sentencias que se consideren estar dentro del cuerpo del bucle while:

```
leer hora
mientras hora sea menor a 8
 imprimir hora
 asignar a hora = hora + 1
```

4. [Después de §2.9] En el ejercicio 3, imaginar que la entrada del usuario para `hora` es 3. ¿Cuántas líneas de salida generará el algoritmo?

5. [Después de §2.11] Trazar el siguiente algoritmo. El libro presenta dos formas de realizar trazados: una corta y una larga. Para facilitar un poco las cosas, la organización para ambas formas se presentan a continuación. Para responder, seleccionar alguna de ellas y utilizarla, e ignorar la otra.

```
1  asignar a y = 0
2  leer x
3  mientras y no sea igual a x
4     asignar a y el valor de x
5     leer x
6     asignar a x = x + y
7     imprimir "x = " x
8     imprimir "y = " y
```

Forma corta:

entrada	x	y	salida
2			
3			
4			
0			

Forma larga:

entrada
2
3
4
0

línea#	x	y	salida

6. [Después de §2.11] Trazar el siguiente algoritmo. El libro presenta dos formas de realizar trazados: una corta y una larga. Para facilitar un poco las cosas, la organización para ambas formas se presentan a continuación. Para responder, seleccionar alguna de ellas y utilizarla, e ignorar la otra:

```
 1 asignar a num = 2
 2 asignar a cuenta = 1
 3 mientras cuenta sea menor a 5
 4  asignar a cuenta = cuenta * num
 5  si cuenta / 2 es menor que 2
 6   imprimir “Hola”
 7  de lo contrario
 8   mientras cuenta sea menor que 7
 9      asignar a cuenta = cuenta + 1
10  imprimir “la cuenta es ” cuenta “.”
```

Forma corta:

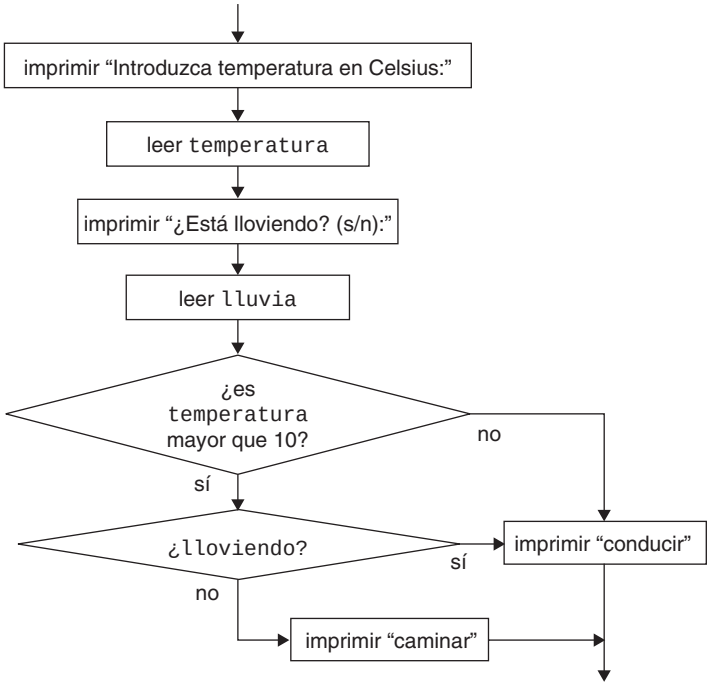
num cuenta salida

Forma larga:

línea#	num	cuenta	salida

Solución a las preguntas de revisión

1. La sentencia imprime literalmente lo que está dentro de las comillas y después imprime el valor actual de la variable `nombreUsuario`.
2. `numeroTotalDeEstudiantes`
3. El seudocódigo que le dice a la computadora que asigne la distancia dividida entre el tiempo en una variable de velocidad sería:
   asigna a `velocidad = distancia / tiempo`
4. Sentencia de seudocódigo:
   lee `altura`
5. Los tres tipos de flujo de control que se estudiaron en el capítulo 2 son secuencial, condicional y de bucle.
6. Cierto. Los bucles son apropiados cuando la siguiente acción a realizar es algo previamente hecho.
7. Después de que el código se ejecuta, el valor de la variable `limiteVelocidad` debería ser 65.
8. Sí. Está bien porque sólo es seudocódigo, y éste se expresa sin ambigüedades. Sin embargo, si éste fuera código de computadora, tendría la posibilidad de compilar, la sintaxis tendría que apegarse exactamente a las reglas de un lenguaje de programación en particular, tal como Java.
9. El diagrama de flujo que implementa la lógica caminar/conducir:



10. Proporcionar una solución al problema anterior en forma de seudocódigo:

```
imprimir “Introduzca temperatura en Celsius:”
leer temperatura
imprimir “¿Está lloviendo? (s/n):”
leer lluvia
```

```
si temperatura es mayor que 10
  si lluvia es igual a "n"
    imprimir "caminar"
de lo contrario
  imprimir "manejar"
```

11. Una decisión de terminación de un bucle se realiza al inicio del bucle.
12. Después de que un bucle termina, la siguiente acción a ejecutar es la primera sentencia después del final del bucle.
13. Sí.
14. Sí.
15. Las tres técnicas de terminación del bucle que se describen en este capítulo son: contador, requerimiento del usuario y valor centinela.
16. Un valor centinela es utilizar para: *c*) señalar el fin de entradas.
17. El bucle interno está enteramente dentro del bucle externo. El bucle interno está alineado a la derecha a diferencia del externo.
18. *d*) Todas las anteriores. El trazado muestra la secuencia de ejecución, ayuda a depurar, y resalta los errores de inicialización y terminación.
19. El trazado del algoritmo de marcador de boliche:

Entrada
94
104
114
−1

línea#	marcador	marcadorTotal	cuenta	promedio	salida
1		0			
2			0		
3					Introduzca marcador (-1 para salir):
4	94				
6		94			
7			1		
8					Introduzca marcador (-1 para salir):
9	104				
6		198			
7			2		
8					Introduzca marcador (-1 para salir):
9	114				
6		312			
7			3		
8					Introduzca marcador (-1 para salir):
9	-1				
10				104	
11					El promedio del marcador es 104

CAPÍTULO 3

Fundamentos de Java

Objetivos

- Escribir ejemplos sencillos de Java.
- Conocer elementos de estilo como los comentarios y la legibilidad.
- Declarar, asignar e inicializar variables.
- Entender los tipos de datos primitivos: enteros, de punto flotante y de carácter.
- Entender las variables de referencia.
- Utilizar los métodos de la clase `String` para manipulación de cadenas.
- Utilizar la clase `Scanner` para lectura de datos.
- Aprender, de manera opcional, acerca de la entrada y salida mediante GUI, haciendo uso de la clase `JOptionPane`.

Relación de temas

3.1 Introducción

En la resolución de un problema, lo mejor es organizar detenidamente nuestros pensamientos en torno a lo que se quiere hacer. En el capítulo 2 nos centramos en el pensamiento y organización de la escritura de soluciones algorítmicas mediante seudocódigo para descripciones de determinados problemas. En este capítulo avanzaremos el siguiente paso: la escritura de soluciones mediante la utilización de un lenguaje real de programación: Java. Usando un lenguaje de programación real, el lector será capaz de ejecutar su programa en una computadora y producir resultados en la pantalla de la misma.

A medida que avance en el presente capítulo, encontrará mayor paralelismo del código Java con el seudocódigo. La diferencia principal entre ambos es que en Java se requiere una sintaxis precisa. La sintaxis del seudocódigo es poco exigente: el seudocódigo debe ser bien claro para que los usuarios lo comprendan, pero la ortografía y gramática no tienen que ser perfectas. La sintaxis en el código de programación es rigurosa: debe ser perfecta en términos ortográficos y gramaticales. ¿Por qué? Porque el código regular de programación se prepara para las computadoras y las computadoras no son capaces de entender las instrucciones, a menos que éstas sean perfectas.

A partir de este capítulo, hará su primera prueba real de Java, enfocando los fundamentos. Un programa de *ejecución secuencial* es aquel en que todas sus sentencias son ejecutadas en el orden en que fueron escritas. Cuando escribamos estos programas, mostraremos las sentencias de salida, asignación y entrada. Además, describiremos los tipos de datos y las operaciones aritméticas. En la parte final del capítulo se presentan algunos temas un poco más avanzados (conversión de datos y métodos de objetos de cadena de caracteres), que agregarán funcionalidad importante sin añadir mucha complejidad. Iniciemos pues el viaje por Java.

3.2 Programa "Tengo un sueño"

En esta sección se presenta un programa sencillo que imprime una simple línea de texto. En varias de las siguientes secciones, analizaremos los diferentes componentes del programa. El análisis puede ser un poco tedioso, pero vale la pena intentarlo. Es importante entender los componentes, pues todos los futuros programas harán uso de ellos. En el resto del capítulo introducimos nuevos conceptos que sirven para desarrollar programas más sustanciales.

Observe la figura 3.1, en ella se muestra un programa que imprime el texto "¡Tengo un sueño!"[1] En las siguientes secciones, nos referiremos a este programa como el Programa del sueño. El programa contiene comentarios para los usuarios e instrucciones que se ejecutan en la computadora. Analizaremos primero los comentarios y después las instrucciones. Puede utilizar este pequeño programa como punto de inicio para otros programas en Java. Es recomendable que lo capture y observe lo que hace. Posteriormente, modifíquelo, ejecútelo otra vez, y así hasta que tenga lo que necesita.

Inicie cada programa con esta estructura de código.

3.3 Comentarios y legibilidad

En la vida real, pasamos mucho tiempo analizando y modificando el código de otras personas. Y las otras personas emplearán mucho de su tiempo analizando y arreglando nuestro código después de que hemos cambiado a algún otro. Con toda esta actividad de análisis del código de otras personas todo el tiempo, todos los códigos deben ser entendibles. Una clave para hacerlos entendibles, es la utilización de buenos comentarios. Los *comentarios* son palabras que las personas leen, pero que el compilador[2] ignora.

[1] El Dr. Martin Luther King presentó su discurso "Tengo un sueño" en los escalones del monumento a Lincoln, como parte de una manifestación por los derechos civiles efectuada el 28 de agosto de 1963 en la ciudad de Washington, D.C. El discurso apoyaba el movimiento contra la segregación racial y ayudó en la conformación de la Ley de los Derechos Civiles de 1964.

[2] Un compilador, definido en el capítulo 1, es un programa especial que convierte el código fuente de un programa en un programa ejecutable. Un programa ejecutable es aquel que la computadora puede ejecutar directamente.

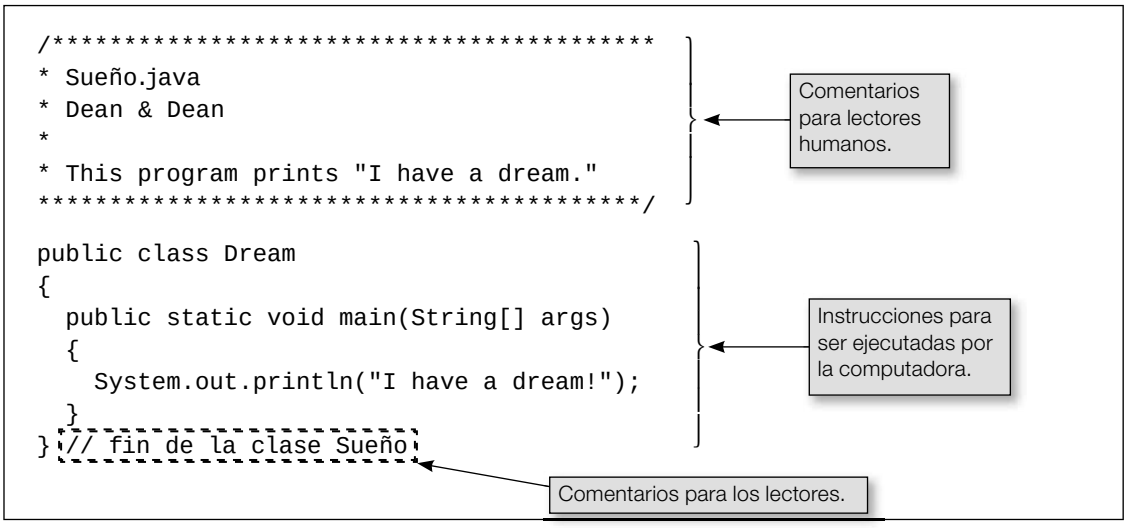


Figura 3.1 Programa Sueño.

Sintaxis de los comentarios de una línea

Hay dos tipos de comentarios: los de una línea y en bloque. Si el texto a comentar es demasiado corto para que quepa en una sola línea, se deben utilizar comentarios de una sola línea. Los comentarios de una línea inician con dos diagonales. A continuación se presenta un ejemplo:

```
} // fin de la clase Sueño
```

El compilador ignora todo desde la primera diagonal hasta el final de la línea. Así, en la línea anterior, el compilador pone atención únicamente en la llave derecha (}) e ignora el resto de la línea. ¿Por qué es útil el comentario? Si se está analizando una larga pieza de código en el monitor y se quiere avanzar hasta el final del mismo, es bueno tener una descripción de éste (por ejemplo, `el final de la clase Sueño`) sin tener que realizar avances y retrocesos de página al inicio y al final del código.

Sintaxis de los comentarios en bloque

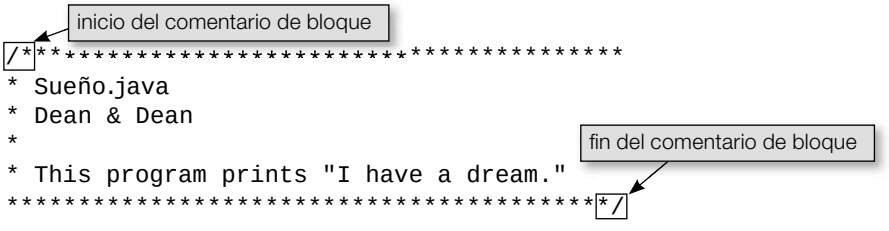
Si el texto del comentario es demasiado largo para ponerlo en una sola línea, se pueden utilizar comentarios de múltiples líneas, pero es un poco tedioso tener que volver a teclear // para cada uno. Como alternativa, se pueden utilizar comentarios en bloque. Los comentarios en bloque inician con los símbolos /* y terminan con los de cierre */. A continuación un ejemplo:

```
/*
El siguiente código despliega los androides en una caza a alta velocidad,
causando estragos cerca de los vehículos.
*/
```

El compilador ignora todo lo que aparece entre la primera y la última diagonal.

Prólogo

Un prólogo es un ejemplo especial de un comentario de bloque. Se debe poner un prólogo al programa para que el programador pueda darle un vistazo rápidamente y pueda tener una idea acerca de lo que hace. Para resaltarlo, es común encerrarlo en una caja de asteriscos. El siguiente es el prólogo del programa Sueño.



Observe que los símbolos de apertura `/*` y de cierre `*/` aparecen al lado de los otros asteriscos. Lo anterior está bien. El compilador aun así reconoce a ambos como puntos de apertura y cierre del bloque de comentarios.

Se deben incluir los siguientes elementos en la sección del prólogo del programa:

- una línea de asteriscos (*)
- nombre del archivo
- nombre del programador
- una línea con un simple * a la derecha
- descripción del programa
- una línea de asteriscos (*)

Legibilidad y líneas en blanco

Se dice que un programa es *legible* si un programador puede entender fácilmente lo que hace. Los comentarios son una forma de mejorar la lectura de un programa. Otra es utilizar líneas en blanco. ¿Qué tan útil son las líneas en blanco? ¿No es más fácil entender muchas instrucciones cortas y simples que una larga? De la misma manera, es más fácil entender pequeños trozos de código que uno largo. La utilización de líneas en blanco permite dividir los largos trozos en otros más pequeños. En un prólogo, se insertan líneas en blanco para separar la sección del autor del nombre del código de la sección de la descripción. También se insertan líneas en blanco abajo del prólogo para separarlo del resto del programa.

Por cierto, a las computadoras no les importa la legibilidad; sólo les importa que el programa funcione. De forma más específica, las computadoras se saltan todos los comentarios, líneas en blanco y caracteres de espacio contiguo. Puesto que las computadoras no se preocupan por la legibilidad, su computadora se sentirá perfectamente feliz de compilar y ejecutar el siguiente código del programa Sueño:

```
public class Sueño{public static void
main(String[]args){System.out.println("¡Tengo un sueño!");}}
```

Pero quien intente leer el programa se molestará porque su legibilidad es pobre.

3.4 El encabezado de la clase

Hasta ahora hemos visto el código que la computadora ignora: los comentarios. Ahora hablaremos del código al que la computadora pone atención. A continuación se presenta la primera línea en el programa Sueño:

```
public class Sueño
```

Esta línea se denomina *encabezado de clase* porque es el encabezado de la definición del programa de *una clase*. ¿Qué es una clase? Por ahora, piense en una clase simplemente como un contenedor para el código del programa.

Examinemos las tres palabras en el encabezado de clase. Primero, la última palabra: `Sueño.Sueño` es el nombre de la clase. El compilador permite al programador elegir cualquier nombre para la clase, pero con la finalidad de hacer entendible el código se debe elegir una palabra que describa al programa. Puesto que el programa Sueño imprime "¡Tengo un sueño!", `Sueño` es un nombre de clase razonable.

Las primeras dos palabras en el encabezado de la clase: `public` y `class` son *palabras reservadas*. Las palabras reservadas, también llamadas *palabras clave*,[3] están definidas por el lenguaje Java para un propósito específico. Un programador no las puede redefinir para que signifiquen otra cosa. Eso significa que los programadores no pueden utilizar palabras reservadas en los nombres de sus programas. Por ejemplo, se puede elegir `Sueño` como nombre de la clase porque Sueño no es una palabra reservada. Por el contrario, no se pueden seleccionar las palabras `public` o `class` como nombre de clase.

[3] En Java, las palabras reservadas y las palabras clave son lo mismo. Pero en algunos lenguajes de programación, hay una sutil diferencia. En dichos lenguajes, ambos términos se refieren a palabras que son definidas por el lenguaje de programación, mientras que las palabras clave pueden ser redefinidas por el programador, y las palabras reservadas, no.

¿Y qué significan las palabras reservadas `public` y `class`? La palabra `class` es un marcador que significa el inicio de la clase. Por ahora, con programas con una sola clase, la palabra `class` también significa el inicio del programa.

La palabra `public` es un modificador de *acceso*: modifica los permisos de la clase para que sea accesible al "público". Hacer una clase públicamente accesible es crucial cuando un usuario intenta ejecutarla, el comando de ejecución del usuario deberá ser capaz de encontrarla.

Hay ciertas convenciones de código que la mayoría de los programadores siguen, mismas que se enlistan en el apéndice "Convenciones de estilo de Java". A lo largo del libro, cuando hagamos referencia a las "convenciones de código estándar", nos estamos refiriendo a las convenciones que se encuentran en dicho apéndice. Las convenciones de código estándar señalan que los nombres de clases deben iniciar con una letra mayúscula como primer carácter; así, la S en el nombre de clase `Sueño` es mayúscula. Java es *sensible a mayúsculas*, lo que significa que el compilador del lenguaje distingue entre letras mayúsculas y minúsculas. Debido a que Java es sensible a mayúsculas, el nombre del archivo debe comenzar con una letra mayúscula.

3.5 El encabezado del método `main`

Se ha hablado acerca de los encabezados de clase. Ahora es tiempo de hablar de los encabezados que van abajo del encabezado de clase: el encabezado del método `main`. Al iniciar un programa, la computadora busca un encabezado para el método `main`, e inicia la ejecución con la primera sentencia después del encabezado del método `main`. El encabezado del método `main` debe tener la siguiente forma:

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

Comenzaremos el análisis del encabezado del método `main` mediante la explicación de la palabra `main`. Hasta ahora lo que el lector sabe es que cuando el programa inicia, la computadora busca esa palabra. Pero `main` es algo más que una palabra; es un *método* de Java. Un método de Java es similar a una función matemática. Una función matemática toma argumentos, ejecuta un cálculo y devuelve una respuesta. Por ejemplo, la función matemática sen(x) toma el argumento x, calcula el seno del ángulo x dado, y devuelve el seno calculado de x. De la misma manera, un método de Java puede tomar argumentos, ejecutar un cálculo y devolver una respuesta.

El resto del encabezado de `main` contiene demasiadas palabras misteriosas, cuya explicación puede resultarle confusa en este punto. En capítulos posteriores, cuando se encuentre mejor preparado, se explicarán dichas palabras con más detalle. Por ahora, es correcto tratar el encabezado del método `main` como una línea de texto que simplemente se copia y se pega abajo del encabezado de la clase. Es comprensible que esto pueda no serle cómodo. Por ello, en el resto de esta sección se explican los detalles del encabezado `main`.

Explicación de los detalles del encabezado del método `main`

Ahora explicaremos los detalles de las palabras reservadas que aparecen a la izquierda del encabezado del método `main: public, static, void`. Como se mencionó anteriormente, la palabra `public` es un modificador de acceso: otorga los permisos para que el método `main` sea accesible al "público". Debido a que `main` es el punto de inicio para todos los programas en Java, debe ser públicamente accesible.

Mientras la palabra `public` especifica quién puede acceder al método `main` (todo mundo), la palabra `static` especifica cómo acceder al método `main.` Con un método no `estático`, se debe realizar trabajo extra antes de acceder a éste.[4] Por otro lado, a un método de tipo `estático` se puede acceder sin realizar ningún tipo de trabajo extra. Ya que `main` es el punto de inicio para todos los programas de Java, debe ser inmediatamente accesible y, por tanto, requiere de la palabra `static.`

Ahora revisaremos la tercera palabra reservada en el encabezado de `main: void`. Es necesario recordar que un método es como una función matemática: calcula algo y devuelve el valor calculado. Bueno, de hecho, un método de Java a veces devuelve un valor y a veces no devuelve nada. La palabra

[4] Para acceder a un método no-`estático` (de manera más formal, llamado método de instancia), se debe crear primero una instancia a un objeto. La creación de instancias a objetos se explica en el capítulo 6.

`void` indica que un método no devuelve nada. Puesto que el método `main` no devuelve nada, entonces se utiliza `void` en el encabezado del método `main`.

Lo siguiente es la porción de código `(String[] args)` en el encabezado del método `main`. Hay que recordar que una función matemática lleva argumentos. El método `main` también los lleva.[5] Estos argumentos se representan con la palabra `args`. En Java, si se tiene algún tipo de argumento, se requiere indicarle a la computadora qué tipo de valor puede almacenar el argumento. En este caso, el tipo del argumento está definido para ser de tipo `String[]`, lo cual indica a la computadora que el argumento `args` puede almacenar un arreglo de cadena de caracteres. Los corchetes `[]`, indican un arreglo. Un *arreglo* es una estructura que almacena una colección de elementos del mismo tipo. En este caso, `String[]` indica un arreglo que almacena una colección de cadenas de caracteres. Una *string* es una secuencia de caracteres. Se estudiará más acerca de las cadenas de caracteres en este capítulo, en la sección 3.22, y sobre arreglos en el capítulo 10.

3.6 Paréntesis de llave

En el programa Sueño, se insertaron paréntesis llave de apertura, {, debajo del encabezado de la clase y del encabezado del método `main`, y se insertaron paréntesis llave de cierre, }, en la parte inferior del programa. Los paréntesis llave identifican agrupaciones para los usuarios y para las computadoras. En el programa Sueño, los paréntesis llave de la parte alta y baja agrupan los contenidos de la clase entera, y los paréntesis llave interiores agrupan los contenidos del método `main`. Para mayor legibilidad, se debe colocar un paréntesis llave de apertura alineada con la misma columna del primer carácter del renglón anterior. Observe en el siguiente fragmento de código cómo los paréntesis llave de apertura se colocan de manera correcta.

```
public class Sueño
{
  public class void main(String[] args)
  {
    System.out.println("¡Tengo un sueño!");
  }
} // fin de la clase Sueño
```

El primer paréntesis llave se coloca inmediatamente después del encabezado de clase, y el segundo aparece inmediatamente después del primer carácter en el encabezado del método `main`. Por simple legibilidad, debe colocarse un paréntesis llave de cierre en una línea en la misma columna como si fuera socio del paréntesis llave de apertura. Vuelva al fragmento de código anterior y observe cómo se colocan correctamente los paréntesis llave.

3.7 `System.out.println`

En el programa Sueño, el método contiene esta sentencia:

```
System.out.println ("¡Tengo un sueño!");
```

La sentencia `System.out.println` le indica a la computadora imprimir algo. La palabra `System` se refiere a la computadora. `System.out` se refiere a la salida en el sistema de cómputo: el monitor. La palabra `println` (se pronuncia "print line") se refiere al método de impresión de un mensaje en la pantalla de la computadora. La sentencia anterior se referiría a la llamada del método `println`. Se *llama* a un método cuando se desea ejecutarlo.

El paréntesis que aparece después del método `println` contiene el mensaje que será impreso. La sentencia anterior imprime este mensaje en la pantalla de una computadora:

```
¡Tengo un sueño!
```

[5] A pesar de que el método `main` lleva argumentos, es raro aquel que hace uso de los mismos. Los programas del libro no utilizan los argumentos del método `main`.

```
/**********************************************************************
* Dichos.java
* Dean & Dean
*
* Este programa imprime varios dichos.
**********************************************************************/

public class Dichos
{
  public class void main(String[] args)
  {
    System.out.println("El futuro no es lo que solía ser.");
    System.out.println(
      "Siempre recuerda que tú eres único, exactamente igual que los demás.");
    System.out.println("Si no eres parte de la solución," +
      " eres parte del problema.");
  } // fin del main
} // fin de la clase Dichos
```

Esto conecta/concatena las dos cadenas separadas en dos.

Salida:

```
El futuro no es lo que solía ser.
Siempre recuerda que tú eres único, exactamente igual que los demás.
Si no eres parte de la solución, eres parte del problema.
```

Figura 3.2 Programa Dichos y su salida asociada.

Observe el uso de las comillas dobles en la sentencia `System.out.println("¡Tengo un sueño!");`. Para imprimir un grupo de caracteres (por ejemplo, Yo, espacio, t, e, n, g, o, …), se deben agrupar. Como se vio en el capítulo 2, las comillas dobles se usan para poner en forma conjunta los caracteres que forman una cadena literal.

Observe el punto y coma al final de la sentencia `System.out.println("¡Tengo un sueño!");`. Un punto y coma en el lenguaje Java es como un punto y coma en el lenguaje natural. Indica el final de una oración. Es necesario poner un punto y coma al final de cada sentencia `System.out.println`.

Se estará llamando constantemente al método `System.out.println`, por lo que convendría su memorización. Para ayudar a memorizarlo se recomienda pensar en las iniciales de cada palabra: "SOP", que representarían cada una de las palabras: `System.out` y `println`. No olvide que la S debe ir en mayúscula, y el resto de las letras en minúsculas.

El método `System.out.println` imprime un mensaje y luego pasa el inicio de la siguiente línea. Lo que significa que si hay otra llamada al método `System.out.println`, éste imprimirá en la siguiente línea. El siguiente ejemplo ilustra lo anterior.

Un ejemplo

En el programa *Sueño*, se imprime una sola línea: "¡Tengo un sueño!" En el siguiente ejemplo se imprimen múltiples líneas de diferentes longitudes. Lea el programa Dichos de la figura 3.2, y su salida asociada. Observe cómo la segunda llamada al segundo método `println` produce una línea de salida separada, y cómo la segunda llamada al método `println` es demasiado larga para caber en una sola línea, y que se colocó el texto en la parte de abajo junto con el otro paréntesis.

La llamada al tercer método `println` es mayor que la segunda, y como tal, no cabría en dos líneas si se dividiera con la pura cadena de texto y el paréntesis restante. En otras palabras, no funcionaría:

```
System.out.println(
  "Si no eres parte de la solución, eres parte del pr
```

Sin espacio suficiente.

Así pues, se dividió la llamada al tercer método `println` a mitad de la cadena que será impresa. Para dividir una cadena de texto, es necesario poner comillas de apertura y de cierre alrededor de las dos subcadenas, y es necesario colocar un signo + entre ambas. Observe las comillas dobles y el signo + en la tercera llamada al método `println` en la figura 3.2.

3.8 Compilación y ejecución

Hasta este punto se ha expuesto únicamente la teoría detrás del código Java (la teoría detrás del código del programa Sueño y la teoría detrás del código del programa Dichos). Ahora es necesario que lo capture en una computadora, lo compile y lo ejecute. Después de todo, para aprender a programar se requiere de mucha práctica. ¡Es un "deporte de contacto"! Se han proporcionado muchos tutoriales sobre el sitio Web del libro que guían al lector a través del proceso de compilación y ejecución de programas sencillos de Java. Es recomendable que ahora se tome su tiempo para trabajar a su propio ritmo con uno más de dichos tutoriales. El resto de la presente sección cubre algunos conceptos básicos relacionados con la compilación y la ejecución. Observe que esos conceptos y otros detalles adicionales se cubren con más detalle en los tutoriales.

Después de la captura del código fuente del programa en una computadora, debe guardarse en un archivo cuyo nombre concuerde con el nombre de la clase más la extensión .java. Por ejemplo, debido a que el nombre del programa es `Sueño`, el nombre del archivo debe ser `Sueño.java`.

Después de salvar el código fuente del programa con el nombre apropiado, es necesario crear el código byte[6] mediante el envío del archivo con el código objeto al compilador de Java. Al compilar el código fuente, el compilador genera un archivo de programa con código byte, cuyo nombre se compone con el nombre de la clase más la extensión `.class`. Por ejemplo, puesto que el nombre del programa es Sueño, y el nombre de la clase es `Sueño`, el nombre del archivo con código byte será `Sueño.class`.

El siguiente paso, después de crear el archivo con código byte, es ejecutarlo. Para ejecutar un programa Java, se requiere enviar el archivo en código byte a la máquina virtual de Java (JVM).

3.9 Identificadores

Hasta esta parte del capítulo ha aprendido Java observando el código. A la larga deberá aprender escribiendo su propio código. Cuando lo haga, deberá asignar nombres a los componentes del programa. Java tiene ciertas reglas para nombrar a los componentes del programa. Estas reglas se estudian a continuación.

Un *identificador* es el término técnico para el nombre de un componente de programa: el nombre de una clase, el nombre de un método, etc. En el programa, `Sueño` era el identificador para el nombre de la clase, y `main` era el identificador para el nombre del método.

Los identificadores deben estar formados enteramente por letras, dígitos y caracteres como signos de moneda ($) y/o guión bajo (_). El primer carácter no debe ser un dígito. Si un identificador no sigue estas reglas, el programa no compilará.

Las reglas de convención de código son menos extensas que las de compilación en lo que se refiere a identificadores. Las convenciones de codificación sugieren que se limiten los identificadores a sólo letras y dígitos. No utilizar signos de moneda ni el guión bajo (excepto para las constantes nombradas que se describirán más adelante). También se sugiere el uso de letras minúsculas para todas las letras identificadoras, excepto:

- El Nombre de la clase debe ir con mayúscula. Por ejemplo, la clase `Sueño` inicia con una S mayúscula.
- Poner juntas las palabras en un identificador de varias palabras, utilizando una letra mayúscula en la primera letra de la segunda palabra, la tercera, etc. Por ejemplo, si un método imprime el color favorito, un nombre apropiado de dicho método sería `imprimirColorFavorito.`

Quizá la regla más importante de las convenciones de codificación de los identificadores sea la que dice que los identificadores deben ser descriptivos. Volviendo al ejemplo del método que imprime el co-

[6] El código byte, definido en el capítulo 1, es una versión binaria codificada del código fuente. La computadora no puede ejecutar código fuente, pero puede ejecutar código byte.

lor favorito: `imprimeColorFavorito`, su nombre es totalmente descriptivo; pero ¿qué tal si utilizamos `colorfav`? Por supuesto que no, en lo absoluto. Algunos programadores gustan de utilizar abreviaturas (como "fav") en sus identificadores. Esto funciona bien a veces, pero no en todos los casos. Se recomienda alejarse de las abreviaturas, a menos que éstas sean estándar. La utilización de palabras completas y con significado en los identificadores ayuda en la autodocumentación. Un programa está *autodocumentado* si el código mismo explica el significado sin necesidad de un manual o de muchos comentarios.

Si rompe una regla de convención de código, no afecta la habilidad del programa para compilar, pero sí perjudica a la legibilidad del mismo. Suponga que tiene el método `cncs` que imprime la lista de las 40 canciones más populares de la semana. Aunque la palabra `cncs` podría funcionar, conviene renombrarla como `imprimirCancionesMasPopulares` para mejorar la legibilidad del programa.

3.10 Variables

Hasta este punto, los programas que se han presentado no han hecho suficiente; sólo han impreso un mensaje. Si se desea hacer más que eso, es necesario poner valores en variables. Una variable en Java puede almacenar sólo un tipo de valor. Por ejemplo, una variable de tipo entero sólo puede almacenar enteros, y una de tipo cadena de caracteres sólo puede almacenar cadena de caracteres.

Declaración de variables

¿Cómo sabe la computadora qué tipo de datos puede almacenar una variable en particular? Antes de utilizar una variable, se debe declarar su tipo en una *sentencia de declaración*.

La sintaxis de las sentencias de declaración es la siguiente:

<tipo> <lista-de-variables-separadas-por-comas>;

Declaraciones ejemplo:

```
int renglon, columna;
String nombreEstudiante;      // el nombre del estudiante
String apellidoEstudiante;    // apellido del estudiante
int idEstudante;
```

En cada sentencia de declaración, la palabra a la izquierda especifica el tipo de variable o variables a la derecha. Por ejemplo, en la primera sentencia de declaración, `int` es el tipo para las variables `renglon` y `columna`. El tener un tipo de variables `int` significa que las variables `renglon`, `columna` sólo podrán almacenar enteros (`int` se utiliza para las variables de tipo entero). En la segunda sentencia de declaración, `String` es el tipo para la variable `nombreEstudiante`. El tener una variable de tipo `String` significa que la variable `nombreEstudiante` sólo puede almacenar cadenas de caracteres. En Java, el `String` es un tipo de datos que también tiene la función de nombre de clase. Como se sabe, las convenciones de código dictan que los nombres de clase inicien con una letra mayúscula. Así, el tipo de dato/clase `String` inicia con una letra S mayúscula. Entonces, al referirse a `String` como el tipo de dato, en el código y en el texto conversacional, se utiliza una S mayúscula.

Cuando se declare una variable(s), es importante no olvidar poner un punto y coma al final de la sentencia de declaración. Cuando se declare más de una variable en una sentencia de declaración, es importante no olvidar separar las variables con comas.

Problemas de estilo

El compilador aceptará la declaración de una variable en cualquier bloque de código, siempre y cuando esté antes de la utilización de la misma. Sin embargo, con el fin de lograr legibilidad, se coloca normalmente en la parte superior del método `main`, lo que hará más fácil encontrarlo.

Aunque pueda parecer una pérdida de espacio, se recomienda declarar una sola variable en cada sentencia de declaración. De esta forma, se podrá proporcionar un comentario para cada variable (que es como normalmente se debe hacer).

En el ejemplo anterior, hicimos una excepción a dicha recomendación. Observe cómo las variables `renglon` y `columna` son declaradas juntas con una sentencia de declaración.

```
int renglon, columna;
```

Esto es aceptable porque ambas se relacionan de manera inmediata. Observe que las variables `renglon` y `columna` son nombres estándar que todo programador debería entender. No se requiere la inserción de un comentario como el siguiente:

```
int renglon, columna;   // renglon y columna almacenan el número de renglón
y columna
```

Observe cómo la variable `idEstudiante` se declara sin comentarios:

```
int idEstudante;
```

Esto es aceptable porque el nombre `idEstudiante` es totalmente descriptivo para cualquier persona. Estaría fuera de lugar incluir un comentario como el siguiente:

```
int idEstudante; //Valor de identificación del estudiante
```

Los nombres de variables son identificadores. Así, cuando se nombren las variables, se deben seguir reglas de identificación de las que ya se habló antes. La variable `idEstudiante` está bien nombrada: utiliza letras mayúsculas, excepto en la primera letra de la segunda palabra, Estudiante.

Una última recomendación para la declaración de las variables: intente alinear los comentarios para que todos inicien en la misma columna. Por ejemplo, observe cómo las // inician en la misma columna.

```
String nombreEstudiante;      // el nombre del estudiante
String apellidoEstudiante;    // apellido del estudiante
```

3.11 Sentencias de asignación

Ahora sabe cómo declarar una variable en Java. Una vez que declara una variable, querrá utilizarla, y el primer paso en la utilización de una variable es poner un valor en la misma. Estudiaremos ahora las sentencias de asignación, que permiten asignar/poner un valor en una variable.

Sentencias de asignación en Java

Java simplemente utiliza signo de igual (=) para las sentencias de asignación. Vea el programa calculadoraDeBonos de la figura 3.3. En particular, observe la línea `salario = 50000;`. Ése es un ejemplo de sentencia de asignación en Java. Asigna el valor de 50000 a la variable `salario.`

En el programa de *calculadoraDeBonos*, observe la línea en blanco debajo de las sentencias de declaración. De acuerdo con los principios del buen estilo, se deben insertar líneas en blanco entre agrupaciones lógicas de código. Un grupo de sentencias de asignación se considera usualmente como una agrupación lógica de código, por lo que debe insertarse una línea en blanco debajo de las sentencias de declaración.

A continuación se analizará la sentencia de asignación del fragmento de código del programa `mensajeDeBono`. Observe el * operador. Este * operador ejecuta una multiplicación. Observe el operador +. Si aparece un operador + entre una cadena y algo más (por ejemplo, otra cadena y un número), entonces el operador ejecuta una *concatenación de cadenas de texto.* Lo que significa que la JVM inserta el elemento que se encuentra a la derecha del signo + al elemento que se encuentra a la izquierda del mismo, conformando una nueva cadena de caracteres. En nuestro ejemplo, la expresión matemática, `.02 * salario` se evalúa antes, ya que está dentro de un paréntesis. La JVM, por tanto, concatena el resultado, 100000 a "Bono = $" para formar una nueva cadena "Bono = $100000".

Observe el paréntesis alrededor de `.02 * salario` en la sentencia de asignación de `mensajeDeBono`. Aunque el compilador no requiere del paréntesis, se decidió incluirlo porque mejora la legibilidad del código. La mejora al aclarar que la operación matemática (.02 × salario) está separada de la operación de concatenación. La utilización de un paréntesis discrecional para mejorar la legibilidad es un arte. A veces es útil, pero no se tiene que realizar necesariamente. Si se utilizan paréntesis de forma muy seguida, el código puede parecer un desorden.

```
/*************************************************************
* CalculadoraDeBonos
* Dean & Dean
*
* Este programa calcula e imprime el bono por trabajo de una persona.
*************************************************************/

public class CalculadoraDeBonos
{
  public class void main(String[] args)
  {
    int salario;           // salario de una persona
    String mensajeDeBono;  // especifica el bono de trabajo

    salario = 50000;
    mensajeDeBono = "Bono = $" + (.02 * salario);
    System.out.println(mensajeDeBono);
  } // fin del main
} // fin de la clase calculadoraDeBonos
```

Figura 3.3 Programa CalculadoraDeBonos.

En la sentencia de asignación del `salario`, observe el 50000. Podría estar tentado a insertar una coma dentro de este número para mejorar su legibilidad; esto es, podría estar tentado a introducir 50 000. Si así lo hace, el programa no compilaría con éxito. En el lenguaje Java, los números no tienen comas. Desafortunadamente, esto facilita introducir accidentalmente números ceros de manera incorrecta en los números largos. ¡Se aconseja contar los ceros!

Rastreo

Como parte de la presentación de un programa, a menudo se pide su rastreo. El rastreo fuerza a entender los detalles del programa a medida que se avanza en su flujo. Para establecer el rastreo, recomendamos proporcionar un encabezado de código para el encabezado de cada variable y para su salida. Posteriormente, ejecute cada sentencia, inicie con la primera sentencia en el método `main`. Para las sentencias de asignación, escriba un signo de interrogación (?) en la columna de las variables declaradas, con lo que se indica que la variable existe, pero aún no tiene valor. Para sentencias de asignación, escriba el valor asignado en la columna de variables. Para una sentencia impresa, escriba el valor impreso en la columna de salida.[7]

Para su primer rastreo con Java, facilitaremos las cosas. En lugar de solicitar al lector realizar un rastreo por su cuenta, se le solicitará únicamente completar el de la figura 3.4. Pero deberá asegurarse de que ha entendido cómo se llenan todos los valores de las columnas.[8]

3.12 Sentencias de inicialización

Una sentencia de declaración especifica un tipo de dato para una variable en particular. Una sentencia de asignación pone un valor dentro de una variable. Una sentencia de inicialización es una combinación de sentencias de declaración y de sentencias de asignación: especifica un tipo de dato para una variable y pone un valor dentro de la misma.

[7] Si el lector requiere mayor información sobre el rastreo, consulte el capítulo 2, sección 2.11.

[8] Si se ejecuta el fragmento de código en una computadora, se verá que se obtendrá un 0 al final de la asignación (`Bono = 1000.0`). El valor del .0 tendrá mayor sentido cuando se aprenda acerca de expresiones mixtas y sobre promoción de tipos de datos en una parte superior del presente capítulo.

```
1  int salario;
2  String mensajeDeBono;
3
4  salario = 50000;
5  mensajeDeBono = "Bono = $" + (.02 * salario);
6  System.out.println(mensajeDeBono);
```

línea#	Salario	mensajeDeBono	Salida
1	?		
2		?	
4	50000		
5		Bono = $1000	
6			Bono = $1000

Figura 3.4 Cálculo de un bono: fragmento de código y su flujo asociado.

El lenguaje Java es un lenguaje que requiere que el tipo de variables sea explícitamente establecido (lo que en inglés se denomina *strongly typed*), lo que significa que todos los tipos de variables sean fijas. Una vez que una variable se declara, no se puede redeclarar. Por tanto, sólo se tiene una sentencia de declaración para una variable en particular. De la misma manera, puesto que una sentencia de inicialización es una forma especializada de sentencia de declaración, sólo se puede tener una sentencia de inicialización para una variable en particular.

He aquí la sintaxis para una sentencia de inicialización:

```
<tipo> <variable> = <valor>;
```

Y he aquí algunos ejemplos de inicialización:

```
String nombre= "John Doe";  // nombre del estudiante
int horasCredito = 0;       // número total de horas
```

La variable `nombre` se declara tipo `String` y se le asigna el valor inicial de "John Doe".[9] La variable `horasCredito` se declara tipo `int` y su valor inicial es 0.

A continuación se presenta una forma alternativa de hacer lo mismo, mediante sentencias de declaración y asignación (en lugar de utilizar sentencias de inicialización):

```
String nombre;          // nombre del estudiante
int horasCredito;       // horas crédito totales del estudiante

nombre = "John Doe";
horasCredito = 0;
```

Cualquiera de las dos técnicas es buena: inicialización o declaración/asignación. Se verán ambas formas en un ejemplo de la vida real. La inicialización tiene el beneficio de compactación. La declaración/asignación tiene el beneficio de dejar más espacio en la declaración para un comentario.

3.13 Tipos de datos numéricos: `int, long, float, double`

Enteros

Antes se mencionó un tipo numérico de Java: `int`. Ahora hablaremos de los tipos numéricos con más detalle. Las variables que almacenan valores enteros (por ejemplo, 1 000, −22) se deben declarar nor-

[9] John Doe es un filtro comúnmente utilizado en Estados Unidos y Gran Bretaña cuando el nombre de una persona real se desconoce. Se utiliza aquí como valor por omisión para el nombre de un estudiante. Sirve como indicativo de que el nombre real del estudiante aún no se ha llenado.

malmente con el tipo de datos `int` o `long`. Un número entero es el que no tiene punto decimal ni parte fraccionaria.

Un tipo de datos `int` utiliza 32 bits de memoria; uno `long`, 64 (dos veces más bits que el `int`). El rango de almacenamiento de valores en una variable de tipo `int` va desde los -2 mil millones hasta los $+2$ mil millones. El rango de almacenamiento de valores en las variables de tipo `long` está entre -9×10^{18} y $+9 \times 10^{18}$, aproximadamente. A continuación se presenta un ejemplo que declara la variable `idEstudiante` como de tipo `int` y a `distanciaViajadaPorSatelite` como de tipo `long`:

```
int idEstudiante;
long distanciaViajadaPorSatelite;
```

Si intenta almacenar un número verdaderamente grande (un número mayor a los 2 mil millones) en una variable de tipo `int`, obtendrá un mensaje de error "*integer number too large*" (número entero demasiado largo) cuando compile su programa. Usted se preguntará ¿por qué no declarar siempre las variables de tipo entero como tipo `long` en lugar de `int`? Una variable tipo `int` ocupa menos memoria de almacenamiento. Y la utilización de menos memoria de almacenamiento significa que la computadora ejecutará más rápido porque hay mayor espacio libre. Por lo que en función de velocidad/eficiencia, se recomienda utilizar un tipo `int` en lugar de una `long` para una variable que almacene menos de 2 mil millones.[10] Si no se tiene la seguridad de que una variable se emplea para almacenar valores mayores a los 2 mil millones, se recomienda ir a la segura y utilizar el tipo `long`. Si se quiere la mayor precisión posible en lo que se refiere a cálculos financieros, se recomienda convertir todo en centésimos y utilizar variables de tipo `long` para almacenar esos valores.

Números de punto flotante

En Java, los números que contienen un punto decimal (por ejemplo, 66. y -1234.5) se denominan números de *punto flotante*. ¿Por qué? Porque un número de punto flotante se puede escribir en diversas formas aumentando (flotando) su punto decimal. Por ejemplo, el número -1234.5 se puede escribir con su equivalente como -1.2345×10^3. ¿Observó cómo el punto decimal ha "flotado" a la izquierda de la segunda versión del número?

Hay dos tipos de números de tipo flotante: `float` y `double`. Un número de tipo `float` utiliza 32 bits de memoria; uno `double`, 64. Un número de tipo `double` se llama "doble" porque utiliza dos veces más bits que el de tipo flotante.

A continuación, un ejemplo que declara la variable `gpa` como tipo `float`, y `costo` como tipo `double`.

```
float gpa;
double costo;
```

El tipo de datos `double` se utiliza mucho más seguido que el tipo de datos `float`. Se recomienda declarar las variables de punto flotante como tipo `double` en lugar de `float` porque 1) las variables de tipo `double` pueden almacenar un rango más amplio de números[11] y porque 2) las variables de tipo `double` pueden almacenar números con una mayor precisión. Una mayor precisión significa más dígitos significativos. Se puede confiar en 15 dígitos significativos para una variable tipo `double`, pero sólo en seis para una de tipo `float`.

Seis dígitos significativos pueden parecer demasiados, pero para muchos casos, no son suficientes. Con sólo seis dígitos significativos se pueden generar errores de precisión en los programas que utilizan punto flotante cuando se presente una operación matemática (adición, multiplicación, etc.). Si dicho programa ejecuta un número significativo de operaciones matemáticas, por tanto los errores de precisión se vuelven no triviales. Así, a manera de regla general, se recomienda utilizar `double` en lugar de `float`

[10] La sugerencia de utilizar `int` por razones de eficiencia es válida, pero es importante recalcar que la diferencia de velocidad es notable sólo en ciertas ocasiones. Se nota únicamente si se tienen muchos números tipo `long` y se tiene sólo una pequeña cantidad de memoria disponible, como cuando se ejecuta un programa en un asistente personal digital (PDA).

[11] Una variable `float` puede almacenar valores positivos entre 1.2×10^{-38} y $3.4 \times 10^{+38}$ y valores negativos entre $-3.4 \times 10^{+38}$ y -1.2×10^{-38}. Una variable `double` puede almacenar números positivos entre 2.2×10^{-308} y $1.8 \times 10^{+308}$ y valores negativos entre $-1.8 \times 10^{+308}$ y -2.2×10^{-308}.

para programas que ejecuten un número significativo de operaciones de punto flotante. Puesto que la precisión es particularmente importante en lo que se refiere a dinero, medidas científicas y medidas en ingeniería, se recomienda utilizar `double` en lugar de `float` para cálculos que involucren estas operaciones.

Asignaciones entre los diferentes tipos

Ha aprendido acerca de las asignaciones de valores enteros en variables enteras y de punto flotante en variables de punto flotante, pero no ha aprendido acerca de asignaciones donde los tipos sean diferentes.

La asignación de un valor entero en una variable de tipo flotante funciona bastante bien. Observe este ejemplo:

```
double saldoCuentaBancaria= 1000;
```

La asignación de un valor entero a una variable de punto flotante es como colocar una caja pequeña dentro de otra grande. El tipo `int` abarca aproximadamente 2 mil millones. Es fácil colocar 2 mil millones dentro de una "caja" `double` porque un `double` abarca hasta 1.8×10^{308}.

Por otro lado, la asignación de un valor de punto flotante a una variable de tipo entero es como poner un objeto largo en una caja pequeña. Por omisión eso es ilegal.[12] Por ejemplo, lo siguiente generaría un error:

```
int temperatura = 26.7;
```

Puesto que 26.7 es un valor de punto flotante, no puede ser asignado a la variable de tipo `int`, `temperatura`. Esto tiene sentido cuando se da uno cuenta de que es imposible almacenar .7, la parte fraccional de 26.7 en una variable de tipo `int`. Después de todo, las variables `int` no almacenan fracciones, sólo números completos.

La siguiente sentencia generaría un error:

```
int cuenta = 0.0;
```

La regla dice que es incorrecto asignar un valor de tipo punto flotante a una variable entera. 0.0 es una variable con valor de tipo punto flotante; no importa que la porción fraccional de 0.0 sea insignificante (es .0); 0.0 es de cualquier manera un valor de punto flotante, y siempre será ilegal asignar un valor de tipo punto flotante a una variable entera. Ese tipo de error se conoce como *compile-time error* (error de tiempo de compilación) o *compilation error* (error de compilación) porque el error es identificado por el compilador durante el proceso de compilación.

Más adelante, en este libro, se proporcionarán detalles adicionales acerca de los tipos de datos entero y de punto flotante. No requiere de estos detalles por ahora, pero si no puede esperar, consulte el capítulo 11, sección 11.2.

3.14 Constantes

Hemos utilizado valores numéricos y valores de cadenas de caracteres en los ejemplos, pero no hemos dado nombres formales para los mismos. Los valores numéricos y de cadena de caracteres se denominan *constantes*. Se denominan constantes porque sus valores son fijos, no cambian. He aquí unos ejemplos:

Constantes enteros	Constantes de punto flotante	Constantes de cadena de caracteres
`8`	`-34.6`	`"Hola, Bob"`
`-45`	`.009`	`"yo"`
`2000000`	`8.`	`"perro"`

Para que una constante sea de tipo punto flotante, debe contener un punto decimal, pero los números a la derecha del punto decimal son opcionales. Así 8. y 8.0 representan la misma constante de punto flotante.

¿Cuál es el tipo por omisión de valor para las constantes enteras: `int` o `long`? El lector probablemente señalará `int`, ya que entero suena parecido a `int`. Y dicha asunción es correcta, el tipo de valor

[12] Aunque esta asignación es normalmente ilegal, se puede hacer agregando más código. Específicamente, se puede hacer si se agrega un operador de conversión. Más adelante, en el presente capítulo, se describirán dichos operadores.

normal para una constante entera es `int`. A continuación se presentan los siguientes ejemplos tipo entero (8, −45 y 2000000), todas son constantes de tipo `int`.

¿Cuál es el tipo por omisión de valor para las constantes de punto flotante: `float` o `double`? Aunque seguramente se sentirá tentado a decir `float`, después de lo señalado en la sección anterior, no debería sorprenderse de que por omisión para una constante de tipo punto flotante sea `double`.

Intente identificar los errores de compilación en el siguiente fragmento de código:

```
float gpa = 2.30;
float mpg;
mpg = 28.6;
```

Las constantes 2.30 y 28.6 entran por omisión como tipo `double`, cuyo tipo de número utiliza 64 bits. Los 64 bits no pueden meterse a la fuerza en las variables de 32 bits `gpa` y mpg, por lo que el código genera mensajes de error "posible loss of precision" (posible pérdida de precisión).

Utilice un tipo de dato más largo.

Hay dos posibles soluciones para este tipo de errores. La solución más sencilla es utilizar variables de tipo `double` en lugar de las de tipo `float` todo el tiempo. A continuación se presenta otra solución: forzar explícitamente las constantes de tipo flotante a ser de tipo `float` mediante la utilización de un sufijo `f` o `F`, tal como se muestra a continuación:

```
float gpa = 2.30f;
float mpg;
mpg = 28.6F;
```

Dos categorías de constantes

Las constantes pueden clasificarse en dos categorías: de código duro y nombradas. Las constantes que se han estudiado hasta ahora corresponden a la categoría de constantes de código duro. Una *constante de código duro* es un valor explícitamente especificado. Las constantes de código duro también se denominan *literales*. "Literal" es un buen término descriptivo porque las literales se refieren a elementos que son interpretados literalmente; por ejemplo, 5 significa 5, "hola" significa "hola". En las siguientes sentencias, la diagonal invertida (/) es el operador de la división, y `299792458.0` es una constante de código duro (o literal):

```
retardoPropagación = distancia / 299792458.0;
```

Asumiendo que este fragmento de código es parte de un programa que calcula el retraso en los mensajes enviados a través del espacio, ¿cuál es el significado detrás del número `299792458.0`? Ninguno obvio, ¿verdad? Continúe leyendo.

En el espacio, las señales de los mensajes viajan a la velocidad de la luz. Debido a que tiempo es = distancia/velocidad, el tiempo que tarda una señal de mensaje en viajar desde un satélite igual a la distancia del satélite dividida entre la velocidad de la luz. Así, en el fragmento correspondiente, el número `299792458.0` representa la velocidad de la luz.

El fragmento del código anterior es un tanto confuso. El significado detrás de la constante del código duro `299792458.0` puede ser muy claro para los expertos en ciencia, pero no lo es para el resto de las demás personas. Para una mejor solución utilice una constante nombrada.

Constantes nombradas

Una *constante nombrada* es una que tiene un nombre asociado. Por ejemplo, en este fragmento de código, `VELOCIDAD_DE_LUZ` es una constante nombrada.

```
final double VELOCIDAD_DE_LUZ = 299792458.0; // en metros/seg
. . .
retrasoPropagacion = distancia / VELOCIDAD_DE_LUZ;
```

Como el lector habrá podido discernir en este código, una constante nombrada es en realidad una variable. Ahora hay un oxímoron: una constante es una variable. Observe cómo `VELOCIDAD_DE_LUZ` se declara como una variable de tipo `double`, y se inicializa con el valor `299792458.0`. ¿En que difiere la inicialización de `VELOCIDAD_DE_LUZ` de las inicializaciones que se han visto anteriormente? La palabra `final` aparece a la izquierda.

La palabra reservada `final` es un *modificador*: modifica `VELOCIDAD_DE_LUZ` para que su valor quede como fijo o "final". Y al ser fijado es el punto completo de una constante nombrada `final`. El modificador `final` le indica a la computadora que genere un error si el programa intenta cambiar el valor de la variable `final` en un momento posterior.

Las convenciones de codificación estándar sugieren que se pongan en mayúsculas todos los caracteres en una constante nombrada y que se utilice un guión bajo para separar las palabras en una constante de múltiples palabras. Ejemplo: `VELOCIDAD_DE_LUZ`. La razón de que se usen mayúsculas es que resaltan las cosas. Y lo que se desea es que las constantes nombradas llamen la atención porque representan valores especiales.

Constantes nombradas *versus* constantes de código duro

No todas las constantes se deberían nombrar así. Por ejemplo, si es necesario inicializar la variable `cuenta` a 0, es mejor emplear código duro 0, como a continuación:

```
int cuenta = 0;
```

Entonces, ¿cómo saber en qué momento utilizar una constante de código duro *versus* una constante nombrada? Utilice una constante nombrada si eso hace el código más fácil de entender. La inicialización anterior de cuenta es claramente la forma de hacerlo. Si se remplaza el 0 con una constante nombrada (por ejemplo, `int cuenta = VALOR_INICIAL_CONTEO`), no se mejora la claridad en relación con la utilización de una constante de código duro. Por otro lado, el siguiente código no resulta claro:

```
retrasoPropagacion = distancia / 299792458.0;
```

Al reemplazar `299792458.0` con una constante nombrada `VELOCIDAD_DE_LUZ`, sí se mejora la claridad, por lo que conviene realizar el cambio a la constante nombrada.

Dos son los principales beneficios de utilizar las constantes nombradas:

1. Las constantes nombradas hacen más fácil la documentación del código y por tanto más entendible.
2. Si un programador necesita cambiar en cualquier momento el valor de una constante nombrada, el cambio es sencillo: se localiza la inicialización de la constante en la parte superior del método y se cambia su valor de inicialización. Esto implementa el cambio automáticamente en cualquier parte del programa. No hay peligro de olvidar cambiar una de las muchas ocurrencias de algún valor constante. Hay consistencia.

Facilite el cambio.

Un ejemplo

Para poner en práctica lo que se ha aprendido sobre el tema de constantes, se presenta un programa completo. En el programa convertidorDeTemperatura de la figura 3.5, se realiza conversión de temperatura de grados Fahrenheit en grados Celsius. No pierda de vista las dos constantes nombradas en la parte superior del programa: 1) `PUNTO_CONGELACION`, la cual se inicializa a 32.0, y 2) `FACTOR_CONVERSION`, la cual se inicializa con la expresión 5.0/9.0. Por lo regular, deseará inicializar cada constante nombrada a una simple constante de código duro. Por ejemplo, el valor de inicialización de `PUNTO_CONGELACION` es 32.0, pero es importante asegurarse de que es válido utilizar una expresión constante para el valor de inicialización de una constante nombrada. Por ejemplo, el valor de inicialización de `FACTOR_CONVERSION` es 5.0/9.0. Esta expresión puede considerarse como una expresión constante porque se utilizan valores constantes y no variables.

En el programa `convertidorDeTemperatura`, la siguiente sentencia se encarga de ejecutar la conversión:

```
celsius = FACTOR_CONVERSION * (fahrenheit – PUNTO_CONGELACION);
```

Con la utilización de constantes nombradas, `FACTOR_CONVERSION` y `PUNTO_CONGELACION`, se puede dar cierto sentido al código de conversión. Sin las constantes nombradas, la sentencia aparecería así:

```
celsius = 5.0 / 9.0 * (fahrenheit – 32.0);
```

La razón matemática 5.0/9.0 podría distraer la atención de algunos lectores, quienes podrían requerir de algún tiempo para preguntarse acerca del significado de los números 5.0 y 9.0. Mediante la utilización

```
/**********************************************************************
* ConvertidorDeTemperatura.java
* Dean & Dean
*
* Este programa convierte temperaturas de grados Fahrenheit a grados Celsius
**********************************************************************/

public class ConvertidorDeTemperatura
{
  public class void main(String[] args)
  {
    final double PUNTO_CONGELACION = 32.0;
    final double FACTOR_CONVERSION = 5.0 / 9.0;
    double fahrenheit = 50;   // temperatura en Fahrenheit
    double celsius;           // temperatura en Celsius

    celsius = FACTOR_CONVERSION * (fahrenheit - PUNTO_CONGELACION);
    System.out.println(fahrenheit + " grados Fahrenheit = " +
      celsius + " grados Celsius.");
  } // fin del main
} // fin de la clase convertidorDeTemperatura
```

Salida:

```
50.0 grados Fahrenheit = 10.0 grados Celsius.
```

Figura 3.5 Programa ConvertidorDeTemperatura y su salida asociada.

de la constante nombrada `FACTOR_CONVERSION`, se le indica al lector "No preocuparse por esto; es sólo un factor de conversión al que algunos científicos llegaron". Si para alguna persona resultan poco familiares las lecturas en escala Fahrenheit, no entenderán el significado de los 32.0. Al utilizar la constante nombrada `PUNTO_CONGELACION` las cosas se vuelven más claras.

3.15 Operadores aritméticos

Hasta ahora se ha hablado acerca de números en lo que se refiere a declaración de variables numéricas, a asignación de números y al funcionamiento de las constantes numéricas. Además, se han mostrado algunos ejemplos de la utilización de números en expresiones matemáticas. En ésta y las dos siguientes secciones, se estudiarán las expresiones con más detalle. Una *expresión* es una combinación de operandos y operadores que ejecutan un cálculo. Los operandos son variables y constantes. Un operando es un símbolo, tal como + o −, que ejecuta una operación. En esta sección se analizarán los operadores aritméticos para los tipos de datos numéricos. Más adelante se analizarán los operadores para otro tipo de datos.

Adición, sustracción y multiplicación

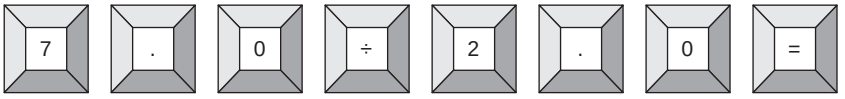
Los operadores aritméticos de Java +, − y * deben ser familiares para el lector. Ejecutan operaciones de adición, sustracción y multiplicación, respectivamente.

División de punto flotante

Java ejecuta la operación de la división de manera diferente, dependiendo de si los números/operandos divididos son enteros o de punto flotante. Se comenzará hablando de la división de punto flotante.

Cuando la Máquina Virtual de Java (JVM) ejecuta divisiones con números de punto flotante, ejecuta "división de calculadora". Se denomina "división de calculadora" porque la división de punto flotante en

Java trabaja de la misma manera que lo hace una calculadora estándar. Por ejemplo, si se introduce la siguiente operación en una calculadora, ¿cuál es el resultado que se obtiene?



El resultado es 3.5. De la misma manera, la siguiente línea en Java imprime 3.5:

```
System.out.println(7.0 / 2.0);
```

Es importante advertir que esa calculadora emplea la tecla ÷ para la división, mientras que Java utiliza el carácter /.

Para explicar los operadores aritméticos, se requerirá evaluar varias expresiones. Para simplificar este análisis, se utilizará el símbolo ⇒, el cual significa "es igual a". Por tanto, la siguiente línea de código dice 7.0/2.0 es igual a 3.5:

7.0/2.0 ⇒ 3.5

La siguiente línea solicita determinar a qué es igual

5/4. ⇒ ?

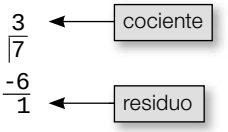
5 es de tipo `int` y 4. es `double`. Éste es un ejemplo de una *expresión mezclada*. Una expresión mezclada es aquella que contiene operandos con diferentes tipos de datos. Los valores `double` se consideran más complejos que los de tipo `int` porque los valores `double` contienen un componente fraccional. Cuando se encuentra una expresión fraccional, la JVM de manera temporal lo *promueve* a un tipo de operando menos complejo para que se ajuste al más complejo de los dos, y después la JVM aplica el operador. En la expresión 5/4., la JVM promueve el número 5 a un `double` y después ejecuta la división de punto flotante con dos valores de punto flotante. La expresión es igual a 1.25.

División de enteros

Cuando la JVM ejecuta la división con números enteros, ejecuta una "división tipo escolar", y se denomina así porque la división entre números enteros es similar a la que el lector efectuaba a mano en su época de educación básica. ¿Recuerda cómo se calculaba la operación de la división con dos operadores? Se obtenía un cociente y un residuo. De la misma manera, Java tiene la capacidad de calcular tanto el cociente como el residuo cuando se efectúa una división entre números enteros. Si se utiliza el operador %, entonces se determina el residuo. El operador % se conoce más formalmente como el operador *módulo*. Observe los siguientes ejemplos:

7 / 2 ⇒ 3
7 % 2 ⇒ 1

Estas operaciones corresponden a la notación aritmética empleada durante la educación básica:



Se proporcionarán muchos problemas de evaluación como éste. Como medida saludable, se recomienda verificar al menos algunos de los resultados mediante su ejecución en un equipo de cómputo. Para ejecutar las expresiones, deberán insertarse éstas en sentencias de impresión en pantalla y ponerlas en un programa de prueba. Por ejemplo, para ejecutar las siguientes expresiones, utilice el programa ExpresionesPrueba de la figura 3.6.

Imprima los detalles para observar lo que hace la computadora.

La figura 3.6 también ilustra estos ejemplos adicionales:

8 / 12 ⇒ 0
8 % 12 ⇒ 8

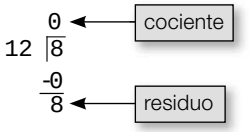
```
public class ExpresionesPrueba
{
  public static void main(String[] args)
  {
    System.out.println("7 / 2 = " + (7 / 2));
    System.out.println("7 % 2 = " + (7 % 2));
    System.out.println("8 / 12 = " + (8 / 12));
    System.out.println("8 % 12 = " + (8 % 12));
  } // fin del main
} // fin de la clase ExpresionesPrueba
```

Salida:

```
7 / 2 = 3
7 % 2 = 1
8 / 12 = 0
8 % 12 = 8
```

Figura 3.6 Programa ExpresionesPrueba y su salida asociada.

Y he aquí la siguiente notación aritmética de educación básica:



3.16 Evaluación de expresiones y precedencia de operadores

Los ejemplos anteriores fueron demasiado básicos, contienen un solo operador, por lo que su evaluación resultaba bastante sencilla. Las expresiones son a veces muy complicadas. En esta sección se tratará de evaluar esas expresiones más complicadas.

Ejemplo del marcador promedio en el boliche

Suponga que intenta calcular el marcador promedio de tres juegos de boliche. ¿Funcionaría la siguiente sentencia?

```
promedioBoliche = juego1 + juego2 + juego3 / 3;
```

El código parece razonable, pero no basta confiar en lo que nuestro instinto nos dice que es razonable. Para ser un buen programador, se requiere estar seguro. Para codificar, es importante enfocar la expresión a la derecha de: `juego1 + juego2 + juego3 / 3`. De manera específica, cabría preguntarse uno mismo: "¿qué operador se ejecuta primero: el de la izquierda o el de la división?" Para responder esta pregunta, conviene consultar la tabla de precedencia de operadores.

Tabla de precedencia de operadores

La clave para entender expresiones complicadas es entender la precedencia de los operadores mostrados en la figura 3.7. Por favor estudie dicha tabla en este momento.

La tabla de precedencia de operadores podría requerir de alguna explicación. Lo que significa que si uno de los operadores superiores aparece en una expresión junto con uno de los operadores de la parte de abajo, entonces el operador superior se ejecuta primero. Por ejemplo, si los operadores * y + aparecen en

1. agrupación con paréntesis:
   (<*expression*>)
2. operadores unarios:
   +x
   -x
   (<*tipo*>) x
3. operadores de multiplicación y division:
   x * y
   x / y
   x % y
4. operadores de suma y resta:
   x + y
   x - y

Figura 3.7 Tabla de precedencia abreviada (ver apéndice 2 para la tabla completa). Los grupos de operadores en la parte alta de la tabla tienen precedencia más alta que los grupos en la parte baja. Todos los operadores dentro de un grupo en particular tienen igual precedencia y se evalúan de izquierda a derecha.

la misma expresión, entonces la operación con el operador * se ejecuta antes que aquella que contiene el signo + (porque el grupo del operador * es mayor en la tabla de precedencia que la del grupo +). Si el paréntesis aparece dentro de una expresión, entonces los elementos dentro del paréntesis se ejecutan antes que los elementos que están fuera de éste (porque el paréntesis aparece arriba de la tabla).

Si una expresión tiene más de dos operadores pertenecientes, las operaciones se deben al mismo grupo (de los grupos de la figura 3.7), entonces evalúe de izquierda a derecha. En matemáticas, eso se conoce como *asociación de derecha a izquierda*. En Java, eso significa que los operadores que aparecen a la izquierda deben ejecutarse antes que los que estén a la derecha. Por ejemplo, puesto que los operadores * y / pertenecen al mismo grupo y ambos aparecen en la misma expresión, pero el signo / está más a la izquierda que *, entonces la división se ejecuta antes que la multiplicación.

Los operadores en el segundo grupo de arriba se denominan *operadores unarios*. Un operador unario es el que se aplica a un solo operando. El operador unario + es cosmético, no hace nada. El operador unario – (negación) invierte el signo del operando. Por ejemplo, si la variable `x` tiene un valor de 6, entonces `–x` se evalúa a un 6 negativo. El operador (<*tipo*>) representa una operación de conversión. Se estudiarán las operaciones de conversión más adelante, en este capítulo.

De vuelta al ejemplo de promedio de marcadores en boliche

Volviendo al ejemplo de promedio de marcadores en boliche y aplicando lo que acaba de estudiar sobre precedencia de operadores, ¿considera que la siguiente sentencia calcula correctamente el promedio de los marcadores en los tres juegos de boliche?

```
promedioBoliche = juego1 + juego2 + juego3 / 3;
```

No. La tabla de precedencia de los operadores indica que el operador / tiene mayor precedencia que el operador +, por lo que se ejecuta primero la operación de división. Después de que la JVM ha efectuado la división de `juego3/3`, entonces suma al resultado la operación de `juego1 + juego2`. La forma correcta de calcular el promedio es obtener la suma de los tres juegos primero y luego dividir dicha suma entre 3. En otras palabras, se necesita forzar el operador + a ejecutarse primero. La solución es utilizar un paréntesis como en la siguiente sentencia:

```
promedioBoliche = (juego1 + juego2 + juego3) / 3;
```

Práctica de evaluación de expresiones

El cálculo manual nos ayuda a entender.

Ahora realizaremos una práctica de evaluación de expresiones para asegurarnos de que realmente se entendió el material de precedencia de operadores. Dadas las siguientes inicializaciones:

```
int a = 5, b = 2;
double c = 3.0;
```

¿Cómo se evalúa la siguiente expresión?

```
(c + a / b) / 10 * 5
```

He aquí la siguiente solución:

```
1. (c + a / b) / 10 * 5 ⇒
2. (3.0 + 5 / 2) / 10 * 5 ⇒
3. (3.0 + 2) / 10 * 5 ⇒
4. 5.0 / 10 * 5 ⇒
5. 0.5 * 5 ⇒
6. 2.5
```

En las expresiones de la resolución de problemas, se recomienda que se muestre cada paso del proceso de evaluación para que la solución sea sencilla de seguir. En la solución anterior, se muestra cada paso, y también los números de línea. No se requiere mostrar dichos números, pero en el ejemplo anterior se hace para ayudar con las explicaciones. De la línea 1 a la 2 se reemplazan las variables con sus valores. De las líneas 2 a 3 se evalúa el operador de prioridad más alta, la / que está dentro del paréntesis. De las líneas 3 a 4 se evalúa el siguiente operador con mayor prioridad, que sería el signo + dentro del paréntesis. Se recomienda el estudio de las líneas siguientes por cuenta propia del lector.

Continuando con la práctica de evaluación de problemas. Dadas las siguientes inicializaciones:

```
int x = 5;
double y = 3.0;
```

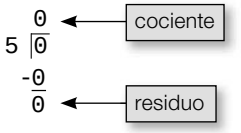
¿Cómo evaluar la siguiente expresión?

```
(0 % x) + y + (0 / x)
```

He aquí la solución:

```
(0 % x) + y + (0 / x) ⇒
(0 % 5) + 3.0 + (0 /5) ⇒
0 + 3.0 + (0 / 5) ⇒
0 + 3.0 + 0 ⇒
3.0
```

Quizá la parte más delicada de la solución anterior es la evaluación de 0 % 5 y de 0 / 5. Ambas se evalúan a 0. La notación de operación en educación básica muestra por qué:



3.17 Más operadores: incremento, decremento y asignación compuesta

Hasta ahora hemos cubierto operadores matemáticos de Java que corresponden a operaciones que se encuentran en libros de matemáticas: adición, sustracción, multiplicación y división. Java proporciona operadores matemáticos adicionales que no tienen contraparte en los libros de matemáticas. En esta sección, se hablará de los operadores de incremento, decremento y asignación compuesta.

Operadores de incremento y decremento

Es una práctica común en un programa de cómputo contar el número de veces que algo ocurre. Por ejemplo, ¿ha visitado una página Web que despliegue el número de "visitantes"? El número de visitantes lo rastrea un programa que cuenta el número de veces que la página Web se descarga en el navegador de alguien. Puesto que el conteo es una práctica común en los programas, existen operadores especiales para realizarlo. El operador de incremento (++) cuenta de uno en uno. El operador de decremento (− −) cuenta hacia abajo de uno en uno.

He aquí una forma de incrementar el valor de la variable *x*:

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

Y he aquí cómo hacerlo utilizando el operador de incremento:

```
x++;
```

Las dos técnicas son equivalentes en términos de funcionalidad. Los programadores experimentados en Java casi siempre utilizan la segunda forma en lugar de la primera. Y el estilo propio sugiere el uso de la segunda. Así pues, se sugiere al usuario utilizar la segunda.

He aquí una forma de decrementar el valor de la variable *x*:

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

Y he aquí cómo hacerlo utilizando el operador de decremento:

```
x--;
```

Una vez más, se sugiere utilizar la segunda forma.

Operadores de asignación compuesta

Ahora se revisarán los operadores *de asignación compuesta*: `+=`, `-=`, `*=`, `/=` y `%=`.

El operador `+=` actualiza una variable mediante la suma de un valor específico a dicha variable. He aquí una forma de incrementar `x` en 3 unidades:

```
x = x + 3;
```

Y he aquí, mediante la utilización del operador `+=`:

```
x += 3;
```

Busque la forma corta.

Las dos técnicas son equivalentes en términos de su funcionalidad. Los programadores expertos en Java casi siempre utilizan la segunda forma, y el estilo propio dicta hacer uso de ésta también. Así pues, se sugiere al lector utilizar la segunda.

El operador `-=` actualiza una variable mediante la resta de un valor específico de dicha variable. He aquí una forma de incrementar en 3:

```
x = x - 3;
```

Y he aquí, mediante la utilización del operador `-=`:

```
x -= 3;
```

Una vez más, se sugiere utilizar la segunda forma.

Los operadores `*=`, `/=` y `%=` funcionan de manera similar a la de los operadores `+=` y `-=`, por lo que no se cansará al lector con explicaciones detalladas de estos tres operadores restantes, aunque se recomienda su estudio con los ejemplos que se presentan a continuación:

```
x += 3;        ≡    x = x + 3;
x -= 4;        ≡    x = x - 4;
x *= y;        ≡    x = x * y;
x /= 4;        ≡    x = x / 4;
x %= 16;       ≡    x = x % 16;
x *= y + 1;    ≡    x = x * (y +1);
```

Los ejemplos muestran las sentencias de operadores de asignación a la izquierda de su forma equivalente en la forma larga. El símbolo ≡ significa "es equivalente a". Es más conveniente hacer uso de las formas a la izquierda, pero no se deben ignorar las del lado derecho. Demuestran cómo funciona el operador de asignación.

El ejemplo en la parte de abajo es el único operador de asignación que utiliza una expresión en lugar de un simple valor; esto es, la expresión a la derecha del operador de asignación `*=` es y `+ 1`, en lugar de simplemente el número 1. Para casos como éstos, la forma de asignación compuesta resulta un poco confusa, por tanto, es aceptable preferir la forma de asignación larga que la de asignación compuesta.

¿Por qué los operadores `+=`, `-=`, `*=`, `/=` y `%=` se denominan operadores de asignación compuesta? Porque componen/combinan la operación matemática con la de asignación. Por ejemplo, el operador `+=` ejecuta una suma y una asignación. La parte de la adición es obvia, pero ¿qué sucede con la de asignación? El operador `+=` sí ejecuta por tanto a la asignación ya que a la variable a la izquierda de dicho operador se le asigna un nuevo valor.

3.18 Rastreo

Para asegurarnos de que realmente estamos entendiendo los operadores de incremento, decremento y los de asignación compuesta, realizaremos el rastreo de un programa que contiene estos operadores. A comienzos del capítulo se mostró un rastreo, pero éste era para un fragmento de código muy limitado: el fragmento de código contenía dos sentencias de asignación y eso era todo. En esta sección se presenta un rastreo con un flujo más complicado.

Observe el programa pruebaConOperadores de la figura 3.8 y la tabla de rastreo asociada. En particular, observe las tres primeras líneas bajo el encabezado de la tabla de rastreo. Contienen los valores iniciales de las variables. Para variables declaradas como partes de una inicialización, su valor inicial es el valor de inicialización. Para variables declaradas sin una inicialización, se dice que su valor inicial es *basura* porque su valor actual se desconoce. Se utilizará un signo de interrogación para indicar que se tiene un valor basura.

Hay que ponerse uno mismo en el lugar de la computadora.

Se sugiere que cubra hasta la parte de abajo del rastreo, y trate de completarlo. Una vez hecho esto, compare la respuesta con la que se presenta en la tabla de trazado de la figura 3.8.

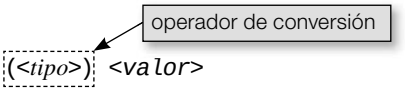
Hay diferentes modos para los operadores de incremento y decremento: el modo prefijo y el modo posfijo. Más adelante, en el libro, se explicarán los modos y se proporcionarán los detalles de cómo trabajar en un ambiente de trazado. No se requiere de estos detalles por ahora, pero si no puede esperar más, consulte el capítulo 11, sección 11.5.

3.19 Conversión de tipos

Hasta ahora se han descrito los operadores aritméticos simples (`+`, `-`, `*`, `/`, `%`), los de incremento y decremento (`++`, `--`), y los de asignación compuesta (`+=`, `-=`, `*=`, `/=`, `%=`). En esta sección se tratará otro operador, el de conversión (*cast*, en inglés).

Operador de conversión

Al escribir un programa, en ocasiones será necesario convertir un valor en un tipo de dato diferente. Puede utilizarse el operador de conversión para ejecutar ese tipo de conversiones. A continuación se presenta la sintaxis:



Como se muestra en la parte de arriba, un operador de conversión consta de un tipo de dato dentro de un paréntesis. Se debe colocar un operador de conversión a la izquierda del valor que se desea convertir.

Suponga que tiene una variable llamada `interes` que almacena el interés de una cuenta bancaria como un dato de tipo `double`. Si deseara extraer la porción en dólares del interés y almacenarla en una

```
1   public class PruebaDeOperadores
2   {
3     public static void main(String []args)
4     {
5       int x;
6       int y = 2;
7       double z = 3.0;
8
9       x = 5;
10      System.out.println("x + y + z = " + (x + y + z));
11      x += y;
12      y++;
13      z--;
14      z *= x;
15      System.out.println("x + y + z = " + (x + y + z));
16    } // fin del main
17  } // fin de la clase PruebaDeOperadores
```

Trace:

línea#	x	y	z	salida
5	?			
6		2		
7			3.0	
9	5			
10				x + y + z = 10.0
11	7			
12		3		
13			2.0	
14			14.0	
15				x + y + z = 24.0

Figura 3.8 Programa pruebaDeOperadores y su flujo asociado.

variable de tipo `int` llamada `interesEnDolares`, tendría que utilizar un operador de conversión de tipo `int`, como se muestra a continuación:

```
interesEnDolares = (int) interes;
```

El operador de conversión tipo `int` regresa la porción del número entero del valor a convertir, truncando la porción fraccionaria. Así, si el `interes` contiene el valor 56.96, después de la asignación, la variable `interesEnDolares` contiene el valor 56. Observe que la operación de conversión no cambia el valor de `interes`. Después de la asignación, `interes` seguirá conteniendo el valor 56.96.

Uso del paréntesis para convertir una expresión

Si es necesario convertir más de un simple valor o una variable, entonces es importante asegurarse de colocar un paréntesis a la expresión entera que se desea convertir. Observe el siguiente ejemplo:

```
double tasaInteres;
double saldo;
int interesEnDolares;
. . .
interesEnDolares = (int) (saldo * tasaInteres);
```

Aquí son necesarios los paréntesis

En la asignación a la variable `interesEnDolares`, `saldo * tasaInteres` es la fórmula para calcular el interés. El fragmento de código ejecuta fundamentalmente la operación como en el anterior fragmento de código. Extrae la porción en dólares del interés y lo almacena en una variable de tipo `int` llamada `interesEnDolares`. La diferencia es que el interés en esta ocasión aparece como una expresión (`saldo * tasaInteres`), en lugar de aparecer como una simple variable, `interes`. Puesto que se desea convertir el operador para aplicarlo a la expresión completa, se debe poner un paréntesis a toda la expresión `saldo * tasaInteres`.

En el fragmento de código anterior, ¿qué habría sucedido si no se hubiese puesto un paréntesis alrededor de toda la expresión `saldo * tasaInteres`? La conversión habría aplicado sólo al primer valor a la derecha del operador, es decir, a `saldo`, en lugar de hacerlo en la expresión completa. Lo anterior tiene sentido cuando se consulta la tabla de precedencia de operadores. La tabla de precedencia de operadores muestra que el operador de conversión tiene una precedencia muy alta; por lo que sin el paréntesis, el operador de conversión se ejecutaría antes que el de multiplicación, y la conversión aplicaría únicamente a la variable `saldo`; lo cual conduce a un cálculo incorrecto de los intereses en dólares.

Utilización de la conversión de punto flotante para forzar la división de punto flotante

Suponga que tiene una variable llamada `puntosObtenidos` que almacena los puntos de calificación que obtiene un estudiante durante el semestre en todas sus clases. Suponga que tiene una variable llamada `numDeClases` que almacena el número de clases tomadas por el estudiante. El promedio de calificaciones del alumno (PCA) se calcula dividiendo los puntos obtenidos entre el número de clases. En la siguiente sentencia `puntosObtenidos` y `numDeClases` son de tipo `int` y `pca` es de tipo `double`. ¿La sentencia calcula de manera correcta el PCA?

```
pca = puntosObtenidos / numDeClases;
```

Compare la salida con lo que esperaba.

Suponga que `puntosObtenidos` suma 14 y `numDeClases` es equivalente a 4. Se desearía que `pca` diera un valor de 3.5 (porque 14 ÷ 4 = 3.5) Pero, sorpresa, `pca` obtiene un valor de 3. ¿Por qué? Porque el operador / ejecuta una división entre dos operandos de tipo `int`. La división entera significa que el valor devuelto es el cociente. El cociente de dividir 14 ÷ 4 es 3. La solución es forzar la división de punto flotante mediante la introducción del operador de conversión. He aquí el código correcto:

```
pca =(double) puntosObtenidos / numDeClases;
```

Después de convertir la variable `puntosObtenidos` en `double`, la JVM mira la expresión mezclada y promueve `numDeClases` a `double`. Por tanto, la siguiente división de punto flotante toma lugar.

Para este ejemplo no se requiere poner paréntesis a la expresión `puntosObtenidos / numDeClases`. Si así se hiciera, el operador / tendría mayor precedencia que el de conversión, y la JVM ejecutaría la división (división entera) antes de ejecutar la operación de conversión.

Más adelante, en este libro, se proporcionan detalles adicionales acerca de las conversiones de tipos. Por ahora no requiere de estos detalles, pero si no puede esperar, puede consultar el capítulo 11, sección 11.4.

3.20 Tipo char y secuencias de escape

Anteriormente, cuando almacenamos o imprimimos texto, lo hicimos con grupos de caracteres de texto (cadenas), no con caracteres individuales. En esta sección utilizaremos el tipo `char` para trabajar con caracteres individuales.

Tipo char

Si se sabe que una variable almacenará un simple carácter, entonces debe utilizarse una de tipo `char`. He aquí un ejemplo que declara una variable tipo `char` llamada `ch` y a la que se le asigna la letra *A*.

```
char ch;
ch = 'A';
```

Observe la letra 'A'. Es una literal tipo char. Las literales char deben declararse con una comilla sencilla. Esta sintaxis difiere de la de las literales tipo cadena, las cuales deben aparecer entre comillas dobles.

¿Cuál es el caso de tener variables de tipo char? ¿Por qué no utilizar cadenas de un solo carácter para todos los procesos que involucren caracteres? Porque para una aplicación que manipula muchos caracteres individuales, es mucho más eficiente (rápido) utilizar variables tipo char, que son simples, en lugar de variables de tipo cadena, que son más complejas. Por ejemplo, el software para ver páginas Web tiene que leer y procesar caracteres individuales mientras éstos son bajados a la computadora. El procesar caracteres individuales es más eficiente si éstos se almacenan como variables tipo char en lugar de variables de caracteres (*String*).

Concatenación de cadenas de caracteres con variables tipo char

¿Recuerda cómo se utiliza el símbolo + para concatenar dos cadenas juntas? Pues bien, también puede utilizar el símbolo + para concatenar una variable de tipo char con una cadena de caracteres. ¿Qué piensa que imprime el siguiente fragmento de código?

```
char nombre, intermedio, apellido;   //iniciales de la persona

nombre = 'J';
intermedio = 'S';
apellido = 'D';
System.out.println("¡Hola, " + nombre + intermedio + apellido + '!' );
```

Y la siguiente sería la salida:

```
¡Hola, JSD!
```

Secuencias de escape

Por lo regular, es fácil imprimir caracteres. Basta con ponerlos dentro de una sentencia System.out.println. Pero algunos son difíciles de imprimir. Para los caracteres de difícil impresión, se utilizan las *secuencias de escape* como el carácter de tabulación. Una secuencia de escape está compuesta por una diagonal invertida (\) y otro carácter. Consulte algunas de las secuencias de escape más populares de Java en la figura 3.9.

Si se imprime el carácter de tabulación (\t), el cursor de la pantalla se mueve al siguiente tabulador. El cursor de la pantalla es la posición en que la computadora imprime a continuación. Si se imprime el carácter de salto de línea (\n), el cursor de la pantalla se mueve al principio de la siguiente línea.

A continuación un ejemplo de cómo se pueden imprimir dos encabezados de columnas: SALDO e INTERÉS, separados por un tabulador, y seguidos por una línea en blanco.

```
System.out.println("SALDO" + '\t' + "INTERÉS" + '\n');
```

Observe que las secuencias de escape aparecen inmediatamente después de los caracteres, por lo que para imprimir un tabulador y un salto de línea, se colocó una comilla sencilla en cada uno de ellos.

Por lo regular, el compilador interpreta las comillas dobles, las sencillas o la diagonal invertida como *caracteres de control*. Un carácter de control tiene como función proporcionar un significado espe-

\t	mover el cursor al siguiente tabulador
\n	salto de línea: avanza a la primera columna en el siguiente renglón
\r	avanza a la primera columna en el renglón actual
\"	imprime una literal que utiliza comilla doble
\'	imprime una literal con comilla sencilla
\\	imprime una diagonal invertida

Figura 3.9 Secuencias de caracteres comunes.

cial al carácter que sigue. El carácter de comilla doble le indica a la computadora que los caracteres subsiguientes son parte de una cadena literal de caracteres. Por el contrario, el carácter de comilla simple le indica a la computadora que el carácter subsiguiente es una literal de tipo `char`. El carácter de control de la diagonal invertida le indica a la computadora que el siguiente carácter a ser interpretado es un carácter secuencia de escape.

Pero ¿qué tal que se quisiera imprimir alguno de estos tres caracteres y omitir la funcionalidad de los caracteres de control? Para hacer eso, se deben anteceder los caracteres de control (comilla doble, comilla simple, diagonal invertida) con una diagonal invertida. La diagonal invertida anula la funcionalidad del carácter de control y por tanto permite que el subsiguiente carácter sea impreso como tal. Si lo anterior no tiene sentido, todo lo que tiene que saber el usuario es lo siguiente:

Para imprimir una sentencia doble, utilizar `\"`.

Para imprimir una comilla sencilla, utilizar `\'`.

Para imprimir una diagonal invertida, utilizar `\\`.

Suponga que quisiera imprimir este mensaje:

```
"Hola.java" está almacenado en el archivo c: \javaPgms.
```

He aquí cómo hacerlo:

```
System.out.println('\"' + "Hola.java" + '\"' +
" está almacenado en el directorio c:" + '\\' "javaPgms.");
```

Inserción de una secuencia de escape dentro de un String

Escriba una sentencia de impresión que genere este encabezado para un reporte de especificaciones de una computadora:

```
TAMAÑO DE DISCO DURO     TAMAÑO DE RAM ("MEMORIA")
```

De manera específica, la sentencia de impresión debería generar un tabulador, un encabezado con el título TAMAÑO DE DISCO DURO, dos tabuladores más y un encabezado de columna con el título TAMAÑO DE RAM (“MEMORIA”) y dos líneas en blanco. He aquí la solución:

```
System.out.println('\t' + TAMAÑO DE DISCO DURO" +'\t'+ '\t'+
  "TAMAÑO DE RAM (" + ' \"' + "MEMORIA" + '\"' + ")" + '\n' + '\n');
```

Busque la forma corta. Lo anterior está bastante enmarañado. Afortunadamente, hay una mejor forma. Una secuencia de caracteres está diseñada para utilizarse como cualquier otro carácter dentro de una cadena de texto, por lo que es perfectamente aceptable insertar secuencias de escape dentro de cadenas de caracteres y omitir los símbolos + y las comillas dobles. Por ejemplo, aquí se presenta una solución alternativa para el encabezado del reporte de las especificaciones de la PC donde los símbolos + y las comillas dobles tienen que omitirse.

```
System.out.printl("\TAMAÑO DISCO DURO" +'\t'+'\t' +
"TAMAÑO MEMORIA (" + '\"' +”RAM”+ '\"' +")" + '\n' +'\n');
```

Ahora todo está dentro de una cadena literal. Al omitir los signos + y las comillas sencillas, el desorden se reduce y todo mundo está feliz. (Excepción: los alumnos preescolares del autor John aman el desorden por lo que no recurrirían a esta segunda solución).

Origen de la palabra “escape” en las secuencias de escape

¿Cuál es el origen de la palabra “escape” en las secuencias de escape? La diagonal invertida provoca un “escape” del comportamiento normal de un carácter en específico. Por ejemplo, si aparece la letra `t` en una sentencia de impresión, la computadora normalmente imprime `t`. Si aparece `\t`, la computadora se escapa de imprimir `t` e imprime un carácter tabulador. Si aparece un carácter de comilla doble (`"`), la computadora lo trata como el principio o el final de una cadena literal de caracteres. Si aparece un `\"`, la computadora escapa del comportamiento inicio/final de una cadena, y en su lugar imprime la doble comilla.

Más adelante, en este libro, se presentan los detalles de una sintaxis relativamente avanzada correspondiente al tipo `char`. No se requieren los detalles por ahora, pero si el usuario no puede esperar, puede encontrar los mismos en el capítulo 11, sección 11.3.

3.21 Variables primitivas *versus* variables de referencia

A través de este capítulo hemos definido y estudiado varios tipos de variables: `String`, `int`, `long`, `float`, `double` y `char`. Ahora es tiempo de obtener una perspectiva amplia de las dos categorías diferentes de variables: variables primitivas y variables de referencia.

Variables primitivas

Una *variable primitiva* almacena una simple pieza de datos. Puede pensarse en una variable primitiva como una pieza de datos que es intrínsecamente indivisible. De manera más formal, se puede decir que es "atómica", porque al igual que el átomo es la base de un "bloque de construcción" y no puede dividirse.[13] Las variables primitivas se declaran con un *tipo primitivo*, que incluye:

`int`, `long`	(tipos enteros)
`float`, `double`	(tipos de punto flotante)
`char`	(tipo carácter)

Existen otros tipos primitivos adicionales (`boolean`, `byte`, `short`) que se estudiarán en los capítulos 4 y 11, pero para la mayoría de las situaciones estos cinco tipos primitivos son suficientes.

Variables de referencia

Mientras una variable primitiva almacena una simple pieza de datos, una *variable de referencia* almacena una localidad de memoria que apunta a una colección de datos. Esta localidad de memoria no es literalmente una dirección de memoria, tal como la calle de un domicilio. Es una abreviación codificada, como la de un número de un apartado postal. Sin embargo, para todo lo que se puede hacer con Java, el valor en una variable de referencia actúa exactamente como una dirección literal de memoria, por lo que se tomará como si así fuera. Se mencionó que la "dirección" de una variable de referencia apunta a una colección de datos. De manera más formal, apunta a un *objeto*. Se estudiarán los detalles acerca de los objetos en el capítulo 6, pero por ahora basta con que el lector sepa que un objeto es una colección de datos relacionados, envueltos en una capa protectora. Para acceder a estos datos, se requiere utilizar una variable de referencia (o *referencia*, más corto) que apunte al objeto.

Las variables de cadena de caracteres (tipo *String*) son un ejemplo de variables de referencia. Una variable de tipo *String* almacena una dirección de memoria que apunta a un objeto de tipo string. El objeto de tipo string almacena los datos: los caracteres de la cadena.

Las variables de referencia se declaran con un *tipo de referencia*. Un tipo de referencia es un tipo que proporciona el almacenamiento para la colección de datos. `String` es un tipo de referencia, y proporciona el almacenamiento para una colección de caracteres. Así, en el siguiente ejemplo, la declaración de `nombre` con una referencia de tipo `String` significa que `nombre` apunta a una colección de caracteres T,h,a,n,h, espacio, N, g, u, y, e, n.

```
String name = "Thanh Nguyen";
```

`String` es sólo uno de muchos tipos de referencia. Las clases, los arreglos y las interfaces son todos tipos de referencia. Los arreglos se estudiarán en el capítulo 10, y las interfaces, en el 13. Los detalles de las clases se estudiarán en el capítulo 6, pero por ahora, es suficiente con saber que una clase es una descripción genérica de los datos en un tipo particular de objeto. Por ejemplo, la clase `String` describe la naturaleza de los datos en objetos de cadena de caracteres. De manera más específica, la clase `String`

[13] La palabra "átomo" viene del griego *a-tomos* que significa indivisible. En 1897, J. J. Thomson descubrió uno de los componentes del átomo (el electrón) y así desvaneció la noción de indivisibilidad del átomo. Sin embargo, a manera de costumbre de su definición original, el término "átomo" todavía se refiere a algo implícitamente indivisible.

indica que cada objeto de cadena de caracteres puede almacenar uno o más caracteres y que los caracteres se almacenan en una secuencia.

Un ejemplo

Ahora analizaremos unos ejemplos de variables primitivas y de variables de referencia. En el siguiente fragmento de código se declaran variables que llevan el registro de los datos básicos de una persona:

```
int nss;           // número de seguridad social
String nombre;     // nombre de la persona
Calendar fechaNac; // fecha de nacimiento de la persona
```

Como se puede observar, tanto `int` como `String` son tipos de datos, `nss` es una variable primitiva y `nombre` es una variable de referencia. En la tercera línea, `Calendar` es una clase, lo que nos indica que `fechaNac` es una variable de referencia. La clase `Calendar` permite almacenar información como mes, día y año.[14] Puesto que `fechaNac` es declarada con la clase `Calendar`, dicha variable puede almacenar elementos como día, mes y año.

3.22 Cadenas de caracteres

Hemos utilizado las cadenas de caracteres de manera reiterada hasta ahora, pero únicamente hemos realizado acciones como el almacenamiento e impresión de sus contenidos, eso es todo. Muchos programas requieren realizar más acciones con las cadenas de caracteres. Por ejemplo, todos los programas de Microsoft Office (Word, Excel, PowerPoint) incluyen utilerías de búsqueda y reemplazo de texto. En esta sección se describe la manera en que Java proporciona la funcionalidad de manejo de cadenas de caracteres en la clase `String`.

Concatenación de cadenas

Como se sabe, las cadenas de caracteres normalmente se concatenan mediante el operador +. Observe que las cadenas de caracteres también pueden concatenarse mediante el operador de asignación compuesta +=. En el siguiente ejemplo, si la cadena de texto `animal` se refiere originalmente a la cadena "perro", posteriormente hace referencia a la cadena "perropez", después de que la sentencia es ejecutada:

```
animal += "pez";
```

Hay que ponerse en el lugar de la computadora.

Le recomendamos realizar el rastreo para asegurarse de que entendió claramente el tema de la concatenación de caracteres. Observe el fragmento de código de la figura 3.10. Intente rastrear el flujo del fragmento de código antes de revisar la solución.

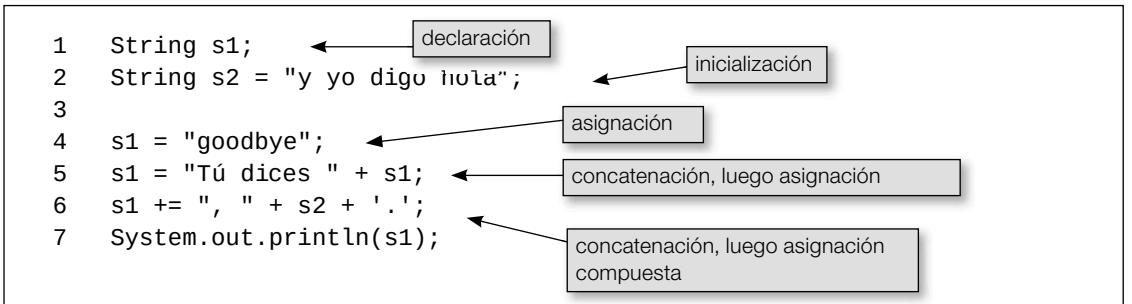
Métodos del objeto String

En la sección anterior se definió al objeto como una colección de datos. Los datos del objeto usualmente están protegidos y se debe acceder a ellos sólo a través de canales especiales. Normalmente, puede accederse a ellos sólo a través de los métodos del objeto. Un objeto de tipo `String` almacena una colección de caracteres, se puede acceder a los caracteres de la cadena a través del método `charAt`, así como a través de otros tres métodos populares: `lenght`, `equals` y `equalsIgnoreCase`. Estos métodos, así como muchos otros de cadenas de caracteres, se encuentran definidos en la clase `String`.

Si el lector desea aprender más acerca de la clase `String` y todos sus métodos, conviene que visite la documentación de Java Sun, en su sitio Web, http://java.sun.com/javase/6/docs/api/, y siga los vínculos que lo llevarán hasta la clase `String`.

Obtenga ayuda de la fuente.

[14] La explicación de la clase `Calendar` con mayor profundidad, está fuera del alcance de este capítulo. Si desea una explicación con más detalle, consulte la documentación de Java Sun en el sitio (http://java.sun.com/javase/6/docs/api/) y buscar `Calendar`.



línea#	s1	s2	salida
1	?		
2		y yo digo hola	
4	adiós		
5	Tú dices adiós		
6	tú dices adiós, y yo digo hola.		
7			Tú dices adiós, y yo digo hola.

Figura 3.10 Fragmento de código y su flujo de código asociado para ilustrar la concatenación de cadena.

El método charAt

Suponga que desea inicializar la variable `animal` con el valor "vaca". Entonces la variable `animal` apunta ahora al objeto de caracteres que contiene cuatro elementos: los cuatro caracteres: 'v', 'a', 'c' y 'a'. Para traer un elemento de los datos (por ejemplo, un carácter), se debe llamar al método `charAt`. `charAt` significa carácter en (del inglés character at). Este método regresa un carácter en una posición específica. Por ejemplo, si `animal` llama al método `charAt` y especifica la tercera posición, entonces `charAt` devuelve 'c' porque la 'c' es el tercer carácter en la palabra "vaca".

Así pues, ¿cómo se llama al método `charAt`? Para responder esta pregunta se comparará este método con la llamada a otro con el que el lector ya se encuentra muy familiarizado: `println`. Vea la figura 3.11.

Observe cómo tanto en la llamada al método `charAt` como al `println` se utiliza esta sintaxis:

<*variable de referencia*> . <*nombre del método*> (<*argumento*>)

En la llamada al método `charAt`, `animal` es la variable de referencia, `charAt` es el nombre del método y 2 es el argumento. El argumento es la parte complicada. El argumento especifica el *índice* del carácter a ser regresado. Las posiciones de los caracteres dentro de una cadena de caracteres em-

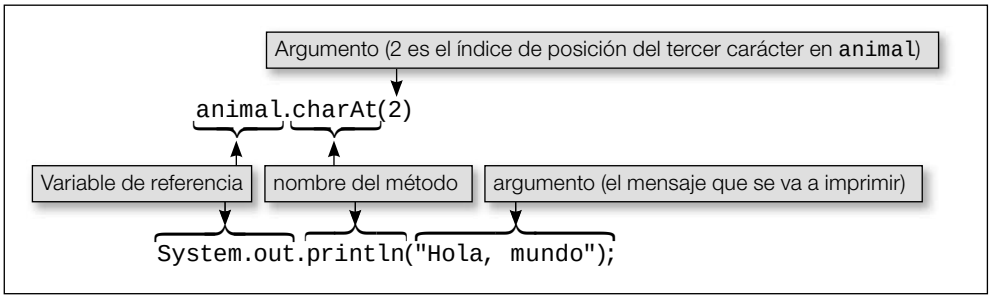


Figura 3.11 Comparación de la llamada del método `charAt` con la llamada al método `println`.

piezan su numeración en la posición del índice cero y no en la del uno. Para enfatizar lo anterior ¡digámoslo de nuevo! Las posiciones en una cadena de caracteres empiezan en el índice cero. Así, si `animal` contiene la cadena "cow". ¿Qué devuelve `animal.charAt(2)`? Como lo indica la siguiente tabla, el carácter 'w' está en el índice 2, por lo que `animal.charAt(2)` devuelve 'w'.

índice:	0	1	2
caracteres de la cadena "cow":	c	o	w

Si se llama al método `charAt` con un argumento que sea negativo o igual o superior al del tamaño de la cadena, el código compilará correctamente, pero no se ejecutará bien. Por ejemplo, suponga que ejecuta el siguiente programa:

```
public class Prueba
{
  public class void main(String[] args)
  {
    String animal = "mamut";
    System.out.println("Ultimo character: " + animal.charAt(5));
  }
}
```

índice inapropiado

Puesto que el último índice en la palabra `mamut` es 4 y no 5, la JVM imprime un mensaje de error. De manera más específica, imprime lo siguiente:

```
Exception in thread "main"
java.lang.StringIndexOutOfBoundsException:
  String index out of range: 5
  at java.lang.String.charAt(String.java:558)
  at Prueba.main(Prueba.java:6)
```

El 5 se refiere al índice en específico; está "fuera de rango".

El 6 se refiere al número de línea en el programa donde ocurrió el error.

Conviene preguntarse: ¿qué está tratando de decirme la computadora?

Al principio, los mensajes de error son intimidatorios y deprimentes, pero poco a poco aprenderá a amarlos. Quizá no los ame, pero aprenderá a apreciar la información que le proporcionan. Se recomienda ver el mensaje de error como una oportunidad de aprendizaje. Hasta este punto, no hay que preocuparse por entender todos los detalles del mensaje de error anterior. Sólo hay que enfocarse en las dos llamadas y las líneas a las que se refieren.

El error de arriba es un ejemplo de *error en tiempo de ejecución*. Un error de tiempo de ejecución es aquel que ocurre mientras el programa se está ejecutando, y provoca que éste termine de forma anormal. Dicho de otra manera, provoca que el programa *colisione*.

El método `length`

El método `length` devuelve el número de caracteres en una cadena de caracteres en particular. ¿Qué imprime el siguiente fragmento de código?

```
String s1 = "hola";
String s2 = "";
System.out.println("número de caracteres en s1 = " + s1.length());
System.out.println("número de caracteres en s2 = " + s2.length());
```

Puesto que la cadena de caracteres `s1` contiene 4 caracteres ('h', 'o', 'l' y 'a'), la primera sentencia imprime lo siguiente:

```
número de caracteres en s1 = 4
```

`s2` es inicializada con el valor `""`. El valor `""` se conoce comúnmente como *cadena vacía*. Una cadena vacía es una cadena que no contiene ningún carácter. Su longitud es cero. La segunda sentencia de impresión presenta lo siguiente:

```
número de caracteres en s2 = 0
```

```
1   String animal1 = "Caballo";
2   String animal2 = "Mosca";
3   String criaturaNueva;
4
5   criaturaNueva = animal1 + animal2;
6   System.out.println(criaturaNueva.equals("CaballoMosca"));
7   System.out.println(criaturaNueva.equals("caballomosca"));
```

línea#	animal 1	animal 2	criaturaNueva	Salida
1	Caballo			
2		Mosca		
3			?	
5			CaballoMosca	
6				verdadero
7				falso

Figura 3.12 Fragmento de código que ilustra al método `equals` y su flujo asociado.

Al llamar al método `charAt`, se requiere insertar un argumento (un índice) en el paréntesis de la llamada al método. Por ejemplo, `animal.charAt(2)`. Por otro lado, al llamar al método `length` no se requiere insertar un argumento en el paréntesis de la llamada al método. Por ejemplo, `s1.length()`. El lector podría pensar: "Sin un argumento, ¿por qué molestarse en poner un paréntesis?" Al llamar a un método, se requiere siempre de un paréntesis, aun cuando aparezca vacío. Sin la presencia de los paréntesis, el compilador no sabrá que la llamada al método es una llamada a un método.

El método `equals`

Para comparar la equivalencia de dos cadenas de caracteres es necesario reejecutar los caracteres en dichas cadenas y verificar que estén colocados en la misma posición, uno a la vez. Afortunadamente, no se requiere escribir código que haga más tediosa la comparación cada vez que se desea ver si dos cadenas son iguales. Sólo se requiere llamar al método `equals`, y éste hará la tediosa comparación de manera automática, más allá de las escenas. De manera más sucinta, el método `equals` devuelve `true` (verdadero) si las dos cadenas contienen la misma secuencia de caracteres. De lo contrario, devuelve `false` (falso).

Hay que ponerse en el lugar de la computadora.

Le recomendamos ahora avanzar paso a paso en el rastreo del flujo para asegurarse de que ha entendido el método `equals`. Vea el fragmento del código en la figura 3.12. Intente realizar el rastreo del fragmento por cuenta propia antes de consultar la solución.

Puesto que `criaturaNueva` contiene el valor "CaballoMosca", el método `equals` devuelve un valor de `true` (verdadero) cuando `criaturaNueva` es comparada con "CaballoMosca". Por otro lado, cuando `criaturaNueva` es comparada con la cadena en minúscula "caballomosca", el método `equals` devuelve un valor de `false` (falso).

El método `equalsIgnoreCase`

A veces, al comparar cadenas de caracteres, se podría desear ignorar el hecho de que las letras sean mayúsculas o minúsculas. En otras palabras, se podría querer que las cadenas "CaballoMosca" y "caballomosca" fueran consideradas iguales. Para probar la igualdad sin sensibilidad a mayúsculas, utilice el método `equalsIgnoreCase`.

¿Qué imprime este fragmento de código?

```
System.out.println("CaballoMOsca".equalsIgnoreCase("caballomosca"));
```

Puesto que el método `equalsIgnoreCase` considera como iguales a las cadenas "CaballoMosca" y "caballomosca", entonces el fragmento de código imprime `true`.

3.23 Entrada: la clase Scanner

Los programas son por lo regular como una calle de doble sentido. Producen salida para desplegar algo en la pantalla de la computadora, y leen entradas del usuario. Hasta este punto, todos los programas de Java y sus fragmentos de código han ido en un solo sentido: han desplegado algo en la pantalla, pero no han leído ninguna entrada. Sin entrada de datos los programas han sido más bien limitados. En esta sección se explicará cómo obtener entradas de datos por parte del usuario. Con las entradas, se podrán escribir programas que sean mucho más flexibles y útiles.

Preguntarse: ¿qué tal si?

Suponga que se le solicita escribir un programa que calcule las ganancias de un fondo de retiro. Si no hay entrada de datos, el programa debe plantear supuestos acerca de las cantidades aportadas, los años antes del retiro, etc. Posteriormente, el programa calcula las ganancias con base en dichos supuestos. Resultado final: el programa sin entradas calcula las ganancias para el plan de un fondo para el retiro específico. Si se utiliza entrada de datos, el programa solicita al usuario proporcionar cantidades de contribución, años antes del retiro, etc. Posteriormente, el programa calculará las ganancias con base en estos datos. Entonces, la pregunta es ¿cuál de los dos programas es mejor: el que solicita los datos o el que no lo hace? El programa que solicita la entrada de datos al usuario es mejor porque le permite hacer planteamientos hipotéticos sobre situaciones como: ¿qué pasaría si contribuyo con más dinero?, ¿qué pasaría si pospongo mi retiro hasta que tenga 90 años?

Fundamentos de entrada de datos

Java proporciona una clase precompilada llamada `Scanner`, que permite obtener entradas ya sea del teclado o de un archivo. La lectura a través de un archivo será tratada en el capítulo 15. Antes de eso, cuando se hable de entrada o lectura de datos, se debe asumir que se está hablando de hacerlo a través del teclado.

La clase `Scanner` no es parte del conjunto fundamental de clases de Java (el core); por lo que si el lector hace uso de la misma, necesita indicarle al compilador dónde encontrarla. Lo cual se realiza importando la clase en el programa. De manera más específica, se requiere incluir la sentencia `import` en la parte superior del programa (justo antes de la sección de inicio).

```
import java.util.Scanner;
```

Los detalles del `import` (como ¿qué es el `java.util`?) serán tratados en el capítulo 5. Por ahora, basta con decir que es necesario importar la clase `Scanner` para preparar el programa para la entrada de datos.

Hay algo más que es necesario saber antes de preparar el programa para la lectura de datos. Se requiere insertar esta sentencia en la parte superior del método `main`.

```
Scanner stdIn = new Scanner(System.in);
```

La expresión `new Scanner(System.in)` crea un objeto. Como el lector ya sabe, un objeto almacena una colección de datos. En este caso, el objeto almacena caracteres introducidos por el usuario a través del teclado. La variable `stdIn` es una variable de referencia, y es inicializada a la dirección del objeto recientemente creado, `Scanner`. Después de la inicialización, la variable `stdIn` permite ejecutar operaciones de entrada de datos (lectura).

Una vez establecido lo anterior, se puede leer y almacenar una línea de entrada llamando al método `nextLine` de la siguiente manera:

```
<variable> = stdIn.nextLine();
```

Ahora se pondrá en práctica lo que se ha aprendido mediante la utilización de la clase `Scanner` y la llamada al método `nextLine` en un programa completo. Vea el programa SaludoAmigable de la figura 3.13. Este programa solicita al usuario introducir su nombre y lo almacena en una variable llamada `nombre`, después imprime un saludo con el nombre del usuario insertado en el saludo.

Observe que la sentencia de impresión "Introduce tu nombre:" en el programa SaludoAmigable utiliza una sentencia `System.out.print` en lugar de `System.out.println`. ¿Recuerda lo que significa el "ln" en el `println`? Significa "línea". Al utilizar `System.out.println` se imprime el mensaje y el cursor en la pantalla se mueve a la siguiente línea. Por otro lado, `System.out.print` imprime un

```
/************************************************************
* SaludoAmigable.java
* Dean & Dean
*
* Este programa despliega un saludo Hola personalizado.
************************************************************/

import java.util.Scanner;

public class SaludoAmigable
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String nombre;
    System.out.print("Introduce tu nombre: ");
    name = stdIn.nextLine();
    System.out.println("¡Hola " + nombre + "!");
  } // fin del main
} // fin de la clase SaludoAmigable
```

Estas dos sentencias crean una conexión de entrada al teclado.

Esto obtiene una línea de entrada.

Figura 3.13 Programa SaludoAmigable.

mensaje y ya. El cursor termina en la misma línea en la que se imprimió el mensaje (al lado derecho del último carácter impreso).

Así pues, cabría preguntarse ¿por qué molestarse en utilizar una sentencia de impresión `print` en lugar de `println`? Porque los usuarios tienden a poner la respuesta justamente al lado del mensaje de petición. Si se hubiese utilizado el `println`, entonces el usuario habría tenido que introducir su respuesta en la siguiente línea. Una cosa más: se colocaron dos puntos y un espacio en blanco al final de la petición. De nueva cuenta, lo razonable es presentar las cosas como los usuarios están acostumbrados a verlas.

Métodos de entrada

En el programa SaludoAmigable, se llamó al método `nextLine` de la clase `Scanner` para obtener una captura de datos. La clase `Scanner` contiene muchos otros métodos que leen datos de entrada de diferentes formas. He aquí algunos de esos métodos:

`nextInt()`	Se salta los espacios dejados en blanco hasta que encuentra un valor de tipo `int`. Devuelve un valor tipo `int`.
`nextLong()`	Se salta los espacios dejados en blanco hasta que encuentra un valor de tipo `long`. Devuelve un valor tipo `long`.
`nextFloat()`	Se salta los espacios dejados en blanco hasta que encuentra un valor de tipo `float`. Devuelve un valor tipo `float`.
`nextDouble()`	Se salta los espacios dejados en blanco hasta que encuentra un valor de tipo `double`. Devuelve un valor tipo `double`.
`next()`	Se salta los espacios dejados en blanco hasta que encuentra un token. Devuelve el token como un valor tipo `String`.

La descripción anterior requiere algunas aclaraciones.

1. ¿Qué son los espacios dejados en blanco?
   Los *espacios en blanco* son aquellos que aparecen vacíos en la pantalla o en la impresora. Esto incluye el espacio entre caracteres, los tabuladores y el carácter de salto de línea. El carácter de nueva línea se genera con la tecla enter. Los *espacios dejados en blanco* son aquellos que aparecen al lado izquierdo de la entrada de datos.

2. ¿Qué sucede si el usuario proporciona una entrada invalidada de datos?
   La JVM imprime un mensaje de error y detiene la ejecución del programa. Por ejemplo, si un usuario introduce 45g o 45.0 en la respuesta a la llamada al método `nextInt()`, la JVM imprime un mensaje de error y detiene el programa.
3. El método `next` busca un token? ¿Qué es un token?
   Se puede pensar en un *token* como una palabra, ya que el método `next` se utiliza usualmente para leer una simple palabra; pero de manera más formal, un *token* es una secuencia de caracteres diferentes a los espacios en blanco. Por ejemplo, "gecko" y "B@a!" son tokens. Pero "monstruo Gila" no es un token por el espacio en blanco que existe entre "monstruo" y "Gila".

Ejemplos

Para asegurarse de que ha comprendido los métodos de la clase `Scanner`, estudie los programas en las figuras 3.14 y 3.15, los cuales muestran cómo utilizar los métodos `nextDouble` y `next`. En particular, ponga atención a las sesiones ejemplo. Las sesiones ejemplo muestran lo que sucede cuando el programa se ejecuta con un conjunto usual de entradas de datos.

En la figura 3.14 observe los caracteres en cursivas para los números `34.14` y `2`. En la 3.15, observe las cursivas en las cadenas `Malallai Zalmai`. Se pusieron cursivas en los valores de entrada con el fin de distinguirlas del resto del programa. Hay que tomar en cuenta que la impresión en cursiva es una técnica pedagógica que se utiliza para un mejor entendimiento de los temas del libro. Los valores de entrada no aparecen con cursivas cuando se muestran en la pantalla de la computadora.

```
/*************************************************************
* ImprimePrecioC.java
* Dean & Dean
*
* Este programa calcula e imprime la cantidad de una orden de compra.
*************************************************************/

import java.util.Scanner;

public class ImprimePrecioC
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double precio;  // precio de compra del artículo
    int cant;       // número de artículos comprados

    System.out.print("Precio de compra del artículo: ");
    precio = stdIn.nextDouble();
    System.out.print("Cantidad: ");
    cant = stdIn.nextInt();
    System.out.println("Total de orden de compra = $" + precio * cant);
  } // fin del main
} // fin de la clase ImprimePrecioC
```

Sesión ejemplo:

```
Precio de compra del artículo: 34.14
Cantidad: 2
Total de orden de compra = $68.28
```

Figura 3.14 Programa ImprimePrecioC que ilustra la utilización del método `nextDouble( )` y `nextInt( )`.

```
/*****************************************************************
* ImprimirIniciales.java
* Dean & Dean
*
* Este programa imprime las iniciales del nombre introducido por el usuario.
*****************************************************************/

import java.util.Scanner;

public class ImprimirIniciales
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String nombre;     // nombre
    String apellido;   // apellido

    System.out.print(
      "Introduzca su nombre y primer apellido separados por un espacio: ");
    nombre = stdIn.next();
    apellido = stdIn.next();
    System.out.println("Sus iniciales son " +
      nombre.charAt(0) + apellido.charAt(0) + ".");
  } // fin del main
} // fin de la clase ImprimirIniciales
```

Sesión ejemplo:

```
Introduzca su nombre y primer apellido separados por un espacio: Malallai Zalmai
Sus iniciales son MZ.
```

Figura 3.15 Programa ImprimirIniciales que ilustra la utilización del método `next()`.

Un problema con el método `nextLine`

El método `nextLine` y otros métodos de la clase `Scanner` no se llevan muy bien al estar juntos. Es correcto utilizar una serie de llamadas al método `nextLine` en un programa. También es correcto utilizar una serie de llamadas a los métodos `nextInt`, `nextLong`, `nextFloat`, `nextDouble` y `next` en un programa; pero, si se utiliza el método `nextLine` y los otros métodos de la clase `Scanner` en el mismo programa, hay que tener cuidado. He aquí por qué.

El método `nextLine` es el único que procesa los espacios dejados en blanco. Los otros métodos los saltan. Suponga que tiene una llamada al método `nextInt` y que teclea 25 y posteriormente presiona la tecla enter. El método `nextInt` lee el número 25 y lo devuelve. El método no lee el carácter de salto de línea (el enter). La llamada al método `nextLine` no salta los espacios dejados en blanco, por lo que se atora con la lectura de lo que se deje de la anterior captura de datos. En el ejemplo anterior, en la llamada a captura de datos se presionó la tecla enter, por lo que la llamada al método `nextLine` se quedó atorada leyéndolo. ¡Oh, oh!

¿Qué pasaría si el método `nextLine` lee un carácter de salto de línea? Abandonaría el proceso porque ha leído un carácter (el carácter de salto de línea marca el final de una línea, aunque sea una muy corta). Por lo que la llamada al método `nextLine` no se espera a leer otra línea, la cual probablemente es la que el usuario espera que lea.

Una solución al problema del método `nextLine` es incluir una llamada extra al método `nextLine` cuyo único propósito sea leer el carácter en blanco del salto de línea. Otra solución es utilizar una variable de referencia al objeto `Scanner` para la entrada a través de `nextLine` (por ejemplo, `stdIn1`) y otra variable de referencia para otra entrada (por ejemplo, `stdIn2`); pero hay que intentar aclarar el

problema juntos. Se intentarán evitar las llamadas al método `nextLine` que sigan cualquiera de las llamadas a los métodos del otro objeto `Scanner`.

A medida que avance en la lectura del libro, verá que la entrada a través del teclado de la computadora y la salida a través de la pantalla son todas las E/S que se requieren para resolver un amplio número de problemas complejos. Aunque, si se tiene una larga cantidad de entradas, podría ser más fácil y más seguro utilizar un simple procesador de textos para escribir esas entradas en un archivo y después releerlas desde el archivo cada que se vuelva a ejecutar el programa. Y si se tiene una larga cantidad de salidas, podría ser más fácil analizar la salida si está almacenada en un archivo. Usted no requiere de la utilización de archivos por ahora, pero si puede esperar, puede encontrar los detalles en el capítulo 15, secciones 15.3 y 15.4. En este momento, tal vez no sea capaz de entender muchos de los detalles en esas secciones posteriores del libro, pero si considera que el pequeño programa que aparece en la figura 15.2 parece sólo una receta, éste le enseñará cómo escribir en un archivo algo que se ha escrito en la pantalla de la computadora. De la misma manera, si lo considera como una receta, también la figura 15.5 le mostrará cómo leer de un archivo algo que leyó en el teclado.

3.24 Apartado GUI: entrada y salida con el objeto `JOptionPane` (opcional)

Esta sección es el segundo acercamiento a la parte opcional de la interfaz gráfica del usuario (GUI). En cada sección del apartado GUI se proporciona una introducción a un concepto de la GUI. En esta sección se describe cómo implementar rudimentarias entradas/salidas (E/S) en una ventana GUI.

Hasta este punto del capítulo se han utilizado *ventanas de consola* para desplegar entradas y salidas de datos. En esta sección se utilizan *ventanas GUI*. ¿Cuál es la diferencia? se preguntará el lector. Una ventana de consola es aquella que puede desplegar únicamente texto, pero no elementos gráficos como botones de textos e imágenes. Por ejemplo, observe la ventana GUI de figura 3.16, que despliega texto (un mensaje de instalación), un botón (de aceptar) y un dibujo (un ícono con una i encerrado en un círculo).

La ventana de la figura 3.16 es un tipo especializado de ventana. Se denomina *ventana de diálogo*. Una ventana de diálogo ejecuta sólo una tarea específica. El cuadro de diálogo de la figura 3.16 ejecuta la tarea de desplegar información (la letra *i* del ícono en el cuadro de texto significa "información"). En secciones posteriores de apartados GUI y en los capítulos 16 y 17 se utilizarán ventanas de propósito general, pero por ahora, nos centraremos en los cuadros de diálogo.

La clase `JOptionPane` y su método `showMessageDialog`

Para desplegar un cuadro de diálogo es necesario utilizar la clase `JOptionPane`. La clase `JOptionPane` no es parte del conjunto fundamental de clases de Java (el core), por lo que si se utiliza la clase `JOptionPane`, es necesario indicarle al compilador dónde encontrarlo. Esto se realiza importando la clase `JOptionPane` en el programa. De manera más específica, se requiere incluir la siguiente sentencia `import` en la parte superior del programa.

```
import java.util.JOptionPane;
```

Observe que el programa DialogoDeInstalacion de la figura 3.17 produce un cuadro de diálogo como el que se muestra en la figura 3.16. El código del programa DialogoDeInstalacion puede parecer familiar al usuario. En la parte alta, tiene una sentencia `import`. Posteriormente, tiene el encabezado estándar de

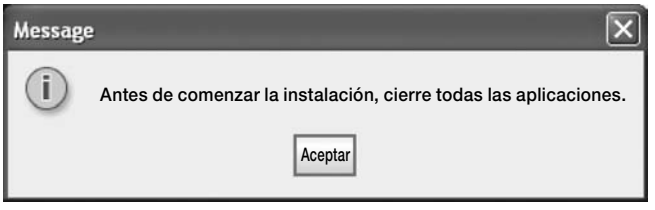


Figura 3.16 Un cuadro de diálogo que despliega información de la instalación.

```
/*********************************************************
* DialogoDeInstalacion.java
* Dean & Dean
*
* Este programa ilustra el cuadro de diálogo del objeto JOptionPane.
*********************************************************/

import javax.swing.JOptionPane;

public class DialogoDeInstalacion
{
  public static void main(String[] args)
  {
    JOptionPane.showMessageDialog(null,
      "Antes de comenzar la instalación, " +
      "cierre todas las aplicaciones.");
  }
} // fin de la clase DialogoDeInstalacion
```

Figura 3.17 Programa DialogoDeInstalacion.

clase, el encabezado estándar del método `main` y los paréntesis de llave. Lo que es nuevo es la llamada al método `JOptionPane.showMessageDialog`.

El método `showMessageDialog` despliega un mensaje en un cuadro de diálogo. He aquí la sintaxis para llamar al método `showMessageDialog`:

```
JOptionPane.showMessageDialog(null, <mensaje>)
```

El método `showMessageDialog` toma dos argumentos. El primero de ellos especifica la posición del cuadro de diálogo en la pantalla de la computadora. Para facilitar las cosas, se mantiene la posición por omisión en el centro de la pantalla. Para colocarlo en la posición por omisión, especifique `null` en el primer argumento. El segundo argumento especifica el mensaje que aparece en el cuadro de diálogo.

Cuadro de diálogo de entrada

Observe que al cuadro de diálogo a menudo se le denomina simplemente *diálogo.* Ambos tipos se consideran como: diálogo de mensaje para salida y diálogo de entrada para lectura. Eso es lo que produce el método `showMessageDialog`. Ahora se estudiará el diálogo de entrada. Ya se ha descrito el diálogo de mensaje. Eso es lo que produce el método `showMessageDialog`. Ahora se analizará el diálogo de entrada.

El diálogo de entrada despliega un mensaje de petición y un cuadro de texto. Los cuatro diálogos de la figura 3.18 son diálogos de entrada. Éstos despliegan un ícono con un signo de interrogación a manera de guía visual de que el diálogo está haciendo una pregunta y está esperando la respuesta del usuario. Al presionar Aceptar se procesa el valor introducido. Al presionar Cancelar se cierra el cuadro de diálogo sin procesar la entrada.

El propósito del diálogo de entrada es leer el valor de un texto introducido por el usuario y almacenarlo en una variable. Para leer valores de texto, llame al método `showInputDialog` de la siguiente manera:

```
<variable tipo String> = JOptionPane.showInputDialog(<mensaje de petición>);
```

Para leer un número es necesario llamar al método `showInputDialog` y luego convertirlo en la cadena de texto leída en número: de manera más específica, para leer un valor tipo `int`, haga esto:

```
<variable tipo int> = Integer.parseInt(JOptionPane.showInputDialog)(<mensaje de
petición>);
```

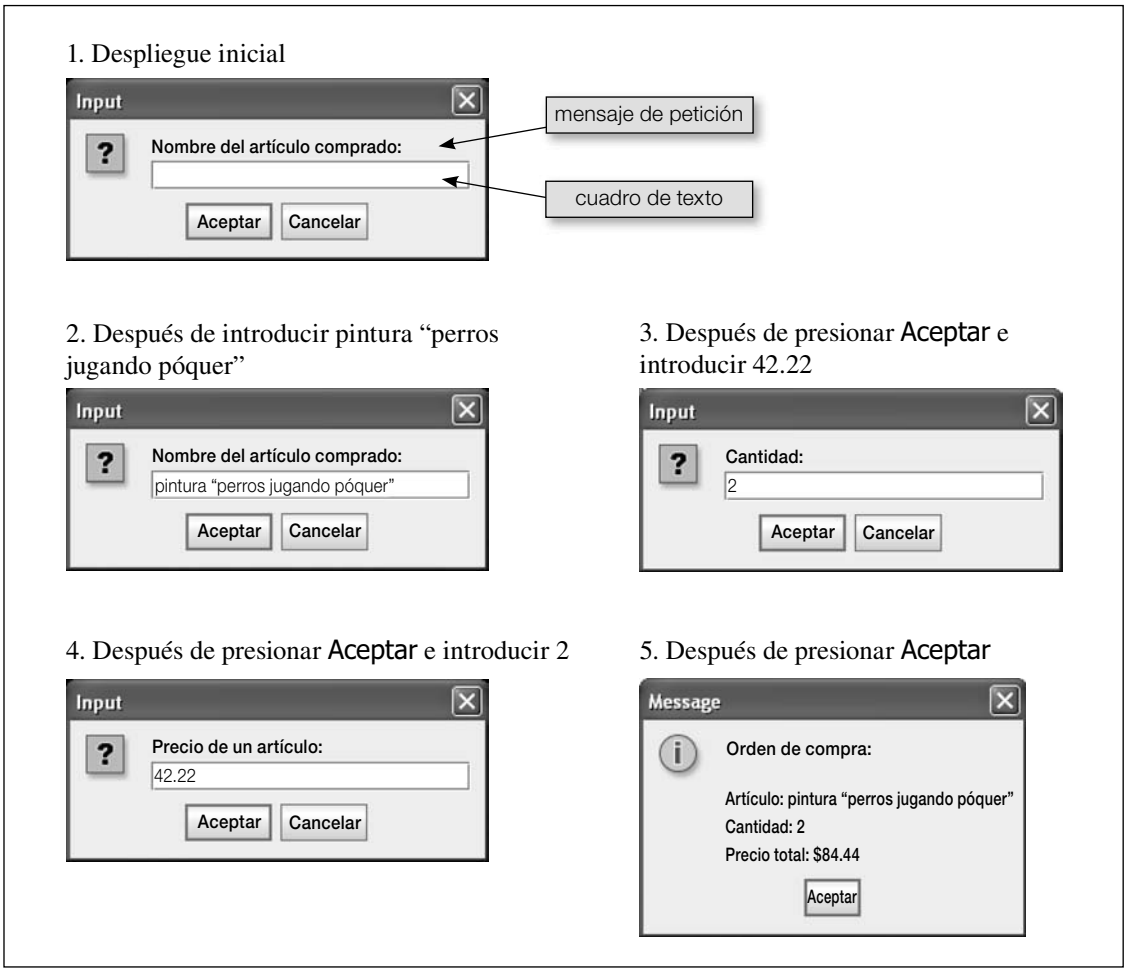


Figura 3.18 Sesión ejemplo del programa ImprimirOrdenCompraGUI.

Y para leer un valor tipo `double` haga esto:

```
<variable tipo double> = Double.parseDouble(JOptionPane.showInputDialog
                         (<variable tipo>));
```

Ahora observe la figura 3.19, que muestra cómo utilizar estas nuevas sentencias para producir la salida de la figura 3.18.

`Integer.parseInt` convierte la cadena de caracteres leídas en un valor tipo `int`, y `Double.parseDouble` convierte la cadena de caracteres leída en un valor tipo `double`. `Integer` y `Double` son clases envoltorio. `parseInt` y `parseDouble` son métodos de clases envoltorio. Las clases envoltorio y sus métodos serán descritos en el capítulo 5.

E/S para el resto del libro

En las secciones de los apartados GUI y en los capítulos GUI al final del libro, por supuesto que se utilizarán ventanas GUI para E/S. Pero, para el resto del libro, se utilizarán ventanas de consola. Se utilizarán ventanas de consola porque eso permite simplificar los programas. Los programas más simples son importantes porque de esa manera se puede simplificar y enfocar el material recientemente introducido. Pero si el usuario ha decidido que le gustan los objetos GUI y cree que no es suficiente, puede tomarse la libertad de convertir todos los programas de consola en programas de ventana GUI. Para hacerlo, deberá reemplazar todo el código de salida con llamadas a `showMessageDialog` y reemplazar todo el código de entrada con llamadas al método `showInputDialog`.

```
/*****************************************************************
* ImprimirOrdenCompraGUI.java
* Dean & Dean
*
* Este programa calcula e imprime un reporte de orden de compra.
*****************************************************************/

import javax.swing.JOptionPane;

public class ImprimirOrdenCompraGUI
{
  public static void main(String[] args)
  {
    String nombreArticulo; // nombre del artículo comprado
    double precio;         // precio de compra del artículo
    int cant;              // número de elementos comprados

    nombreArticulo = JOptionPane.showInputDialog("Nombre del artículo comprado:");
    precio = Double.parseDouble(
      JOptionPane.showInputDialog("Cantidad:"));
    cant = Integer.parseInt(
      JOptionPane.showInputDialog("Cantidad:"));
    JOptionPane.showMessageDialog(null,
      "ORDEN DE COMPRA:\n\n" +
      "Artículo: " + nombreArticulo + "\nCantidad: " + cant +
      "\nPrecio total: $" + precio * cant);
  } // fin del metodo main
} // fin de la clase ImprimirOrdenCompraGUI
```

Figura 3.19 Programa ImprimirOrdenCompraGUI.

Resumen

- Los comentarios se utilizan para mejorar la legibilidad y el entendimiento de un programa.
- El método `System.out.print` imprime un mensaje y después deja el cursor en la siguiente línea. El método `System.out.print` imprime un mensaje y después deja el cursor en la misma línea del mensaje.
- Las variables pueden almacenar sólo un tipo de datos y ese tipo se define mediante una sentencia de declaración de variable.
- Una sentencia de asignación utiliza el operador = y asigna un valor a una variable.
- Una sentencia de inicialización es una combinación de una sentencia de declaración y una de asignación. Declara un tipo de variable y también le otorga a ésta un valor inicial.
- Las variables que almacenan números enteros deben ser declaradas con el tipo de datos `int` o el tipo `long`.
- Las variables que almacenan números de punto flotante deben ser declaradas con el tipo de datos `double`. Si se está seguro de que la variable está limitada a unos cuantos números de punto flotante, es adecuado utilizar el tipo de datos `float`.
- Las constantes nombradas utilizan el modificador `final`.
- Hay dos tipos de división de enteros. Uno obtiene el cociente (mediante el operador /). El otro obtiene el residuo (mediante el operador %).
- Las expresiones se evalúan con la utilización de un conjunto de reglas de precedencia de operadores bien definidas.

- El operador de conversión permite devolver un tipo de dato diferente al valor dado.
- Se usa una secuencia de escape (mediante una diagonal invertida) para imprimir caracteres de difícil impresión como el de tabulación.
- Una variable de referencia almacena una dirección de memoria que apunta a un objeto. Un objeto es una colección de datos relacionados inmersos en una capa protegida.
- La clase `String` proporciona métodos que se pueden utilizar para procesamiento de cadenas de texto.
- La clase `Scanner` proporciona métodos que se pueden utilizar para entrada de datos.

Preguntas de revisión

§3.2 Programa "Tengo un sueño"

1. ¿Qué hace el programa `Sueño.java`?

2. ¿Cuáles son las extensiones del nombre del archivo fuente y de código byte, respectivamente?

§3.3 Comentarios y legibilidad

3. ¿Por qué el código fuente tiene comentarios?

§3.4 El encabezado de clase

4. Para un archivo con una clase `pública`, el nombre del archivo debe concordar con el nombre de la clase, excepto cuando el archivo tenga extensión `.java` agregada a su nombre. (C/F)

5. Las convenciones estándar de código señalan que los nombres de clase inician con una letra minúscula en el primer carácter. (C/F)

6. En Java, la utilización de mayúsculas o minúsculas no tiene importancia. (C/F)

§3.5 El encabezado del método `main`

7. Un método de inicio `main` debe estar en una clase de tipo `público`. (C/F)

8. El método `main` mismo debe ser de tipo `público`. (C/F)

9. Haciendo uso de su memoria (sin mirar la respuesta en el libro) escriba el encabezado del método `main`.

§3.6 Paréntesis de llave

10. Identifique dos tipos de agrupamiento de código que deban ser encerrados entre paréntesis de llave.

§3.7 `System.out.println`

11. Haciendo uso de su memoria (sin utilizar el libro), escriba la sentencia que le indique a la computadora desplegar la siguiente cadena de texto:

```
Este es un ejemplo
```

§3.9 Identificadores

12. Enliste todos los tipos de caracteres que se pueden utilizar para formar un identificador.

13. Enliste todos los caracteres que se pueden utilizar como el primer carácter de un identificador.

§3.10 Variables

14. Los nombres de las variables se deben abreviar mediante la omisión de vocales para ahorrar espacio. (C/F)

15. ¿Por qué es una buena práctica utilizar una línea de separación para declarar cada variable?

§3.11 Sentencias de asignación

16. Debe haber un punto y coma al final de cada sentencia. (C/F)

§3.12 Sentencias de asignación

17. En la inicialización "mata dos pájaros de un tiro", ¿cuáles son estos "dos pájaros"?

§3.13 Tipos de datos numéricos: `int`, `long`, `float`, `double`

18. El tipo de dato más apropiado para utilizar en cifras financieras es _____.

19. En cada una de las oraciones siguientes, indique cierto o falso.

a) `1234.5` es un número de tipo punto flotante. (C/F)

b) `1234` es un número de tipo punto flotante. (C/F)

c) `1234.` es un número de tipo punto flotante. (C/F)

20. Si intenta asignar un valor de tipo `int` a uno de tipo `double`, la computadora realiza automáticamente la conversión sin problema, pero si intenta asignar a una variable de tipo `int` un valor de tipo `double`, el compilador genera un error. ¿Por qué?

§3.14 Constantes

21. En cada una de las oraciones siguientes, indique cierto o falso.
- ***a***) `0.1234` es de tipo `float`. (C/F)
- ***b***) `0.1234f` es de tipo `float`. (C/F)
- ***c***) `0.1234` es de tipo `double`. (C/F)
- ***d***) `1234.0` es de tipo `double`. (C/F)

22. ¿Qué tipo de modificador especifica que el valor de una variable es de tipo fija/constante?

§3.15 Operadores aritméticos

23. ¿Cuál es el operador para obtener el residuo en una división?

24. Escriba las siguientes operaciones matemáticas como expresiones Java válidas:
- ***a***) $\frac{3x-1}{x^2}$
- ***b***) $\frac{1}{2}+\frac{1}{xy}$

§3.16 Evaluación de expresiones y precedencia de operadores

25. Asumiendo lo siguiente:

```
int m = 3, n = 2;
double x = 7.5;
```

Evalúe las siguientes expresiones:
- ***a***) `(7 - n) % 2 * 7.5 + 9`
- ***b***) `(4 + n / m) / 6.0 * x`

§3.17 Más operadores: incremento, decremento y asignación compuesta

26. Escriba la sentencia de Java más corta para incrementar la variable `cuenta` en uno.

27. Escriba la sentencia de Java más corta para incrementar la variable `cuenta` en tres.

28. Escriba la sentencia de Java más corta que multiplique a la variable `numero` por (`numero -1`) y asigne el resultado del producto a la misma variable `numero`.

§3.18 Más operadores: incremento, decremento y asignación compuesta

29. ¿Qué significa que una variable contenga basura?

30. En una lista de rastreo de flujo, ¿qué función tienen los números de línea?

§3.19 Conversión de tipos

31. Escriba una sentencia de Java que asigne a la variable `miEntero`, de tipo `int`, el valor de la variable `miDouble`, que es de tipo `double`.

§3.20 Tipo `char` y secuencias de escape

32. ¿Qué error aparece en la siguiente inicialización?

```
char letra = "y";
```

33. Si se intenta poner comillas dobles en cualquier parte dentro de una cadena literal de caracteres, la computadora lo interpreta como el fin de dicha cadena. ¿Cómo se soluciona este problema y cómo se fuerza a la computadora a reconocer las comillas dobles como algo a imprimir dentro de la cadena?

34. Cuando se describe la ubicación de un archivo o un directorio, las computadoras utilizan rutas de directorios. En los ambientes Windows se utiliza el carácter diagonal invertida (\) para separar los directorios y archivos dentro de una ruta de directorios. Si se requiere imprimir la ruta dentro de un directorio en un programa de Java, ¿de qué manera debe escribirse el carácter diagonal invertido?

§3.21 Variables primitivas *versus* variables de referencia

35. El nombre del tipo para una variable primitiva no se escribe con mayúsculas, pero para una de tipo referencia, por lo regular se escribe con mayúsculas. (C/F)

36. Enumere los tipos primitivos que se describen en este capítulo, en las siguientes categorías:
- ***a***) Números enteros.
- ***b***) Números de punto flotante.
- ***c***) Caracteres individuales de texto y símbolos especiales.

§3.22 Cadenas de caracteres

37. ¿Cuáles son los dos operadores que ejecutan concatenación de caracteres, y cuál es la diferencia entre ambos?
38. ¿Cuál es el método que se puede utilizar para traer un carácter a una posición específica dentro de una cadena de caracteres?
39. ¿Cuáles son los dos métodos que se pueden utilizar para comparar la igualdad de dos cadenas?

§3.23 Entrada: la clase `Scanner`

40. ¿Qué es un espacio en blanco?
41. Escriba una sentencia que se deba poner antes que cualquier otro código y que le indique al compilador que se estará utilizando la clase `Scanner`.
42. Escriba una sentencia que cree una conexión entre un programa y el teclado de la computadora.
43. Escriba una sentencia que lea una línea a través del teclado y lo escriba en una variable llamada `linea`.
44. Escriba una sentencia que lea un número de tipo `double` desde el teclado y que lo asigne a una variable llamada `numero`.

Ejercicios

1. [Después de §3.3] Demuestre las dos formas de proporcionar comentarios en un programa Java mediante la escritura de las siguientes líneas como si fueran comentarios. Hágalo en los dos formatos.

```
Éste es un comentario largo con muchas palabras innecesarias que obli-
gan a utilizar infinidad de líneas para incluir todo.
```

2. [Después de §3.5] ¿Por qué el siguiente encabezado del `main` genera un error: `public static void Main (String[] args)`?
3. [Después de §3.6] ¿Para qué se utilizan los paréntesis de llave?
4. [Después de §3.8] ¿Qué programa se encarga de
   a) Leer el código fuente de Java y generar código byte?
   b) Ejecutar el código byte?
5. [Después de §3.9] Para mejorar la legibilidad de un identificador que está compuesto de varias palabras, conviene utilizar puntos entre las palabras. (C/F)
6. [Después de §3.10] Para cada uno de los siguientes nombres de variables, indique (con una s/n) si es válido y si se utiliza el estilo adecuado. Atención: puede omitirse la pregunta sobre el estilo para todos los nombres inapropiados de variables, ya que el estilo es irrelevante en cada caso.

	Valido (s/n)?	Estilo adecuado(s/n)?
a) `_estaLista`		
b) `3erNOmbre`		
c) `num de ruedas`		
d) `dinero#en#mano`		
e) `tasaImpuesto`		
f) `NumeroAsiento`		

7. [Después de §3.10] No se necesita un punto y coma después de la declaración de una variable. (C/F)
8. [Después de §3.13] Si simplemente se escribe un número de tipo punto flotante sin especificar su tipo, ¿de qué tipo de número asume la computadora que es éste?
9. [Después de §3.14] ¿Cómo se especificaría la raíz cuadrada del número 2 como una constante nombrada? Utilice 1.41421356237309 para el valor de la constante nombrada.
10. [Después de §3.15] Escriba las siguientes expresiones matemáticas como expresiones válidas de Java.
   a) $\left(\frac{3-k}{4}\right)^2$
   b) $\frac{9x-(4.5+y)}{2x}$
11. [Después de §3.16] Asuma lo siguiente:

```
int a = 9;
double b = 0.5;
int c = 0;
```

Evalúe cada una de las siguientes expresiones a mano. Realice el trabajo utilizando una línea separada para cada paso de evaluación. Verifique el trabajo escribiendo y ejecutando un programa que evalúe estas expresiones y despliegue los resultados.

a) `a + 3 / a`
b) `25 / ((a - 4) * b)`
c) `a / b * a`
d) `a % 2 - 2 % a`

12. [Después de §3.19] Conversión de datos:
Asuma las siguientes declaraciones:

```
int entero;
double realPreciso;
float realImpreciso;
long enteroGrande;
```

De las siguientes sentencias rescriba las que generarían un error de compilación al utilizar un tipo de conversión inapropiado que permita continuar con el error. No proporcione una conversión para cualquier sentencia que el compilador promovería de manera automática.

a) `entero = realPreciso;`
b) `enteroGrande = realImpreciso;`
c) `realPreciso = entero;`
d) `realImpreciso = enteroGrande;`
e) `entero = realImpreciso;`
f) `enteroGrande = realPreciso;`
g) `realImpreciso = entero;`
h) `realPreciso = enteroGrande;`
i) `entero = enteroGrande;`
j) `realImpreciso = realPreciso;`
k) `realPreciso = realImpreciso;`
l) `enteroGrande = entero;`

13. [Después de §3.20] Asuma que los tabuladores están a cuatro columnas de donde se está imprimiendo, ¿qué salida generará la siguiente sentencia?

```
System.out.println("\"Ruta:\" \n\tD:\\miJava\\Hola.java");
```

14. [Después de §3.21] Los tipos de referencia inician con una letra mayúscula. (C/F)

15. [Después de §3.22] Asuma que se tiene una variable de tipo String. Proporcione un fragmento de código que imprima el tercer carácter de la misma.

16. [Después de §3.22] ¿Qué imprime el siguiente fragmento de código?

```
String s = "hedge";
s += "hog";
System.out.println(s.equals("hedgehog"));
System.out.println((s.length()-6)+ "" + s.charAt(0)+ "\'s"));
```

Solución a las preguntas de revisión

1. Genera la salida:
`¡Tengo un sueño!`

2. La extensión del código fuente de Java es `.java`. La extensión del código byte es `.class`.

3. El código fuente incluye comentarios para ayudar a los programadores a recordar o determinar cómo funciona un programa. (Los comentarios son ignorados por la computadora, y no son accesibles al usuario común y corriente.) Los bloques de comentarios iniciales incluyen el nombre del archivo como un recordatorio continuo para el programador. Contiene los nombres de los autores del programa para ayuda y referencia. Puede incluir la fecha y la versión de la revisión para identificar el contexto. Incluye una pequeña descripción para facilitar el entendimiento. Los símbolos en los comentarios como los siguientes `//fin de clase` <*nombre de clase*> ayudan a mantener orientado al lector. Los comentarios especiales identifican variables y aclaran fórmulas de difícil entendimiento.

4. Cierto. Si un nombre de archivo tiene una clase `pública`, entonces el nombre del archivo debe ser igual al de la clase.

5. Falso. Los nombres de las clases deben comenzar con la primera letra en mayúscula.

6. Cierto. El lenguaje Java es sensible a mayúsculas y minúsculas. Al cambiar de mayúsculas a minúsculas o viceversa, se crea un identificador completamente diferente.

7. Cierto.

8. Cierto. De otra manera, el procedimiento de inicio no puede ser accedido.
9. `public static void main(String[] args)`
10. Se deben utilizar paréntesis de llave para 1) todos los contenidos de clase y 2) para el contenido de un método.
11. `System.out.println("He aquí un ejemplo");`
12. Caracteres en mayúscula, en minúscula, guión bajo y símbolo de dólar.
13. Caracteres en mayúscula, caracteres en minúscula, guión bajo y símbolo de dólar. No se permiten números.
14. Falso. En el código fuente, el ahorrar espacio no es tan importante como una buena comunicación. Las abreviaturas extrañas son difíciles de escribir y no tan fáciles de recordar como las palabras reales.
15. Si cada variable se encuentra en una línea separada, cada variable tiene su espacio a la derecha para un comentario elaborado.
16. Cierto.
17. La declaración de variables y asignación de un valor en una variable.
18. Tipo `double`, o tipo `long`, con valor en centésimas.
19. ***a***) Cierto; ***b***) Falso; ***c***) Cierto.
20. La asignación de un valor entero a una variable de punto flotante es como poner un objeto pequeño dentro de una grande. El tipo `int` abarca hasta aproximadamente 2 mil millones. Es fácil de acomodar 2 mil millones en una "caja" `double` porque ésta abarca hasta 1.8×10^{308}. Por otro lado, la asignación de una variable de punto flotante a una variable entera es como poner un objeto grande en una caja chica. Por omisión, eso es inválido.
21. ***a***) Falso; ***b***) Cierto; ***c***) Cierto; ***d***) Cierto.
22. El modificador `final` especifica que el valor de una variable es fijo/constante.
23. El operador de residuo es un signo de porcentaje %.
24. Escriba las siguientes expresiones matemáticas como expresiones válidas en Java.
a) (3 * x − 1) / (x * x)
b) 1.0 / 2 + 1.0 / (x * y)
o
.5 + 1.0 / (x * y)
25. Evaluación de expresiones:
a)
```
(7 - n) % 2 * 7.5 + 9 ⇒
5 % 2 * 7.5 + 9 ⇒
1 * 7.5 + 9 ⇒
7.5 + 9 ⇒
16.5
```
b)
```
(4 + n / m) / 6.0 * x ⇒
(4 + 2 / 3) / 6.0 * 7.5 ⇒
(4 + 0) / 6.0 * 7.5 =>
4 / 6.0 * 7.5 ⇒
0.666666666666666667 * 7.5 ⇒
5.0
```
26. `cuenta ++;`
27. `cuenta -= 3;`
28. `numero *= (numero - 1);`
29. Para variables declaradas sin una inicialización, el valor inicial es referido como *basura* porque su valor actual se desconoce. Utilice un signo de interrogación para indicar un valor basura.
30. Los números de línea indican qué sentencia en el código genera los resultados actuales de trazado.
31. `miEntero = (int) miDoble;`
32. La variable `letra` es de tipo `char`, pero las comillas en `"y"` especifican que el valor inicial es de tipo `String`, por lo que los tipos son incompatibles. Debe escribirse como:
```
char letra = 'y';
```
33. Para imprimir una comilla doble en una cadena de texto, inserte una diagonal invertida enfrente de ella, es decir, `\"`.
34. Para imprimir una diagonal invertida, utilice dos diagonales invertidas, es decir, `\\`.
35. Cierto.
36. Enumere los tipos primitivos que se describen en el presente capítulo, en las siguientes categorías:
a) Números enteros: `int`, `long`.
b) Números de punto flotante: `float`, `double`.
c) Caracteres individuales y símbolos especiales: `char`.

37. Los operadores + y += ejecutan la *concatenación*. El operador + no actualiza el operando a su izquierda. El operador += sí lo hace.
38. El método `charAt` se puede utilizar para traer un carácter a una posición especificada dentro de una cadena de caracteres.
39. Los métodos `equals` y `equalsIgnoreCase` se pueden utilizar para comparar cadenas de caracteres para efectos de igualdad.
40. Los caracteres en blanco son aquellos asociados con la línea espaciadora, el tabulador y la tecla Enter.
41.
```
import java.util.Scanner;
```
42.
```
Scanner stdIn = new Scanner(System.in);
```
43.
```
line = stdIn.nextLine();
```
44.
```
numero = stdIn.nextDouble();
```

Para fortuna de los hispanohablantes, Java permite el uso del carácter "ñ", así como de caracteres acentuados para nombrar métodos, variables y clases. Sin embargo, aquí se evitará el uso de nombres con acento, y sólo cuando se crea conveniente, se utilizará la letra "ñ", tal y como se hizo en el programa Sueño. (*Nota del traductor.*)

CAPÍTULO 4

Sentencias de control

Objetivos

- Aprender a utilizar la sentencia `if` para modificar la secuencia de ejecución de un programa.
- Familiarizarse con los operadores lógicos y de comparación de Java, y aprender a utilizarlos para describir condiciones complejas.
- Aprender a utilizar la sentencia `switch` para alterar la secuencia de ejecución de un programa.
- Reconocer operaciones repetitivas, entender los diferentes tipos de ciclos que existen en Java, y aprender a seleccionar el tipo de ciclo más adecuado para cada problema que requiera una evaluación repetitiva.
- Ser capaz de realizar una operación de trazado de flujo en un ciclo.
- Aprender cómo y cuándo anidar un ciclo dentro de otro ciclo.
- Aprender a utilizar variables `boolean` para escribir código más elegante.
- Aprender a validar datos de entrada.
- Opcionalmente, aprender a simplificar expresiones lógicas complicadas.

Relación de temas

4.1 Introducción

En el capítulo 3, las cosas se mantuvieron simples y se escribieron únicamente programas de tipo secuencial. En un programa secuencial puro, las sentencias se ejecutan en la secuencia/orden en que fueron es-

critas; esto es, después de ejecutar una sentencia, la computadora ejecuta la que sigue inmediatamente después. La programación secuencial pura funciona bien para problemas triviales, pero para algo sustancial se requiere de habilidad para ejecutarla de una manera no secuencial. Por ejemplo, si escribe un programa para recuperación de recetas, lo más seguro es que no desee imprimir cada una de éstas una tras otra. Si el usuario indica inclinación por las galletas de chocolate, deseará ejecutar la receta de preparación de galletas con chips de chocolate, y deseará ejecutar la impresión de los pasos para preparar un quiché de cangrejo, si es que indica inclinación por este platillo. Esa clase de funcionalidad requiere del uso de *sentencias de control*. Una sentencia de control controla el orden de ejecución dentro de un algoritmo. En el capítulo 2 se utilizaron las sentencias if (si) y while (mientras) en el seudocódigo para controlar el orden de ejecución dentro de un algoritmo. En este capítulo se utilizan las sentencias `if` y `while`, además de otras adicionales, para controlar el orden de ejecución dentro de un programa.

Para controlar el orden de ejecución, una sentencia de control utiliza una condición (una pregunta) para decidir qué camino seguir. El capítulo 4 inicia con una introducción a las condiciones en Java. Posteriormente, se describen las sentencias de control de Java: el `if`, el `switch`, los ciclos `while`, `do` y `for`. Además, se describen los operadores lógicos en Java `&&`, `||` y `!`, necesarios cuando se trabaja con condiciones más complicadas. El capítulo concluye con varios conceptos relacionados con ciclos: ciclos anidados, validación de entradas y variables `boolean`. ¡Demasiado trabajo!

4.2 Condiciones y valores *boolean*

En los diagramas de flujo del capítulo 2 se usaron formas de rombo para representar puntos de decisión lógica: puntos en que el flujo de control podría ir para un lado o para el otro. En esos rombos se insertaron diferentes preguntas abreviadas como "¿es el `sueldo del director general` mayor a $500 000?" y ¿`cuenta` es menor o igual a 100?" Entonces se marcó con una etiqueta cada una de las dos rutas alternativas provenientes del rombo, con respuestas de "sí" o "no" para cada pregunta. En el seudocódigo del capítulo 2 se utilizaron cláusulas "si" y "mientras" para describir las condiciones lógicas. Los ejemplos incluyeron sentencias como: "si `figura` es un círculo", "si `calificación` es mayor o igual a 60" y "mientras `marcador` no sea igual a −1". Se consideraron las condiciones de seudocódigo como "verdaderas" o "falsas".

Expresiones de condición informal como éstas son adecuadas para diagramas de flujo y seudocódigo, pero cuando se inicia escribiendo código real de Java, se debe hacer con precisión. La computadora interpreta cada condición "if" o cada ciclo como uno de dos caminos. ¿Cuáles son los dos posibles valores reconocidos por Java? Son los valores `verdadero` y `falso`. Estos valores, `verdadero` y `falso`, se denominan valores de tipo *boolean*, en honor a George Boolean, un famoso lógico del siglo XIX. En el resto del libro se verán sentencias `if` y los ciclos donde las *condiciones* aparecen como pequeños fragmentos de código dentro de un paréntesis, como en el siguiente:

```
if (<condición>)
{
    . . .
}
while (<condición>)
{
    . . .
}
```

Lo que sea que se coloque dentro de <*condición*> siempre será evaluado como `verdadero` o `falso`.

De manera típica, cada condición involucra algún tipo de comparación. Con el seudocódigo se pueden emplear palabras para describir comparaciones; pero con código real de Java, se deben utilizar *operadores de comparación* especiales. Los operadores de comparación (también llamados de igualdad y operadores relacionales) son como los operadores matemáticos en que se ligan a operandos adyacentes, pero en lugar de combinar los operandos, de alguna manera los comparan. Cuando un operador matemático combina dos números, la combinación es un número como los operandos que se combinan. Pero, cuando un operador de comparación, compara dos números, el resultado es de un tipo diferente. No es un

número como los operandos que se están comparando, sino un valor real boolean, `verdadero` o `falso`.

Éstos son los operadores de comparación de Java:

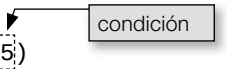
```
==, !=, <, >, <=, >=
```

El operador == determina si dos valores son iguales. Observe que este símbolo se escribe con dos signos de igual. Éste es diferente del símbolo de igual que se utiliza para representar igualdad. La pregunta sería ¿por qué Java usa dos signos de igual para representar igualdad en una comparación? Porque Java ya usa el signo de igual para representar asignación, y el contexto no es suficiente para distinguir entre igualdad y asignación. ¡No se debe utilizar un solo signo de = para comparación! Al compilador de Java no le gustará.

El operador != verifica si dos valores son desiguales. Como podría esperarse, el operador < prueba si el valor a la izquierda es menor que el valor a la derecha. El operador > verifica si el valor a la izquierda es mayor que el de la derecha. El operador <= prueba si el valor a la izquierda es menor o igual que el valor a la derecha. El operador >= verifica si el valor a la izquierda es mayor o igual que el valor a la derecha. El resultado de cualquiera de estas comparaciones es siempre un valor `true` (verdadero) o `false` (falso).

4.3 Sentencias `if`

Ahora se analizará un ejemplo simple de la condición en una sentencia `if`. He aquí una sentencia `if` que verifica el valor de medición de la temperatura e imprime una advertencia si la temperatura se encuentra arriba de los 38 grados Celsius.




```
if (temperatura > 215)
{
  System.out.println("¡Alerta! El anticongelante está demasiado caliente.");
  System.out.println("Deje de manejar y deje que el motor enfríe.");
}
```

La condición utiliza el operador > para generar un valor `verdadero` si la temperatura está por arriba de los 38 grados o un valor `falso` si no lo está. Las sentencias subordinadas se ejecutan sólo si la condición genera un valor `true` (verdadero).

Sintaxis

En el ejemplo anterior observe el paréntesis alrededor de la condición. Los paréntesis son necesarios siempre que se tenga una condición, independientemente de si es o no una sentencia `if`, un ciclo `while`, o alguna otra estructura de control. Observe los paréntesis de llave alrededor de las sentencias subordinadas de impresión. Es necesario usar paréntesis de llave alrededor de sentencias que estén lógicamente dentro de algo más. Por ejemplo, las llaves se requieren abajo del encabezado del método `main` y al final del mismo porque las sentencias dentro de las llaves están lógicamente dentro del método. De la misma manera, se deben utilizar paréntesis de llave para encerrar las sentencias que estén lógicamente dentro de una sentencia `if`. Para destacar el punto de que las sentencias dentro de llaves están lógicamente dentro de algo más, se debe hacer sangría dentro de las llaves. Puesto que esto es importante, se repite una vez más: siempre hay que hacer sangría en las sentencias que estén dentro de las llaves.

Cuando una sentencia `if` incluye dos o más sentencias subordinadas, se deben encerrar entre paréntesis de llave. Dicho de otra manera, se debe utilizar un *bloque*. Un bloque, también llamado *sentencia compuesta*, es un conjunto de cero o más sentencias rodeadas por llaves. Se puede utilizar un bloque en cualquier parte en que una sentencia estándar pueda ser utilizada. Si no se emplean paréntesis de llave para dos sentencias subordinadas al `if`, la computadora considera sólo la primera como subordinada al `if`. Cuando se supone que es sólo una sentencia subordinada, no es necesario encerrarla entre llaves, pero se recomienda hacerlo de cualquier manera. Así, no se tendrán problemas cuando haya que volver al código e insertar sentencias subordinadas adicionales en ese punto del programa.

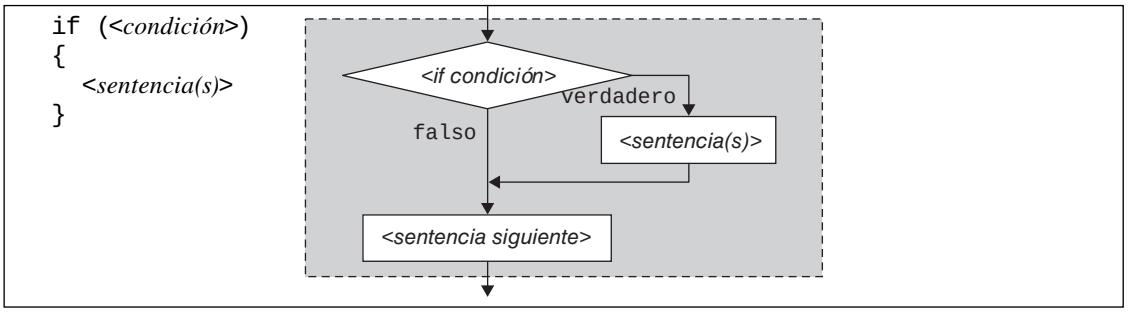


Figura 4.1 Sintaxis y semántica para la forma del "if" simple de la sentencia `if`.

Tres formas de la sentencia `if`

Existen tres formas básicas de una sentencia `if`:

- "if", se utiliza cuando se quiere hacer una cosa o no hacer nada.
- "if, else", se utiliza cuando se quiere hacer una u otra cosa.
- "if, else if", se utiliza cuando hay tres o más posibilidades.

En el capítulo 2 se presentaron versiones de seudocódigo de esas formas. Las figuras 4.1, 4.2 y 4.3 muestran las formas Java.

Tómese unos minutos y examine las figuras 4.1, 4.2 y 4.3, que muestran la sintaxis y la semántica de las tres formas de la sentencia `if`. La *semántica* de una sentencia es la descripción de cómo trabaja una

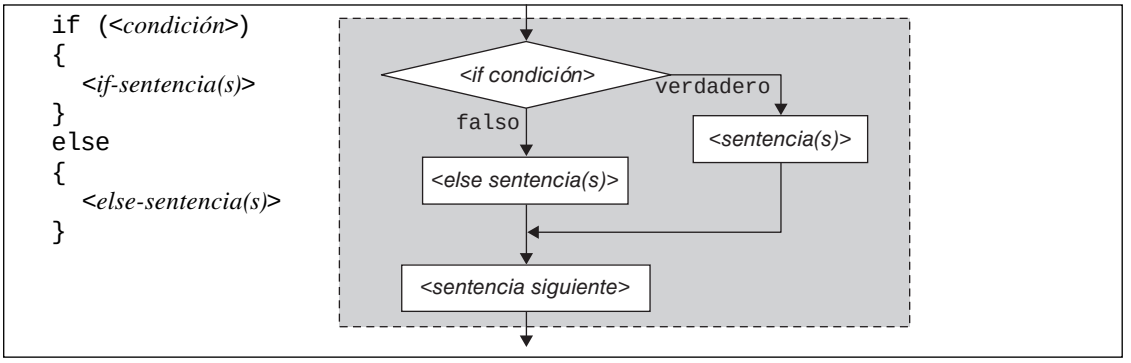


Figura 4.2 Sintaxis y semántica para la forma "if, else" de la sentencia `if`.

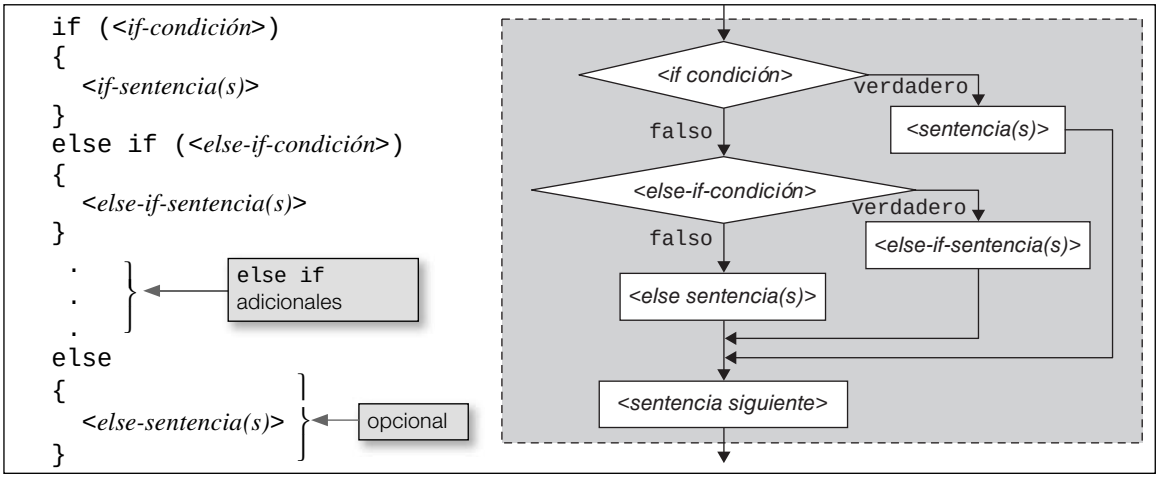


Figura 4.3 Sintaxis y semántica para la forma "if, else if" de la sentencia `if`.

sentencia. Por ejemplo, el diagrama de flujo de la figura 4.1 ilustra la semántica de la forma "if" de la sentencia `if` mostrando el flujo de control para diferentes valores en la sentencia de la condición `if`.

La mayor parte de lo que se observa en las sentencias de las figuras habrá de parecer familiar al lector, ya que concuerda con lo que aprendió en el capítulo 2; aunque la sentencia de la forma "if, else if" requiere atención extra. Se pueden incluir tantos bloques "else if" como se desee: mientras más bloques "else if" haya, mayores opciones se tendrán. Observe que el bloque "else" es opcional. Si todas las condiciones son falsas y no hay un bloque "else" final, ninguno de los anteriores se ejecuta. He aquí un fragmento de código que utiliza la forma "if, else if" de la sentencia `if` para resolver problemas en el iPod.[1]

```
if (problemaiPod.equals("sin respuesta"))
{
  System.out.println("Desbloquee el cordón de iPod.");
}
else if (problemaiPod.equals("no toca música"))
{
  System.out.println("Actualizar software de su iPod.");
}
else
{
  System.out.println("Visite http://www.apple.com/support.");
}
```

Problema de práctica

Ahora pondrá en práctica lo que ha aprendido utilizando la sentencia `if` dentro de un programa completo. Suponga que se le solicita escribir un programa de prueba de sentencias que verifique que el usuario haya introducido una oración que termine con un punto. El programa debe imprimir un mensaje de error si el último carácter de la línea no es un punto. Al escribir el programa se debe utilizar una sesión muestra a manera de guía. Observe que la frase de Mahatma Gandhi en letras cursivas se captura como valor de entrada del usuario.

Utilice la salida deseada para especificar el problema.

Sesión muestra:

```
Introduzca una oración:
El bien permanente nunca puede ser resultado de la violencia.
```

Otra sesión muestra:

```
Introduzca una oración:
El bien permanente nunca puede ser resultado de
Captura inválida: ¡la oración está incompleta!
```

Como primer paso para implementar una solución se utiliza seudocódigo para generar una salida informal de la lógica básica:

```
imprimir "Introduzca una oración":
leer oración
si el último carácter de la oración es diferente de '.'
  imprimir "Captura inválida: ¡la oración no está completa!"
```

Observe la forma simple "if" de la sentencia `if`. Ésta es apropiada porque se necesita hacer algo (imprimir un mensaje de captura inválida) o nada. ¿Por qué nada? Porque la descripción del problema no indica que se deba imprimir algo cuando el usuario introduzca una entrada válida. En otras palabras, el programa debe saltarse lo que está en la sentencia `if` si la oración se introduce correctamente. Ahora sugerimos intentar escribir el código Java que implemente el algoritmo. Será necesario utilizar un par de

[1] El iPod es un dispositivo móvil reproductor de medios, diseñado y comercializado por Apple Computer.

```
/*****************************************************************
* PruebaOracion.java
* Dean & Dean
*
* Este programa verifica que exista un punto al final de la oración.
*****************************************************************/

import java.util.Scanner;

public class PruebaOracion
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String oracion;
    String carUltimaPosicion;

    System.out.println("Introduzca una oración:");
    sentence = stdIn.nextLine();
    carUltimaPosicion = oracion.length() - 1;
    if (oracion.charAt(carUltimaPosicion) != '.')
    {
      System.out.println(
        "Captura inválida: ¡su oración requiere un punto!");
    }
  } // fin del main
} // fin de la clase PruebaOracion
```

Esta condición comprueba una terminación correcta.

Figura 4.4 Programa PruebaOracion.

métodos del objeto `String` descritos casi al final del capítulo 3. Después, mire la solución de `PruebaOracion` en la figura 4.4.

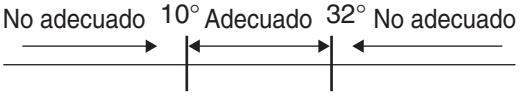
¿Cómo determina el programa PruebaOracion si el último carácter es un punto? Suponga que el usuario introduce "Hola." En ese caso, ¿qué valor se asignaría a la variable `carUltimaPosicion`? El método `length` del objeto `String` devuelve el número de caracteres en una cadena de texto. El número de caracteres en "Hola." es cinco. Debido a que la primera posición es cero, la variable `carUltimaPosicion` generaría el valor de (5-1) o 4. ¿Por qué es necesario `carUltimaPosicion`? Es necesario para verificar si el último carácter introducido por el usuario es un punto. Para hacerlo, se utiliza `carUltimaPosicion` como el argumento en el método `charAt` para que devuelva el carácter especificado como índice en la cadena de caracteres. La posición índice del punto en la cadena "Hola." es 4, y si la condición `if` verifica el último carácter introducido por el usuario encontrará que éste es un punto.

4.4 Operador lógico &&

Hasta este punto, en todos los ejemplos de sentencia `if` se han utilizado condiciones simples. Una condición simple se evalúa directamente como `true` (verdadera) o `false` (falsa). En las siguientes tres secciones se explicarán los operadores lógicos, como el operador "y" (`&&`) y el operador "o" (`||`), que hacen posible construir condiciones *compuestas*. Una condición compuesta es una conjunción (ya sea conjuntiva o disyuntiva) de dos o más condiciones. Cuando se tiene una condición compuesta, cada parte de la misma debe evaluarse como `verdadera` o `falsa`. Y después las partes se combinan para producir un `cierto` o `falso` para la condición combinada. Las reglas de combinación son lo que se podría esperar: cuando se conjugan dos condiciones, el resultado es `verdadero` sólo si la primera condición es `verdadera` y la segunda también. Cuando se realiza la operación de disyunción de dos condiciones, la combinación da como resultado `verdadero` si la primera condición es verdadera o si lo es la segunda. Se verán otros ejemplos más adelante.

Ejemplo con operador &&

Aquí se inicia el estudio de operadores lógicos con un ejemplo en que se utiliza el operador &&. El operador && significa "y" (*and*, en inglés). Suponga que quiere imprimir "Adecuado" si la temperatura se encuentra entre 10 y 32 grados, e imprimir "No adecuado" de lo contrario:



He aquí el seudocódigo para la descripción del problema:

```
if temp ≥ 10 y ≤ 32
  imprimir "Adecuado"
else
  imprimir "No adecuado"
```

Observe que las condiciones en el seudocódigo incluyen ≥ y ≤ en lugar de > y <. La especificación original del problema indica imprimir "Adecuado" si la temperatura se encuentra entre los 10 y los 32 grados. Cuando la gente dice "entre" por lo regular, aunque no siempre, quiere decir incluir los puntos finales. Así pues, se asume que los 10 y los 32 son los puntos que deben incluirse en los rangos de temperatura, por lo que se utilizan los símbolos ≥ y ≤; pero, en general, si al escribir un programa no se está seguro acerca de los puntos finales para un rango en particular, éstos no deben asumirse. En lugar de ello, se debe consultar con el cliente acerca de lo que desea. Los puntos finales son importantes.

Piense en los límites a que debe llegar.

Observe la figura 4.5, que muestra la implementación de Java para el problema de la temperatura entre los 10 y los 32 grados. En Java, si ambos criterios se deben satisfacer (por ejemplo, `temp >= 10` y `temp <= 32`), entonces se deben separar los dos criterios con el operador &&. Tal como se indica en la primera llamada en la figura 4.5. Si ambos criterios utilizan la misma variable (por ejemplo, `temp`), entonces se debe incluir esa variable en ambos lados del operador &&. Observe el uso de los símbolos >= y <=. En seudocódigo no hay problema si se utiliza ≥ y ≤, o "mayor o igual que" y "menor o igual que"; pero en Java sólo deben utilizarse >= y <=.

Precedencia de operadores

En la figura 4.5 observe que el paréntesis alrededor de cada una de las dos comparaciones de la temperatura, fuerza la evaluación de las comparaciones antes de la evaluación del &&. ¿Qué sucede si se omite el paréntesis? Para responder esta pregunta, consulte la tabla de precedencia de operadores. El apéndice 2 proporciona una tabla completa de precedencia de operadores, pero en la mayoría de los casos se cubren en la tabla abreviada de precedencia que se presenta en la figura 4.6. Todos los operadores dentro de un grupo numerado en particular tienen la misma precedencia, pero los operadores en la parte alta de la figura (en los grupos 1, 2, …) tienen mayor precedencia que los operadores en la parte de abajo de la figura (los grupos … 7, 8).

La figura 4.6 muestra que los operadores de comparación >= y <= tienen una precedencia más alta que el operador lógico &&. Así, las operaciones >= y <= se ejecutan antes que la operación && (aun si el

```
if ((temp >= 10) && (temp <= 32))
{
  System.out.println("Adecuado");
}
else
{
  System.out.println("No adecuado");
}
```

`temp` debe repetirse

Utilizar <=, no ≤.

Figura 4.5 Implementación en Java para el problema de la temperatura entre 10 y 32°C.

1. agrupación con paréntesis:
   (<*expression*>)
2. operadores unarios:
   +x
   -x
   (<*tipo*>) x
   x++
   x--
   !x
3. operadores de multiplicación y división:
   x * y
   x / y
   x % y
4. operadores de suma y resta:
   x + y
   x – y
5. operadores relacionales menos que y mayor que:
   x < y
   x > y
   x <= y
   x >= y
6. operadores de igualdad:
   x == y
   x != y
7. operador lógico "and":
   x && y
8. operador lógico "or":
   x || y

Figura 4.6 Tabla de precedencia abreviada (ver apéndice 2 para la tabla completa). Los grupos de operadores en la parte alta de la tabla tienen precedencia más alta que los grupos en la parte baja. Todos los operadores dentro de un grupo en particular tienen igual precedencia. Si una expresión tiene dos o más operadores con la misma precedencia, entonces se ejecutan primero las expresiones de la izquierda y luego las de la derecha.

paréntesis interno en la condición de la figura 4.5 se omite). En otras palabras, se podría haber escrito la condición de la figura 4.5 de una manera más simple, como a continuación:

```
if (temp >= 10 && temp <= 32)
```

Se puede incluir este paréntesis extra o no, según el deseo del programador. En la figura 4.5 se incluye para destacar el orden de evaluación en esta presentación inicial, pero en el futuro se omitirá a menudo para simplificar un poco las cosas.

Otro ejemplo

Este ejemplo es de promociones comerciales en eventos deportivos. Suponga que el restaurante Yummi Burgers está dispuesto a regalar papas fritas a todos los aficionados de un juego de basquetbol cuando el equipo gane en casa y obtenga un marcador de al menos 100 puntos. El problema es escribir un programa que imprima el siguiente mensaje cuando se satisfaga dicha condición:

"Aficionado, cambia tu boleto usado por una orden de papas fritas en Yummy Burgers."

La figura 4.7 muestra el diseño. Observe en la figura donde dice <*insertar código aquí*>. Antes de continuar con la lectura de la respuesta, observe si puede proporcionar el código insertado por cuenta propia.

```
/*****************************************************************
 * PapasGratis.java
 * Dean & Dean
 *
 * Este programa lee los puntos obtenidos por el equipo anfitrión y
 * por el oponente y determina si los aficionados obtienen papas
 * fritas gratis.
*****************************************************************/

import java.util.Scanner;

public class FreeFries
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int ptsCasa;      // puntos obtenidos por el equipo anfitrión
    int ptsOponente;  // puntos obtenidos por el oponente

    System.out.print("Puntos obtenidos por el equipo anfitrión: ");
    ptsCasa = stdIn.nextInt();
    System.out.print("Puntos obtenidos por el equipo oponente: ");
    ptsOponente = stdIn.nextInt();

    <insertar código aquí>

  } // fin del main
} // fin de la clase PapasGratis
```

Sesión muestra:

```
Puntos obtenidos por el equipo anfitrión: 103
Puntos obtenidos por el equipo oponente: 87
Aficionado, cambia tu boleto usado por una orden de papas fritas en
Yummy Burgers.
```

Figura 4.7 Programa PapasGratis con la condición “and”.

He aquí lo que se debe insertar:

los puntos del equipo anfitrión deben repetirse

```
if (ptsCasa > ptsOponente && ptsCasa >= 100)
{
  System.out.println("Aficionado, cambia tu boleto usado por una orden" +
    " de papas fritas en Yummy Burgers.");
}
```

4.5 Operador lógico | |

Ahora se echará un vistazo al complemento del operador “and”: el operador “or” (o). Asuma que tiene una variable llamada `respuesta` que contiene 1) una “s” mayúscula o minúscula si el usuario desea salir o 2) algún otro carácter si el usuario desea continuar. Escriba un fragmento de código que imprima “Adiós” si el usuario introduce una “s” ya sea mayúscula o minúscula. Al utilizar seudocódigo, podría escribir algo como lo siguiente para la parte crítica del algoritmo:

si `respuesta` es igual a “s” o “S”
  imprimir “Adiós”

Observe la "o" en la condición de la sentencia `if`. Esto funciona bien para el seudocódigo, donde las reglas de sintaxis son poco exigentes, pero en Java se tendría que utilizar el operador `||` en la computadora, buscar ese símbolo en el teclado y presionarlo dos veces. He aquí una implementación tentativa del fragmento de código en Java:

```
Scanner stdIn = new Scanner(System.in);
String respuesta;

System.out.print("Introduzca s o S: ");
respuesta = stdIn.nextLine();
if (respuesta == "s" || respuesta == "S")
{
  System.out.println("Adiós");
}
```

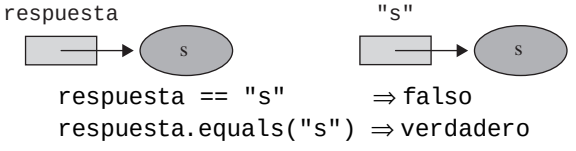
Cuando se inserta en un método `main`, esto compila, pero no funciona.

Observe que la variable `respuesta` aparece dos veces en la sentencia de condición `if`. Esto es necesario porque si ambos lados de una condición `||` involucran la misma variable, ésta debe repetirse.

La llamada indica que algo está mal, ¿qué es esto? Si se inserta este fragmento de código en un programa, éste compila y ejecuta pero cuando el usuario responda a la petición introduciendo ya sea una "s" o una "S" nada sucederá. El programa no imprime "Adiós". ¿Por qué no? ¿Se debió haber utilizado un paréntesis interior en la condición "if"? La figura 4.6 muestra que el operador `==` tiene una mayor precedencia que el operador `||`, entonces lo que se hizo es correcto. El problema radica en algo más.

⚠ No utilizar == para comparar cadenas de caracteres

El problema es con las expresiones `respuesta == "s"` y `respuesta == "S"`. Enfoquemos la expresión con la `respuesta == "s"`. La variable de cadena de caracteres `respuesta` y la cadena literal de caracteres "s" apuntan ambas a la dirección de memoria que apunta a objetos de tipo cadena de caracteres; no almacenan en sí objetos de tipo String. Así, cuando se utiliza el doble signo de igual (`==`) se están comparando las direcciones de memoria almacenadas en la variable tipo String `respuesta` y la cadena literal "s". Si la variable de `respuesta` tipo cadena de caracteres y la cadena literal de caracteres "s" contienen direcciones de memoria diferentes (apuntan a objetos tipo String diferentes), entonces la comparación da como resultado el valor `falso`, aunque ambos objetos de cadena de caracteres contengan la misma secuencia de caracteres. La siguiente imagen muestra esto. Las flechas representan direcciones de memoria. Puesto que apuntan a diferentes objetos, `respuesta == "s"` se evalúa como `falso`.



Entonces, ¿qué se puede hacer para resolver este problema? En el capítulo 3 se aprendió a utilizar el método `respuesta` para probar la igualdad de dos cadenas de caracteres. El método `respuesta` compara los objetos de tipo String referenciados por una dirección de memoria. En la figura de arriba, los objetos de tipo String tienen la misma secuencia de caracteres, s y s, por lo que la llamada al método `respuesta.respuesta("s")` devuelve un valor de `verdadero`, que es lo que se desea. He aquí el fragmento de código corregido:

```
if (respuesta.equals("s") || respuesta.respuesta("S"))
{
  System.out.println("Adiós");
}
```

O, como una alternativa más compacta, se puede utilizar el método `respuestaIgnoreCase` como a continuación:

```
if (respuesta.respuestaIgnoreCase("s"))
{
  System.out.println("Adiós");
}
```

Una tercera alternativa es utilizar el método `charAt` de la clase `String` para convertir la cadena de entrada en un carácter y después utilizar el operado == para comparar ese carácter con los caracteres literales ‘s’ y ‘S’.

```
char resp = respuesta.charAt(0);
if (resp == 'q' || resp == 'Q')
{
  System.out.println("Adiós");
}
```

La gran molestia está en los detalles.

Estas implementaciones no son traducciones triviales del seudocódigo que especificó el algoritmo. Es importante organizar nuestros pensamientos antes de comenzar a escribir código Java; pero ni siquiera una mejor preparación elimina la necesidad de pensar en cómo proceder. ¡Los detalles también importan!

Errores

Se hizo un gran esfuerzo por no utilizar los signos == para comparar cadenas de caracteres porque es muy fácil cometer el error y muy difícil encontrarlo. Es fácil cometer el error porque se utilizan los == todo el tiempo para comparar valores primitivos. Es difícil encontrar el error porque los programas que utilizan el == para comparación de cadenas de caracteres compilan y ejecutan sin errores reportados. ¿Sin errores reportados? Entonces, ¿por qué preocuparse? Porque aunque no se reporten errores, éstos existen: se denominan errores lógicos.

Sea cuidadoso. Pruebe cada aspecto.

Un *error lógico* ocurre cuando el programa ejecuta por completo sin mensajes de error, y la salida es incorrecta. Los errores lógicos son los más difíciles de encontrar y de solucionar porque no hay mensaje que los saque a la luz, indicando lo que se hizo mal. Para empeorar las cosas, la utilización de == para comparación de cadenas de caracteres genera un error lógico sólo en ocasiones, no siempre. Debido a que el error lógico ocurre sólo en ocasiones, los programadores pueden confiarse y pensar que su código está correcto cuando en realidad no es así.

Hay tres categorías principales de errores: de compilación, de ejecución y lógicos. Un error de compilación es aquel que es identificado por el proceso de compilación. Un error de ejecución es el que ocurre mientras se ejecuta el programa y provoca que el programa termine anormalmente. El compilador genera un mensaje por un error de compilación, y la máquina virtual de Java (JVM) genera un mensaje por un error de ejecución. Desafortunadamente, no hay mensajes de error para los lógicos. Depende del programador solucionarlos mediante el análisis de las salidas y planeando cuidadosamente el código.

4.6 Operador lógico !

Ahora es tiempo de considerar al operador lógico de “negación” (!). Asumiendo que se tiene una variable `char` llamada `resp` que contiene 1) una ‘s’ mayúscula o minúscula para cuando el usuario desea salir o 2) algún otro tipo de carácter si el usuario desea continuar. Esta vez, la meta es imprimir “Comencemos…” si `resp` contiene algo diferente a una ‘s’ mayúscula o minúscula. Se puede utilizar una sentencia “if, else” con un bloque “if” vacío como éste:

```
if (resp == 's' || resp == 'S')
{ }
else
{
  System.out.println("Comencemos. . . .");
  . . .
```

Pero esto no es muy *elegante*. Los programadores a menudo utilizan el término elegante para describir el código que está bien escrito y que tiene "belleza". De manera más específica, el código elegante es fácil de entender y de actualizar, es robusto, razonablemente compacto y eficiente. El código vacío anterior del bloque de la sentencia "if" no es elegante porque no es compacto. Si el lector ha incluido un bloque "if" vacío sin que el bloque "else" así lo esté, debe tratar de rescribirlo como un bloque "if" sin bloque "else". El truco es invertir la condición del `if`. En el ejemplo anterior, representaría probar la ausencia de una 's' mayúscula o minúscula en lugar de la presencia de esta letra. Para probar la ausencia de la letra 's' ya sea mayúscula o minúscula, se utiliza el operador `!`.

El operador `!` cambia valores `verdaderos` a valores `falsos`, y viceversa. Esta funcionalidad de cambio de `verdadero` a `falso` y de `falso` a `verdadero` se conoce como una operación de "negación", por ello es que el operador `!` se conoce como operador de "negación". Debido a que se desea imprimir el mensaje "Comencemos . . ." si la condición de la sentencia `if` anterior no es verdadera, entonces se inserta el operador `!` a la izquierda de la condición de la siguiente manera:

```
if (!(resp == 's' || resp == 'S'))
{
  System.out.println("Comencemos. . . .");
  . . .
```

Observe que el `!` está dentro de un paréntesis y fuera de otro. Se requiere de ambos paréntesis porque el compilador los necesita alrededor de la condición completa. El paréntesis interno es necesario también porque sin él el operador `!` actuaría sobre la variable `resp` en lugar de hacerlo sobre la condición completa. ¿Por qué? Porque la tabla de precedencia de operadores (figura 4.6) muestra que el operador `!` tiene mayor precedencia que los operadores `==` y `||`. La solución para forzar los operadores a que se ejecuten `==` y `||` es ponerlos dentro de un paréntesis.

No hay que confundir el operador de negación (!) con el de no igualdad (`!=`). El operador de negación devuelve el valor opuesto a la expresión dada (una expresión `verdadera` devuelve `falsa` y una `falsa` devuelve una `verdadera`). El operador `!=` hace una pregunta: ¿son desiguales estas dos expresiones?

4.7 Sentencia `switch`

La sentencia `switch` funciona de manera similar a como lo hace la forma "if, else if" de la sentencia `if`, en el sentido que permite seguir una de varias rutas; pero una diferencia clave entre la sentencia `switch` y la sentencia `if` es que la determinación sobre la ruta a seguir en la sentencia `switch` se basa en un simple valor. (En una sentencia `if`, la determinación sobre la ruta a seguir se basa en múltiples condicio-

nes, una para cada ruta.) Tener la determinación con base en un simple valor puede resultar en una implementación más compacta y más entendible. Piense en la ruta de manejo que se sigue por la costa de California y la llegada a un cruce con rutas alternativas a través y alrededor de la ciudad. Las rutas diferentes son mejores a ciertas horas del día. Si son las 8 a.m. o las 5 p.m., debería tomarse la ruta externa a la de los negocios para evitar las horas pico de tráfico. Si son las 8 p.m. convendría tomar la ruta que permita apreciar la vista con la puesta del sol. Si es otra hora, se debería tomar la ruta recta porque es la más directa y más rápida. Utilice un simple valor, hora del día, para determinar las rutas alternativas para el proceso de toma de decisiones en una sentencia `switch`.

Sintaxis y semántica

Se recomienda estudiar la sintaxis de las sentencias `switch` en la figura 4.8. Cuando se ejecuta una sentencia `switch`, el control salta a la constante `case` que concuerda con el valor de expresión de control, y la computadora ejecuta todas las sentencias subsiguientes hasta toparse con la sentencia `break`. La sentencia `break` provoca que el control salga de la sentencia `switch` (debajo del paréntesis de llave de cierre). Si no hay constantes `case` que concuerden con el valor de control de la expresión, entonces el control salta a la etiqueta `default` (si es que la hay) o fuera de la sentencia `switch` si no hay sentencia `default`.

Usualmente, las sentencias `break` se colocan al final de cada bloque `case`. Esto se debe a que normalmente se quiere ejecutar sólo la(s) sentencia(s) subordinada(s) de un bloque `case` y después salir de la sentencia `switch`. Sin embargo, las sentencias `break` no son necesarias. A veces se querrá omitirlas, y poderlas omitir es una característica especial de la construcción `switch`; pero es un error muy común olvidar accidentalmente la sentencia `break`. Si no hay una sentencia `break` al final de un bloque de control `case`, el flujo continúa en las subsiguientes constantes `case` y ejecuta todas las sentencias subordinadas hasta que se encuentra con una.

Si no hay una sentencia `break` al final del bloque `case`, el flujo de control avanza a través de las sentencias subordinadas en el bloque `default` (si es que existe).

En relación con la figura 4.8 se deben tomar en cuenta estos detalles:

- Debe haber paréntesis alrededor de la expresión de control.
- La expresión de control a evaluar debe ser tipo `int` o `char`.[2] No es válido utilizar un valor boolean.
- Aunque es común que la expresión de control conste de una simple variable, también puede estar conformada por una expresión más complicada: variables múltiples, operadores e incluso llamadas a métodos, todos se permiten. Estas expresiones deben ser evaluadas a `int` o `char`.
- Debe haber paréntesis de llave alrededor del cuerpo de la sentencia `switch`.

[2] De hecho, una expresión de control puede evaluar *tipos* `byte`, `short` o *enum*. Los tipos `byte` y `short` se estudian en el capítulo 12. Los tipos enum están fuera del alcance de este libro, pero si el lector desea aprender acerca de ellos, puede consultar el sitio http://java.sun.com/docs/books/tutorial/java/javaOO/enum.html

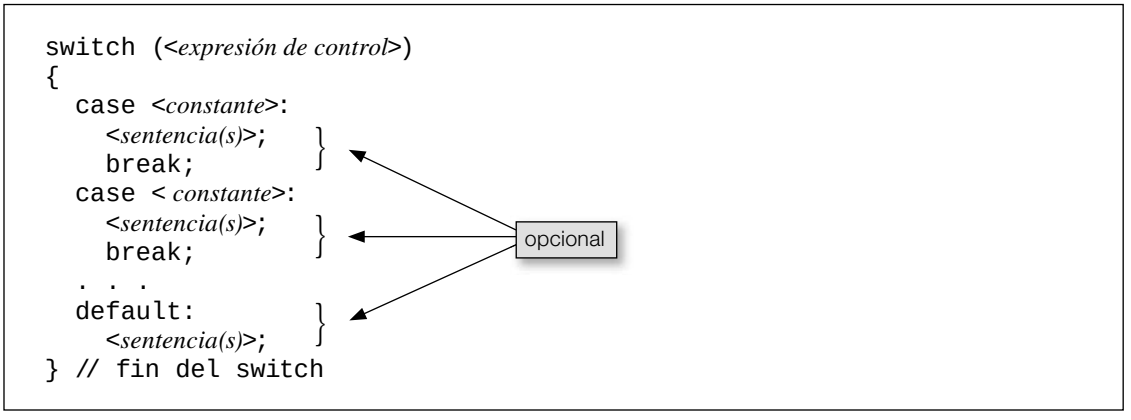


Figura 4.8 Sintaxis de la sentencia `switch`.

- Deben colocarse dos puntos al final de cada constante `case`.
- Aunque las sentencias subordinadas a cada constante `case` tienen sangría, las llaves no son necesarias. Lo anterior es inusual en Java, es la única parte en la que no se requiere de paréntesis de llave alrededor de sentencias que pertenecen lógicamente a algo más.
- Es una buena práctica incluir el comentario `// fin del switch` después de la llave de cierre de cada sentencia `switch`.

Ejemplo de código postal

Para aplicar sus conocimientos de la sentencia `switch`, escriba un programa que lea el código postal y que utilice el primer dígito del mismo para imprimir el área geográfica asociada, con el mismo. He aquí la explicación ejemplificada:

Si el CP inicia con:	Imprimir el mensaje
0, 2, 3	<*cp*> está en la costa este.
4-6	<*cp*> está en el área central.
7	<*cp*> está en el sur.
8-9	<*cp*> está en la costa oeste.
Otra	<*cp*> es un código postal inválido.

El primer dígito de un código postal en Estados Unidos identifica un área geográfica en particular dentro del país. Los códigos postales que inician ya sea con 0, 2 o 3 pertenecen al este, los que inician con 4, 5 o 6 corresponden a la región del área central, etc.[3] El programa debe solicitar al usuario su código postal y utilizar el primer carácter del valor introducido para imprimir la región geográfica del usuario. Además de imprimir la región geográfica, también debe hacerlo con el código postal, tal como el usuario lo introduce. A continuación un ejemplo de lo que el programa debe hacer:

Sesión muestra:

```
Introducir el código postal: 56044
56044 corresponde al área central.
```

Utilice el punto de vista del cliente para especificar el programa.

Ése es el punto de vista del cliente sobre el programa. Ahora se analizará la implementación del mismo: su solución, la cual se muestra en la figura 4.9.

Observe la expresión de control (`cp.charAt(0)`), que obtiene el primer carácter del c.p. De manera alternativa, se podría haber iniciado mediante la lectura del primer carácter en una variable separada (por ejemplo, `primerCar`), y después insertarla en una expresión de control; pero, debido a que el primer carácter fue necesario sólo en un punto, el código se vuelve más compacto al insertar la llamada directa al método `cp.charAt(0)` en el paréntesis de la expresión de control.

La sentencia `switch` compara el carácter en su expresión de control con cada una de las constantes en las etiquetas `case` hasta que concuerda con una de ellas. Debido a que el método `charAt` en la expresión de control devuelve un valor tipo `char`, las constantes en el `case` deben ser todas de tipo `char`. Por tanto, las constantes en el `case` deben tener comillas sencillas alrededor de ellas. Si no se utilizan las comillas sencillas (si en su lugar se utilizan dobles), entonces se obtendrá un error de compilación. ¡La sentencia `switch` no es muy flexible!

Como se mencionó anteriormente, es un error común omitir accidentalmente la sentencia `break` al final de un bloque `case` de la sentencia `switch`. Por ejemplo, suponga que se comete el error en el análisis del c.p.:


```
case '4': case '5': case '6':
  System.out.println(
    cp + " está en el área central plana.");
case '7':
  System.out.println(cp + " está en el sur.");
  break;
```

[3] http://www.nass.usda.gov-census/census97/zipcode/zipcode.htm

Observe que no hay una sentencia `break` al final del bloque del *case* 4, 5, 6. La siguiente sesión de muestra ilustra lo que sucede. Con una entrada de 56044, la sentencia `switch` busca un '5' y se detiene cuando encuentra la etiqueta con el `case '5':`. La ejecución inicia ahí y continúa hasta que encuentra una sentencia `break`; así, el flujo llega a la etiqueta del `case '6':` e imprime el mensaje de área central plana. El flujo posteriormente continúa en el bloque del `case '7':` y equivocadamente imprime el mensaje Sur.

Sesión muestra:

```
Introduzca el código postal: 56044
56044 está en el área central.
56044 está en el Sur.
```

¡error!

Sentencia switch *versus* forma "if, else if" de la sentencia if

Ahora se sabe que la sentencia `switch` permite hacer una o más cosas de una lista con múltiples posibilidades, pero también lo hace la forma "if, else if" de la sentencia `if`, entonces ¿por qué utilizar la sen-

```
/*************************************************************
* CodigoPostal.java
* Dean & Dean
*
* Este programa identifica la región geográfica de acuerdo con el código postal.
*************************************************************/

import java.util.Scanner;

public class CodigoPostal
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String cp;  // c.p. introducido por el usuario

    System.out.print("Introduzca código postal: ");
    cp = stdIn.nextLine();

    switch (cp.charAt(0))
    {
      case '0': case '2': case '3':
        System.out.println(zip + " está en la Costa Este.");
        break;
      case '4': case '5': case '6':
        System.out.println(
          cp + " está en el área del Plano Central.");
        break;
      case '7':
        System.out.println(cp + " está en el Sur.");
        break;
      case '8': case '9':
        System.out.println(cp + " está en el Oeste.");
        break;
      default:
        System.out.println(cp + " es un código postal inválido.");
    } // fin del switch
  } // fin del main
} // fin de la clase CodigoPostal
```

Figura 4.9 Utilización de la sentencia `switch` para encontrar la región geográfica de acuerdo con el c.p.

tencia `switch`? Porque la sentencia `switch` proporciona una solución más elegante (más limpia, con mejor apariencia de organización) para cierta clase de problemas.

Ahora, la pregunta opuesta, ¿por qué utilizar la forma "if else if" de la sentencia `if` en lugar de la sentencia `switch`? Porque las sentencias `if` son más flexibles. Con una sentencia `switch` cada prueba (cada etiqueta `case`) se limita a una correspondencia exacta con una constante tipo `int` o tipo `char`. Con una sentencia `if`, cada prueba puede ser una expresión completa, con operadores, variables y llamadas a métodos.

En resumen, cuando es necesario realizar una acción de entre una lista de posibilidades:

- Utilice la sentencia `switch` si es necesario que corresponda con un valor tipo `int` o `char`.
- Utilice una sentencia `if` si es necesaria mayor flexibilidad.

4.8 Ciclo `while`

Existen dos categorías básicas de sentencias de control: las de ramificación y las de ciclo. La sentencia `if` y la sentencia `switch` implementan funcionalidad de *ramificación* (llamadas así porque las decisiones provocan que el control se ramifique a una sentencia hacia arriba de la que se está analizando). Las sentencias de ciclo `while`, `do` y `for` implementan funcionalidad de ciclos de control. El ciclo `while` se describe en esta sección y el `for` y `do` se explicarán en las dos siguientes; pero se iniciará con una introducción a los ciclos en general.

No duplique código. Use un ciclo.

Para resolver un problema en particular, una de las primeras y más importantes cosas en las que hay que pensar es si existen tareas repetitivas. Las tareas repetitivas deben implementarse con la ayuda de un ciclo. En algunos problemas se puede evitar la implementación de tareas repetitivas mediante sentencias consecutivas secuenciales. Por ejemplo, si se le solicita al lector imprimir 10 veces la frase "¡Feliz cumpleaños!", se podría implementar esta solución mediante 10 sentencias de impresión consecutivas, pero ésa sería una pobre solución. Una mejor sería insertar una sentencia de impresión sencilla dentro de un ciclo que la repita 10 veces consecutivas. La implementación del ciclo es mejor porque es más compacta. Por ejemplo, si es necesario cambiar la frase "¡Feliz cumpleaños!" por "¡Bon anniversaire!" (feliz cumpleaños en francés), entonces sería cuestión de cambiar únicamente la sentencia dentro del ciclo en lugar de actualizar 10 sentencias de impresión separadas.

Sintaxis y semántica del ciclo `while`

Ahora se verá la forma más simple de ciclo, el ciclo `while`. En la figura 4.10 se muestran la sintaxis y la semántica del bucle `while`. La sintaxis del bucle `while` es semejante a la sintaxis de la declaración `if`, excepto que la palabra `while` se usa en lugar de la palabra `if`. No hay que olvidar el paréntesis alrededor de la condición, las llaves, o el hacer sangría en las sentencias incluidas.

Una condición en un ciclo `while` es igual que en una sentencia de condición `if`. Por lo regular emplea operadores de comparación y lógicos, y se evalúa a `verdadero` o `falso`. He aquí la manera en que trabaja el ciclo `while`:

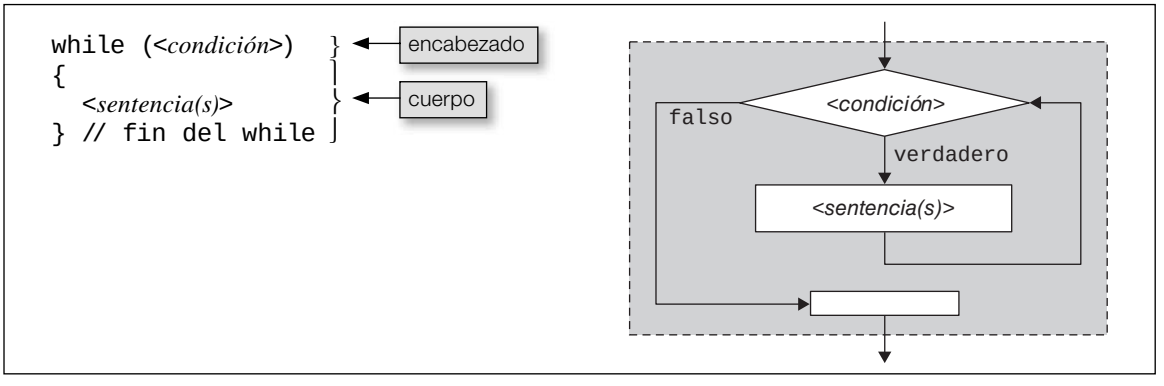


Figura 4.10 Sintaxis y semántica del ciclo `while`.

1. Verifica la condición del ciclo `while`.
2. Si la condición es `verdadera`, ejecuta el cuerpo del ciclo `while` (las sentencias que están dentro de las llaves), después regresa a la condición del ciclo `while` y repite el paso 1.
3. Si la condición es `falsa`, salta hacia abajo del cuerpo del ciclo `while` y continúa con la siguiente sentencia.

Ejemplo

Ahora un ejemplo: un programa que crea un registro de mesa de regalos para una boda. De manera más específica, el programa solicita al usuario información de dos cosas: un regalo y la tienda donde puede comprarse. Cuando el usuario ha introducido y almacenado los valores, el programa imprime la lista del registro. Estudie esta sesión muestra:

Sesión muestra:

```
¿Desea crear una lista de registro de bodas? (s/n): s
Introduzca el nombre del artículo: candelabro
Tienda: Sears
¿Algún otro artículo? (s/n): s
Introduzca el nombre del artículo: podadora
Tienda: Home Depot
¿Algún otro artículo? (s/n: n

Registro de boda:
candelabro - Sears
podadora - Home Depot
```

Utilice E/S simples para especificar el problema.

Ésta es la especificación del problema. La solución aparece en la figura 4.11. Como el lector puede observar por la condición `mas == 's'` del ciclo `while` y la petición al final del ciclo, el programa emplea un ciclo por petición del usuario. El requerimiento inicial arriba del ciclo `while` hace posible salir sin tener que pasar por ningún ciclo. Si se desea forzar el paso por lo menos una vez a través del ciclo, se debe borrar la petición inicial e inicializar de la siguiente manera:

```
char mas = 's';
```

El programa RegistroDeBoda ilustra muchos conceptos periféricos que el lector deseará recordar para programas futuros. Dentro del ciclo `while` observe la sentencia de asignación `+=`, que se repite para conveniencia del lector:

```
registro += stdIn.nextLine() + " - ";
registro += stdIn.nextLine() + "\n";
```

El operador `+=` se incluye cuando se desea agregar algo a una cadena de manera incremental. El programa de RegistroDeBoda almacena todos los valores de regalo y tienda en una sola variable tipo `String` denominada `registro`. Cada captura de un nuevo regalo y tienda se concatena a la variable `registro` mediante el operador `+=`.

En la parte alta y final del ciclo `while` del programa RegistroDeBoda existen llamadas a los métodos `charAt` y `nextLine`, que se repiten para conveniencia del lector:

```
mas = stdIn.nextLine().charAt(0);
```

Las llamadas a los métodos están *encadenadas* juntas mediante la inserción de un punto entre ellas. La llamada al método `nextLine()` lee una línea de entrada del usuario y devuelve la entrada como tipo `String`. Esta cadena de caracteres llama al método `charAt(0)`, que devuelve el primer carácter de la misma. Observe que esto es aceptable y bastante común para encadenar múltiples llamadas a métodos juntos como éste.

Ciclos infinitos

Suponga que está tratando de imprimir los números 1 a 10. ¿Funcionará el siguiente código?

```
int x = 0;
while (x < 10)
{
  System.out.println(x + 1);
}
```

El cuerpo del ciclo `while` hace una sola cosa: imprime 1 (ya que 0 + 1 es 1). Puesto que no hay una sentencia de asignación o incremento para la misma, no actualiza el valor de `x`. Al no haber actualización de `x`, la condición del ciclo `while` (`x < 10`) siempre se evalúa como `verdadera`. Ése es un ejemplo de un *ciclo infinito*. La computadora ejecuta las sentencias en el cuerpo del ciclo una y otra vez, por siempre. Cuando se tiene un ciclo infinito, la computadora parece congelarse o "colgarse".

Inserte sentencias de impresión temporales para ver los detalles.

A veces lo que parece ser un ciclo infinito, no es más que un algoritmo ineficiente al que le toma mucho tiempo terminar. En cualquier caso, el lector puede intentar investigar qué está sucediendo mediante la inserción de una sentencia de diagnóstico dentro del ciclo, que imprima el valor que el programador piense que debería estar cambiando cada vez. Después puede ejecutar el programa y observar lo que pasa con ese valor.

```
/*************************************************************
* RegistroDeBoda.java
* Dean & Dean
*
* Este programa solicita los registros para regalos en una boda.
*************************************************************/

import java.util.Scanner;

public class RegistroDeBoda
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String registro = "";
    char mas;

    System.out.print(
      "¿Desea crear una lista de registro de bodas? (s/n): ");
    mas = stdIn.nextLine().charAt(0);

    while (mas == 'y')
    {
      System.out.print("Introduzca el nombre del artículo: ");
      registro += stdIn.nextLine() + " - ";
      System.out.print("Tienda: ");
      registro += stdIn.nextLine() + "\n";
      System.out.print("¿Algún otro artículo? (s/n): ");
      mas = stdIn.nextLine().charAt(0);
    } // fin del while

    if (!registro.equals(""))
    {
      System.out.println("\nRegistro de bodas:\n" + registro);
    }
  } // fin del main
} // fin de la clase RegistroDeBoda
```

Figura 4.11 Programa RegistroDeBoda con un ciclo `while` y una terminación por petición del usuario.

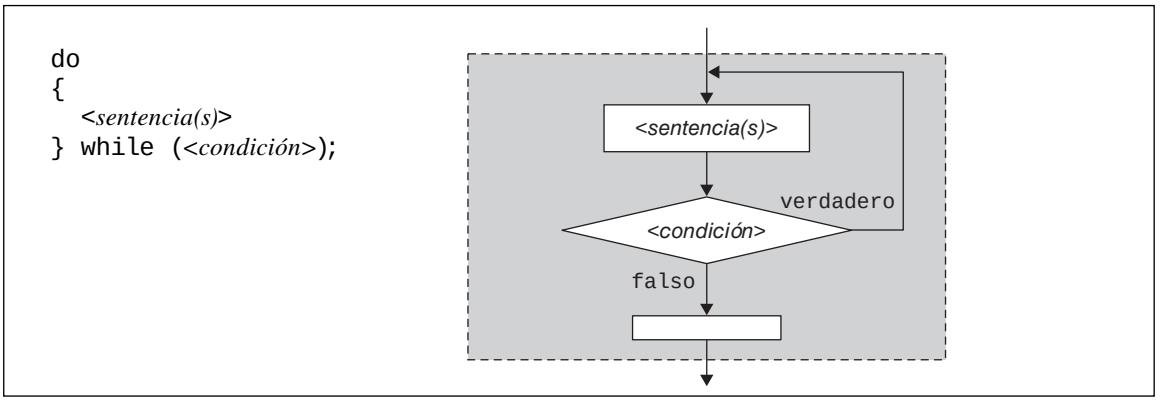


Figura 4.12 Sintaxis y semántica del ciclo `do`.

4.9 Ciclo do

Ahora se considerará un segundo tipo de ciclo en Java: el ciclo `do`. Un ciclo `do` es apropiado cuando se está seguro de querer que el cuerpo del ciclo se repita al menos una sola vez. Debido a que el ciclo `do` se adapta a la forma en que la mayoría del hardware de cómputo ejecuta operaciones de ciclo, es un poco más eficiente que otro tipo de ciclos. Desafortunadamente, su estructura lo hace propenso a errores de programación y, por tanto, a algunos programadores no les gusta utilizarlo. Pero, por lo menos, es importante que el lector lo conozca.

Sintaxis y semántica

La figura 4.12 muestra la sintaxis y semántica del ciclo `do`. Observe que la condición del ciclo `do` está hasta el final. Esto contrasta con el ciclo `while`, donde la condición se presenta al principio. El tener la condición probada hasta el final es la manera en que el ciclo `do` garantiza que sus instrucciones se ejecuten al menos una vez. Observe el punto y coma a la derecha de la condición, requerido por el compilador, omitirlo es un error muy común. Finalmente, observe que la parte de la condición `while` está en la misma línea que el paréntesis de llave de cerrado: esto es un buen estilo. Es posible poner un `while` (<*condición*>); en la línea después del cierre de llaves, pero esto es un mal estilo porque parecería que se está intentando iniciar un ciclo `while`.

He aquí cómo trabaja el ciclo `do`:

1. Ejecuta el cuerpo del ciclo `do`.
2. Verifica la condición final.
3. Si la condición es `verdadera`, vuelve a la parte de arriba del ciclo `do` y repite el paso 1.
4. Si la condición es `falsa` continúa con la sentencia que aparece inmediatamente después del ciclo.

Problema de práctica

Ahora se ilustrará el ciclo `do` mediante un problema de ejemplo. Suponga que se le pide al lector escribir un programa que solicite al usuario introducir las dimensiones de longitud y la anchura de cada recámara de una casa propuesta, con el fin de calcular el espacio total de la casa completa. Después de introducir cada longitud/anchura, se le pregunta al usuario si existe otra recámara. Cuando no hay más recámaras, se imprime la dimensión del espacio de piso total.

¿Cuántas repeticiones?

Para resolver este problema, lo primero que hay que preguntarse es si realmente se requiere un ciclo. ¿Se requiere repetir algo? Sí, se deseará leer las dimensiones de manera repetitiva, para que el ciclo sea apropiado. Para determinar el tipo de ciclo hay que preguntarse: ¿se requerirá la lectura de las dimensiones de la recámara al menos una vez? Sí, cada casa debe tener al menos una, por lo que se necesitará leer al menos un conjunto de dimensiones. Entonces, es apropiada la utilización de un ciclo `do` para este problema. Ahora que se ha avanzado en los aspectos del ciclo, se está listo para tomar lápiz y papel y escribir la solución. Vayamos por éste.

```
/*************************************************************
* EspacioDePiso.java
* Dean & Dean
*
* Este programa calcula el espacio total de piso en una casa.
*************************************************************/

import java.util.Scanner;

public class EspacioDePiso
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double longitud, anchura;    // room dimensions
    double espacioPiso = 0;      // house's total floor space
    char respuesta;              // user's y/n response

    do
    {
      System.out.print("Introduzca la longitud: ");
      longitud = stdIn.nextDouble();
      System.out.print("Introduzca la anchura: ");
      anchura = stdIn.nextDouble();
      espacioPiso += longitud * anchura;
      System.out.print("¿Otra recámara (s/n): ");
      respuesta = stdIn.next().charAt(0);
    } while (respuesta == 's' || respuesta == 'S');

    System.out.println("El espacio total del piso es " + espacioPiso);
  } // fin del main
} // fin de la clase EspacioDePiso
```

Figura 4.13 Utilización de un ciclo `do` para calcular el espacio total de piso.

Cuando tenga la solución por su cuenta, observe la solución en la figura 4.13. ¿Solicitó los valores de longitud y anchura dentro del ciclo `do` y después sumó el producto de longitud por anchura a la variable de espacio total del piso? ¿Se le presentó después al usuario una decisión de continuar o no?

Compare la técnica de terminación de ciclo utilizada en el programa EspacioDePiso con la técnica de terminación de ciclo utilizada en el programa RegistroDeBoda de la figura 4.11. En el programa RegistroDeBoda se le hicieron dos preguntas al usuario: una antes del inicio del ciclo y otra dentro del ciclo justo antes del final. En el programa EspacioDePiso se requiere de una sola pregunta al usuario: dentro del ciclo justo antes del final. El ciclo `do` requiere que haya al menos un paso, pero si éste se acepta, se requiere de algunas líneas más de código que en el ciclo `while`.

Antes de dejar el programa EspacioDePiso observe una característica de estilo. ¿Observa el lector las líneas en blanco arriba y abajo del ciclo `do`? Es un buen estilo separar pedazos de código con líneas en blanco. Debido a que un ciclo es un pedazo lógico de código, es correcto complementar los ciclos con líneas en blanco, a menos que el ciclo sea muy corto, es decir, de menos de cuatro líneas.

4.10 Ciclo `for`

Ahora se considerará un tercer tipo de ciclo: el ciclo `for`. Un ciclo `for` es apropiado cuando se sabe el número exacto de iteraciones antes de que inicie. Por ejemplo, suponga que se desea realizar un conteo del 10 hacia abajo, como éste:

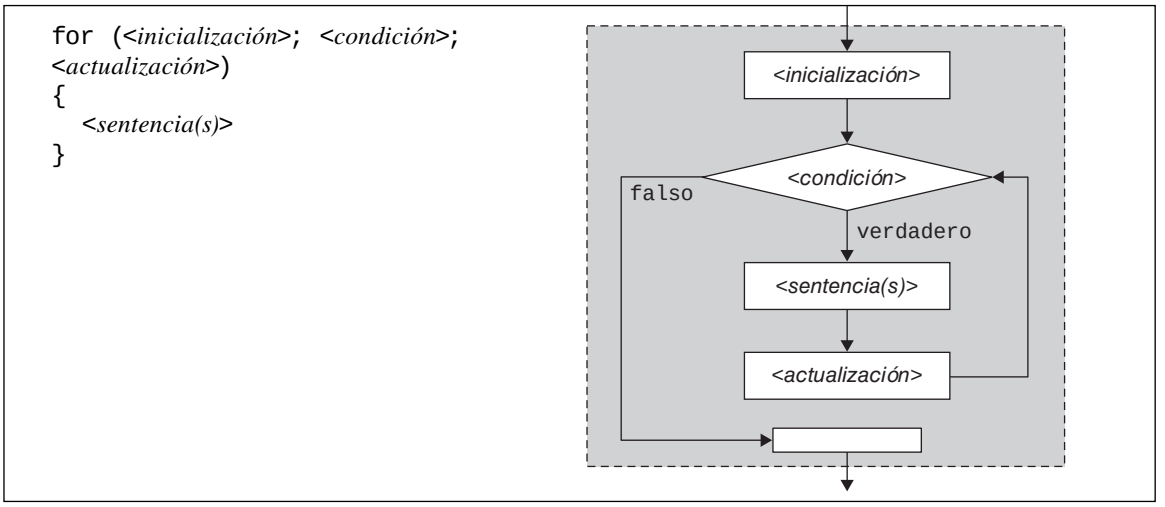


Figura 4.14 Sintaxis y semántica del ciclo `for`.

Sesión muestra:

```
10 9 8 7 6 5 4 3 2 1 ¡Despegamos!
```

En el programa es necesario imprimir 10 números y se debe imprimir cada uno de ellos con la ayuda de una sentencia de impresión dentro del ciclo. Puesto que la sentencia de impresión se debe ejecutar 10 veces, se conoce el número exacto de iteraciones para el ciclo. Por tanto, se debe utilizar un ciclo `for`.

Otro ejemplo: suponga que se quiere encontrar el factorial del número introducido por el usuario, como a continuación:

Sesión muestra:

```
Introducir un número entero: 4
4! = 24
```

Para el 4 factorial es necesario multiplicar los valores de 1 a 4: $1 \times 2 \times 3 \times 4 = 24$. Las tres $\times$ indican que se requiere de tres multiplicaciones. Por lo que el 4 factorial requiere de iteraciones de ciclo. Para el caso general, donde se requiere encontrar el número factorial para el número capturado por el usuario, se debe almacenar dicho número en una variable `cuenta`. Posteriormente, se multiplican los valores de 1 a `cuenta` como a continuación:

$1 \underbrace{* 2 * 3 * \ldots *}_{\texttt{cuenta - 1 número de *}} \texttt{count}$

Los * indican que las multiplicaciones de `cuenta - 1` son necesarias. Así, factorial de `cuenta` requiere de iteraciones de `cuenta - 1`. Puesto que se conoce el número de iteraciones (`cuenta - 1`), se utiliza un ciclo `for`.

Sintaxis y semántica

La figura 4.14 muestra la sintaxis y semántica del ciclo `for`. El encabezado del ciclo `for` realiza mucho trabajo. Ese trabajo se divide en tres componentes: *inicialización*, *condición* y *actualización* de componentes. La siguiente lista explica cómo el ciclo `for` utiliza esos tres componentes. Conforme se lea la lista, conviene consultar el diagrama de flujo de la figura 4.14 para tener una mejor idea de lo que está sucediendo.

1. Inicialización de componente
   Antes de que se ejecute la primera instrucción dentro del ciclo, ejecutar la inicialización del componente.

2. Condición del componente
   Antes de que se realice cualquier iteración, evaluar la condición del componente:
   - Si la condición es `verdadera`, ejecutar el cuerpo del ciclo.
   - Si la condición es `falsa`, terminar el ciclo (pasar a la sentencia debajo del paréntesis de llave de cierre del ciclo).
3. Actualización del componente
   Después de cada ejecución dentro del cuerpo del ciclo, volver al encabezado del ciclo y ejecutar la actualización del componente. Posteriormente, volver a verificar la continuación de la condición del segundo componente, y si se satisface, ir al cuerpo del ciclo otra vez.

Ejemplo de cuenta regresiva

He aquí un fragmento del ejemplo de cuenta regresiva mencionado al principio de esta sección:

```
for (int i=10; i>0; i--)
{
  System.out.print(i + " ");
}
System.out.println("¡Despegamos!");
```

Observe que la misma variable `i` aparece en los tres componentes del encabezado del ciclo `for`. Esa variable tiene un nombre especial. Se denomina *variable índice*. Las variables índice en los ciclos `for` son a menudo, no siempre, nombradas `i` por "índice". Las variables índice con frecuencia inician con un valor bajo, que se va incrementando y se detiene cuando se ha alcanzado el umbral establecido en la condición del componente. Pero, en el ejemplo anterior, la variable índice hace exactamente lo contrario. Comienza en el valor más alto (10), sufre decremento y se detiene cuando alcanza el valor umbral de 0. De manera informal se rastrea un ejemplo:

La inicialización del componente asigna 10 al índice `i`.
La condición del componente pregunta "¿es `i` > 0? La `respuesta` es "sí", por lo que se ejecuta el cuerpo del ciclo.
Imprime 10 (porque `i` tiene el valor de 10), y deja un espacio.
Puesto que se está en la parte baja del ciclo se realiza un decremento en el componente de actualización de `i` de 10 a 9.
La condición del componente pregunta "¿es `i` > 0? La respuesta es "sí", por lo que se ejecuta el cuerpo del ciclo.
Imprime 9 (porque `i` tiene el valor de 9), y deja un espacio.
Puesto que se está en la parte baja del ciclo se realiza un decremento en el componente de actualización de `i` de 9 a 8.
La condición del componente pregunta "¿es `i` > 0? La respuesta es "sí", por lo que se ejecuta el cuerpo del ciclo.
Repite la impresión previa y realiza un decremento en el valor hasta que se imprime 1.
. . .
Después de imprimir 1, ya que se está en la parte más baja del ciclo, se realiza el decremento de `i` de 1 a 0.
La condición del componente pregunta "¿es `i` > 0? La respuesta es "no", por lo que se sale del cuerpo del ciclo, se pasa a la sentencia debajo de la llave de cierre y se imprime "¡Despegamos!"

Alternativamente, se podría haber implementado la solución con el ciclo `while` o con `do`. ¿Por qué es preferible el ciclo `do`? Porque con los ciclos `while` y `do` se requiere de sentencias extra para inicializar y actualizar la variable cuenta. Eso funcionaría bien, pero la utilización del ciclo `for` es más elegante.

Ejemplo de factorial

Ahora es momento de asegurarse de que realmente se entendió el funcionamiento del ciclo `for` mediante el estudio formal de rastreo del segundo ejemplo mencionado al inicio de la sección: el cálculo de un factorial. La figura 4.15 muestra el código de cálculo del factorial y el rastreo de su flujo asociado. Ob-

```
 1  Scanner stdIn = new Scanner(System.in);
 2  int number;
 3  double factorial = 1.0;
 4
 5  System.out.print("Introduzca un número entero: ");
 6  numero = stdIn.nextInt();
 7
 8  for (int i=2; i<=numero; i++)
 9  {
10    factorial *= i;
11  }
12
13  System.out.println(numero + "! = " + factorial);
```

Declare las variables índice del ciclo `for` en el encabezado del ciclo `for`.

Entrada

4

línea#	numero	factorial	i	salida
2	?			
3		1.0		
5				Introduzca un número entero:
6	4			
8			2	
10		2.0		
8			3	
10		6.0		
8			4	
10		24.0		
8			5	
13				4! = 24.0

Figura 4.15 Fragmento de código que ilustra el cálculo del factorial con el rastreo de su flujo.

serve la columna de entrada en la parte de la esquina superior izquierda del rastreo. En los ejemplos de rastreo del capítulo 3 no se tenían ejemplos con entradas, por lo que la mención de las entradas es valiosa ahora. Cuando el programa lee un valor de entrada, se copia la siguiente entrada de la columna de entrada al siguiente renglón debajo de la variable a la cual se asigna la entrada. En este caso, cuando se obtiene `numero = stdIn.nextInt()` se copia el 4 de la columna de entrada al siguiente renglón en la columna `numero`.

Este rastreo muestra que las secuencias 8, 10 se repiten tres veces, por lo que hay tres iteraciones con sangría, como se esperaba. Suponga que se introduce `numero` = 0. ¿Trabaja el programa en ese caso extremo? El encabezado del ciclo inicializa con `int i=2` e inmediatamente después se prueba para ver si `i<= número`. Puesto que esta condición es `falsa`, el ciclo termina antes de que inicie, y el código imprime el valor inicial de `factorial`, el cual es 1.0. Esto es correcto, ya que el factorial de 0 es igual a 1.

¿Qué sucede con el otro caso extremo cuando el valor de entrada es muy largo? El factorial de un número se incrementa mucho más rápidamente de lo que el mismo número lo hace. Si se hubiera declarado que `factorial` fuese de tipo `int`, entonces los valores mayores a 12 provocarían un sobreflujo en la variable de salida, ¡lo que estaría terriblemente mal! Es por ello que se declara `factorial` de tipo `double`. Un `double` tiene mayor precisión que un `int`, y brinda respuestas aproximadamente correctas aun cuando su precisión sea inadecuada. Esto hace al programa más robusto, porque falla *con más gracia*. Esto es, cuando falla, falla sólo un poco, no mucho.

Los pequeños errores son mejores que los grandes.

Alcance del índice del ciclo for

En los ejemplos del ciclo `for` presentados hasta ahora, la variable de índice del ciclo (`i`) se inicializa (es declarada y se le asigna un valor inicial) en el encabezado del ciclo `for`. Esto limita el *alcance* del rango reconocible de la variable de índice al ciclo `for` mismo. En otras palabras, en cualquier momento que se declare una variable en el ciclo `for`, ésta existe y puede ser reconocida y utilizada sólo por el código que está dentro del cuerpo del propio ciclo. Por ejemplo, si se intentara utilizar el valor de la variable de índice `i` en la sentencia de impresión que aparece al final del paréntesis de llave del ciclo `for` en la figura 4.15, el compilador señalaría "cannot find symbol … variable i" ("no se puede encontrar la variable símbolo i").

A veces, las variables que se utilizan en un ciclo deben tener un alcance más allá del alcance del ciclo. El programa Factorial anterior ilustra esto. La variable `factorial` debe estar disponible para la sentencia de impresión después del final del ciclo, por lo que debe ser declarada fuera de éste. Puesto que se requiere en el ciclo, debe declararse antes del ciclo, entonces se declaran otras variables al inicio del método cuyo alcance se extiende a través del mismo.

4.11 Resolución del problema de qué ciclo utilizar

Ahora se compararán los diversos tipos de ciclos.

El punto de decisión del ciclo `do` se encuentra al final del ciclo. Esto contrasta con los ciclos `while` y `for`, donde el punto de decisión está en la parte superior del ciclo. Cuando el punto de decisión está en la parte superior del ciclo, la decisión cuenta más y el código es menos vulnerable a errores de programación.

En la programación como en la vida, hay normalmente muchas formas diferentes de conseguir la misma cosa. Por ejemplo, para un problema que requiere repetición, de hecho se puede utilizar cualquiera de los tres ciclos para resolver problemas de repetición. Aunque ése sea el caso, el programador debe intentar hacer sus programas elegantes, y eso significa que debe elegir el ciclo más apropiado, aunque cualquiera de ellos pueda funcionar.

Un equipo de herramientas requiere de más de una herramienta.

La flexibilidad hace divertida la programación, si se desea ser creativo. Pero si se está iniciando en este campo, tal flexibilidad puede conducirle a la confusión. En la figura 4.16 se proporciona una tabla que in-

Tipo de ciclo	Cuándo utilizarlo	Patrón

Ciclo `for`:	Cuando se sepa cuántas veces se quiere repetir el ciclo antes de iniciarlo.	`for (i=0; i<max; i++)`
		`{`
		`  <`*sentencias*`>`
		`}`

Ciclo `do`:	Cuando se requiera en todos los casos realizar las acciones repetidas al menos una vez.	`do`
		`{`
		`  <`*sentencias*`>`
		`  <`*petición ¿realizar otra vez(s/n)?*`>`
		`} while (<`*respuesta == 's'*`>);`

Ciclo `while`:	Cuando el ciclo es "manejado por eventos", esto es, se realiza el ciclo hasta que las condiciones especiales cambian.	`<`*petición ¿hacerlo (s/n)?*`>`
		`while (<`*respuesta == 's'*`>)`
		`{`
		`  <`*sentencias*`>`
		`  <`*petición ¿realizar otra vez(s/n)?*`>`
		`}`

Figura 4.16 Elección del ciclo adecuado e inicio con el código del ciclo.

tenta aclarar un poco esta confusión; esta tabla sugiere una forma de elegir el tipo apropiado de ciclo y cómo iniciar con el código de ese ciclo. Se utilizan los corchetes de ángulo (< >) para indicar que el texto que aparece dentro de ellos es una descripción del código, no el código real que se codificará. Así, al utilizar los patrones de ciclos `do`, `while` en la figura 4.16, será necesario reemplazar <*petición: ¿realizar otra vez (s/n)?*> con el código real. Por ejemplo, para el programa de un juego, se podría utilizar este código real:

```
System.out.print("¿Deseas jugar otra vez (s/n)? ");
respuesta = stdIn.nextLine().charAt(0);
```

Cuando se está buscando el ciclo más adecuado a utilizar, lo mejor es pensar en los ciclos en el orden que aparecen en la figura 4.16. ¿Por qué? Observe que el ciclo `for` utiliza el menor número de líneas, el ciclo `do` le sigue en menor número, y el ciclo `while` es el que utiliza el mayor número. Así pues, el ciclo `for` es el más compacto y le sigue el `do`; pero el ciclo `while` es más popular que el ciclo `do` porque su condición aparece al principio del mismo ciclo, lo cual lo hace más fácil de encontrar. Así, se puede desear evitar el ciclo `do` por su estructura relativamente difícil; en general, se debe optar por el ciclo que sea más apropiado para el problema en particular.

Cuando se decide cómo escribir el código del ciclo, se pueden usar las plantillas que se muestran en la figura 4.16 como punto de inicio. Es importante saber que se requiere algo más que sólo copiar dicho código, es necesario adaptarlo al problema en particular. Por ejemplo, al escribir el ciclo `for` es común utilizar `i=0` para el componente de inicialización y he ahí el porqué en la inicialización del ciclo `for` aparece ésta. Sin embargo, si algún otro tipo de componente de inicialización es más apropiado, por ejemplo, `cuenta = 10`, entonces debe utilizarse.

4.12 Ciclos anidados

Un *ciclo anidado* es un ciclo que se encuentra en otro ciclo. Se verán ciclos anidados muy a menudo en programas de la vida real. En esta sección se hablará de algunas características comunes inherentes a este tipo de ciclos.

Suponga que se le solicita escribir un programa que imprima un rectángulo de caracteres donde el usuario especifique el largo y el ancho de la figura, así como el carácter específico a utilizar.

Sesión muestra:

```
Introduzca la altura: 4
Introduzca la anchura: 3
Introduzca el carácter: <
<<<
<<<
<<<
<<<
```

Seleccione la mejor herramienta para la tarea a realizar.

Para determinar los ciclos se debe pensar primero en lo que es necesario repetir. Así pues, ¿qué es necesario repetir? Es necesario imprimir renglones de caracteres repetidamente. ¿Qué tipo de ciclo se debería utilizar para imprimir los renglones de forma repetitiva? Primero se intenta utilizar un ciclo `for`. La prueba para un ciclo `for` es si se conoce el número de veces que será necesario imprimir el ciclo. ¿Se conoce el número de veces? Sí, el usuario introduce la altura, y ese valor se puede utilizar para determinar el número de renglones, y también indica el número de veces que se repite el ciclo. Por tanto, se debe utilizar un ciclo `for` para imprimir los renglones sucesivos.

Ahora que ya sabe cómo imprimir múltiples renglones, necesita saber cómo imprimir un renglón individual ¿Es necesario repetir algo cuando se imprime un renglón individual? Sí, es necesario imprimir caracteres de manera repetida. Entonces, ¿qué tipo de ciclo se debe utilizar para eso? Es necesario usar otro ciclo `for` porque se puede utilizar la anchura introducida por el usuario para determinar el número de caracteres a imprimir.

Así pues, se tienen los datos: se necesitan dos ciclos `for`. ¿Debe ponerse uno después del otro? ¡No! Se debe anidar el segundo ciclo dentro del primero que se encarga de imprimir los renglones individua-

```
/*************************************************************
* RectanguloCicloAnidado.java
* Dean & Dean
*
* Este programa calcula el espacio total de piso en una casa.
*************************************************************/

import java.util.Scanner;

public class RectanguloCicloAnidado
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int altura, anchura;         // rectangle's dimensions
    char carImpresion;

    System.out.print("Introduzca la altura: ");
    altura = stdIn.nextInt();
    System.out.print("Introduzca la anchura: ");
    anchura = stdIn.nextInt();
    System.out.print("Introduzca el carácter: ");
    carImpresion = stdIn.next().charAt(0);

    for (int reng=1; reng<=altura; reng++)
    {
      for (int col=1; col<=anchura; col++)
      {
        System.out.print(carImpresion);
      }
      System.out.println();
    }
  } // fin del main
} // fin de la clase RectanguloCicloAnidado
```

Utilizar `print` aquí para permanecer en la misma línea.

Utilizar `println` aquí para moverse a la siguiente línea.

Figura 4.17 Programa que utiliza ciclos anidados para dibujar un rectángulo.

les. Esto tiene sentido si se observa la meta detenidamente: "Imprimir múltiples renglones, y dentro de cada renglón imprimir una secuencia de caracteres." La palabra clave es "dentro de". Esto indica que se debe insertar el segundo ciclo `for` dentro de los paréntesis de llave del primero.

Esta explicación se puede usar como una guía para desarrollar la solución del programa. Cuando se haya hecho, debe compararse la respuesta con el programa RectanguloCicloAnidado de la figura 4.17.

Observe que se utiliza el método `print` para desplegar la sentencia dentro del ciclo interno para mantener impresos los caracteres subsiguientes en la misma línea. Posteriormente, cuando el ciclo interno finaliza, se utiliza un método `println` separado para pasar a la siguiente línea.

Para la mayoría de los problemas que tengan relación con un dibujo de dos dimensiones como el del ejemplo del rectángulo, se querrá utilizar ciclos `for` internos, con variables internas llamadas `reng` y `col` (`reng` y `col` como abreviatura de renglón y columna) ¿Por qué? Esto hace más entendible el código. Por ejemplo, en el primer encabezado del ciclo `for`, la variable `reng` va de 1 a 2 a 3, etc., y eso corresponde perfectamente con los renglones impresos en ese momento por el programa. Sin embargo, se debe considerar que es una práctica común para los ciclos internos utilizar las variables de índice `i` y `j`. ¿Por qué estas letras? Porque la `i` significa *"índice"* y la `j` es la letra que sigue después de la `i`.

En el programa RectanguloCicloAnidado hay dos niveles de anidamiento, pero en general puede haber cualquier número de éstos. Cada nivel agrega otra dimensión al problema. El programa RectanguloCicloAnidado es bastante simétrico. Ambos ciclos son el mismo tipo (ambos son ciclos `for`) y ambos

ciclos hacen el mismo tipo de tareas (imprimir algo). En general, sin embargo, los ciclos anidados no tienen que tener el mismo tipo, y no tienen que realizar el mismo tipo de tareas.

4.13 Variables boolean

Todas las condiciones que aparecen en las sentencias `if` y los ciclos se evalúan como `verdaderas` o `falsas`. En la sección 4.2 se evalúan estos valores boolean. Java también permite definir una variable de este tipo, una variable que puede almacenar un valor de `verdadero` o `falso`. Para declarar una variable `boolean`, se debe especificar que es de este tipo de la siguiente manera:

```
boolean direccionArriba;
```

En esta sección se describe cuándo usar variables `boolean` en general, y se proporciona un programa que usa estas variables, incluida la variable `direccionArriba` que se muestra más adelante.

¿Cuándo utilizar una variable boolean?

Los programas a menudo requieren realizar un rastreo del estado de una condición. Se puede utilizar una variable `boolean` para rastrear cualquiera de los dos *estados* de atributo de una entidad: un sí/no, arriba/abajo, encendido/apagado. Por ejemplo, para escribir un programa que simule las operaciones de apertura de la puerta de una cochera, será necesario rastrear el estado de la dirección de la puerta: ¿está la dirección hacia abajo? Es necesario rastrear el "estado" de dirección porque la dirección determina lo que sucede cuando se presiona el botón de apertura de la puerta. Si el estado de dirección está hacia arriba, cuando se presiona el botón de la puerta la dirección cambia hacia abajo. Si el estado de la dirección es hacia abajo, al presionar el botón de la puerta de la cochera la dirección cambia hacia arriba.

Las variables `boolean` son útiles para mantener el rastro del estado de alguna condición cuando el estado tiene dos valores, por ejemplo:

Valores para el estado de dirección de apertura de una puerta de cochera	Valores comparativos para una variable tipo `boolean` llamada `direccionArriba`
Arriba	`verdadero`
Abajo	`falso`

Ejemplo del abridor de la puerta de cochera

El siguiente esqueleto del código demuestra cómo funciona la variable `direccionArriba`:

```
boolean direccionArriba = true;
do
{
  . . .
  direccionArriba = !direccionArriba;
  . . .
} while (<el usuario presiona el botón de abrir la puerta de la cochera>);
```

La sentencia `boolean direccionArriba = true;` le indica al programa iniciar en la posición de abajo/cerrado e ir hacia arriba cuando el botón de control de la puerta de la cochera se presione. Cada iteración del ciclo representa lo que pasa cuando el usuario presiona el botón. La sentencia `direccionArriba = !direccionArriba` implementa la operación de cambio para abrir la puerta. Si `direccionArriba` contiene el valor de `verdadero`, la sentencia cambia a `falso`, y viceversa.

Ahora se analizará la variable `direccionArriba` en el contexto de un programa completo. En el programa, cada vez que se presiona la tecla Enter en el teclado de la computadora, se simula presionar el botón que abre la puerta. La primera vez que se presiona, la puerta se abre. La segunda vez que se presiona, la puerta se detiene. Con la tercera presión la puerta se cierra; con la cuarta, se detiene, y así hasta que el usuario presiona 's' para hacer que el programa salga. Analice la visión del cliente para el programa PuertaDeCochera:

Sesión muestra:

```
SIMULADOR DE CONTROL DE APERTURA DE PUERTA

Presione Enter o introduzca la letra 's' para salir:
moviendo hacia arriba
Presione Enter o introduzca la letra 's' para salir:
detenida
Presione Enter o introduzca la letra 's' para salir:
moviendo hacia abajo
Presione Enter o introduzca la letra 's' para salir:
detenida
Presione Enter o introduzca la letra 's' para salir: s
```

La figura 4.18 contiene una visión de la implementación de este programa: el código. En el programa advierta que la variable `direccionArriba` se utiliza en la manera que se explicó anteriormente. Observe que hay una segunda variable `boolean`: `enMovimiento`. La variable `direccionArriba` mantiene el control del estado de apertura o cierre. Esa variable de estado sería suficiente si la presión del botón de apertura/cierre generara siempre un movimiento de apertura o cierre; pero, como se mostró en la sesión muestra, ése no es siempre el caso. La mitad del tiempo, la presión del botón de apertura provoca que la puerta de la cochera no se mueva. He aquí la explicación: si la puerta se está moviendo, este movimiento se detiene, y si el movimiento se ha detenido, la puerta lo inicia otra vez. Con ayuda de la segunda variable se monitorea si la puerta está en ese momento en movimiento, `enMovimiento`. La variable de estado `enMovimiento` realiza el cambio (pasa de `falso` a `verdadero`, y viceversa) en cada presión que se hace del botón, mientras que la variable de estado `direccionArriba` verifica únicamente cuándo está detenido el movimiento de la puerta: cada vez que se presiona el otro botón.

Observe cómo se utilizan las variables `enMovimiento` y `direccionArriba` como condiciones para las sentencias `if`*:*

```
if (enMovimiento)
{
  if (direccionArriba)
  {
    . . .
```

Previamente se utilizaron operadores relacionales dentro de las condiciones (por ejemplo, ==, <=); pero la única regla para una condición es que es necesario evaluarla como `verdadera` o `falsa`. Una variable `boolean` es `verdadera` o `falsa`, por lo que su utilización, por sí misma, es válida para una condición. De hecho, la utilización de una variable `boolean` por sí misma para una condición se considera con frecuencia como elegante. Por ejemplo, las condiciones `if` anteriores son más elegantes que la funcionalidad de las siguientes condiciones `if`*:*

```
if (enMovimiento == true)
{
  if (direccionArriba == true)
  {
    . . .
```

El programa PuertaDeCochera *es amigable* porque requiere una mínima cantidad de entradas del usuario. Una entrada del usuario sirve para uno de dos propósitos: la forma simple de entrada (presionar la tecla Enter) simula presionar el botón del control de apertura de la puerta. Cualquier otro tipo de entrada (no sólo una 's') finaliza con el proceso del flujo. Cuando se da un tipo especial de valor (en este caso, cualquiera excepto Enter) se indica al programa detener el ciclo, y se dice que se está utilizando un *valor centinela* para terminar el proceso del ciclo. Debido a que el programa impone un límite mínimo al usuario en términos de entradas, y a que el código es relativamente conciso y eficiente, es apropiado llamar a esto una implementación elegante.

```
/*****************************************************************
* PuertaDeCochera.java
* Dean & Dean
*
* Ésta es la simulación de la operación de una puerta de cochera.
*****************************************************************/

import java.util.Scanner;

public class PuertaDeCochera
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String captura;                  // captura del usuario: tecla enter o s
    boolean direccionArriba = true; // ¿Es la dirección actual hacia arriba?
    boolean enMovimiento = false;   // ¿Esta la puerta de la cochera en movimiento?

    System.out.println("SIMULACION DE APERTURA DE PUERTA DE COCHERA\n");

    do
    {
      System.out.print("Presione Enter, o introduzca 's' para salir: ");
      entry = stdIn.nextLine();

      if (captura.equals(""))       // pressing Enter generates ""
      {
        enMovimiento = !enMovimiento;     // button toggles run state
        if (enMovimiento)
        {
          if (direccionArriba)
          {
            System.out.println("moviendo hacia arriba");
          }
          else
          {
            System.out.println("moviendo hacia abajo");
          }
        }
        else
        {
          System.out.println("detenida");
          direccionArriba = !direccionArriba;  // direction reverses at stop
        }
      } // fin de captura = ""
    } while (captura.equals(""));
  } // fin del main
} // fin de la clase PuertaDeCochera
```

el operador ! cambia el movimiento siempre

cuando se detiene el operador !, la dirección cambia

Figura 4.18 Programa PuertaDeCochera.

4.14 Validación de entradas

En la sección anterior se aprendió a utilizar las variables `boolean` para monitorear un estado en dos sentidos. En esta sección se aprenderá a utilizarlas en particular para un estado común en dos sentidos: el estado de entrada de un usuario en términos de validez o invalidez de su entrada.

```
while (!captura.equals("") && !captura.equalsIgnoreCase("s"))
{
  System.out.println("Tecla inválida.");
  System.out.print("Presione Enter o la tecla 's': ");
  entry = stdIn.nextLine();
}
```

Figura 4.19 Ciclo con validación de captura después de la sentencia de entrada de la figura 4.18.

En la *validación de entrada* el programa verifica que la entrada de un usuario sea válida, esto es, correcta y razonable. Si es válida, el programa continúa; si no lo es, el programa entra en un ciclo que advierte al usuario de la entrada errónea y le solicita volver a introducirla.

En el programa PuertaDeCochera, observe cómo el programa verifica una cadena de caracteres vacía (lo que indica que el usuario desea continuar). Si la cadena no está vacía, asume que el usuario introdujo una 's'. Por consecuencia, no considera la posibilidad de que el usuario, de manera accidental, haya presionado otra tecla antes de presionar la tecla Enter, sino que interpreta esa entrada como una orden de salida y no como un error.

Para hacer más robusto el programa se debe proporcionar una validez de entrada. Hay muchas formas de hacer esto. Una de las más simples es insertar un ciclo `while` cuya condición y todas sus posibilidades erróneas adviertan al usuario de la entrada errónea y le solicite volver a introducirla. Para el programa PuertaDeCochera de la figura 4.18, la validación de entrada la proporciona el fragmento del código de la figura 4.19.

¿Dónde debe insertarse este fragmento de código? Se desea validar la entrada inmediatamente después que se presiona una tecla; entonces, para hacer el programa PuertaCochera más robusto, se debe insertar el fragmento de código anterior en la figura 4.18, inmediatamente después de esta sentencia:

```
captura = stdIn.nextLine();
```

Al ejecutar el programa modificado se produce la siguiente sesión muestra:

Sesión muestra:

```
SIMULADOR DE CONTROL DE APERTURA DE PUERTA

Presione Enter o introduzca la letra 's' para salir:
moviendo hacia arriba
Presione Enter o introduzca la letra 's' para salir: alto
Tecla inválida.
Presione Enter o introduzca la letra 's'
detenida
Presione Enter o introduzca la letra 's' para salir: s
```


Referencias opcionales adelantadas

Hasta este punto, algunos lectores quizá deseen aprender acerca de arreglos. Un arreglo es una colección de elementos relacionados del mismo tipo. La manipulación de arreglos requiere del uso de ciclos. Los arreglos proporcionan al lector un medio para adquirir mayor práctica con el material presentado en el capítulo 4, de manera específica el material de ciclos. Por ahora no es necesario saber de arreglos, pero si el lector no puede esperar, puede leer el capítulo 10, secciones 10.1 a 10.6.

Más adelante se presentan detalles avanzados de sintaxis que corresponden a las sentencias de control. Por ejemplo, la inserción de una expresión de asignación en el encabezado de un ciclo o la utilización de la sentencia `break` para salir de un ciclo. Por ahora no es necesario conocer estos detalles, pero si el lector no puede esperar, puede encontrarlos en el capítulo 11, secciones 11.6 a 11.12.

4.15 Resolución de problemas con lógica boolean (opcional)

Haga la lógica lo más limpia posible.

Las condiciones para las sentencias `if` y los ciclos pueden complicarse en ocasiones. Para entender mejor las condiciones complicadas, ahora se estudiará la lógica que involucra una condición. El aprendizaje de cómo manipular la lógica debe ayudar al lector 1) a simplificar el código de la condición y 2) a depurar problemas lógicos. Este tema se conoce como *lógica boolean* o *álgebra boolean*.

Los bloques de construcción de la lógica boolean ya se han visto antes: los operadores lógicos `&&`, `||` y `!` El lector ya ha visto cómo funcionan los operadores lógicos cuando se aplican en condiciones de comparación de operadores. Por ejemplo, este código (que utiliza el operador `&&` en conjunción con los operadores de comparación `>=` y `<=`) seguramente ya tiene sentido para él:

```
(temp >= 32.0 && temp <= 10.0)
```

Identidades básicas del álgebra boolean

A veces, sin embargo, una expresión lógica es difícil de entender. Esto es particularmente cierto cuando incluye varios operadores "negación" (!). Para entender mejor lo que significa el código y lo que se supone que hace, en ocasiones es útil transformar la expresión lógica. El álgebra boolean proporciona un conjunto de fórmulas llamadas *identidades básicas* que pueden ser utilizadas por cualquiera para realizar transformaciones. Estas identidades básicas se listan en la figura 4.20. La precedencia de varios operadores es la que aparece en la figura 4.6. Esto es, `!` tiene mayor precedencia, `&&` tiene el siguiente nivel de precedencia, `||` tiene el nivel de precedencia más bajo. El símbolo ⟷ significa equivalencia, esto es, lo que sea que se tenga del lado izquierdo de la flecha doble, se puede reemplazar por lo que se tenga del lado derecho, y viceversa.

Las 13 primeras entidades son relativamente sencillas, por lo que el lector debe ser capaz de satisfacer sus validaciones simplemente pensando en ellas. De la misma manera, no se requiere su memoriza-

```
1.   !!x ⟷ x

2.   x || falso ⟷ x
3.   x && verdadero ⟷ x

4.   x || verdadero ⟷ verdadero
5.   x && falso ⟷ falso

6.   x || x ⟷ x
7.   x && x ⟷ x

8.   x || !x ⟷ verdadero
9.   x && !x ⟷ falso

10.  x || y ⟷ y || x                          } ← conmutación
11.  x && y ⟷ y && x

12.  x || (y || z) ⟷ (x || y) || z            } ← asociación
13.  x && (y && z) ⟷ (x && y) && z

14.  x && (y || z) ⟷ x && y || x && z         } ← distribución
15.  x || y && z ⟷ (x || y) && (x || z)

16.  !(x || y) ⟷ !x && !y                     } ← DeMorgan
17.  !(x && y) ⟷ !x || !y
```

Figura 4.20 Identidades básicas del álgebra boolean. Se pueden utilizar estas identidades en cualquier combinación para cambiar la forma de cualquier expresión condicional.

```
/*************************************************************
* TablaVerdad.java
* Dean & Dean
*
* Esto prueba la equivalencia de dos expresiones de tipo boolean
*************************************************************/

public class TablaVerdad
{
  public static void main(String[] args)
  {
    boolean x = false;
    boolean y = false;
    boolean resultado1, resultado2;

    System.out.println("x\ty\tresultado1\tresultado2");
    for (int i=0; i<2; i++)
    {
      for (int j=0; j<2; j++)
      {
        resultado1 = !(x || y);
        resultado2 = !x && !y;
        System.out.println(x + "\t" + y +
          "\t" + resultado1 + "\t" + resultado2);
        y = !y;
      } // fin para j
      x = !x;
    } // fin para i
  } // fin del main
} // fin de la clase TablaVerdad
```

Para probar cualquiera de las dos expresiones boolean, sustitúyalas por estas dos expresiones (sombreadas).

Salida muestra:

```
x          y          resultado1 resultado2
falso      falso      verdadero  verdadero
falso      verdadero  falso      falso
verdadero  falso      falso      falso
verdadero  verdadero  falso      falso
```

Figura 4.21 Programa que genera una tabla lógica para dos expresiones de este tipo. Si los valores `resultado1` y `resultado2` son los mismos en todos los renglones, entonces las expresiones son equivalentes.

ción, debe poder utilizarlas de manera instintiva. Por ejemplo, la *conmutación* significa que se puede cambiar el orden sin cambiar nada más, y la *asociación* significa que se puede mover el paréntesis sin tener que mover algo más. Las últimas cuatro identidades son más misteriosas y algunas podrían incluso parecer irrazonables al principio. Por ejemplo, la *distribución* es una clase de revoltijo, y el *teorema de DeMorgan* señala que se puede negar todo e intercambiar todos los and y or.

Prueba de las identidades boolean

Ahora que se han visto las identidades básicas se estudiará cómo probarlas. La prueba técnica es escribir un programa que compare dos expresiones lógicas de manera arbitraria para todos los posibles valores de las variables `boolean` que éste contenga. Si ambas expresiones evalúan los mismos valores verdaderos para todas las posibles variables verdaderas, entonces son lógicamente equivalentes. La figura 4.21 con-

tiene un programa que lo hace sólo para el caso especial de las expresiones en cualquiera de los dos lados de la identidad básica 16 de la figura 4.20.

Es fácil modificar el programa TablaVerdad de la figura 4.21 para probar cualquiera de las identidades básicas de la figura 4.20. De hecho, se puede modificar el programa para probar cualquier equivalencia prospecto para las expresiones asignadas a `resultado1` y `resultado2`, respectivamente.

Aplicaciones

Hay muchas maneras de utilizar las identidades boolean.

Por ejemplo, considere la condición en la sentencia `if` de la figura 4.5, la cual aparece de la siguiente manera:

```
((temp >= 10) && (temp <= 32))
```

Al utilizar la definición estándar del operador de negación `!` se puede aplicar `!` para cada una de las operaciones de comparación anteriores y obtener su condición equivalente:

```
(!(temp < 10) && !(temp > 32))
```

Se puede aplicar la identidad básica 16 a la condición anterior y obtener la siguiente que es equivalente:

```
!((temp < 10) || (temp > 32))
```

Se puede utilizar la condición anterior como parte de un reemplazo de la sentencia `if` original de la figura 4.5, donde las sentencias subordinadas `if` y `else` se intercambian. He aquí la equivalencia funcional resultante de la sentencia `if`:

```
if ((temp < 10) || (temp > 32))
{
  System.out.println("No adecuada");
}
else
{
  System.out.println("Adecuada");
}
```

Para el siguiente ejemplo, considere la condición en el ciclo `while` de la figura 4.19, que se asemeja a ésta:

```
(!captura.equals("") && !captura.equalsIgnoreCase("s"))
```

Se puede aplicar la identidad básica número 16 a la condición y obtener su condición equivalente:

```
!(captura.equals("") || captura.equalsIgnoreCase("s"))
```

Resumen

- Se puede alterar la secuencia de ejecución de un programa usando una sentencia `if`. La elección entre las dos rutas alternativas la determina la veracidad de la condición de las sentencias.
- Utilice la forma “if, else if” de la sentencia `if` para elegir entre tres o más alternativas.
- Se deben utilizar paréntesis de llave alrededor de dos o más sentencias subordinadas dentro de una sentencia `if`, y es aconsejable utilizarlos aun cuando haya una sola sentencia subordinada.
- Los operadores de comparación de una condición (<, >, <= >=, ==, y !=) tienen mayor prioridad que sus operadores lógicos “and” (&&) y “or” (||).
- Para negar el resultado de las operaciones lógicas && and y/o || or, debe encerrarse entre paréntesis y precederlo con un operador !.
- Utilice la sentencia `switch` para elegir entre varias alternativas con base en identificadores de tipo `int` o `char`.

- Utilice `case` <*número*>: o `case` <*carácter*>: seguido de un `break`; para delimitar cada alternativa en una sentencia `switch`.
- Si la condición en el encabezado de un ciclo `while` es `verdadera`, lo que esté en el bloque subsiguiente se ejecuta y si la condición continúa siendo verdadera, la ejecución se repite.
- Un ciclo `do` ejecuta su bloque al menos una vez, y repite la ejecución siempre que la condición que aparece en el `while` final continúe siendo `verdadera`.
- Un ciclo `for` ejecuta su bloque siempre y cuando la condición en el segundo componente de su encabezado continúe siendo `verdadera`. El primer componente en el encabezado inicializa una variable de cuenta antes de la primera ejecución, y el tercer componente en el encabezado actualiza esa variable de conteo después de cada ejecución y antes de la siguiente evaluación de la condición del segundo componente.
- Se pueden realizar iteraciones multidimensionales insertando un ciclo en otro ciclo.
- Para evitar la duplicación y/o el desorden, se recomienda asignar expresiones lógicas complicadas a variables `boolean`, y utilizar esas variables en sentencias `if` o en condiciones de ciclos.
- Utilice validaciones de entrada para evitar introducir datos erróneos en los programas.
- Opcionalmente, utilice la lógica boolean para simplificar las expresiones en sentencias `if` y condiciones de ciclo, y utilice tablas de verdad para verificar la equivalencia de expresiones lógicas alternativas.

Preguntas de revisión

§4.2 Condiciones y valores `boolean`

1. ¿Cuáles son los dos valores boolean en Java?
2. Proporcione una lista de operadores de comparación en Java.

§4.3 Sentencias `if`

3. Proporcione una sentencia `if` que implemente la siguiente lógica:
   Cuando la temperatura del agua sea menor a 48°C encender el calentador mediante la asignación del valor "encendido" a la variable de tipo string, `calentador`. Cuando la temperatura del agua sea superior a los 60°C, apagar el calentador mediante la asignación del valor "apagado" a la variable de tipo string, `calentador`. No hacer nada cuando la temperatura del agua se encuentre entre estas dos medidas.
4. ¿Cuál es el número máximo de bloques "else if" permitidos en una sentencia `if` que utiliza la forma "if, else if"?

§4.4 Operador lógico `&&`

5. Los operadores relacionales y de igualdad tienen mayor precedencia que los aritméticos (F/V).

§4.5 Operador lógico `||`

6. Corrija el siguiente fragmento de código para que se ejecute y devuelva `Correcto` si la variable `a`, de tipo `int`, es igual ya sea a 2 o 3:

```
if (a = 2 || 3)
{
  imprimir("Correcto\n");
}
```

§4.6 Operador lógico `!`

7. ¿Qué operador de Java revierte la veracidad o falsedad de una condición?

§4.7 Sentencia `switch`

8. ¿Qué sucede si olvida incluir el `break`; al final de un bloque de sentencias después de una etiqueta `case:` en particular?
9. Si se está tratando de sustituir la sentencia `switch` por una sentencia "if else", puede utilizar la condición `if` como la expresión de control en la sentencia `switch`. (F/V)
10. Suponga que la expresión de control en una sentencia `switch` es (`stdIn.next().charAt(0)`), y que se desea que tanto la `'S'` como la `'s'` produzcan el mismo resultado, el cual es:
    `System.out.println("saliendo");`
    Escriba el fragmento de código para el `case` que produzca ese resultado.

§4.8 Ciclo `while`

11. ¿A qué debe ser evaluada la condición en el ciclo `while`?

12. Suponga que desea utilizar la técnica de petición por el usuario para terminar un simple ciclo `while`. ¿Dónde debe colocar la petición?

§4.9 Ciclo `do`

13. ¿Qué está mal en el siguiente fragmento de código?

```
int x = 3;
do
{
  x -= 2;
} while (x >= 0)
```

§4.10 Ciclo `for`

14. Si de antemano sabe el número de iteraciones a través del ciclo, ¿qué tipo de ciclo debe utilizar?

15. Implemente lo siguiente como un ciclo `for`:

```
int edad = 0;
while (edad < 5)
{
  System.out.println("Feliz cumpleaños# " + edad);
  edad = edad + 1;
} // fin del while
```

¿Qué salida generaría el ciclo `for` equivalente?

§4.11 Resolución del problema de qué ciclo utilizar

16. Si un ciclo debe ejecutarse al menos una vez, ¿qué tipo de ciclo es el más apropiado?

§4.12 Ciclos anidados

17. Construya un patrón para un ciclo `for` dentro de otro ciclo `for`. Utilice la letra `i` para la variable de índice externo y la letra `j` para la variable de índice interno.

§4.13 Variables boolean

18. Asuma que la variable `Correcto` se declaró como `boolean`. Reemplace el siguiente código con un ciclo `for` equivalente:

```
Correcto = false;
while (!Correcto)
{
  <sentencia(s)>
}
```

§4.15 Resolución de problemas con lógica boolean (opcional)

19. Dada la expresión lógica:

```
!(!a || !b)
```

Reemplácela con una expresión lógica equivalente desprovista de operaciones de "negación".

Ejercicios

1. [Después de §4.3] Siempre que se envía una carta se debe decidir qué cantidad de timbres poner en el sobre. El usuario desearía usar esta regla genérica: un timbre postal por cada cinco hojas de papel o fracción. Por ejemplo, si son 11 hojas de papel, entonces se utilizan tres timbres. Para ahorrar dinero, simplemente no se envía la carta si un sobre requiere de más de tres timbres postales.

Dado que el número de hojas se almacena en una variable llamada `numHojas`, escriba un fragmento de código que solicite al usuario el número de hojas, que calcule el número de timbres necesarios y que imprima "Utilizar <*no_de_timbres*> timbres" o "No enviar carta", donde <*no_de_timbres*> es un valor entero apropiado.

2. [Después de §4.8] Dado el siguiente fragmento de código:

```
1   double x = 2.1;
2
3   while (x * x <= 50)
```

```
4   {
5     switch ((int) x)
6     {
7       case 6:
8         x--;
9         System.out.println("caso 6, x= " + x);
10      case 5:
11        System.out.println("caso 5, x= " + x);
12      case 4:
13        System.out.println("caso 4, x= " + x);
14        break;
15      default:
16        System.out.println("algo más, x= " + x);
17    } // fin del switch
18    x +=2;
19  } // fin del while
```

Realice el rastreo del flujo del código utilizando la forma larga o la forma corta. Para ayudar al lector a comenzar he aquí el esquema de rastreo. Para la forma corta no se requiere de la columna #línea.

#línea	x	salida

3. [Después de §4.9] Se supone que el siguiente método `main` imprime la suma de los números 1 a 5 y el producto de los números 1 a 5. Encuentre los errores en el programa y arréglelos. No agregue o elimine sentencias. Sólo arregle las existentes. Se recomienda verificar el trabajo del lector probando la ejecución del código en una computadora.

```
public static void main(String[] args)
{
  int cuenta = 0;
  int suma = 0;
  int producto = 0;
  do
  {
    cuenta++;
    suma += cuenta;
    producto *= cuenta;
    if (cuenta == 5)
      System.out.println("Suma = " + suma);
      System.out.println("Producto = " + producto);
  } while (cuenta < 5)
} // fin del main
```

Salida propuesta:

```
Suma = 15
Producto = 120
```

4. [Después de §4.10] Dado el siguiente método `main`:

```
1   public static void main(String[] args)
2   {
3     int i;
4     String depura;
5     for (int i=0; i<3; i++)
6     {
7       switch (i * i)
8       {
9         case 0:
10          depura = "primero";
11          break;
```

```
12        case 1: case 2:
13          depura = "segundo";
14        case 3:
15          depura = "tercero";
16        default:
17          System.out.println("En el default");
18      } // fin del switch
19    } // fin del for
20    System.out.println("i = " + i);
21  } // fin del main
```

Realice el rastreo del flujo del código utilizando la forma larga o la forma corta. Para ayudar al lector a comenzar he aquí el esquema de rastreo. Para la forma corta no se requiere de la columna #línea.

#línea	i	depura	salida

5. [Después de §4.10] Dado el siguiente esqueleto de un programa, inserte código en la sección <*insertar-código-aquí*> de tal manera que el programa imprima el producto de los enteros nones de 2 a num. No se requiere efectuar validación de entradas.

```
public class ProductoEnterosNones
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int i, num, producto;

    System.out.print("Introduzca un número non positivo: ");
    num = stdIn.nextInt();

    <insertar-código-aquí>

    System.out.println("Producto = " + producto);
  } // fin del main
} // fin de la clase ProductoEnterosNones
```

Sesión muestra:

```
Introduzca un número non positivo: 8
Producto = 384
```

6. [Después de §4.12] Dado el siguiente método main:

```
1     public static void main(String[] args)
2     {
3       for (int inicia=1; inicia<=5; inicia+=2)
4       {
5         for (int cuenta=inicia; cuenta>=1; cuenta--)
6         {
7           System.out.print(cuenta + " ");
8         }
9         System.out.println("!Despegamos!");
10      }
11    } // fin del main
```

Realice el rastreo del flujo del código utilizando la forma larga o la forma corta. Para ayudar al lector a comenzar he aquí el esquema de rastreo. Para la forma corta no se requiere de la columna #línea.

#línea	inicia	cuenta	salida

7. [Después de §4.13] Dado el siguiente método main:

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

```
2   {
3     boolean meAma = true;
4
5     for (int num=0; num<4; num++)
6     {
7       meAma = !meAma;
8     }
9     if (meAma)
10    {
11      System.out.println("¡Ella me ama!");
12    }
13    else
14    {
15      System.out.println("¡Ella no me ama!");
16    }
17  } // fin del main
```

Realice el rastreo del flujo del código utilizando la forma larga o la forma corta. Para ayudar al lector a comenzar he aquí el esquema de rastreo. Para la forma corta no se requiere de la columna #línea.

#línea	meAma	num	salida

8. [Después de §4.13] Considere el programa de MarcadorDeBoliche que aparece a continuación:

```
/*************************************************************
* MarcadorDeBoliche.java
* Dean & Dean
*
* Este programa implementa un algoritmo e marcador en el boliche.
*************************************************************/

import java.util.Scanner;

public class MarcadorDeBoliche
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int marcador;
    int marcadorTotal = 0;
    int cuenta = 0;
    double promedio;

    System.out.print("Introduzca el marcador (-1 para salir): ");
    score = stdIn.nextInt();

    while (marcador >= 0)
    {
      marcadorTotal += marcador;
      cuenta++;
      System.out.print("Introduzca el marcador (-1 para salir): ");
      marcador = stdIn.nextInt();
    }

    promedio = (double) marcadorTotal / cuenta;
    System.out.println("El marcador promedio es " + promedio);
  } // fin del main
} // fin de la clase MarcadorDeBoliche
```

Modifique el programa para evitar la división entre cero. Inicialice una variable tipo `boolean` llamada `mas` con valor de `true` y utilícela mientras dure la condición del ciclo. Elimine la petición al usuario y la entrada antes del ciclo y mueva ambas (la petición y la entrada) a la parte alta del ciclo. Utilice una estructura "if, else" en el ciclo para asignar a `mas` el valor de `false` y dejar que se realice el cálculo si la entrada es negativa.

9. [Después de §4.13] Considere el siguiente fragmento de código. Sin cambiar el tipo de ciclo, modifique el código como a continuación. Incorpore una sentencia `if` en el cuerpo del ciclo para prevenir la impresión cuando la entrada iguale el valor centinela de cero.

```
int x;
do
{
  x = stdIn.nextInt();
  System.out.println("cuadrado = " + (x * x));
} while (x != 0);
```

10. [Después de §4.15] He aquí una pregunta capciosa que utiliza la lógica boolean.
Usted viaja por una carretera y llega a una encrucijada. Sabe que una ruta conduce a una olla de oro y la otra a un dragón. Hay dos duendes en la encrucijada, ambos saben la manera de llegar a la olla de oro. Usted sabe que uno de los duendes siempre dice la verdad y que el otro siempre miente, pero no sabe cuál de ellos es. ¿Qué pregunta simple le haría para determinar la ruta a la olla de oro?

Solución a las preguntas de revisión

1. Los valores boolean en Java son `verdadero` (`true`) y `falso` (`false`).
2. Los operadores de comparación en Java son:

```
==, !=, <, >, <=, >=
```

3. Utilice una sentencia "if, else if" como ésta:

```
if (temp < 48)
{
  calentador = "encendido";
}
else if (temp > 76)
{
  calentador = "encendido";
}
```

No incluir un `else` al final.
4. No hay límite en el número de bloques "else if" permitidos.
5. Falso. Los operadores aritméticos tienen mayor precedencia que los de comparación.
6. Las correcciones están subrayadas.

```
(a == 2 || a == 3)
{
  System.out.print("Correcto\n");
}
```

7. El operador de negación `!` revierte la veracidad o falsedad de una condición.
8. Si se omite la sentencia `break`, el control fluye al siguiente bloque `case`, y las sentencias de bloque se ejecutan también.
9. Falso. Una condición "if, else" se evalúa como `verdadera` o `falsa`. La expresión de control en una sentencia `switch` debe evaluarse ya sea como `int` o como `char` (o `byte` o `short`).
10. Cuando más de un identificador produce el mismo resultado, es posible su concatenación en la misma línea utilizando un `case` <*identificador*>: separado para cada identificador:

```
case 'S': case 's':
  System.out.println("saliendo");
```

11. Una condición `while` se evalúa ya sea como `verdadera` o `falsa`.

12. El requerimiento del usuario ocurre antes que la condición de finalización sea probada. Un ciclo `while` prueba la condición de terminación al principio del ciclo. Posteriormente, el requerimiento del usuario debe ocurrir exactamente antes de la parte superior del ciclo y también antes de la parte final del mismo. Si desea que el ciclo siempre se ejecute al menos una vez, entonces omita el requerimiento en la parte superior y reemplácelo con una asignación que haga que la condición de terminación sea `verdadera`.

13. No hay punto y coma después de la condición `while`.

14. Si de antemano sabe el número exacto de iteraciones a través de un ciclo utilice el ciclo `for`.

15. Feliz cumpleaños para un ciclo `for`*:*

```
for (int edad=0; edad < 5; edad++)
{
  System.out.println("Feliz cumpleaños# " + edad);
} // fin del for
```

Salida:

```
Feliz cumpleaños# 0
Feliz cumpleaños# 1
Feliz cumpleaños# 2
Feliz cumpleaños# 3
Feliz cumpleaños# 4
```

16. Un ciclo `do` es más apropiado en situaciones simples donde se requiera de al menos una ejecución de las sentencias dentro del mismo.

17. Un patrón para un par de ciclos `for` anidados:

```
for (int i=0; i<imax; i++)
{
  for (int j=0; j<jmax; j++)
  {
    <sentencia(s)>
  } // fin del for j
} // fin del for i
```

18. La representación de un ciclo `while` en un ciclo `for`:

```
for (boolean correcto=false; !correcto;)
{
  <sentencia(s)>
}
```

19. Dadas las expresiones:

```
!(!a || !b)
```

Comenzando con el lado izquierdo de la identidad básica 16 y avanzando hacia el lado derecho nos da esto:

```
!!a && !!b
```

Después, al utilizar la identidad básica, se obtiene esto:

```
a && b
```

CAPÍTULO 5

Utilización de métodos preconstruidos

Objetivos

- Aprender a incorporar software API preconstruido en los programas y conocer la documentación del software API de Sun.
- Utilizar los métodos y constantes nombradas definidas en la clase `Java Math`.
- Utilizar los métodos de análisis sintáctico (*parsing*) en las clases envoltorio para convertir las representaciones de texto en formatos numéricos, y el uso del método `toString` para hacer lo contrario.
- Utilizar los métodos de la clase `Character` para identificar y modificar los tipos de caracteres y sus formatos.
- Utilizar los métodos de la clase `String` para encontrar el primer índice de un carácter en particular, extraer o reemplazar subcadenas de texto, cambiar de mayúsculas a minúsculas y viceversa, y eliminar espacios en blanco al principio y al final.
- Dar formato de salida con el método `System.out.printf`.
- De manera opcional, utilizar la clase `Random` para generar distribuciones de números aleatorios no uniformes.
- De manera opcional, entender cómo dibujar figuras geométricas, desplegar imágenes y texto en ventanas de despliegue gráfico y ejecutar un applet de Java.

Relación de temas

5.1 Introducción

En los capítulo 3 y 4 la atención se centró en los constructores básicos de Java: variables, operadores de asignación, etc. También se introdujo una nueva técnica de programación más avanzada: la llamada a métodos. Las llamadas a métodos proveen muchas más armas para la programación. En otras palabras, hacen mucho y requieren muy poco trabajo de parte del programador. Por ejemplo, se obtiene un gran beneficio con poco esfuerzo cuando se llama a los métodos de salida `print` y `println`; a los de en-

trada, `next`, `nextLine`, `nextInt` y `nextDouble`; así como a `charAt`, `length`, `equals` y `equalsIgnoreCase` para la manipulación de cadenas de texto. En este capítulo se exponen otros métodos ya escritos y probados, y que están disponibles para cualquier programador.

Al mismo tiempo que este capítulo incrementa la conciencia sobre el valor de los métodos ya escritos, ofrece una mejor perspectiva de lo que éstos pueden hacer de manera general. Y el conocimiento de lo que los métodos pueden hacer es un paso importante en el aprendizaje de la *programación orientada a objetos* (POO). La POO se describe con todo detalle en el siguiente capítulo, pero por ahora se proporciona una breve explicación: la POO es la idea de que los programas deben ser organizados como objetos. Un *objeto* es un conjunto de datos relacionados más un conjunto de comportamientos. Por ejemplo, un string (cadena de caracteres) es un objeto: sus caracteres son un "conjunto de datos relacionados" y sus métodos (`length`, `charAt`, etc.) son un conjunto de comportamientos. Cada objeto es una instancia de una clase. Por ejemplo, un simple objeto de cadena de caracteres "hola," es una instancia de la clase `String`. Este capítulo sirve de transición de los fundamentos de Java vistos en los capítulos 3 y 4 al código de la POO sin tener que implementarlo. De manera más específica, en este capítulo se aprenderá a utilizar métodos, y en el siguiente se aprenderá a escribir nuestras propias clases y los métodos pertenecientes a las mismas.

Existen dos tipos básicos de métodos, los de *instancia* y los de *clase*; de ambos se proporcionarán ejemplos en este capítulo. Los métodos de instancia están asociados con una instancia particular de una clase. Por ejemplo, para llamar al método `length` de la clase `String`, se tiene que asociar con una cadena de texto en particular. Así, en el ejemplo siguiente, observe cómo la variable de cadena de caracteres `primerNombre` está asociada con el método `length`:

```
primerNombre = primerNombre.length();
```

La cadena de caracteres `primerNombre` es un ejemplo de un *objeto llamador*. Como su nombre lo indica, un objeto llamador es un objeto que llama a un método. Cuando se desee ejecutar el método de una instancia, se tiene que anteponer al nombre del método, el nombre del objeto y después un punto.

Los métodos de clase están asociados con una clase entera, no con una instancia en particular. Por ejemplo, existe una clase llamada `Math` que contiene muchos métodos de clase. Sus métodos están asociados con `Math` en general, no con una instancia de ella (de hecho, ni siquiera se pueden crear instancias de la clase `Math`). Para llamar a un método de la clase, se coloca como prefijo en el nombre del método, el nombre de la clase que lo define. Por ejemplo, la clase `Math` contiene el método `round` que devuelve la versión redondeada de un valor en particular. Para llamar al método `round`, se antepone `Math` de la siguiente manera:

```
pagoEnDolares = Math.round(gananciasCalculadas);
```

El presente capítulo inicia con una introducción a la biblioteca API, que es una colección de clases preconstruida de Sun. Después se examina la clase `Math`, que provee de métodos para cálculos matemáticos. Después la atención se centra en las clases envoltorio (*wrapper* en inglés), que encapsulan (envuelven) tipos de datos primitivos. En seguida, se amplía el estudio de la clase `String`, proveyendo métodos adicionales de cadenas de caracteres. Después de esto, se describe el método `printf`, el cual provee funcionalidad para salida formateada. En seguida se habla de la clase `Random`, la cual provee métodos para generar números aleatorios. Y finaliza el capítulo con la sección del apartado adicional GUI, en el que se tratan los métodos proveídos por la clase `Graphics` y se describe cómo llamar a los métodos gráficos desde un applet de Java. ¡Muchas cosas que ver!

5.2 La biblioteca API

Al trabajar en un problema de programación, normalmente se debe verificar si hay clases preconstruidas que satisfagan las necesidades del programa. Si existen esas clases, entonces hay que utilizarlas: "no tratar de reinventar la rueda". Por ejemplo, la captura de datos que hace el usuario es una tarea un tanto complicada que puede realizar la clase `Scanner` de Java. Entonces, cuando es necesario capturar datos en un programa se recomienda hacer uso de esta clase en lugar de desarrollar una nueva.

Hay dos ventajas principales de usar clases preconstruidas: una, se puede ahorrar tiempo ya que no es necesario escribir unas nuevas; y dos, el uso de clases preconstruidas también puede mejorar la cali-

dad de los programas ya que han sido probadas completamente, depuradas y sometidas a un proceso de escrutinio para asegurar su eficiencia.

Búsqueda de la documentación de la biblioteca de clases API

Las clases preconstruidas de Java están almacenadas en la *biblioteca de clases de la Interfase de Programación de Aplicaciones* (*API*), que se conoce simplemente como biblioteca API. El usuario puede encontrar la documentación para la biblioteca API en el sitio Web de Java Sun:

http://java.sun.com/javase/6/docs/api/

Utilice los recursos disponibles.

La biblioteca API contiene decenas de miles de métodos preconstruidos definidos en miles de clases. Las clases están organizadas en casi 200 grupos denominados *paquetes* (un paquete es un grupo de clases). Es poco probable que usted pueda memorizar los nombres de todos estos métodos, dónde están y qué hacen. Entonces, ¿cómo localizar la pieza particular del software preconstruido que se requiere para cubrir las necesidades del proyecto de programación actual?

Puede utilizar un libro (como el que tiene en sus manos ☺) para comenzar a seleccionar clases y métodos de ejemplo. Después visitar el sitio Web de los API de Java Sun y navegar. Observe la figura 5.1, en la que se muestra que la ventana del sitio Web está dividida en tres secciones. La sección superior izquierda despliega una lista de todos los paquetes de Java. La sección inferior izquierda despliega una lista de todas las clases de Java. La sección derecha despliega una variedad de contenidos diversos, donde el tipo de contenido depende de lo que el usuario especifique.

El sitio Web provee varias formas de buscar las cosas:

1. Si cree que la biblioteca API contiene un método o clase que puede ayudarle en el proyecto de programación actual, pero no sabe cuál es, tendrá que navegar para encontrarlo. Hay que iniciar asegurándose de que el vínculo Overview, en la parte superior del sitio Web, esté seleccionado. Cuando el vínculo Overview está seleccionado, la sección derecha de la ventana despliega una lista de todos los paquetes, así como una breve descripción de cada uno de ellos. Si encuentra un paquete que puede ayudarle, dé clic en el nombre. Eso hará que en la sección derecha se desplieguen todas las clases del paquete seleccionado, así como una breve descripción de cada una. Si encuentra una clase que crea útil, dé clic en el nombre. Eso hará que en la sección derecha se desplieguen todos los métodos de la clase y una breve descripción de cada uno. Si encuentra un método que quizá le puede ser útil, dé clic en el nombre, con ello se mostrarán los detalles completos de éste en el lado derecho.

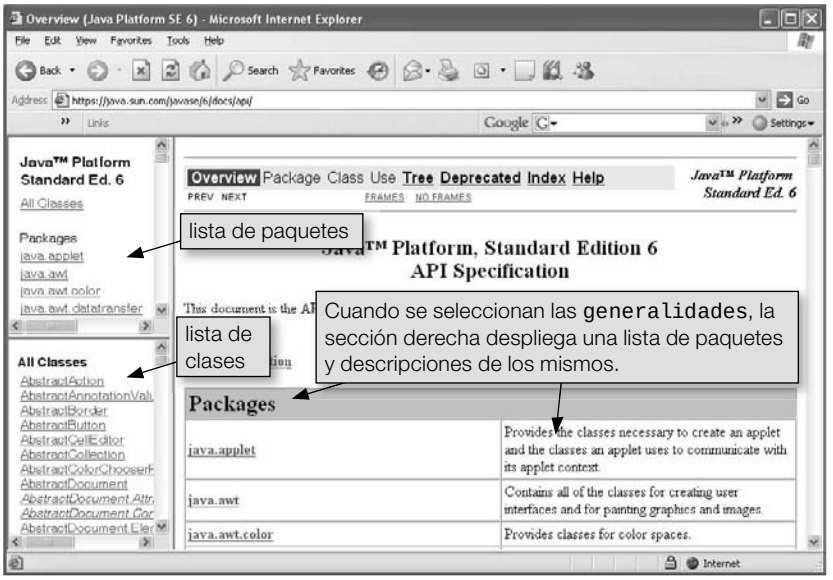


Figura 5.1 Sitio Web del API de Java Sun.

2. Si conoce el nombre de una clase en particular de la cual desea obtener detalles, dé clic en cualquier parte de la sección inferior izquierda de la ventana y presione Ctrl+b (presione ambas teclas al mismo tiempo). La b significa buscar, y cuando se presionan ambas teclas se abre el cuadro de diálogo Buscar. Introduzca el nombre de la clase que desea encontrar en el cuadro de diálogo y dé clic en el botón Buscar siguiente; con esto la clase será encontrada y aparecerá resaltada en la sección inferior izquierda. Al dar clic en dicha clase se desplegarán los detalles de la misma en la sección derecha de la ventana.
3. Si conoce el nombre de un método en particular del cual desee obtener detalles, dé clic en el vínculo Index, en la parte superior de la ventana. Al hacerlo, la sección derecha desplegará las letras del alfabeto; dé clic en la letra que corresponda con la primera letra del método en que está interesado. Esto hará que la sección derecha de la ventana despliegue métodos (y otras entidades, como las clases) que inicien con la letra seleccionada. Busque el método que le interesa y dé clic en el nombre para desplegar todos sus detalles.

Navegar en el sitio Web de API de Java Sun es como navegar en la red, pero no se navega por el mundo entero, sino sólo a través de las bibliotecas API de Java. Se puede hacer y se exhorta a hacerlo cuando se tenga curiosidad.

Utilización de la biblioteca de clases API

Para emplear una clase API en un programa se requiere primero importarla, es decir, cargarla en el programa. Por ejemplo, para utilizar la clase `Scanner` se debe introducir la siguiente sentencia al principio del programa:

```
import java.util.Scanner;
```

Observe que la parte `java.util` en `java.util.Scanner`, corresponde al nombre del paquete. La palabra "`util`" significa "utilidad" y el paquete `java.util` contiene clases de utilidad de propósito general; sin embargo, la única clase `java.util` que es necesaria ahora es la clase `Scanner`; pero hay otras muchas clases útiles en el paquete `java.util`, por ejemplo:

- La clase `Random`, que es útil para trabajar con números aleatorios. Se hablará de esta clase en una sección opcional al final de este capítulo.
- La clase `Calendar` es útil para trabajar con datos de horas y fechas. Se hablará de esta clase en una sección opcional al final del capítulo 8.
- Las clases `Arrays`, `ArrayList`, `LinkedList` y `Collections` son útiles para trabajar con listas o colecciones de datos similares. De la clase `ArrayList` se hablará en una sección opcional al final del capítulo 10.

Si un programa requiere utilizar más de una de las clases de un paquete particular, como dos o más de las clases del paquete `util`, recientemente mencionadas, se pueden `importar` utilizando una sentencia como la siguiente:

```
import java.util.*;
```

El asterisco es un carácter *comodín*. En las sentencias anteriores, el asterisco causa que todas las clases en el paquete `java.util` sean importadas: no sólo la clase `Scanner`. No hay ineficiencia al utilizar la notación con caracteres comodines. El compilador incluirá sólo lo que requiera para que el programa sea compilado.

Muchas clases son tan importantes, que el compilador de Java las importará de manera automática. Estas clases importadas en forma automática se encuentran en el paquete `java.lang`, donde `lang` significa "lenguaje". En efecto, el compilador de Java inserta automáticamente esta sentencia al principio de cada programa:

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

Esto es automático y entendible, por tanto, no hay necesidad de describirlo de manera explícita.

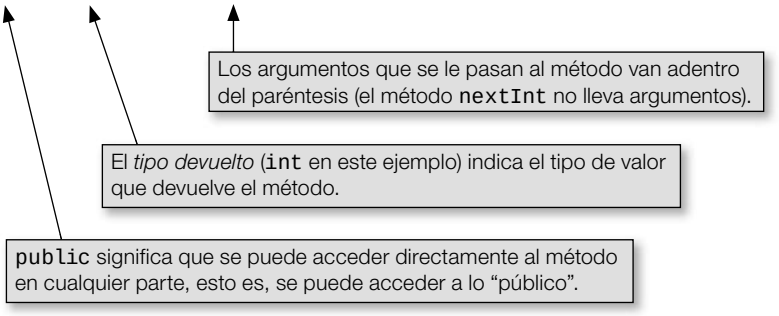
La clase `Math` está en el paquete `java.lang`, por lo que no es necesario importarla si se desea ejecutar operaciones matemáticas. Asimismo, la clase `System` está en el mismo paquete, por lo que no es necesario importarla si se desea ejecutar el comando `System.out.println`.

Encabezados para los métodos API

Para utilizar una clase API no es necesario conocer detalles internos de la misma, sólo es necesario saber cómo interactuar con ella. Para interactuar con la clase, es necesario saber cómo utilizar los métodos pertenecientes a la clase. Por ejemplo, para ejecutar la lectura de datos es necesario saber cómo utilizar los métodos de la clase `Scanner`: `next`, `nextLine`, `nextInt`, `nextDouble`, etc. Para emplear un método es necesario saber qué tipo de *argumentos* pasarle y qué tipo de valores *devuelve*. Los argumentos son las entradas que se proveen a un método cuando se le llama, o que se le pide hacer algo por nosotros, y el valor que devuelve es la respuesta que se regresa.

La forma estándar de presentar la información del método-interface es mostrar el encabezado del código fuente del método. Por ejemplo, he aquí el encabezado del código fuente del método `nextInt` de la clase `Scanner`:

```
public int nextInt()
```



En el encabezado anterior `nextInt`, el modificador de acceso `public` debe parecer conocido al lector, pues todos los encabezados del método `main` utilizan esa palabra. Se hablará de los métodos `private` en el capítulo 8, los cuales son accesibles sólo dentro de la clase que los define. Observe que el método `nextInt` devuelve un valor tipo `Int` y no contiene argumentos dentro del paréntesis. He aquí el ejemplo de una sentencia Java que muestra cómo se podría mandar a llamar al método `nextInt`:

```
int dias = stdIn.nextInt();
```

5.3 Clase `Math`

La clase `Math` es una de las clases preconstruidas en el paquete siempre disponible `java.lang`. Esta clase contiene métodos que implementan *funciones* matemáticas estándar. Una función matemática genera un valor numérico con base en uno o más valores numéricos. Por ejemplo, una función de raíz cuadrada genera la raíz cuadrada de un número. De la misma manera, el método `sqrt` de la clase `Math` devuelve la raíz cuadrada de un número dado. Además de proveer métodos matemáticos, la clase `Math` también provee dos constantes matemáticas: π (el radio de la circunferencia de un círculo a su diámetro) y e (la base de los logaritmos naturales).

Métodos `Math` básicos

Ahora se estudiarán algunos métodos de la clase `Math`. A lo largo del libro, cuando sea necesario presentar un grupo de métodos de la librería API, se introducirán mostrando una lista de sus encabezados y una breve descripción asociada. Los encabezados para los métodos API, por lo común, son referidos como *encabezados API*. La figura 5.2 contiene los encabezados API para algunos de los métodos más populares en la clase `Math`, con una breve descripción asociada.

Como puede leerse en la figura 5.2, se espera que usted encuentre la mayoría de los métodos sencillos; aunque algunos de ellos requieran de una explicación más profunda. Observe el método modificador `static` a la izquierda de los métodos `Math`. Todos los métodos en la clase `Math` son `static`. El modificador `static` significa que son métodos de clase y deben llamarse antecediendo el nombre del método con el de la clase en la cual están definidos. Por ejemplo, he aquí cómo se llamaría al método `abs`:

`public static double abs(double num)`
Devuelve el valor absoluto de un valor tipo `double`.

`public static int abs(int num)`
Devuelve el valor absoluto de un valor tipo `int`.

`public static double ceil(double num)`
Devuelve el número entero más pequeño redondeado hacia arriba y que sea el más cercano a `num.ceil` significa *"ceiling"* (techo en castellano, se le conoce como "función techo").

`public static double exp(double power)`
Devuelve el valor `E` (base de los algoritmos naturales) elevado a `power`.

`public static double floor(double num)`
Devuelve el número entero más grande que sea menor o igual que `num`.

`public static double log(double num)`
Devuelve el logaritmo natural (base `E`) de `num`.

`public static double log10(double num)`
Devuelve el logaritmo 10 de `num`.

`public static double max(double x, double y)`
Devuelve el valor mayor de dos números tipo `double` `x` y `y`.

`public static int max(int x, int y)`
Devuelve el valor mayor de dos números tipo `int` `x` y `y`.

`public static double min(double x, double y)`
Devuelve el valor menor de dos números tipo `double` `x` y `y`.

`public static int min(int x, int y)`
Devuelve el valor menor de dos números tipo `int` `x` y `y`.

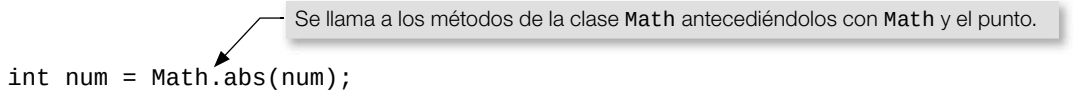
`public static double pow(double num, double power)`
Devuelve `num` elevado a `power`.

`public static double random()`
Devuelve un valor uniformemente distribuido entre 0.0 y 1.0, pero excluyendo a 1.0.

`public static long round(double num)`
Devuelve el número entero que está más cercano a `num`.

`public static double sqrt(double num)`
Devuelve la raíz cuadrada de `num`.

Figura 5.2 Encabezados API y la descripción resumida de algunos de los métodos en la clase `java.lang.Math`.



La sentencia actual actualiza el valor del valor de `num`, para que `num` obtenga el valor absoluto de su valor original. Por ejemplo, si `num` tiene el valor inicial de −15, terminará con el de 15.

Observe cómo la siguiente sentencia no funciona de manera adecuada:

```
Math.abs(num);
```

Se encuentra el valor absoluto de num, pero no se actualiza el contenido almacenado dentro de num. Los métodos de Math devuelven un valor; no actualizan uno. Por ello, si se desea actualizar un valor se debe utilizar el operador de asignación.

En la figura 5.2 observe que se tiene sólo un método pow: uno con parámetros tipo double. No existe un método pow con parámetros tipo Int; pero lo anterior no representa un gran problema porque se puede pasar un valor de tipo Int al método pow. De manera más general, es válido pasar un valor entero a un método que acepta un argumento de tipo punto flotante. Es como asignar un valor entero a una variable de punto flotante, que se trató en el capítulo 3. Ahora se verá cómo funciona éste dentro de un fragmento de código. Hay una regla empírica llamada "Ley de Horton", que dice que la longitud de un río está en proporción con el área drenada por el río de acuerdo con esta fórmula:

$$\text{longitud} \approx 1.4\ (\text{area})^{0.6}$$

He aquí cómo se podría implementar el código de la Ley de Horton en Java:

Es correcto pasar un valor tipo int (area) al método pow, el cual acepta argumentos tipo double.

```
int area = 10000;    // millas cuadradas drenadas
System.out.println("longitud del río = " + 1.4 * Math.pow(area, 0.6));
```

Salida:

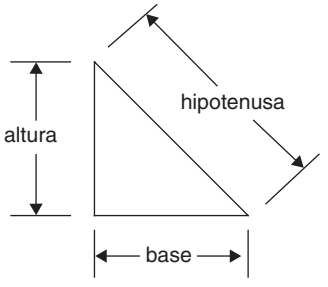
```
longitud del río = 351.66410041134117
```

Observe el método round en la figura 5.2. ¿Qué tanto difiere de la utilización de un operador de conversión (int) o de la utilización de un valor tipo double? El operador (int) trunca la fracción mientras que el método round lo redondea si la fracción es ≥ 0.5.

Como se muestra en la figura 5.2, el método random de la clase Math devuelve un valor uniformemente distribuido entre 0 y 1.0, sin incluir 1.0. "Uniformemente distribuido" significa que existe la misma oportunidad de obtener cualquier valor en el rango especificado. En otras palabras, si se tiene un programa que llame al método random, las oportunidades de obtener los valores 0.317, 0.87, 0.02, o cualquier valor entre 0.0 y 1.0, sin incluir 1.0, son las mismas.

¿Por qué utilizar el método random? Si es necesario analizar una situación de la vida real, que suponga el uso de pruebas aleatorias, se deberá considerar el diseño de un programa que use el método random para modelar este tipo de pruebas. Por ejemplo, si usted trabajara para el departamento de transporte de la ciudad y estuviera a cargo de mejorar el flujo de tránsito en las intersecciones donde hubiera semáforos, podría escribir un programa que utilizara el método random para modelar los eventos aleatorios. Para cada semáforo en que esté interesado, establecería el ciclo de tiempo para cada señalización (por ejemplo, dos minutos entre cada señal verde), y después simularía la llegada de los automóviles a los semáforos a intervalos de tiempos aleatorios. El programa se ejecutaría para simular el flujo de tránsito de una semana, y se mantendría el monitoreo del tiempo de espera de los vehículos. Únicamente tendría que agregar el tiempo del ciclo de cada señal (por ejemplo, un minuto con cuarenta y cinco segundos entre cada señal verde), ejecutaría la simulación una vez más, y determinaría qué ciclo de tiempo de la señal de tránsito produce el promedio de tiempo de espera más bajo.

Es momento de involucrarse en el tema de los métodos de Math utilizados en la figura 5.2, con el ejemplo de un programa completo. Suponga que se desea calcular la longitud de la hipotenusa de un triángulo rectángulo, tomando como base las longitudes de su base y su altura, tal como se muestra en la imagen.



$$hipotenusa = \sqrt{altura^2 + base^2}$$

```
/*****************************************************************
* encontrarHipotenusa.java
* Dean & Dean
*
* Este programa calcula la hipotenusa de un triángulo rectángulo.
*****************************************************************/

import java.util.Scanner;

public class encontrarHipotenusa
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double base;
    double altura;
    double hipotenusa;

    System.out.print("Introduzca la base del triángulo rectángulo: ");
    base = stdIn.nextDouble();
    System.out.print("Introduzca la altura del triángulo rectángulo: ");
    altura = stdIn.nextDouble();
    hipotenusa = Math.sqrt(base * base + altura * altura);

    System.out.println("Longitud de la hipotenusa = " + hipotenusa);
  } // fin del main
} // fin de la clase encontrarHipotenusa
```

se llama a la clase `Math` y método `sqrt`

Sesión muestra:

```
Introduzca la base del triángulo rectángulo: 3.0
Introduzca la altura del triángulo rectángulo: 4.0
Longitud de la hipotenusa = 5.0
```

Figura 5.3 Programa encontrarHipotenusa que demuestra el uso de una de las funciones matemáticas preconstruidas.

La figura 5.3 contiene un sencillo programa que solicita al usuario introducir los valores de base y altura. Después, utiliza el método `sqrt` de la clase `Math` para calcular e imprimir la raíz cuadrada de la suma de los cuadrados de ambas variables. Observe que no se utilizó el método `pow` de la clase `Math` para elevar la base y la altura. Para potencias pequeñas es más eficiente sólo multiplicarlas.

Métodos trigonométricos de la clase `Math`

La figura 5.4 contiene los encabezados API y descripciones de algunos de los métodos de la clase `Math`, que pueden ser útiles para resolver problemas de trigonometría. Los métodos `sin`, `cos` y `tan` implementan las funciones de seno, coseno y tangente, respectivamente. Los métodos `asin`, `acos` y `atan` implementan las funciones de arcoseno, arcocoseno y arcotangente, respectivamente. Todas las funciones trigonométricas e inversas utilizan o devuelven valores de ángulos en radianes, no grados. La utilización o adjudicación de los grados es un error común en la programación. ¡Hay que tener cuidado!

Constantes nombradas

La clase `Math` también contiene valores `double` para dos importantes constantes nombradas:

```
PI = 3.14159265358979323846
E = 2.7182818284590452354
```

`public static double acos(double ratio)`
Devuelve el ángulo en radianes entre 0.0 y π cuyo coseno iguala al valor dado.

`public static double asin(double ratio)`
Devuelve el ángulo en radianes entre $-\pi/2$ y $+\pi/2$ cuyo seno iguala al valor dado.

`public static double atan(double ratio)`
Devuelve el ángulo en radianes entre $-\pi/2$ y $+\pi/2$ cuyo tangente iguala al valor dado.

`public static double cos(double radians)`
Devuelve el coseno de un ángulo expresado en radianes.

`public static double sin(double radians)`
Devuelve el seno de un ángulo expresado en radianes.

`public static double tan(double radians)`
Devuelve la tangente de un ángulo expresado en radianes.

`public static double toDegrees(double radians)`
Convierte un ángulo medido en radianes en un ángulo medido en grados.

`public static double toRadians(double degrees)`
Convierte un ángulo medido en grados en un ángulo medido en radianes.

Figura 5.4 Encabezados API y una breve descripción de algunos métodos trigonométricos en la clase `java.lang.Math`.

`PI` y `E` son constantes matemáticas estándar. La `PI` es el radio del perímetro de un círculo a su diámetro. La `E` es el número Euler, la base para el cálculo de los logaritmos naturales. Los nombres `PI` y `E` aparecen en mayúsculas, porque ése es el estilo estándar para las constantes nombradas. Las constantes tienen valores fijos, y si se intenta asignarles un valor, se obtiene un error de compilación. Así como los métodos de `Math` son llamados métodos de clase, estas constantes se denominan *constantes de clase*, y se puede acceder a ellas a través del nombre de la clase `Math`. En otras palabras, si es necesario traer el valor de π, especificar `Math.PI`.

Suponga que desea calcular el agua necesaria para llenar un globo de agua de 10 centímetros de diámetro. He aquí la fórmula para el volumen de una esfera:

$$\frac{\pi}{6} diámetro^3$$

Y he aquí el código y la salida resultante para calcular el volumen de agua de un globo de agua:

```
double diámetro = 10.0;
double volume = Math.PI / 6.0 * diámetro * diámetro * diámetro;
System.out.print("El volumen del globo en cm cúbicos = " + volumen);
```

Salida:

```
Volumen del globo en cm cúbicos = 523.5987755982989
```

Algunos métodos de la clase `Math` de Java son extremadamente útiles cuando se requiere evaluar una función matemática no trivial, como elevar un número de punto flotante a una potencia fraccional. Otras cosas que sí son sencillas las puede hacer usted mismo. Por ejemplo, ¿puede pensar en una forma primitiva de hacer la misma acción que realiza el método `Math.round`? Es muy sencillo, únicamente se debe sumar `0.5` al número tipo `double` original y después utilizar un operador `long` de conversión en dicho valor `double` para finalizar con la versión redondeada del número original (así es como se hacía en el pasado). Si es así de fácil, ¿por qué preocuparse por utilizar el método `Math.round`? Porque el código se vuelve más legible. La expresión `Math.round(numero)` es autodocumentable, es más informativa que la expresión extraña `((long) (0.5 + numero))`.

5.4 Clases envoltorio (*wrapper*) para tipos primitivos

Una clase *envoltorio* (*wrapper*, en inglés) es un constructor que envuelve (contiene) un tipo de dato primitivo y lo convierte en un objeto con un nombre similar, para que pueda utilizarse en una situación en la que sólo se permiten objetos. Sin embargo, las clases envoltorio hacen más que envolver. También proveen algunos métodos de clases útiles y constantes de clase. El paquete `java.lang` provee clases envoltorio para todos los tipos de datos primitivos de Java. Puesto que este paquete siempre está disponible, no es necesario utilizar el `import` para acceder a esas clases. He aquí las clases envoltorio que serán consideradas, junto con los tipos primitivos que encapsulan:

Clase envoltorio	Tipo primitivo
`Integer`	`int`
`Long`	`long`
`Float`	`float`
`Double`	`double`
`Character`	`char`

Para la mayoría de las clases envoltorio, el nombre de la clase envoltorio es el mismo que su tipo primitivo asociado, con la salvedad de que se utiliza una letra mayúscula como primer carácter. Hay dos excepciones: la clase envoltorio para `int` es `Integer`, y la clase envoltorio para `char` es `Character`.

Métodos

Así como la clase `Math`, las clases envoltorio también contienen métodos y constantes. Se iniciará con los métodos. La explicación se limitará a dos clases de métodos: los métodos que convierten cadenas de caracteres en tipos primitivos y los que convierten tipos primitivos en cadenas de caracteres. Entonces, ¿cuándo sería necesario convertir una cadena de caracteres en un tipo de datos primitivo? Por ejemplo, ¿cuándo sería necesario convertir la cadena "4" en un `int` 4? Si es necesario leer un valor como si fuera una cadena de caracteres y después manipular el valor como si fuera un número, se deberá ejecutar una conversión de cadena de caracteres en número (para la elección de un número de la lotería) o una "s" (para salir). El programa lee la captura del usuario como tipo string, y si el valor no es una "s", entonces convierte dicha captura en número.

Ahora, en el otro sentido, ¿cuándo es necesario convertir un tipo primitivo en una cadena de caracteres? Si se llama a un método que tome un argumento de cadena de caracteres y lo que se obtiene es un argumento numérico, entonces se debe realizar una conversión de número en cadena de caracteres. Con los programas de interfaz gráfica del usuario (GUI), todas las salidas numéricas deben ser string (cadena de caracteres). Así, para desplegar el número, se requiere convertir el número en una cadena de caracteres antes de llamar al método de despliegue GUI. Con los programas GUI, todas las entradas numéricas también deben ser string. Se verán muchos ejemplos de estos procesos más adelante, en los capítulos 16 y 17.

He aquí la sintaxis para convertir cadenas de caracteres en tipos primitivos y tipos primitivos en cadenas de caracteres:

Clase envoltorio	String → primitivo	Primitivo → String
`Integer`	`Integer.parseInt(`<*string*>`)`	`Integer.toString(`<*#*>`)`
`Long`	`Long.parseLong(`<*string*>`)`	`Long.toString(`<*#*>`)`
`Float`	`Float.parseFloat(`<*string*>`)`	`Float.toString(`<*#*>`)`
`Double`	`Double.parseDouble(`<*string*>`)`	`Double.toString(`<*#*>`)`

Todas las clases envoltorio numéricas funcionan de manera similar, por lo que, si se entiende cómo convertir una cadena de caracteres en un `int`, entonces también se entenderá cómo convertir una cadena en otro tipo primitivo. Para convertir una cadena de caracteres en un `int` utilice la clase envoltorio de `int`, `integer` para llamar al método `parseint`. En otras palabras, llame a `Integer.parseint` (<*string*>) y la cadena de caracteres `int` correspondiente será devuelta. De la misma manera, para convertir una cadena de caracteres en `double` se utiliza la clase envoltorio `double` denominada `double` para llamar al `parseDouble`. En otras palabras, llame a `Double.parseDouble`(<*string*>) y la cadena de caracteres `double` correspondiente será devuelta. Más adelante, en esta sección, se presentan

ejemplos no triviales que utilizan los métodos de conversión de las clases envoltorio; pero primero se mostrarán algunos ejemplos sencillos para acostumbrar al usuario a la sintaxis de la llamada a métodos. Aquí se utilizan los métodos `parseInt` y `parseDouble` para convertir cadena de caracteres en datos primitivos:

```
String añoStr = "2002";
String marcadorStr = "78.5";
int año = Integer.parseInt(añoStr);
double marcador = Double.parseDouble(marcadorStr);
```

Para recordar la sintaxis de las llamadas a los métodos de cadena-a-número, piense en el formato <*type*>.`parse`<*type*> para `Integer.parseInt`, `Long.parseLong`, etcétera.

Para convertir un `int` en una cadena de caracteres utilice la clase envoltorio de `int`, `Integer` para llamar al método `toString`. En otras palabras, llame a `Integer.toString`(<*valor-int*>) y será devuelto el valor `int`. De la misma manera, para convertir un `double` en una cadena de caracteres, utilice la clase envoltorio de `double`, `Double`, para llamar a `toString`. En otras palabras, llame a `Double.toString`(<*valor-double*>) y la cadena de caracteres correspondiente del valor `double` será devuelta. Observe el siguiente ejemplo:

```
int año = 2002;
float marcador = 78.5;
String añoStr = Integer.toString(año);
String marcadorStr = Float.toString(marcador);
```

Casi la mitad de los métodos numéricos de las clases envoltorio son métodos clase. Se estudiarán estos métodos. Puesto que son métodos clase se pueden llamar antecediendo el método que se llama con el nombre de la clase envoltorio, justo como se ha hecho.

Constantes nombradas

Las clases envoltorio contienen algo más que sólo métodos: contienen constantes nombradas. Todos los envoltorios de números proveen constantes nombradas para valores máximos y mínimos. Los envoltorios de punto flotante también proveen constantes nombradas para valores mínimo y máximo. Los envoltorios de punto flotante también proveen constantes nombradas para más infinito y menos infinito y para "Not a number" (no un número), un valor indeterminado que se obtiene cuando se intenta dividir cero entre cero. He aquí cómo acceder a las constantes nombradas más importantes definidas en las clases envoltorio `Integer` y `Double`:

```
Integer.MAX_VALUE
Integer.MIN_VALUE
Double.MAX_VALUE
Double.MIN_VALUE
Double.POSITIVE_INFINITY
Double.NEGATIVE_INFINITY
Double.NaN
```

NaN significa "no número".

Hay constantes nombradas comparables para los envoltorios `Long` y `Float`.

Un ejemplo

Ahora se pondrá en práctica el material de los envoltorios y el método `Math.random` mostrándolo en el contexto de un programa completo. El programa Lotería de la figura 5.5 solicita al usuario adivinar un número generado de manera aleatoria entre el rango de 0 y el valor máximo de `int`. El usuario paga $1.00 por cada adivinanza y gana $1 000 000 si adivina la respuesta correcta. Introduzca "s" para salir.

En la inicialización de la variable `numeroGanador` observe cómo el programa genera el valor de un número aleatorio ganador:

```
numeroGanador = (int) (Math.random() * Integer.MAX_VALUE);
```

```
/*************************************************************************
* Loteria.java
* Dean & Dean
*
* Este programa solicita al usuario adivinar un número seleccionado
* de manera aleatoria.
*************************************************************************/

import java.util.Scanner;

public class Loteria
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String entrada;
    int numeroGanador = (int) (Math.random() * Integer.MAX_VALUE);

    System.out.println("¿Desea ganar un millón de dólares?");
    System.out.println("De ser así, adivine el número ganador a" +
      " un número entre 0 y " + (Integer.MAX_VALUE - 1) + ").");
    do
    {
      System.out.print(
        "Inserte $1.00 e introduzca su número o 's' para salir: ");
      entrada = stdIn.nextLine();
      if (entrada.equals("dame una pista"))    //  a back door
      {
        System.out.println("intente: " + numeroGanador);
      }
      else if (!entrada.equals("s"))
      {
        if (Integer.parseInt(entrada) == numeroGanador)
        {
          System.out.println("¡HA GANADO!");
          entrada = "s"; // si alguien gana, está forzado a salir
        }
        else
        {
          System.out.println(
            "Lo sentimos, buen intento, pero no lo suficientemente bueno.");
        }
      } // fin del else if
    } while (!entrada.equals("s"));
    System.out.println("Gracias por jugar. !Vuelva pronto!");
  } // fin del main
} // fin de la clase Loteria
```

Inicialice con números aleatorios a escala.

El método `Integer.parseInt` convierte el tipo de `String` en `int`.

Figura 5.5 El programa Lotería ilustra el uso de la clase envoltorio `Integer`.

Adapte el software existente a sus necesidades.

El punto de inicio es `Math.random( )`, un número aleatorio entre 0.0 y 1.0. Después, la máquina virtual de Java (JVM) lo multiplica por `Integer.MAX_VALUE` para expandir el rango de (0.0 a 1.0) a (0.0 a 2147483647.0). Después, la JVM ejecuta una conversión (`int`) para truncar el componente fraccional.

Observe que el programa lee el número adivinado por el usuario como un string.

```
input = stdIn.nextLine();
```

Al leer el número adivinado como un string y no como número, el programa puede manejar la captura del usuario como una "s" o como la cadena "dame una pista", para brindarle una pista. Si el usuario introduce "s", el programa sale. Si el usuario captura "dame una pista", el programa imprime el número ganador. Gran pista, ¿no cree usted? En ese caso, la pista es en realidad una *puerta trasera*. Una puerta trasera es una técnica secreta para ganar acceso a un programa. La puerta trasera del programa Loteria puede utilizarse con fines de prueba.

Si el usuario no introduce "s" o "dame una pista", el programa intenta convertir la entrada del usuario en un número, llamando a `Integer.parseInt`. El programa, entonces, compara el número convertido con el número ganador y responde por consiguiente.

El programa Loteria podría producir la siguiente salida:

Sesión muestra:

```
¿Desea ganar un millón de dólares?
De ser así, adivine el número ganador (un número entre 0 y 2147483646).
Inserte $1.00 e introduzca su numero o 's' para salir: 66761
Lo sentimos, buen intento, pero no lo suficientemente bueno.
Inserte $1.00 e introduzca su numero o 's' para salir: 1234567890
Lo sentimos, buen intento, pero no lo suficientemente bueno.
Inserte $1.00 e introduzca su numero o 's' para salir: dame una pista
intente: 1661533855
Inserte $1.00 e introduzca su numero o 's' para salir: 1661533855
¡HA GANADO!
Gracias por jugar. ¡Vuelva pronto!
```

5.5 Clase `Character`

En la sección anterior se mencionó la clase envoltorio `Character`, pero no se explicó; es hora de hacerlo. Con frecuencia es necesario escribir programas que manipulen caracteres individuales en una cadena de caracteres de texto. Por ejemplo, quizás usted necesita leer un número telefónico y almacenar sólo los dígitos, saltándose los otros caracteres (guiones, espacios, etc.). Para verificar los dígitos, utilice el método `isDigit` de la clase `Character`. La figura 5.6 muestra algunos de los métodos más populares de la clase `Character`, incluyendo el método `isDigit`.

La mayoría de los métodos que se muestran en la figura 5.6 son sencillos, pero los métodos `toUpperCase` y `toLowerCase` quizá necesiten alguna clarificación. Puesto que ambos métodos son tan similares, clarificaremos sólo uno de ellos, `toUpperCase`. Si se llama al método `toUpperCase` y se le pasa una letra minúscula como parámetro, el método devuelve la versión de esta letra en mayúscula; pero, ¿qué pasa si se llama a ese método y se le pasa una letra en mayúscula o un carácter que no sea una letra? El método devuelve el carácter que se le pasó sin cambio alguno. Y ¿qué sucede si se le pasa una variable tipo `char` en el método `toUpperCase` en lugar de una constante tipo `char`? El método devuelve la versión en mayúscula de la variable `char` que se pasó como argumento, pero no cambia el valor en ésta.

Como es evidente por los modificadores `static` que se muestran en la figura 5.6, la mayoría de los métodos `Character` son métodos de clase. Puesto que éstos son métodos de clase se puede llamar a los mismos, antecediéndolos con el nombre de la clase envoltorio. Observemos un ejemplo. Suponga que tiene una variable tipo `char` llamada `inicialSegundoNombre` y que quiere tener su contenido convertido en una letra mayúscula. He aquí el primer intento de cambiar el contenido de la variable a mayúscula:

```
Character.toUpperCase(inicialSegundoNombre);
```

Esa sentencia compila y ejecuta, pero no cambia el contenido de la variable `inicialSegundoNombre`. He aquí la forma correcta de hacerlo:

```
inicialSegundoNombre = Character.toUpperCase(inicialSegundoNombre);
```

```
public static boolean isDigit(char ch)
```
Devuelve `true` si el carácter especificado es un dígito numérico.

```
public static boolean isLetter(char ch)
```
Devuelve `true` si el carácter especificado es una letra del alfabeto.

```
public static boolean isUpperCase(char ch)
```
Devuelve `true` si el carácter especificado es una letra en mayúscula.

```
public static boolean isLowerCase(char ch)
```
Devuelve `true` si el carácter especificado es una letra en minúscula.

```
public static boolean isLetterOrDigit(char ch)
```
Devuelve `true` si el carácter especificado es una letra o dígito.

```
public static boolean isWhitespace(char ch)
```
Devuelve `true` si el carácter especificado es cualquier clase de espacio en blanco (blanco, tabulador, salto de línea).

```
public static char toUpperCase(char ch)
```
Devuelve el carácter de entrada como carácter en mayúscula.

```
public static char toLowerCase(char ch)
```
Devuelve el carácter de entrada como carácter en minúscula.

Figura 5.6 Encabezados API y breve descripción de algunos de los métodos de la clase `Character`.

El programa VerificaIdentificador de la figura 5.7 ilustra la clase `character` en el contexto de un programa completo. Utiliza los métodos `isLetter` e `isLetterOrDigit` de la clase `Character` para verificar si el usuario introdujo un identificador válido.

5.6 Métodos de `String`

La clase `String` es otra de las clases del siempre disponible paquete `java.lang`. En el capítulo 3 se vieron muchos ejemplos de métodos útiles asociados con objetos de la clase `String`, como `length`, `charAt` y `equalsIgnoreCase`. En esta sección se describen algunos métodos adicionales de la clase `String`, mismos que se muestran en la figura 5.8. Estos métodos de la clase `String` no tienen el modificador de acceso `static`, y no se puede acceder a ellos sino mediante el nombre de la clase. Son métodos de instancia y se debe acceder a ellos con una instancia de string en particular. O, dicho de otra manera, se debe acceder a ellos a través de una llamada a un objeto de tipo string.

Ordenación lexicográfica de los objetos tipo String

Sabemos que los números se pueden comparar para determinar cuál es mayor. Las cadenas de caracteres también se pueden comparar. Cuando la computadora lo hace, utiliza el *orden lexicográfico*. Para la mayoría, el orden lexicográfico es lo mismo que el orden en un diccionario. La cadena "hiena" es mayor que la cadena "halcón" porque *hiena* aparece después de *halcón* en el diccionario.

El método `compareTo` de la clase `String` compara dos cadenas para determinar cuál es mayor. Como se explicó en la figura 5.8, el método `compareTo` devuelve un número positivo si el objeto tipo `String` que llama al método es mayor que la cadena de caracteres utilizada como argumento; un número negativo, si es menor; y cero, si ambos son iguales. El siguiente fragmento de código ilustra esta explicación. En ella, se comparan los títulos de videos de YouTube[1] y se imprimen los resultados de las comparaciones. Si se corre este fragmento de código, no hay que sorprenderse si los dos primeros resul-

[1] YouTube es un popular sitio Web de videos compartidos, adquirido por Google en octubre de 2006, el cual permite a los usuarios cargar, ver y compartir videos.

```
/*************************************************************
* VerificaIdentificador.java
* Dean & Dean
*
* Verifica la entrada de un usuario para determinar si es un identificador válido.
*************************************************************/

import java.util.Scanner;

public class VerificaIdentificador
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String linea;            // captura del usuario
    char ch;
    boolean valido = true; // ¿La línea introducida es un identificador válido?

    System.out.println("Este programa verifica la validez" +
      " un identificador de Java propuesto.");
    System.out.print("Proponga un identificador de Java: ");
    line = stdIn.nextLine();
    ch = line.charAt(0);
    if (!(Character.isLetter(ch) || ch == '$' || ch == '_'))
    {
      valido = false;
    }
    for (int i=1; i<linea.length() && valido; i++)
    {
      ch = linea.charAt(i);
      if (!(Character.isLetterOrDigit(ch) || ch == '$' || ch == '_'))
      {
        valido = false;
      }
    }
    if (valido)
    {
      System.out.println(
        "Felicidades, " + linea + " es un identificador válido de Java.");
    }
    else
    {
      System.out.println(
        "Lo sentimos, " + linea + " no es un identificador válido de Java.");
    }
  } // fin del main
} // fin de la clase VerificaIdentificador
```

Llamadas a métodos de la clase `Character`.

Figura 5.7 Programa VerificaIdentificador.

tados de los valores de salida que se obtengan puedan estar entre 1 y −14. De acuerdo con la especificación de Sun, los primeros dos valores de salida pueden ser cualquier número positivo y cualquier negativo, respectivamente.*

* Aun cuando en la mayoría, si no es que en todos los casos, se traducen los letreros de salida en la pantalla de la computadora, aquí se respetan dichas etiquetas, pues los autores están haciendo uso de un criterio de búsqueda que puede ser localizado en el sitio YouTube, la cual fallaría si se capturara en castellano. (*N. del T.*)

```
String videoyouTube = "Colbert Invades Cuba";
System.out.println(
  videoyouTube.compareTo("Bad Day at Work") + " " +
  videoyouTube.compareTo("Colbert Whitehouse Dinner") + " " +
  videoyouTube.compareTo("Colbert Invades Cuba"));
```

Salida:

```
1 -14 0
```

`public String compareTo(String str)`
Devuelve *un entero que indica el orden lexicográfico de la cadena de caracteres que llama a este método, comparado con la cadena que se envía como parámetro.* Si la cadena que lo llama es "mayor que" la cadena enviada como argumento, devuelve un número positivo; si es "menor que", devuelve uno negativo; y si es igual que aquél, devuelve un cero.

`public int indexOf(int ch)`
Devuelve la posición de la primera ocurrencia del carácter específico.

`public int indexOf(int ch, int fromIndex)`
Devuelve la primera posición del carácter específico en o después de `fromIndex`.

`public int indexOf(String str)`
Devuelve la primera posición de la primera ocurrencia del carácter específico.

`public int indexOf(String str, int fromIndex)`
Devuelve la primera posición de la primera ocurrencia de la cadena de caracteres específica en o después de `fromIndex`.

`public boolean isEmpty()`
Devuelve `true` si la cadena que lo llama es una una cadena vacía (""). De lo contrario, devuelve `false`.

`public String replaceAll(String target, String replacement)`
Devuelve una nueva cadena de caracteres reemplazando todas las ocurrencias de la cadena `target` con la cadena `replacement`.

`public String replaceFirst(String target, String replacement)`
Devuelve una nueva cadena de caracteres reemplazando la primera ocurrencia de la cadena `target` con la cadena `replacement`.

`public String substring(int beginIndex)`
Devuelve la porción de la cadena que lo llama, de la posición especificada en `beginIndex` hasta el final de la misma.

`public String substring(int beginIndex, int afterEndIndex)`
Devuelve la porción de la cadena que lo llama, de la posición especificada en `beginIndex` hasta antes del índice `afterEndIndex`.

`public String toLowerCase()`
Devuelve una nueva cadena de caracteres con todos los caracteres de la cadena que lo llama en minúscula.

`public String toUpperCase()`
Devuelve una nueva cadena de caracteres con todos los caracteres de la cadena que lo llama en mayúscula.

`public String trim()`
Devuelve una nueva cadena de caracteres, eliminando los caracteres en blanco de la cadena que lo llama, del principio al final de la misma.

Figura 5.8 Encabezados API y breves descripciones de algunos de los métodos en la clase `String`.

Análisis de una cadena de caracteres vacía

Antes se aprendió que una cadena vacía no contiene caracteres, y que está representada por dos comillas sin nada entre ellas: "". En ocasiones será necesario analizar una variable tipo string para verificar si contiene la cadena vacía. Por ejemplo, como parte de la validación de una entrada del usuario, cuando se lee ésta, se podría desear verificar si la misma está vacía. El siguiente fragmento de código ejemplifica lo anterior:

```
if (entradaUsuario.equals(""))
...
```

Puesto que la verificación de una cadena vacía es una necesidad común, Sun proporciona un método para cubrir esta necesidad. El método `isEmpty` devuelve `true` si la cadena que llama contiene una cadena vacía y `false`, en caso contrario. El programa de la figura 5.9 utiliza el método `isEmpty` como parte de una validación de captura del usuario en un ciclo `while`. El ciclo `while` fuerza al usuario a introducir un nombre no vacío.

Obtención de resultados con el método substring

Observe los dos métodos `substring` en la figura 5.8. El método `substring` de un solo parámetro devuelve una cadena que es un subconjunto del objeto string que lo llama, empezando en el parámetro de posición nombrado `indiceInicio`, hasta el final de la cadena de caracteres del mismo objeto. El mé-

```
/*****************************************************************
* DemoMetodoString.java
* Dean & Dean
*
* Este programa usa el método isEmpty de la clase String.
*****************************************************************/

import java.util.Scanner;

public class DemoMetodoString
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String nombre;

    System.out.print("Introduzca su nombre: ");
    name = stdIn.nextLine();

    while (nombre.isEmpty())
    {
      System.out.print("captura inválida. Debe de introducir su nombre: ");
      nombre = stdIn.nextLine();
    }
    System.out.println("Hola, " + nombre + "!");
  } // fin del main
} // fin de la clase DemoMetodoString
```

Esto verifica la cadena vacía.

Sesión muestra:

El usuario inmediatamente presiona la tecla Enter.

```
Enter your name:
Invalid entry. You must enter your name: Virginia Maikweki
Hello, Virginia Maikweki!
```

Figura 5.9 Programa `DemoMetodoString`.

todo `substring` de dos parámetros devuelve una cadena que es un subconjunto de caracteres de la cadena que los llama; en este caso, la cadena resultante comienza en la posición `indiceInicio` y se extiende hasta la posición `despuesIndiceFin-1`, donde `indiceInicio` y `despuesIndiceFin` son los dos parámetros del método `substring`.

El siguiente fragmento de código procesa una cita de *Candide*.[2] En esta llamada al método `candide.substring(8)`, `candide` es el objeto que llama, y 8 es el valor del parámetro de `beginIndex`. Como se recordará, los índices en las cadenas de caracteres inician en el índice 0. Así, el 8 se refiere al carácter número 9, el cual es 'c'. Por tanto, la primera sentencia de impresión despliega `cultivar nuestro jardín`. Observe el fragmento de código con la llamada al método `candide.substring(3, 16)`. El 3 y el 16 se refieren a los caracteres 3 y 17, que son 'e' y un espacio. Así, la segunda sentencia de `println` desplegaría `emos cultivar.`

```
String candide = "debemos cultivar nuestro jardín";
System.out.println(candide.substring(8));
System.out.println(candide.substring(3,17));
```

Salida:

```
cultivar nuestro jardín
emos cultivar
```

Si desea probar el fragmento de texto antes presentado, o cualquier código de método `String`, utilice el programa de la figura 5.9 como base. De manera más específica, reemplace las sentencias del método `main` del cuerpo del programa en la figura 5.9 con el nuevo fragmento de código. Posteriormente, compile y ejecute el programa resultante.

Determinación de la posición

Observe los métodos `indexOf` de un solo parámetro en la figura 5.8, que devuelven la posición de la primera ocurrencia de un carácter dado o la subcadena dentro de una cadena de caracteres que lo llama. Si el carácter o la subcadena no existe dentro de la cadena que lo llama, devuelve como resultado −1.

Observe los métodos `indexOf` de dos parámetros en la figura 5.8. Devuelven la posición de la primera ocurrencia de un carácter dado o la subcadena dentro de una cadena de caracteres que lo llama, iniciando la búsqueda en la posición especificada por el segundo parámetro `indexOf`. Si el carácter dado o la subcadena no se encuentran, `indexOf` devuelve como resultado −1.

Es común utilizar uno de los métodos `indexOf` para localizar un carácter o subcadena de interés y posteriormente utilizar uno de los métodos `substring` para extraerlo. Por ejemplo, considere este fragmento de código:[3]


```
String hamlet = "Ser o no ser: he ahí el dilema;";
int índice = hamlet.indexOf(':');
String hamlet2 = hamlet.substring(indice + 1);
System.out.println(hamlet2);
```

Salida:

```
 he ahí el dilema;
```

Observe que el caracter impreso es un espacio en blanco.

Reemplazo de texto

Observe los métodos `replaceAll` y `ReplaceFirst` en la figura 5.8. El método `replaceAll` busca dentro del objeto que lo llama el primer parámetro, y devuelve una nueva cadena de texto, en la que todas

[2] Voltaire, *Candide*, traducido por Lowell Bair, Bantam Books, 1959, versión final.

[3] Shakespeare, *Hamlet,* acto III, esc. 1.

las ocurrencias de `destino` son sustituidas con el segundo parámetro denominado `reemplazo`. El método `replaceFirst` trabaja como `replaceAll`, excepto que sólo la primera ocurrencia de la cadena destino que se busca es reemplazada. He aquí un ejemplo que ilustra ambos métodos:[4]

```
String ladyMacBeth = "¡Fuera, maldita mancha! ¡Fuera, te digo!";
System.out.println(ladyMacBeth.replaceAll("Fuera", "Invádeme"));
ladyMacBeth = ladyMacBeth.replaceFirst(", maldita mancha", "");
System.out.println(ladyMacBeth);
```

Actualiza el contenido de la variable tipo String ladyMacBeth.

Salida:

```
¡Invádeme, maldita mancha! ¡Invádeme, te digo!
Fuera! ¡Fuera, te digo!
```

Observe cómo la segunda sentencia imprime la segunda cita de Lady MacBeth, y cómo se reemplazan las ocurrencias de la palabra "Fuera" por "Invádeme", pero no se cambia el contenido del objeto tipo string. Se puede decir que no se cambia el contenido de éste porque las dos siguientes sentencias generan `¡Fuera!, ¡Fuera, te digo!`; se observa que aparece "*Fuera*" en lugar de *"Invádeme*". La razón de que el método `replaceAll` no cambie el contenido del objeto de tipo string, es que este tipo de objetos son *inmutables*. Inmutable quiere decir sin cambios. Los métodos de la clase String, tales como `replaceAll` y `replaceFirst` devuelven una nueva cadena de caracteres, no una nueva versión del objeto que lo llama. Si realmente se quiere cambiar el contenido de una variable de tipo string, es necesario asignar el contenido de la misma a una nueva variable de este tipo. Esto es lo que sucede con la tercera sentencia, donde la JVM asigna el resultado de la llamada al método `replaceFirst` a la variable `ladyMacBeth`.

En el ejemplo de Lady Mac Beth, la llamada al método `replaceFirst` borra la frase "maldita mancha" y la reemplaza con una cadena vacía. Puesto que hay una sola ocurrencia de "maldita mancha", tanto `replaceAll` como `replaceFirst` devolverían el mismo resultado; pero `replaceFirst` es un poco más eficiente y es por ello que se utiliza en este caso.

Eliminación de espacios en blanco y conversión de caracteres

Observe la utilización de los métodos `trim`, `toLowerCase` y `toUpperCase` en la figura 5.8. El método `trim` elimina los espacios en blanco antes y después de la cadena de caracteres del objeto que lo llama. El método `toLowerCase` devuelve una cadena de caracteres idéntica a la de la que lo llama, excepto que todos los caracteres aparecen en minúscula. El método `toUpperCase` devuelve una versión en mayúscula del objeto tipo string que lo llama. Para ver cómo funcionan estos métodos se intentará cambiar el código anterior de Hamlet de la siguiente manera:

```
String hamlet = "Ser o no ser: he ahí el dilema;";
int indice = hamlet.indexOf(':');
String hamlet2 = hamlet.substring(indice + 1);
System.out.println(hamlet2);
hamlet2 = hamlet2.trim();
hamlet2 = hamlet2.toUpperCase();
System.out.println(hamlet2);
```

Ahora la salida aparece así:

Salida:

```
 he ahí el dilema;
HE AHÍ EL DILEMA;
```

Observe cómo el método `trim` elimina el espacio en blanco de la cadena de caracteres `hamlet2`. También observe cómo el método `toUpperCase` devuelve una versión en mayúsculas de `hamlet2`.

[4] Shakespeare, *MacBeth,* acto V, esc. 1.

Inserción

Para hacer una inserción se debe saber dónde se desea hacerla. Si se desconoce el índice desde el que se desea iniciarla, se puede encontrar mediante el método `indexOf` con un solo argumento en el método substring. Posteriormente, se sustrae la subcadena hasta dicho índice, se concatena la inserción deseada y se concatena la subcadena después de dicho índice. El siguiente fragmento de código ejecuta dos inserciones dentro de una cadena de caracteres. De manera más específica, el fragmento de código inicia con una cita atribuida al filósofo y matemático francés del siglo XVII, René Descartes: "Toda la naturaleza hará lo que deseo." Posteriormente, inserta dos cadenas de caracteres y transforma el mensaje en una frase crudamente contrastante de Charles Darwin: "Toda la naturaleza es perversa y no hará lo que deseo."[5]

```
String descartes = "Toda la naturaleza hará lo que deseo.";
String darwin;
int indice;
index = descartes.indexOf("hará");
darwin = descartes.substring(0, indice) +
  "is perversa & " +
  descartes.substring(indice);
indice = darwin.indexOf("hará");
darwin = darwin.substring(0, indice) +
  "no " +
  darwin.substring(indice);
System.out.println(darwin);
```

Salida:

```
Toda naturaleza es perversa y no hará lo que deseo.
```

5.7 Salida formateada con el método `printf`

Hasta ahora se han utilizado de manera reiterada los métodos `System.out.print` y `System.out.println`. Éstos funcionan bien la mayoría de las veces, pero hay un tercer método de `System.out` que se querrá utilizar de vez en cuando para formatear la salida. Es el método `printf`, donde la "f" significa "formateada". En esta sección se describirá este método.

Salida formateada

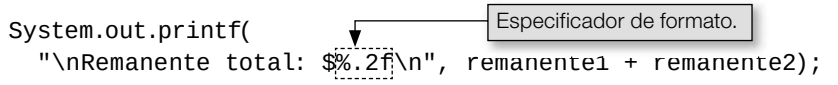
La mayoría de los programas tienen como meta calcular algo y desplegar el resultado. Es importante que el resultado desplegado sea entendible. Si no es entendible, entonces nadie se molestará en utilizar el programa, aun cuando realice sus cálculos impecablemente. Una forma de hacer que los resultados desplegados sean entendibles es formatear la salida. Lo anterior significa tener los datos alineados correctamente en columnas, tener números de punto flotante que muestren el mismo número de dígitos después del punto decimal, etc. Observe el formato en el reporte de presupuesto siguiente. La columna izquierda está alineada a la izquierda; las otras columnas, a la derecha. Los números muestran dos dígitos a la derecha del punto decimal. Los números muestran comas entre cada grupo de tres dígitos a la izquierda del punto decimal. Y finalmente, los números muestran paréntesis para indicar que un número es negativo.

[5] *Letras de Charles Darwin*, editado por Frederick Burkhardt, Cambridge (1996). Charles Darwin inició sus estudios universitarios en la Universidad de Edimburgo en 1825, en el área de medicina como lo hizo su padre. La carrera de medicina no le atrajo mucho, por lo que decidió emigrar a la Universidad de Cambridge, donde obtuvo un grado académico en preparación para la carrera de ministro de la iglesia; pero lo que él realmente disfrutaba era el estudio de los insectos en el granero de la familia. Justamente después de su graduación y poco antes de iniciar su carrera como ministro de la iglesia, algunos contactos familiares, una buena referencia de un profesor del colegio y una personalidad agradable le dio la oportunidad de viajar alrededor del mundo en compañía de un brillante capitán marino llamado Robert FitzRoy (quien posteriormente inventaría el pronóstico del tiempo). Este viaje llevó a Darwin a convertirse en uno de los científicos más influyentes del mundo moderno.

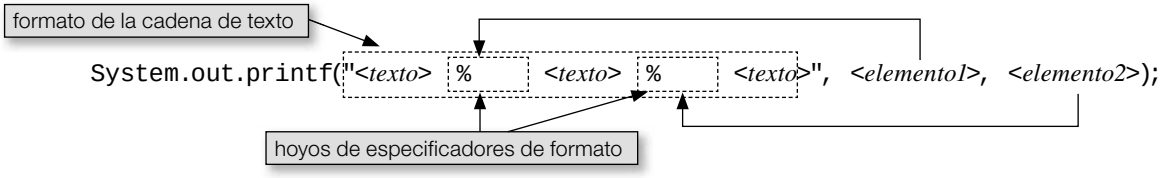
```
Cuenta                          Actual     Presupuesto     Remanente
-------                         ------     -----------     ---------
Arts. de oficina              1,150.00       1,400.00        250.00
Fotocopiado                   2,100.11       2,000.00      (100.11)

Remanente total: $149.89
```

Aprenda a utilizar herramientas versátiles.

El método `System.out.printf` se encarga de generar la salida formateada. El método `printf` tiene muchas características de formateo. Se tratará de hacerlo simple y explicar sólo unas de las más populares. Se iniciará la explicación del método `printf` mostrando cómo generar la línea "Remanente total" en el reporte anterior. He aquí el código:



El primer argumento del método `printf` es conocido como *cadena de formato*. Contiene texto que se imprime tal cual, más especificadores de formato que manejan la impresión de formato. En el ejemplo anterior, `"\nRemanente total: $...\n"` es el texto que se imprime tal cual. Y `%.2f` es el especificador de formato. Se puede pensar en el *especificador de formato* como un hoyo en el cual se insertan datos. En el ejemplo anterior, `remanente1 + remanente2` son los datos que se colocan. Si `remanente1` tiene 250 y `remanente2` tiene −100.11, la suma es 149.89 y dicha cifra se coloca en el hoyo de especificador de formato. El especificador de formato inicia con `%` porque todos los especificadores de formato deben comenzar con ese símbolo. El especificador de formato `.2` causa que se desplieguen dos dígitos después del punto decimal. El especificador de formato `f` indica que el dato es un número de punto flotante. El ejemplo muestra sólo un especificador de formato, por lo que se debe tener un argumento de dato correspondiente. He aquí un ejemplo de lo que se está hablando:



Detalles de los especificadores de formato

Los especificadores de formato son unos pequeños bichos poderosos. No se intentará describir todo su poder, pero se proporcionarán suficientes detalles para ponerlo de pie y hacerlo ejecutar. Si usted se encontrara con algún problema de formato que no puede resolver con la cobertura limitada, localice el tema de `printf` en el sitio Web de Java Sun y busque detalles de formato de cadenas de caracteres; pero prepárese para muchos de ellos. Sun proporciona un número tremendo de opciones con el método `printf`.

He aquí la sintaxis para un especificador de formato:

%[*banderas*][*anchura*][*.precisión*]*carácter de conversión*

Antes se vio que el símbolo % indica el inicio de un especificador de formato. Las banderas, anchura, precisión y el carácter de conversión representan las diferentes partes del especificador. Cada una de ellas especifica un rasgo de formato diferente. Se cubrirán en el orden de derecha a izquierda. Así, se describirá primero el carácter de conversión, pero antes de pasar a los detalles del mismo, observe las llaves cuadradas, que indican que algo es opcional. Así las banderas, anchura y partes de precisión son opcionales. Sólo los caracteres % y los de conversión son obligatorios.

Caracteres de conversión

El carácter de conversión le indica a la JVM el tipo de dato que debe imprimir. Por ejemplo, podría indicarle imprimir una cadena de caracteres o un número de punto flotante. He aquí una lista parcial de caracteres de conversión:

- `s` Despliega una cadena de caracteres.
- `d` Despliega un entero decimal (un `int` o un `long`).
- `f` Despliega un número de punto flotante (`float` o `double`) con un dígito a la derecha del punto decimal.
- `e` Despliega un número de punto flotante (`float` o `double`) en notación científica.

Al explicar cada parte del especificador de formato (carácter de conversión, precisión, anchura y banderas) se presentan pequeños ejemplos que ilustran su sintaxis y semántica. Una vez finalizadas las explicaciones se muestra un ejemplo con un programa. Observe este fragmento de código y su salida asociada:

```
System.out.printf("Planeta: %s\n", "Neptuno");
System.out.printf("Número de lunas: %d\n", 13);
System.out.printf("Periodo orbital (en años terrestres): %f\n", 164.79);
System.out.printf(
  "Distancia promedio del sol (en km): %e\n", 4498252900.0);
```

Salida:

```
Planeta: Neptuno
Número de lunas: 13
Periodo orbital (en años terrestres): 164.790000
Distancia promedio del sol (en km): 4.498253e+09
```

Los especificadores de conversión `f` y `e` imprimen seis dígitos por omisión.

Observe que por omisión, los especificadores `f` y `e` generan seis dígitos a la derecha del punto decimal.

Precisión y anchura

La parte de precisión de un especificador de formato trabaja en conjunción con los caracteres de conversión `f` y `e`; esto es, trabaja con datos de punto flotante. Especifica el número de dígitos a ser impresos a la derecha del punto decimal. Se hará referencia a estos dígitos como los dígitos fraccionales. Si los datos tienen más dígitos fraccionales que el valor de precisión, se realiza un redondeo. Si los datos tienen menos dígitos fraccionales que los del valor de precisión, entonces se agregan ceros a la derecha para que el valor impreso tenga el número específico de dígitos fraccionales.

La parte de anchura del especificador de formato establece el número mínimo de caracteres mínimos que habrán de imprimirse. Si el dato contiene más del número de caracteres especificados, entonces todos se imprimen. Si el dato contiene menos caracteres del número especificado, entonces se agregan espacios. Por omisión, los valores de salida se alinean a la derecha, entonces cuando se adicionan espacios se agregan a la izquierda.

Observe este fragmento de código y su salida asociada:

```
System.out.printf("Las vacas son %6s\n", "agradables");
System.out.printf("Pero los perros %2s\n", "mandan");
System.out.printf("PI = %7.4f\n", Math.PI);
```

Salida

```
Las vacas son agradables
Pero los perros mandan
PI =  3.1416
```

6 caracteres

7 caracteres

En la tercera sentencia anterior, observe el especificador %7.4f. Es fácil confundirse con 7.4. Parece estar diciendo: "siete lugares a la izquierda del punto decimal punto y cuatro lugares a la derecha del punto decimal", pero en realidad está diciendo: "siete espacios en total, con cuatro lugares a la derecha del punto decimal". Y no hay que olvidar que el punto decimal se considera como uno de esos siete espacios en total. El valor `Math.PI` es 3.141592653589793, y cuando se imprime con cuatro lugares a la derecha del punto decimal, se redondea a 3.1416.

Banderas

A continuación se presenta, a manera de recordatorio, la sintaxis para un especificador de formato:

%[*banderas*][*anchura*][.*precisión*]*carácter de conversión*

Se han descrito las partes de la conversión, precisión y la anchura de un especificador de formato. Es tiempo de hablar acerca de las banderas. Las banderas nos permiten agregar características adicionales de formato, un carácter de bandera para cada característica de formato. He aquí una lista parcial de los caracteres de bandera:

- `-` Despliega el valor impreso utilizando la alineación a la izquierda.
- `0` Si un dato numérico contiene menos caracteres que el valor de anchura del especificador, entonces agrega ceros al valor impreso (despliega ceros a la derecha del número).
- `,` Despliega un dato numérico con separadores de agrupación locales específicos. En Estados Unidos, esto significa que se insertan comas entre cada grupo de tres dígitos a la izquierda del punto decimal.
- `(` Despliega un dato de número negativo utilizando paréntesis en lugar del signo menos. La utilización de paréntesis para números negativos es una práctica común en el campo de la contabilidad.

Ahora se verá cómo funcionan los especificadores de formato en el contexto de un programa completo. Analice el programa ReporteDePresupuesto de la figura 5.10. Observe que se utiliza el mismo formato de cadena para imprimir los encabezados de columna y las columnas bajo líneas, y cómo la cadena de formato es almacenada en una constante llamada `CAD_FM_ENCABEZADO`. Si se utiliza una cadena de formato en más de un lugar, es una buena idea guardarla en una constante nombrada y utilizarla en las sentencias `printf`. Al guardar la cadena de formato en un lugar común (una constante numérica) se asegura la consistencia y se facilita la actualización del formato de cadenas de caracteres en el futuro.

En el programa ReporteDePresupuesto, observe el signo menos en las cadenas de formato `CAD_FMT_ENCABEZADO` y `CAD_FMT_DATOS`. Esto alinea a la izquierda los datos de la primera columna. Observe las comas en la cadena de formato `CAD_FMT_DATOS`, que causan que aparezcan los caracteres locales específicos de separación (comas en Estados Unidos) entre cada grupo de tres dígitos a la izquierda del punto decimal. Observe el paréntesis a la izquierda en la cadena de formato `CAD_FMT_DATOS` que causa que los números negativos utilicen paréntesis en lugar de un signo negativo.

5.8 Resolución de problemas con números aleatorios (opcional)

En esta sección se mostrará cómo generar variables aleatorias que tengan distribuciones de probabilidad distintas a la distribución uniforme de 0.0 a 1.0 asumidas en una simple llamada al método `Math.random`.

Utilización de `Math.random` para generar números aleatorios con otras distribuciones de probabilidad

Como se indica en la figura 5.2, sección 5.3, cuando se requiere de un número aleatorio, se puede utilizar el método `Math.random` para generar uno. Suponga que se desea un número aleatorio de un rango que sea diferente del que existe entre 0.0 y 1.0. Como se realizó en la inicialización de `numeroGanador` de la figura 5.5, se puede expandir el rango a cualquier valor máximo multiplicando el número generado de manera aleatoria por el valor máximo deseado. Se puede modificar el rango también mediante la suma o la resta de una constante. Por ejemplo, suponga que desea obtener un número que se encuentre uniformemente distribuido en el rango entre −5.0 y +15.0. En lugar de utilizar el simple método `Math.random()`, haga lo siguiente:

```
(20.0 * Math.Random()) – 5.0.
```

Es posible manipular los números producidos por `Math.random()` para obtener el tipo de distribución que se desee. Por ejemplo, se puede generar cualquier distribución mostrada en la figura 5.11.

```
/*************************************************************
* ReporteDePresupuesto.java
* Dean & Dean
*
* Este programa genera un reporte de presupuesto.
*************************************************************/

public class ReporteDePresupuesto
{
  public static void main(String[] args)
  {
    final String CAD_FMT_ENCABEZADO = "%-25s%13s%13s%15s\n";
    final String CAD_FMT_DATOS = "%-25s%,13.2f%,13.2f%(,15.2f\n";
    double actual1 = 1149.999; // cantidad gastada en la 1era cuenta
    double presup1 = 1400;     // presupuesto para la 1era cuenta
    double actual2 = 2100.111; // cantidad gastada en la 2da cuenta
    double presup2 = 2000;     // presupuesto para la 2da cuenta
    double remanente1, remanente2; // cantidades no gastadas

    System.out.printf(CAD_FMT_ENCABEZADO,
      "Cuenta", "Actual", "Presupuesto", "Remanente");
    System.out.printf(CAD_FMT_ENCABEZADO,
      "-------", "------", "------", "---------");

    remanente1 = presup1 - actual1 ;
    System.out.printf(CAD_FMT_DATOS,
      "Artículos de oficina", actual1, presup1, remanente1);
    remanente2 = presup2 - actual2;
    System.out.printf(CAD_FMT_DATOS,
      "Fotocopiado", actual2, presup2, remanente2);

    System.out.printf(
      "\nRemanente total: $%(,.2f\n", remanente1 + remanente2);
  } // fin del main
} // fin de la clase ReporteDePresupuesto
```

justificación a la izquierda

paréntesis para negativos, coma para separadores de grupo

Salida:

```
Cuenta                        Actual    Presupuesto        Remanente
-------                       ------    -----------        ---------
Artículos de oficina       1,150.00       1,400.00           250.00
Fotocopiado                2,100.11       2,000.00         (100.11)

Remanente total: $149.89
```

Figura 5.10 Programa ReporteDePresupuesto y su salida.

Ahora se analizará cómo generar estos cinco tipos de números aleatorios a partir de `Math.random`.

1. El primer tipo (una distribución continua uniforme) es sencillo. Para obtener un valor para un número aleatorio, `x`, uniformemente distribuido en el intervalo entre cero y la unidad ($0.0 \leq x < 1.0$), utilice una sentencia como la siguiente:

   ```
   double r1 = Math.random();
   ```

   Este primer tipo de número aleatorio es la base de todos los demás tipos de números aleatorios.

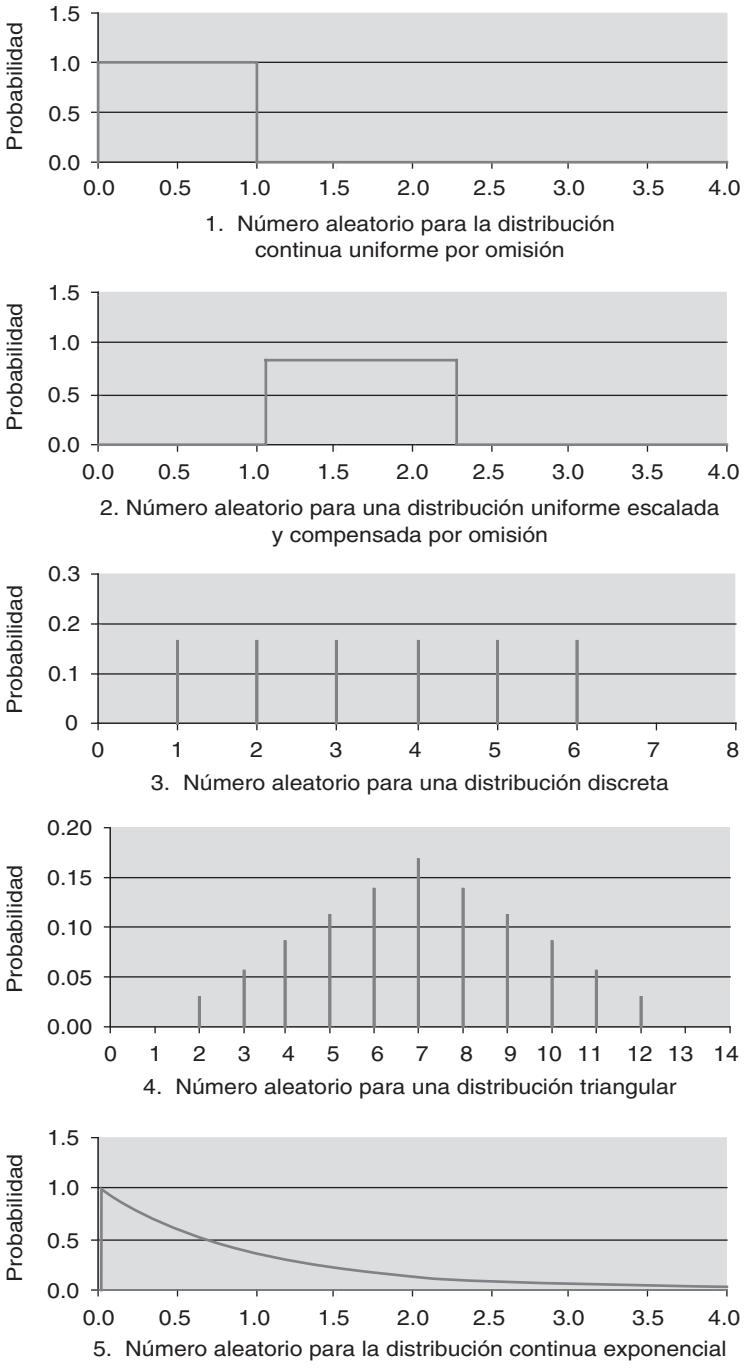


Figura 5.11 Tipos importantes de distribuciones numéricas aleatorias.

2. Para el segundo tipo (una distribución continua uniforme, compensada y expandida) se deben tener valores máximos y mínimos, por ejemplo:

```
double minReal = 1.07; // metros para el adulto humano más pequeño
double maxReal = 2.28; // metros para el adulto humano más alto
```

Después se aumenta y se expande el número aleatorio mediante la utilización de una sentencia como la siguiente:

```
double r2 = minReal + Math.random() * (maxReal - minReal);
```

3. Para el tercer tipo (una distribución discreta uniforme) se crean versiones enteras de los límites, por ejemplo:

```
int min = 1;     // el menor número de puntos en un dado
int max = 6;     // el mayor número en un dado
```

Después se mueve y se expande el número aleatorio básico; tal y como se hace en el segundo tipo:

```
double r3 = min + (int) (Math.random() * (max - min + 1));
```

Esta vez se debe recordar que la resta de enteros produce una distancia que es un número menor que los enteros en el rango (6 menos 1 igual a 5, no a 6), por lo que se tiene que agregar 1 a la diferencia como en el siguiente caso: `max – min + 1`. El número `double` devuelto por `Math.random` promueve todo automáticamente a `double`, por lo que el rango corrido y expandido está entre 1.0 y 6.99999. La selección aleatoria da un peso igual a cada uno de los seis intervalos encima de los enteros de interés (1, 2, 3, 4, 5 y 6). Las funciones de conversión (`int`) eliminan las fracciones.

4. Para el cuarto tipo (una distribución discreta triangular), al principio se podría pensar que sólo se puede utilizar el tercer tipo con min = 2 y max = 12, pero no es así. Se generarían únicamente tantos 2 y 12 como 7, pero ¡la oportunidad de obtener un 7 es de hecho seis veces más alta que obtener ya sea un 2 o un 12! La forma más sencilla de obtener la respuesta correcta es llamando a `Math.random` dos veces, y agregar los resultados:

```
int dadoDos = r3 + r3;
```

5. El quinto tipo de distribución (una distribución exponencial continua) se ha incluido porque se utiliza en modelos de muchos fenómenos importantes del mundo real, como los siguientes:
   - El tiempo intermedio de llegada de automóviles a un semáforo aislado.
   - El tiempo entre llamadas telefónicas no frecuentes.
   - El tiempo entre emisiones radiactivas de un átomo no estable.
   - El tiempo para que falle una pieza de maquinaria.
   - El tiempo para que falle el dispositivo de un semiconductor.

   Para generar una variable aleatoria con una distribución exponencial continua, utilice una sentencia como la siguiente:

```
double r5 = -Math.log(1.0 – Math.Random()) * tiempoPromedioEntreEventos;
```

El logaritmo de cero es –infinito, pero eso nunca ocurre porque `Math.random` nunca genera un número tan alto como `1.0`, por lo que (`1.0` – `Math.random()`) nunca es menor a cero.

Utilización de la clase Random

Aunque es posible obtener cualquier clase de distribución de `Math.random` no siempre es sencillo. Por ejemplo, el algoritmo que se requiere para convertir la distribución uniforme de `Math.random` en una de Gauss (curva de campana) es algo complicado; por lo que sería adecuado tener algunos métodos preconstruidos que generen de manera inmediata números aleatorios para ésta y otras distribuciones. La clase `Random` en el paquete `java.util` nos ayuda a lograr esto. He aquí los encabezados API de algunos de los métodos de la clase `Random`:

Utilice el recurso que mejor se ajuste.

```
public double nextDouble()
public int nextInt()
public int nextInt(int n)
public boolean nextBoolean()
public double nextGaussian()
```

El método `nextDouble` hace esencialmente lo mismo que `Math.random`. Este tipo de distribución aparece en la gráfica superior de la figura 5.11. El método `nextInt` sin parámetros genera enteros distribuidos uniformemente en el rango completo de los enteros, esto es de −2147483648 a +2147483647, incluidos. El método `nextInt` con un parámetro genera enteros aleatorios uniformemente de cero a uno menos el valor del parámetro. Esta distribución es casi como la que aparece en la tercera gráfica de la fi-

gura 5.11 para el caso especial donde n = 7, excepto que también se permite el cero. El método `nextBoolean` genera valores aleatorios de `true` o `false` (verdadero o falso). El método `nextGaussian` genera un valor tipo `double` de una distribución con una media de 0.0 y una desviación estándar de 1.0.

Observe que los métodos de la clase `Random` no tienen el modificador `static`, por lo que no son métodos de clase, y no se puede utilizar el nombre de la clase `Random` para acceder a esos métodos. Primero se debe crear un objeto, y después utilizar el nombre de ese objeto para acceder a esos métodos. Para crear un objeto a partir de cualquier clase diferente a la clase `String`, se requiere llamar a un *constructor*. Un constructor es un tipo de método especial que crea e inicializa objetos. Para llamar a un constructor, se requiere especificar la palabra reservada de Java `new`, el nombre del constructor, y después una lista de argumentos encerrados entre paréntesis. Por ejemplo, observe la llamada al constructor de `Random` en la figura 5.12. Observe que no se pasan argumentos al constructor, por lo que los paréntesis en la llamada están vacíos. Observe también que la llamada al constructor de `Random` es asignada a una variable de referencia llamada `aleatorio`. El objeto `aleatorio` posteriormente genera dos números aleatorios mediante la llamada a los métodos `nextInt` y `nextGaussian`. Debido al argumento `INTEGER.MAX_VALUE`, el método `nextInt` genera un número aleatorio entre cero y uno menos que el valor entero máximo. El método `nextGaussian` genera un número aleatorio tomado de la distribución gaussiana con una media de 5.0 y una desviación estándar de 0.8.

Utilización de una semilla fija para una secuencia de números aleatorios

Se puede llamar al constructor `Random` sin argumentos tal como se muestra en la figura 5.12 y se puede llamar también con un argumento, donde el argumento es una *semilla*. Una semilla proporciona un punto de inicio para el estado interno del generador de números aleatorios. Suponiendo que se cambia el cuerpo del método `main` en la figura 5.12 con el siguiente código:

```
Random aleatorio = new Random(123);

System.out.println(5.0 + 0.8 * random.nextGaussian());
System.out.println(5.0 + 0.8 * random.nextGaussian());
System.out.println(5.0 + 0.8 * random.nextGaussian());
```

```
/**********************************************************************
* PruebaAleatoria.java
* Dean & Dean
*
* Este programa ejecuta métodos de la clase Random.
**********************************************************************/

import java.util.Random;

public class PruebaAleatoria
{
  public static void main(String[] args)
  {
    Random aleatorio = new Random();

    System.out.println(aleatorio.nextInt(Integer.MAX_VALUE));
    System.out.println(5.0 + 0.8 * aleatorio.nextGaussian());
  } // fin del main
} // fin de la clase PruebaAleatoria
```

Para invocar un constructor de objetos explícito se usa **new**

Sesión muestra:

```
1842579217
4.242694469045554
```

Figura 5.12 Programa PruebaAleatoria que utiliza la clase `Random` para generar números aleatorios de diferentes distribuciones.

Ahora, si se corre el programa, se obtendrá lo siguiente:

Sesión muestra:

```
3.8495605526872745
5.507356060142144
5.1808496102657315
```

Si se ejecuta el programa una y otra vez, se obtendrán exactamente los mismos tres números "aleatorios" cada vez. La semilla 123 establece un punto de inicio, y esto determina precisamente la secuencia "aleatoria". Si se coloca una semilla diferente se obtiene una secuencia diferente, pero esa secuencia siempre será la misma mientras se mantenga esa semilla. Ahora ya conoce por qué los métodos de la clase `Random` no son métodos de clase. Se requiere que un objeto los llame porque esos métodos requieren algún tipo de información de la que contiene el objeto, la semilla y la posición actual en la secuencia del número aleatorio.

Cuando pruebe, arregle sus números aleatorios.

Puede utilizar la naturaleza determinística del generador semilla de números aleatorios para hacer su vida más sencilla cuando esté desarrollando y depurando programas que utilicen números aleatorios.

Si no se utiliza un generador semilla de números aleatorios en el momento en que un programa genera uno de éstos, lo que resulta sería una sorpresa porque ¡es un aleatorio! Este resultado no predecible puede ser bastante frustrante cuando se está tratando de desarrollar y probar un programa que utilice números aleatorios, porque cada prueba ejecutada produce diferentes valores numéricos. Durante el desarrollo y prueba lo que se desearía es un conjunto fijo de números "aleatorios", que parecería ser exactamente el mismo cada vez que se vuelve a correr el programa que se está probando.

Para establecer un conjunto de pruebas de números aleatorios fijo, se podría escribir un sencillo programa que imprima un conjunto particular de números aleatorios. Se podrían copiar estos números particulares en sentencias de asignación en el programa, esto es, código duro en el programa para desarrollo y prueba. Posteriormente, después de que el programa ha sido probado y verificado, se podría reemplazar cada "número aleatorio" de código duro por un generador de números aleatorios que genere números diferentes cada vez que se invoque.

Pero la clase `Random` proporciona una forma más elegante de desarrollar programas que tengan variables aleatorias. Durante el desarrollo, utilice el constructor de un parámetro con una semilla fija para producir exactamente la misma secuencia de números distribuidos de forma aleatoria cada vez que se ejecute el programa. Posteriormente, cuando se solucionan todos los errores, basta con borrar el número semilla desde el constructor `Random` en la sentencia de inicialización al principio del código, y —*voilà*— el generador de números aleatorios produce números completamente diferentes de ahí en adelante.

5.9 Apartado GUI: diseño de imágenes, líneas, rectángulos y óvalos en applets de Java (opcional)

En esta sección se muestra cómo desplegar imágenes, líneas, rectángulos y óvalos en una ventana GUI. La forma fácil de hacer esto es llamando a los métodos en la clase `Graphics` en Java desde el interior del applet de Java. Como podrá recordar del capítulo 1, un applet es un programa de Java que se inserta en una página Web. Este programa se ejecuta llamándolo en el código HTML de la página Web (HTML es el lenguaje base para la mayoría de las páginas Web). Se puede ejecutar la página Web cargándola en un navegador de Internet.

Archivos de imagen

Java puede manipular muchos tipos de imágenes. Algunas imágenes son íconos estilizados, otras son fotografías digitalizadas. Por ejemplo, suponga que tiene una fotografía digitalizada de un miembro de la familia, como la de Max, el sobrino de John, el autor, que se muestra en la figura 5.13.

Suponga que esta fotografía se almacena en un archivo llamado `huracanes.jpg`. "Huracanes" es el nombre del equipo de fútbol soccer de Max. La extensión ".jpg" es la abreviatura de JPEG, que significa Joint Photographic Experts Group (Unión de Grupo de Expertos Fotográficos). Los archivos con

Figura 5.13 Una imagen típica almacenada en un archivo `.jpg`.

esta extensión deben adecuarse a los estándares JPEG para comprensión digital de imágenes fotográficas. Las representaciones exactas de imágenes dibujadas sencillas por lo común se almacenan en archivos con extensión ".gif". GIF significa Graphics Interchange Format (formato de intercambio gráfico). Por simplicidad se asumirá que el archivo de imagen está en el directorio actual: el mismo directorio donde se guardan los programas Java que la desplegarán.

Antes de que se pueda utilizar una imagen en un programa se requiere saber qué tan grande es, su anchura y altura, en términos de números de *pixeles*. Los pixeles son los diminutos puntos de color que una computadora utiliza para desplegar algo en una pantalla. La figura 5.14 contiene un programa que se puede utilizar para determinar la anchura y altura del contenido de un archivo de imagen en pixeles. El programa importa la clase `Scanner` de Java para traer la captura del nombre del archivo, e importa la clase de Java `ImageIcon` para leer el archivo de imagen y determinar las propiedades de la imagen. Después de solicitar al usuario el nombre del archivo de la imagen, el programa utiliza ese nombre para crear un objeto llamado `icono`, el cual maneja la transferencia de información a partir del archivo de imagen, como el objeto `stdIn` que maneja transferencia de información desde el teclado. Los métodos `getWidth` y `getHeight` devuelven la anchura y altura de la imagen en pixeles. Esto da el tamaño por omisión del área requerida para desplegar la imagen en una pantalla de computadora.

Métodos de clase `Graphics`

La clase API de Java llamada `Graphics` contiene muchos métodos para desplegar imágenes y figuras geométricas. La figura 5.15 presenta encabezados API y descripciones para algunos de estos métodos. Observe que estos encabezados no tienen modificadores `static`. Esto significa que se debe utilizar un objeto de la clase `Graphics` para llamar a todos estos métodos, tal como se hace cuando se utiliza un objeto `String` para llamar a los métodos de la clase `String`. Este objeto `Graphics` contiene una referencia a la ventana en la cual se dibujan los objetos, y contiene otra información necesaria, como la posición actual de dicha ventana en la pantalla de la computadora, color y tipo de letra actuales.

La mayoría de los métodos de la figura 5.15 tienen pares de parámetros (como `int x, int y`) que indican las coordenadas `x` y `y` en una imagen o ventana. Estas coordenadas siempre se miden en pixeles. La coordenada `x` es el número de pixeles en el lado izquierdo de la imagen o ventana. La coordenada `y` es el número de pixeles desde la parte alta de la imagen o ventana. La coordenada `x` es lo que probable-

```
/*************************************************************
* InfoImagen.java
* Dean & Dean
*
* Este programa proporciona el ancho y largo de una imagen.
*************************************************************/

import java.util.Scanner;
import javax.swing.ImageIcon;

public class InfoImagen
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    ImageIcon icono;

    System.out.print("Introduzca el nombre de la imagen: ");
    icon = new ImageIcon(stdIn.nextLine());
    System.out.println("anchura de la imagen = " + icono.getIconWidth());
    System.out.println("altura de la imagen = " + icono.getIconHeight());
  }
} // fin de InfoImagen
```

Sesión muestra:

```
Enter image filename: hurricanes.jpg
image width = 640
image height = 427
```

Figura 5.14 Programa InfoImagen que determina la anchura y la altura de una imagen en un archivo de imagen.

mente se esperaría, pero la coordenada `y` podría parecer rara porque normalmente se piensa que la `y` crece hacia arriba. Sin embargo, en una pantalla de cómputo, la `y` se incrementa hacia abajo porque una computadora pinta algo en una pantalla, pinta la línea superior primero, la línea siguiente después, y así continúa, en una secuencia de arriba hacia abajo.

En un encabezado de método, los nombres de los parámetros de las coordenadas `x` y `y` a veces utilizan sufijos numéricos para distinguir un punto de otro. En los parámetros del método `drawImage` hay también un prefijo de carácter antes de cada identificador de coordenada. El prefijo d significa "destino", lo que significa posición en la ventana del monitor. El prefijo s significa "fuente" (*source*, en inglés), lo que significa la posición en la imagen original.

A veces los números en una secuencia de parámetros no son las coordenadas `x` y `y`. En lugar de ello, se tiene anchura y altura, que son las diferencias de coordenadas. Éste es el caso de los métodos `drawRect` y `fillOval`. Es fácil olvidar cuál técnica utiliza un método en particular para especificar la anchura y la altura. ¿Especifica posiciones de las esquinas superior izquierda e inferior derecha, o especifica las posiciones de las esquinas superior izquierda y después anchura y altura? Tenga cuidado, éste es uno de los orígenes de los errores de programación GUI.

El método `drawImage` copia una porción rectangular de la imagen fuente y pega una versión expandida o contraída de ésta en un rectángulo especificado en la ventana destino. El primer parámetro es una referencia a la imagen fuente; el segundo y tercero son las coordenadas destino (ventana de despliegue) de la esquina superior izquierda de la parte copiada de la imagen; el cuarto y quinto son las coordenadas destino de la esquina inferior derecha de la parte copiada de la imagen. Los parámetros seis y siete representan las coordenadas de la esquina superior izquierda de la parte de la imagen fuente a ser co-

```
public boolean drawImage(Image img,
  int dx1, int dy1, int dx2, int dy2,
  int sx1, int sy1, int sx2, int sy2,
  ImageObserver observer)
```
Selecciona lo que sea que esté entre los pixeles `sx1` y `sx2` a la derecha del margen izquierdo de la imagen fuente y entre `sy1` y `sy2` pixeles bajo la parte alta de la imagen fuente. Realiza la escala de esta selección como se requiera para ajustar entre `dx1` y `dx2` pixeles a la derecha del margen izquierdo de la ventana destino y entre `dy1` y `dy2` bajo la parte superior de la ventana destino.

```
public void setColor(Color c)
```
Establece un color de pintado específico.

```
public void drawRect(int x, int y, int anchura, int altura)
```
Dibuja las líneas de un rectángulo, cuya esquina superior izquierda es `x` pixeles a la derecha de la parte izquierda de la ventana y `y` pixeles bajo la parte superior de la ventana. Utiliza el color establecido más recientemente.

```
public void drawLine(int x1, int y1, int x2, int y2)
```
Dibuja una línea recta desde un punto `x1` pixeles a la derecha de la parte izquierda de la ventana y `y1` pixeles bajo la parte superior de la ventana a `x2` pixeles de la derecha de la parte izquierda de la ventana y `y2` pixeles bajo la parte superior de la ventana. Utiliza el color establecido más recientemente.

```
public void fillOval(int x, int y, int anchura, int altura)
```
Llena una elipse limitada por el rectángulo especificado con el color establecido más recientemente.

```
public void drawString(String texto, int x, int y)
```
Imprime el texto especificado en una línea que empieza `x` pixeles a la derecha del lado izquierdo de la ventana y `y` pixeles debajo de la parte superior de la ventana. Utiliza el color establecido más recientemente.

Figura 5.15 Métodos seleccionados de la clase API de Java `Graphics`.

piada; mientras que el ocho y el nueve son las coordenadas de la esquina inferior derecha de la parte de la imagen a ser copiada. El último parámetro permite al método enviar la información actual.

El método `setColor` establece el color a ser utilizado en operaciones subsiguientes que dibujen líneas o figuras geométricas o que escriban texto. A este método se le pasa un argumento, tal como `Color.BLUE`, que identifica una o varias constantes nombradas definidas en la clase API de Java, `Color`. La mayoría de estos nombres son bastantes obvios, por lo que de manera práctica, sólo se requiere adivinar y ver qué sucede.

El método `drawRect` dibuja los márgenes de un rectángulo con el color utilizado más recientemente. Los parámetros uno y dos son las coordenadas de la esquina superior izquierda del rectángulo en la ventana de despliegue. Los parámetros tres y cuatro representan el largo y ancho del rectángulo.

El método `drawLine` dibuja una línea recta entre dos puntos específicos, utilizando el color establecido más recientemente. Los parámetros son como los utilizados en `drawRect`: el primero y segundo son las coordenadas del punto de inicio. El tercero y cuarto son las coordenadas del punto final.

El método `fillOval` dibuja una elipse en el rectángulo especificado y lo rellena con el color más recientemente establecido. Los parámetros son como los del método `drawRect`: el primer y segundo parámetros son las coordenadas de la esquina superior izquierda del rectángulo encerrado en la ventana de despliegue; el tercero y cuarto son el ancho y largo de dicha figura, y el ancho y largo del óvalo mismo.

El método `drawString` imprime texto en la posición especificada utilizando el color establecido más recientemente. El primer parámetro es la cadena de texto a ser impresa. Los parámetros dos y tres son las coordenadas de la esquina superior izquierda de la cadena de texto.

Utilización de métodos gráficos en un applet de Java

La figura 5.16 proporciona un ejemplo de cómo se pueden emplear los métodos gráficos en la figura 5.15 para manipular una imagen fotográfica y agregar nuestras propias líneas, figuras y texto a la misma. La ventana desplegada en la figura 5.16 tiene 640 pixeles de ancho y 640 de largo.

La figura 5.17 muestra el código de Java necesario para realizar el despliegue que se muestra en la figura 5.16. Todo el código se encuentra en el cuerpo del método denominado `paint`. El parámetro *g* del método `paint` se utiliza como prefijo en cada llamada al método `paint`. Se refiere al objeto `Graphics` que realiza la operación de pintado. Dentro del método `paint`, la primera sentencia trae una referencia a la imagen fuente. La siguiente sentencia que llama al método `drawImage` emplea una referencia a la imagen fuente para acceder a esta imagen.

El método `drawImage` reduce la imagen `huracanes.jpg` de la figura 5.13 a dos tercios de su tamaño original y la coloca en la parte de la esquina superior izquierda de la ventana de despliegue. El método `setColor` establece el color actual a azul. El método `drawRect` dibuja un cuadrado alrededor de un área de interés. Después, las cuatro llamadas al método `drawLine` dibujan las cuatro líneas rectas en las esquinas en donde irá el alargamiento triple del área de interés en esta ubicación. Otra llamada al método

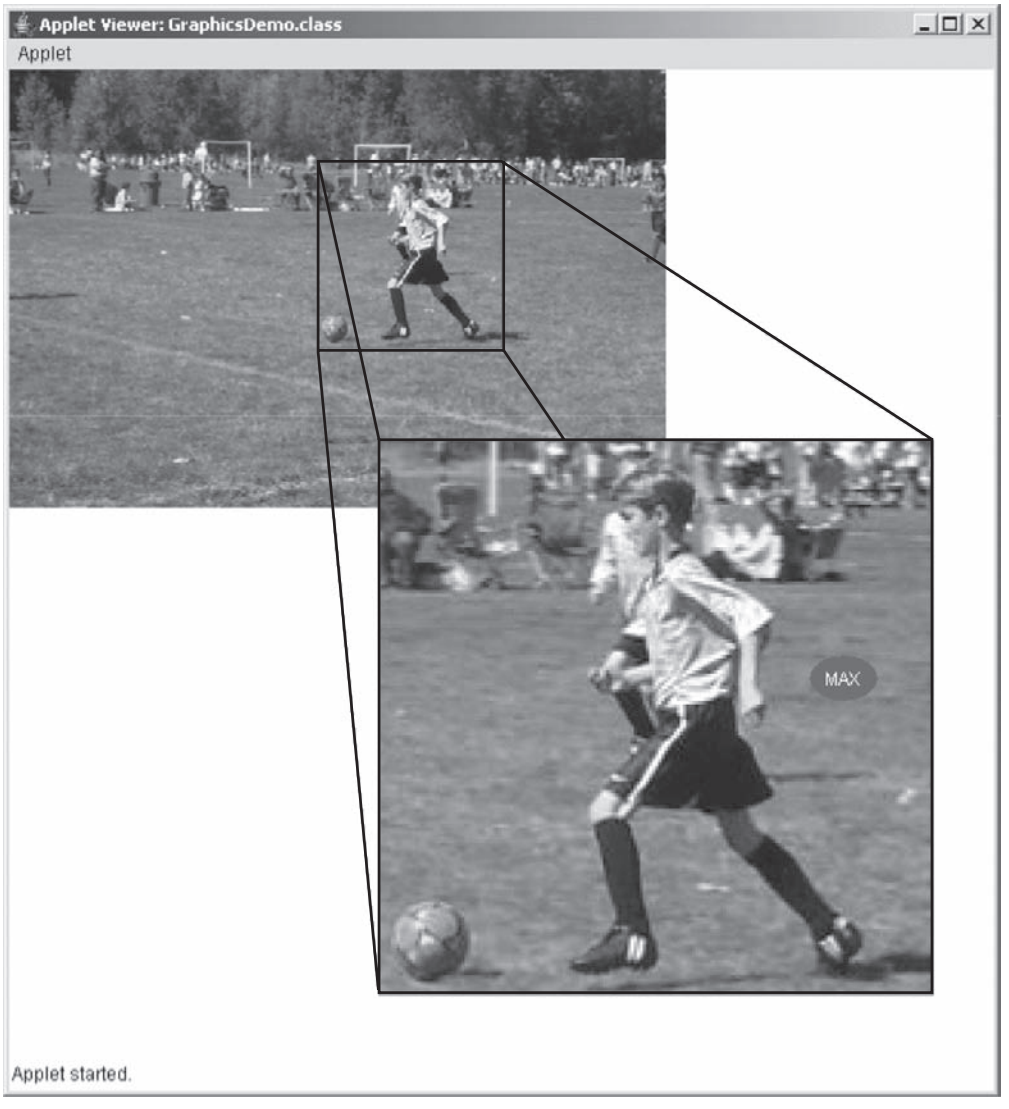


Figura 5.16 Salida producida por programa de figura la 5.17.

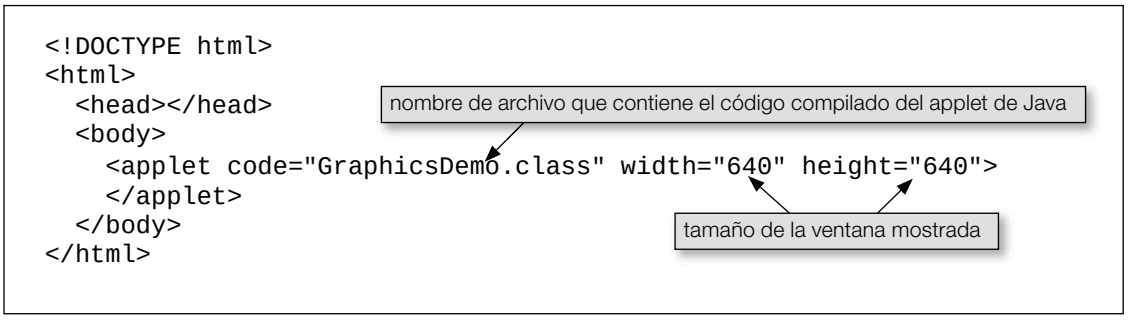
```
/*************************************************************************
* GraphicsDemo.java
* Dean & Dean
*
* Este programa define un applet de Java que despliega una imagen y gráficas.
*************************************************************************/

import java.awt.*;       // para clases Graphics, Image and Color
import java.applet.Applet;

public class GraphicsDemo extends Applet
{
  public void paint (Graphics g)
  {
    Image image =
      this.getImage(getDocumentBase(),"huracanes.jpg");

    // despliega la imagen completa más reducida en la esquina superior
    // izquierda de la ventana
    g.drawImage(image, 0, 0, 427, 284,     // destino, supIzq, infDer
      0, 0, 640, 427, this);               // fuente supizq, infDer

    // establece el color de todas las líneas a ser dibujadas
    g.setColor(Color.BLUE);

    // dibuja un rectángulo alrededor de la región a ser expandida
    g.drawRect(200, 60, 120, 120);         // supIzq, ancho & largo

    // Dibuja las líneas entre las esquinas de los rectángulos
    g.drawLine(200, 60, 240, 240);         // superior izquierda
    g.drawLine(320, 60, 600, 240);         // superior derecha
    g.drawLine(200, 180, 240, 600);        // inferior izquierda
    g.drawLine(320, 180, 600, 600);        // inferior derecha

    // despliega la parte expandida de la imagen original
    g.drawImage(image, 240, 240, 600, 600, // destino, supIzq, infDer
      300, 90, 480, 270, this);            // fuente supIzq, infDer

    // draw rectangle around expanded part of image
    g.drawRect(240, 240, 360, 360);        // supIzq, ancho & largo

    // crea un óvalo iluminado de azul (BLUE) y escribe sobre éste
    g.fillOval(520, 380, 45, 30);          // supIzq, ancho & largo
    g.setColor(Color.WHITE);               // cambia el color para el texto
    g.drawString("MAX", 530, 400);         // cadena y posición de inicio
  } // fin de paint
} // fin de la clase GraphicsDemo
```

La sentencia `extends Applet` insertada en el encabezado de la clase permite a esta clase "pedir prestado" el método `getImage` de la clase API de Java ya definida `Applet`.

Figura 5.17 Applet de Java GraphicsDemo que ejemplifica el uso de métodos listados en la figura 5.15. El applet produce la salida mostrada en la figura 5.16.

`drawImage` pega una versión alargada del área de interés en esta ubicación. Otra llamada a `drawRect` coloca un rectángulo alrededor de esta imagen alargada. Luego, el método `fillOval` pinta un óvalo azul en versión alargada, y, finalmente, el método de impresión `drawString` imprime "MAX" en el centro del óvalo azul. Observe cómo cada operación subsiguiente sobrescribe o cubre todas las operaciones previas.

El código de la figura 5.17 es muy corto. Por ejemplo, no incluye una definición del método `getImage`. Entonces, ¿cómo llamar a ese método? Como se descubrirá en el capítulo 12, Java permite a




```
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <applet code="GraphicsDemo.class" width="640" height="640">
    </applet>
  </body>
</html>
```

Figura 5.18 Código de un archivo HTML que corre el código GraphicsDemo de la figura 5.17.

cualquier clase que se defina pedir prestado los métodos de otra clase que ya haya sido definida. En particular, la clase `GraphicsDemo` definida en la figura 5.17 solicita el uso del método `getImage` de la clase API de Java `Applet`. Esto lo realiza mediante la inserción de la cláusula `extends Applet` en el encabezado de la clase. Por supuesto, el compilador debe saber dónde encontrar la clase `Applet`, para que el programa pueda importarla. También `importa` el paquete `java.awt` para proporcionar acceso a las clases `Graphics`, `Image` y `Color` utilizadas por las sentencias en el método `paint`.

Observe que la llamada al método `getImage` tiene la palabra `this` como prefijo, y que esa palabra también aparece en el último argumento en las dos llamadas al método `drawImage`. En el siguiente capítulo se explicará que el término especial `this` en Java se refiere a cualquier objeto que en un momento esté llamando al método actual de ejecución. En el programa GraphicsDemo, `this` se refiere al objeto que llama al método `paint`, y dicho objeto es una instancia de la clase `GraphicsDemo` definida en el código de la figura 5.17. Pero, ¿dónde está el código que crea un objeto `GraphicsDemo`?, ¿dónde está el código que trae una referencia al objeto asociado `Graphics`, y dónde está el código que llama al método `paint`? Está en un archivo separado…

Ejecución de un applet

¿Se dio cuenta de que no hay un método `main` en la figura 5.17? Un applet de Java es diferente de una aplicación Java, porque el applet no tiene un método `main`, no se puede ejecutar en la forma normal. Típicamente, se inserta en otro programa, de tal manera que su código se ejecuta cuando el otro programa lo llama. El propósito principal de un applet de Java es animar una página Web. Por lo que normalmente se llamará a los applets de Java desde programas HTML (*HyperText Markup Language* [Lenguaje de Marcado de Hipertexto]) que definen páginas Web. La figura 5.18 contiene un programa mínimo de HTML que llama al applet `GraphicsDemo` definido en la figura 5.17.

Observe que la parte de este código HTML que es específico al applet en particular, se encuentra en la quinta línea. Ésta identifica la versión compilada del applet, y especifica la anchura y altura en pixeles de la ventana que contendrá cualquier cosa que sea desplegada por el applet. Este código HTML se puede escribir con cualquier editor primitivo de texto, como Microsoft Notepad o vi de UNIX. Después, se salva en el mismo directorio del código que tiene la versión compilada del applet que maneja. Por ejemplo, en el directorio que contiene el archivo `graphicsDemo.class`. Cuando se salva, se le da un nombre con la extensión `html`, como `GraphicsDemo.html`.

Existen tres formas alternativas para correr un archivo HTML (y su applet de Java asociado):

1. Abrir un navegador como Microsoft Internet Explorer, navegar al directorio que contiene el archivo HTML, y dar doble clic en el nombre del archivo HTML.
2. Abrir el editor de comandos de Windows, moverse al directorio que contiene el archivo HTML, e introducir:
   appletviewer `graphicsDemo.html`
3. Seleccionar “Ejecutar un applet de Java” en su IDE (Integrated Development Environment) local, y seleccionar el archivo HTML deseado.

Resumen

- La documentación de Java Sun identifica la interfaz pública de todo el software API de Java. También proporciona una breve descripción de lo que hace y cómo utilizarlo. El paquete `java.lang` siempre está disponible.
- La clase `Math` proporciona métodos que permiten calcular potencias y raíces, máximos y mínimos, conversiones de ángulo y muchas funciones trigonométricas. La función `random` genera un número aleatorio cuya distribución es uniforme en el rango de 0.0 a 0.9 de manera repetitiva. Esta clase también proporciona valores de constantes nombradas para PI y E.
- Las clases envoltorio numéricas como `integer`, `Long`, `Float` y `double` contienen métodos de análisis sintáctico (*parsing*) tales como `parseInt` que permiten convertir representaciones de números en cadenas de caracteres en formato numérico. Las constantes nombradas `MIN_VALUE` y `MAX_VALUE` proporcionan los valores mínimo y máximo permitidos para los diferentes tipos de datos numéricos.
- La clase `Character` proporciona métodos que indican si un carácter es un espacio en blanco, un dígito o una letra, y, en caso de ser una letra, si está en mayúscula o minúscula. Otros métodos nos permiten realizar el cambio de minúscula a mayúscula y viceversa.
- El método `indexOf` de la clase `String` ayuda a encontrar la posición de un carácter en particular dentro de una cadena de texto. El método `substring` permite extraer una parte de una determinada cadena de texto. Los métodos `replaceAll` y `replaceFirst` realizan sustituciones dentro de una cadena de texto. Se puede realizar la conversión de mayúscula a minúscula y viceversa con los métodos `toLowerCase` y `toUpperCase`, respectivamente, y se puede utilizar el método `trim` para eliminar espacios en blanco de la parte final de una cadena de texto.
- El primer argumento en el método `System.out.printf` es una cadena de texto de formato, la cual permite utilizar un código especial para especificar el formato de salida de texto y números. Por ejemplo, para desplegar un número tipo `double` llamado `precio` como dólares y centavos con comas entre grupos de tres dígitos y un cero a la izquierda de los decimales para valores menores a $1.00, se podría escribir:

  ```
  System.out.printf("$%,04.2f\n", precio);
  ```
- Utilice la clase `Random` en el paquete `java.util` para obtener varias distribuciones de números aleatorios o exactamente la misma lista de números aleatorios cada vez que se corre un programa en particular.
- Utilice métodos de la clase `Graphics` del API de Java para desplegar imágenes fotográficas, figuras geométricas y texto en ventanas gráficas.
- Para ejecutar un applet de Java se requiere crear un archivo HTML que especifique el applet de Java, y cargar el archivo HTML dentro de un navegador Web.

Preguntas de revisión

§5.3 Clase `Math`

1. Dadas las siguientes declaraciones:

```
double diametro = 3.0;
double perimetro;
```

Proporcione una sentencia que asigne la longitud del perímetro de un círculo a la variable `perimetro`. Utilice la variable `diametro`.

2. ¿Cuál es el nombre de la clase que contiene los métodos `abs`, `min` y `round`?

a) Arithmetic
b) Math
c) Number

§5.4 Clases envoltorio para tipos primitivos

3. Proporcione una sentencia que asigne infinito positivo a una variable tipo `double` llamada `num`.

4. Proporcione una sentencia que convierta una cadena de caracteres llamada `s` a una `long` y asigne el resultado a una variable tipo `long` llamada `num`.

5. Proporcione una sentencia que convierta una variable tipo `int` llamada `num` en una cadena de caracteres y asigne el resultado a una variable tipo `String` llamada `numStr`. Utilice un método de una clase envoltorio.

§5.5 Clase `Character`

6. ¿Qué imprime el siguiente fragmento de código?

```
System.out.println(Character.isDigit('#'));
System.out.println(Character.isWhitespace('\t'));
System.out.println(Character.toLowerCase('B'));
```

§5.6 Métodos `String`

7. Dada esta declaración.[6]

```
String snyder = "Unirse. \nAprender las flores. \nIr tranquilo.";
```

Escriba una sentencia Java que encuentre el índice de la letra 'I' e imprima todo el contenido de la variable `snyder`, a partir de dicho índice. En otras palabras, que imprima `Ir tranquilo.`

§5.7 Salida formateada con el método `printf`

8. Escriba un formato de cadena que maneje y despliegue tres datos en tres columnas. La primera columna debe tener 20 espacios de ancho, y debe imprimir la cadena de caracteres alineada a la izquierda. La segunda columna debe tener una anchura de 10 espacios, y debe imprimir un entero alineado a la derecha. La tercera columna debe tener 16 espacios de ancho y debe imprimir un número de punto flotante alineado a la derecha, con formato científico con seis puntos decimales. La cadena de formato debe provocar que el cursor de la pantalla se mueva a la siguiente línea después de imprimir el tercer dato.

9. Proporcione un especificador de formato que maneje el despliegue de un dato de tipo punto flotante. Debe imprimir una versión redondeada del dato, sin lugares decimales. Debe insertar separadores de agrupamiento y utilizar paréntesis si el número es negativo.

§5.8 Resolución de problemas con números aleatorios (opcional)

10. Escriba una sentencia de Java que imprima un número aleatorio para el número total de puntos al lanzar un par de dados.

11. Escriba un programa que imprima cinco valores tipo `boolean` aleatorios con la semilla `123L`, y después despliegue dichos valores.

Ejercicios

1. [Después de §5.3] Escriba una sentencia que calcule e imprima la raíz cúbica de una variable tipo `double` llamada `numero`. [*Sugerencia:* busque un método apropiado en la documentación de Sun de la clase `Math`.]

2. [Después de §5.3] En cálculos de probabilidad, a menudo se requiere calcular el valor del factorial de algún número n. El factorial de un número `n` (designado n!) está dado por la fórmula:

$n! \leftarrow n * (n-1) * (n-2) * ... * 3 * 2 * 1.$

Cuando la n es un número muy largo, el cálculo toma mucho tiempo. Afortunadamente, hay una fórmula sencilla, llamada *Fórmula de Stirling*, que proporciona una muy buena aproximación a n!, cuando la n es larga. La fórmula de Stirling es:

$n! \approx (1 + 1/(12n -1)) * sqrt(2n\pi) * (n/E)^n$

El símbolo π es el radio del perímetro de un círculo a su diámetro, y el símbolo E es la base de los logaritmos naturales. El valor actual de n! es siempre ligeramente más pequeño que el valor dado por su fórmula. Para este ejercicio, escriba un fragmento de código en Java que implemente la fórmula de Stirling.

3. [Después de §5.3] Escriba un método `main` que solicite al usuario un ángulo θ, en grados, e imprima los valores de sen(θ), cos(θ) y tan(θ).

Sesión muestra:

```
Introduzca un ángulo en grados: 30
sen(deg) = 0.49999999999999994
cos(deg) = 0.8660254037844387
tan(deg) = 0.5773502691896257
```

[6] Gary Snyder, "For the Children", en *Turtle Island*. New Directions (1974).

4. [Después de §5.3] Proporcione una sentencia que imprima la longitud de la hipotenusa de un triángulo rectángulo cuya base está dada por la variable `base`, y cuya altura está dada por la variable `altura`. En la sentencia se debe hacer uso del método `hypot` de la clase `Math`. Para aprender acerca del método `hypot` consulte el sitio Web de Java Sun.
5. [Después de §5.3] Dado el log base-e, siempre es posible encontrar el log de cualquier otra base, con la fórmula $\log_{base}(x) = \log_e(x) / \log_e(base)$. Por ejemplo, un científico de cómputo podría estar interesado en el número de bits que son necesarios para expresar un entero positivo x dado en binario. En ese caso, el número total de bits requeridos es $\log_2(x)$, redondeado hasta el siguiente entero más alto. Escriba una sentencia de Java que 1) calcule el número de bits requeridos para almacenar el valor de la variable x y 2) que asigne ese valor calculado en una variable tipo `int` llamada `bits`.
6. [Después de §5.6] En el siguiente bosquejo de programa reemplace <*Insertar código aquí*> con su propio código. *Sugerencia:* utilice las variables que ya han sido declaradas por usted (`canciones`, `textoBuscado`, `indiceEncontrado` y `Cuenta`). El programa resultante debe solicitar al usuario una cadena de texto de búsqueda, y después despliega el número de ocurrencias de la cadena a buscar en una lista determinada de canciones. Estudie la sesión muestra.

```
import java.util.Scanner;

public class ContarOcurrenciasDeSubtexto
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String canciones =
      "1. Green Day - American Idiot\n" +
      "2. Jesus Jones - Right Here, Right Now\n" +
      "3. Indigo Girls - Closer to Fine\n" +
      "4. Peter Tosh - Equal Rights\n";

    String textoBuscado;    // texto a ser buscado
    int indiceEncontrado;   // posición en la que es encontrado el texto
    int cuenta = 0;        // núm de ocurrencias del texto

    System.out.print("Introduzca el texto a buscar: ");
    textoBuscado = stdIn.nextLine();

    <Insertar código aquí.>

    System.out.println("Número de ocurrencias de \"" +
      textoBuscado + "\": " + cuenta);
  } // fin del main
} // fin de la clase ContarOcurrenciasDeSubtexto
```

Sesión muestra:

```
Introduzca el texto a buscar: Right
Número de ocurrencias de "Right": 3
```

7. [Después de §5.6] En el siguiente bosquejo de programa, reemplace <*Insertar código aquí*> con su propio código. *Sugerencia:* utilice las variables que ya han sido declaradas por usted (`canciones`, `numCancion`, `indiceCancion`, `indiceFL` y `cancion`). El programa resultante debe solicitar al usuario un número de canción y después extraer el número más el resto de la línea de la cadena de la línea de una lista determinada de canciones. Estudie la sesión muestra. Se puede asumir que el usuario introduce un número de canción válido (no se requiere validación de entrada).

```
import java.util.Scanner;

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

```
    Scanner stdIn = new Scanner(System.in);
    String canciones =
      "1. Bow Wow - Fresh Azimiz\n" +
      "2. Weezer - Beverly Hills\n" +
      "3. Dave Matthews Band - Crash Into Me\n" +
      "4. Sheryl Crow - Leaving Las Vegas\n";

    String numCancion;  / Número de canción a ser buscada
    int indiceCancion;  // posición en la que es encontrada el número de
                        // canción
    int indiceFL;       // posición del carácter de fin de línea
    String cancion;     // la línea especificada

    System.out.print("Introduzca un número de canción: ");
    numCancion = stdIn.nextLine();

    <Insertar código aquí.>

    System.out.println(cancion);
  } // fin del main
} // fin de la clase ExtraerLinea
```

Sesión muestra:

```
Introduzca un número de canción: 3
3. Dave Matthews Band - Crash Into Me
```

8. [Después de §5.7] En el siguiente bosquejo de programa, reemplace los cuatro elementos <*Agregar código aquí*> de tal manera que el programa produzca la siguiente salida. Trate de imitar exactamente el formato de salida, aunque está bien si los anchos de columna varían ligeramente de los de las columnas mostradas aquí.

```
public class ReporteInventarioAutos
{
  public static void main(String[] args)
  {
    final String CADENA_FMT_ENCABEZADO = <agregar código aquí>;
    final String CADENA_FMT_DATO = <agregar código aquí>;
    String elemento1 = "Mazda RX-8";
    int cant1 = 10;
    double price1 = 27999.99;
    String elemento2 = "MINI Cooper";
    int cant2 = 100;
    double price2 = 23000.25;

    System.out.printf(CADENA_FMT_ENCABEZADO,
      "Elemento", "Cantidad", "Precio", "Valor");
    System.out.printf(CADENA_FMT_DATO,
      "-----", "-------", "------", "------");
    System.out.printf(CADENA_FMT_DATO, <agregar código aquí>);
    System.out.printf(CADENA_FMT_DATO, <agregar código aquí>);
  } // fin del main
} // fin de la clase ReporteInventarioAutos
```

Salida:

```
Elemento        Cantidad     Precio        Valor
-----          ---------     ------       ------
Mazda RX-8            10     28,000      280,000
MINI Cooper          100     23,000    2,300,025
```

9. [Después de §5.8] Proporcione una sentencia que utilice `Math.Random` para generar el número total de puntos en un par de dados lanzados.

Solución a las preguntas de revisión

1. `perímetro = Math.PI * diámetro;`
2. La clase que contiene los métodos `abs`, `min` y `round` es: b) `Math`.
3. `num = Double.POSITIVE _ INFINITY;`
4. `num = Long.parseLong(s);`
5. `numStr = Integer.toString(num);`
6. He aquí la salida del fragmento de código:

```
false
true
b
```

7. `System.out.println(snyder.substring(snyder.indexOf('I')));`
8. `"%-20s%10d%16.6e\n"`
   o
   `"%-20s%10d%16e\n"`
   (es correcto omitir el .6 porque el especificador de conversión e imprime seis lugares decimales por omisión).
9. `"%(,.0f"`
   o
   `"%,(.0f"`
   (el orden de la bandera especificadora de caracteres es irrelevante).
10. La sentencia que imprime el número total de puntos en un par de dados lanzados:

```
System.out.println(2 + (int) (6 * (Math.random())) +
  (int) (6 * (Math.random())));
```

11. El programa que imprime cinco valores tipo `boolean` aleatorios con semilla 123L:

```
import java.util.Random;

public class BooleanAleatorio
{
  public static void main(String[] args)
  {
    Random aleatorio = new Random(123L);

    for (int i=0; i<5; i++)
    {
      System.out.println(aleatorio.nextBoolean());
    }
  } // fin del main
} // fin de BooleanAleatorio
```

Los valores son:

```
true
false
true
false
false
```

CAPÍTULO 6

Programación orientada a objetos

Objetivos

- Entender qué es un objeto y cómo se relaciona con una clase.
- Aprender a encapsular y acceder a datos dentro de un objeto.
- Aprender a dividir programas en clases "controladoras" y "controladas" para crear un objeto de la clase controlada, y para darle a la controladora una referencia de ese objeto.
- Entender las diferencias entre los datos de un objeto y los datos que son locales a un método, y aprender a distinguir entre estas piezas de datos cuando ambos tengan el mismo nombre.
- Entender la inicialización implícita (valores por omisión) de varios tipos de variables.
- Aprender a rastrear un programa orientado a objetos.
- Aprender a utilizar un diagrama UML de clases.
- Hacer que un método devuelva un valor adecuado.
- Aprender a pasar los valores a los métodos.
- Escribir métodos que obtengan, asignen y prueben los valores de los datos de un objeto.
- De manera opcional, aprender a mejorar la velocidad y la seguridad de una simulación.

Relación de temas

6.1 Introducción

Como se señaló en el prólogo, este libro se escribió con cierta flexibilidad respecto al orden del contenido. Los lectores que deseen una primera introducción a la programación orientada a objetos (POO), tienen la opción de leer las secciones 6.1 a 6.8 después de completar el capítulo 3.

El capítulo 5 sirvió como puente entre los constructores básicos del lenguaje de programación (variables, operadores de asignación, sentencias condicionales, ciclos, etc.) y los conceptos de la POO. Se resaltó principalmente un aspecto de la POO: aprender a utilizar métodos preconstruidos. Se utilizaron métodos asociados a un objeto, como `substring` e `indexOf` para objetos de tipo string; asimismo, se utilizaron métodos asociados a una clase, como `abs` y `pow` de la clase `Math`. En este capítulo se aprenderá a ir más allá de la utilización de clases y métodos preconstruidos; se aprenderá a construir clases y métodos propios.

Como se ha visto, la POO facilita el trabajo con programas largos y facilitar el trabajo con programas largos es muy importante porque las computadoras utilizan programas muy largos. La tensión en el aprendizaje de la POO es que los primeros programas con POO que un estudiante puede entender son necesariamente pequeños y no pueden mostrar muy bien el poder de esta programación; pero alto, tome el estudio de este capítulo y de los que están por venir como una inversión. Al final del siguiente capítulo estará recogiendo algunos de los rendimientos de esta inversión.

Este capítulo inicia con una breve introducción a los términos básicos y conceptos de la POO. Después se estudia el diseño e implementación de un sencillo programa con POO. Normalmente, el diseño de la POO inicia con un sencillo diagrama de clases con el lenguaje de modelado de datos (UML, por sus siglas en inglés), que proporciona una descripción pictográfica de alto nivel de lo que se desea que el programa modele. Luego, el diseño de POO sigue los detalles del programa. Se mostrará cómo adaptar las técnicas de rastreo a un ambiente de POO. Se mostrará cómo especificar los detalles del método. En el capítulo anterior se vieron los métodos desde afuera: desde el punto de vista del usuario o *cliente*. Ahora se estudiarán los métodos desde adentro: desde el punto de vista del *servidor* o desde la implementación.

El capítulo finaliza con una sección opcional de resolución de problemas que es una introducción a la aplicación importante del área: la simulación de cómputo. Con la simulación de cómputo se pueden resolver problemas que son difíciles o imposibles de resolver a mano. Se describe una estrategia especial que permite mejorar en forma sustancial tanto la seguridad como la eficiencia de las simulaciones de cómputo.

6.2 Introducción a la programación orientada a objetos

Los lectores que deseen una primera introducción a la POO tienen la opción de leer esta sección después de completar la sección 1.3 (Desarrollo de programas) del capítulo 1.

Antes de la POO, la técnica estándar de programación era la *programación procedural*. Se denomina programación procedural porque en ella se destacan los procedimientos o tareas que resuelven un problema. Se piensa primero en lo que se quiere hacer: los procedimientos. En contraste, el paradigma de la POO invita a pensar en lo que se desea que represente el programa. Normalmente se responde esta invitación identificando algunas cosas en el mundo que se desea que el programa modele. Estas cosas podrían ser entidades físicas o conceptuales. Una vez identificadas las cosas que se quiere modelar, se identifican sus propiedades/atributos básicos. Después se determina qué pueden hacer las cosas (sus comportamientos) o qué se podría haber hecho con ellas. Se pueden agrupar todas las propiedades y comportamientos juntos en una estructura coherente llamada objeto. Al escribir un programa de POO se definen objetos, se crean y se hace interactuar uno con otro.

Objetos

Un objeto es:

un conjunto de datos relacionados que identifican el *estado* actual del objeto
\+ un conjunto de *comportamientos*.

El estado de un objeto se refiere a las características que lo definen en este momento. Por ejemplo, si se está escribiendo un programa que monitoree los salarios de los empleados, probablemente se querrá tener objetos del empleado, donde el estado de un objeto empleado conste del nombre y salario actual.

Los comportamientos de un objeto se refieren a las actividades asociadas con el objeto. Una vez más, si se está escribiendo un programa que monitoree los salarios de un empleado, probablemente se querrá definir un comportamiento que ajuste el salario del empleado. Este tipo de comportamiento se asemeja al del mundo real: un aumento de pago o una disminución. En Java se implementan los com-

portamientos de un objeto como métodos. Por ejemplo, el comportamiento de ajuste del salario se implementaría como un método `ajustarSalario`. Los detalles de comportamiento de ajuste de salario se describirán en un momento; pero es importante completar primero la introducción a la POO.

He aquí algunas entidades que bien podrían ser objetos en un programa orientado a objetos:

Objetos físicos	Objetos humanos	Objetos matemáticos
autos en una simulación de flujo de tránsito	empleados	puntos en un sistema de coordenadas
avión en un sistema de control aéreo	clientes	números complejos
componentes eléctricos en un programa de diseño de circuitos eléctricos	estudiantes	tiempo

Pensemos en el primer ejemplo de objeto. Si un auto se considera como un objeto en un programa de simulación de flujo de tránsito, ¿qué datos se deben almacenar en cada objeto auto? Para analizar el flujo de tránsito, se debe monitorear cada posición y velocidad del auto. Por tanto, estos dos datos se deben almacenar como parte del estado del objeto auto. Y ¿qué comportamientos están asociados con los objetos auto? Se necesita poder arrancarlo, detenerlo, etc., por lo que probablemente se desearía implementar estos métodos:

```
arrancar, detener, disminuirVelocidad
```

Los comportamientos de un objeto pueden cambiar el estado del mismo. Por ejemplo, el método `arrancar` causa que los datos posición y velocidad cambien.

Encapsulamiento

Los objetos proporcionan *encapsulamiento.* En general, el encapsulamiento sucede cuando algo es envuelto en una capa protectora. Cuando el encapsulamiento se aplica a los objetos, significa que los datos del objeto están protegidos, "ocultos" dentro del objeto. Con los datos ocultos, ¿cómo puede el resto del programa acceder a ellos? (El *acceso* a los datos de un objeto se refiere a leerlos o modificarlos.) El resto del programa no puede acceder de manera directa a los datos de un objeto; lo tiene que hacer con ayuda de los métodos del objeto. En el supuesto de que los métodos de un objeto estén bien escritos, los métodos aseguran que se pueda acceder a los datos de manera adecuada. Volviendo al programa ejemplo empleado-salarios, el salario de un objeto empleado debe modificarse sólo con llamar al método `ajustarSalario`. El método `ajustarSalario` garantiza que el salario de un objeto empleado se modifique en forma apropiada. Por ejemplo, con este método se puede evitar que el salario de un objeto empleado llegue a ser negativo.

La figura 6.1 ilustra cómo los métodos de un objeto constituyen la interfaz entre los datos de un objeto y el resto del programa.

Beneficios de la POO

Ahora que tiene una idea básica de lo que es la POO, podría preguntarse por qué se le promociona tanto. ¿Por qué se prefiere la POO y no la programación procedural en la mayoría de los programas de hoy en día? He aquí algunos de sus beneficios:

- Los programas en POO tienen una organización más natural. Puesto que la gente tiende a pensar en los problemas del mundo real en términos de objetos del mundo real, es más fácil para la gente entender un programa que está organizado en torno a objetos.
- La POO facilita el desarrollo y mantenimiento de programas largos. Sin embargo, el cambio a la programación POO, normalmente hace más complicado un programa pequeño, de manera natural divide las cosas, por lo que el programa crece de manera elegante y no implica un desastre enorme. Puesto que los objetos proporcionan encapsulamiento, los *bugs* (errores) y la reparación de errores pueden ser localizados.

La segunda viñeta requiere alguna explicación. Cuando los datos de un objeto se pueden modificar sólo con la utilización de métodos de ese objeto, es difícil que un programador estropee los datos de un objeto de manera accidental. Volviendo otra vez al programa ejemplo empleado-salarios, se asume que la

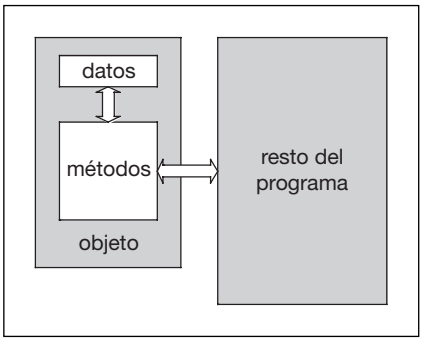


Figura 6.1 Para acceder a los datos de un objeto, se deben utilizar los métodos de un objeto como una interfaz.

única manera de cambiar el salario en el objeto empleado es mediante el método `ajustarSalario`. Después, si hay un error relacionado con el salario del empleado, el programador sabe inmediatamente dónde buscar el problema: en el método `ajustarSalario` o en alguna de las llamadas al mismo.

Clases

Habiendo tratado el tema de los objetos, es hora de hablar acerca de una entidad relacionada estrechamente a ellos: una *clase*. Se comenzará con una amplia definición de clase, que más adelante se detallará. Hablando ampliamente, una clase es una descripción de todos los objetos que define. Como tal, es una *abstracción*: un concepto aparte de cualquier instancia en particular. Observe en la figura 6.2 tres computadoras en una banda transportadora en una planta de manufactura. Las tres computadoras representan objetos. El documento que establece las especificaciones de las computadoras es un plano que describe las características de las computadoras. El documento de especificaciones representa una clase. Cada objeto es una instancia de su clase. Así pues, para propósitos prácticos, "objeto" e "instancia" son sinónimos.

Una clase puede tener cualquier o ningún número de objetos asociadas con ella. Lo anterior debe tener sentido si se piensa en un ejemplo de una fábrica de computadoras. ¿Sería posible tener un patrón de diseño para la computadora, pero no tener ninguna computadora construida a partir de dicho patrón?

Ahora se presentará una descripción más completa de una clase. Antes se mencionó que una clase es una descripción de un conjunto de objetos. La descripción consiste en:

una lista de variables
\+ una lista de métodos

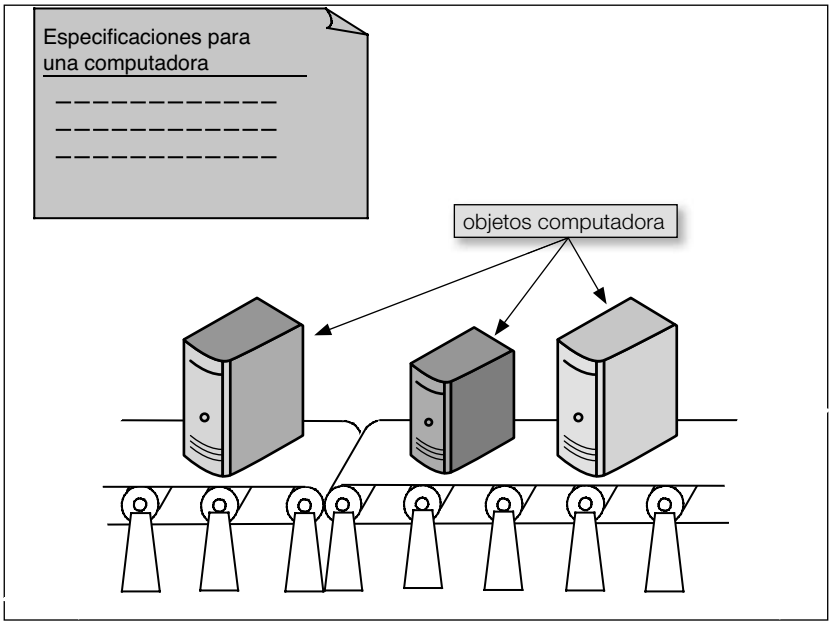


Figura 6.2 Representación de una banda transportadora de la relación de clase-objetos.

Las clases pueden definir dos tipos de variables: *variables de clase* y *variables de instancia*. Y las clases también pueden definir dos tipos de métodos: *métodos de clase* y *métodos de instancia*. En el capítulo 5 se mostró cómo utilizar los métodos de clase de la clase `Math`, y se ha estado definiendo un método de clase llamado `main` desde el principio. En el capítulo 9 se mostrará cuándo es apropiado definir otros métodos de clase y definir y utilizar variables de clase; pero es fácil caer en la trampa de definir y utilizar métodos y variables de clase de manera inapropiada. Queremos alejarlo de esta trampa hasta después de desarrollar buenos hábitos de POO. Por tanto, centraremos la atención en variables y métodos de instancia a lo largo de este capítulo y en varios posteriores.

Las variables de instancia de una clase especifican el tipo de dato que un objeto puede almacenar. Por ejemplo, se tiene un objeto computadora que almacena el valor del tamaño del disco duro de la computadora. El método de instancia de una clase especifica el comportamiento que un objeto puede exhibir; así, por ejemplo, si se tiene una clase para objetos computadora, y la clase `Computadora` contiene un método de instancia `imprimirEspecificaciones`, entonces cada objeto computadora puede imprimir un reporte específico (el reporte de especificaciones muestra el tamaño del disco duro de la computadora, la velocidad del CPU, el costo, etcétera).

Observe el uso del término "instancia" en "variables de instancia". Esto refuerza el hecho de que las variables y los métodos de instancia están asociados con la instancia de un objeto en particular. Por ejemplo, cada objeto `empleado` tendría su propio valor para la variable de instancia `salario`, que sería accedida a través de su método de instancia `ajustarSalario`. Esto contrasta con los métodos de clase. Los métodos de clase están asociados con una clase completa. Por ejemplo, la clase `Math` contiene el método de clase `round`, que no está asociado con una instancia en particular de la clase `Math`.

6.3 Primera clase con POO

En las siguientes secciones se pondrá en práctica lo que se ha aprendido, mediante la implementación de un programa. El programa contendrá una clase llamada `Raton` que simulará el crecimiento de dos objetos `Raton` (se está hablando de roedores, no de dispositivos de cómputo). Como es costumbre en los programas de la POO, se inicia el proceso de implementación con la descripción gráfica de la solución mediante un *diagrama UML de clase*. Un diagrama UML de clase es una técnica diagramática para describir clases, objetos y las relaciones entre ellos. Es ampliamente aceptado en la industria del software como estándar para modelar diseños de POO. Después de describir la solución de la simulación del ratón con un diagrama UML de clase, se presentará el código fuente del programa Raton y se hará un análisis del mismo.

Utilice UML para especificar POO.

Diagrama UML de clase

Como puede verse, la figura 6.3 contiene un diagrama UML de clase abreviado para una clase `Raton`. Un cuadro de diagrama UML de clase está dividido en tres partes: nombre de clase hasta arriba, *atributos* en la parte de en medio y *operaciones* al final. En los programas de Java, los atributos igualan a las variables y las operaciones igualan a los métodos. A partir de este momento se utilizan los términos de Java, variables y métodos, en lugar de los términos formales de UML, que son atributos y operaciones. De manera conjunta, se hace referencia a las variables y métodos de la clase como *miembros* de clase. Ahora se describirá cada miembro de la clase `Raton`.

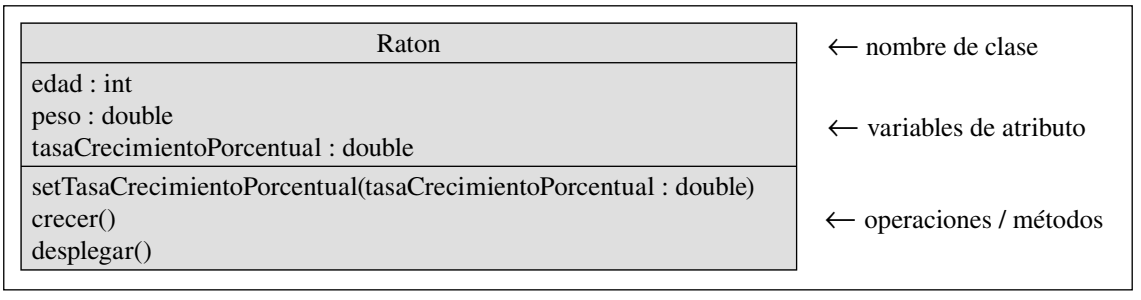


Figura 6.3 Diagrama UML de clase abreviado para una clase `Raton`.

La clase `Raton` tiene tres variables de instancia: `edad`, `peso` y `tasaCrecimientoPorcentual`. La variable de instancia edad mantiene el registro de la edad del objeto `Raton` en días. La variable de instancia `peso` mantiene el registro del peso del objeto `Raton` en gramos. La variable de instancia `tasaCrecimientoPorcentual` es el porcentaje del peso actual que se va agregando al peso cada día. Si la `tasaCrecimientoPorcentual` es de 10 por ciento y el peso actual es de 10 gramos, entonces el ratón sube 1 gramo de peso por día.

La clase `Raton` tiene tres métodos de instancia: `setTasaCrecimientoPorcentual`, `crecer` y `desplegar`. El método `setTasaCrecimientoPorcentual` asigna un valor específico a la variable de instancia `TasaCrecimientoPorcentual`. El método `crecer` simula el aumento de peso en un día para un ratón. El método `desplegar` imprime la edad y peso del ratón.

Con referencia a la figura 6.3 observe cómo se especifican los tipos de variables en un diagrama de clases. El tipo aparece a la derecha de la variable (por ejemplo, edad: int). Esto es lo contrario a la manera en que se realizan las declaraciones en Java, donde se escribe el tipo a la izquierda de la variable (por ejemplo, `int edad;`).

Iniciar documentación temprano.

Algunos programadores utilizan los diagramas UML de clase como medio para documentar programas después de que éstos ya han sido escritos. Lo anterior es correcto, pero no es para lo que fueron concebidos los diagramas. Se recomienda comenzar a dibujar los diagramas de clase como un primer paso para manejar la complejidad del programa y para lograr afinidad con el seudocódigo, probablemente se deseará codificar los métodos directamente con Java o codificar primero con seudocódigo como un paso intermedio. Para el ejemplo de la clase `Raton`, los métodos de la clase `Raton` son sencillos y se codificarán directamente con Java. A continuación se revisará el código fuente en Java de la clase `Raton`.

Código fuente de la clase `Raton`

La figura 6.4 muestra la clase `Raton` implementada con Java. Observe la declaración de las tres variables de instancia de la clase `Raton`: `edad`, `peso` y `TasaCrecimientoPorcentual`. Las variables de instancia deben ser declaradas fuera de todos los métodos, y para hacer el código más autodocumentable se deben declarar al principio de la definición de la clase. Las declaraciones de variables de instancia son muy similares a las declaraciones de variables, que se estudiaron anteriormente: los tipos de variables van a la izquierda de la variable y, de manera opcional, se puede asignar un valor inicial a la variable. ¿Recuerda cómo se llama la asignación de un valor a una variable como parte de una declaración? Se llama inicialización. Observe la inicialización para la `edad` y el `peso`. Se inicializa la `edad` a 0 porque el ratón recién nacido pesa aproximadamente 1 gramo.

La diferencia principal entre la declaración de variables de instancias y las otras variables que se vieron anteriormente es el modificador de acceso `private` (privado). Si se declara un miembro como `private`, entonces el miembro puede ser accedido sólo desde dentro de la clase del miembro y no desde "fuera del mundo" (esto es, por código que esté fuera de la clase en la cual reside el miembro). Las variables de instancia son declaradas casi siempre con el modificador de acceso `private` porque siempre se querrá que los datos del objeto estén ocultos. Al hacer `privada` una variable de instancia tenemos control sobre la manera en que su valor puede ser cambiado. Por ejemplo, se podría asegurar que la variable peso nunca resulte negativa. El acceso a datos restringidos es lo que conlleva el encapsulamiento, y es una de las bases de la POO.

Además del modificador de acceso `private` hay un modificador de acceso `public` (pública). Dadas las definiciones estándar de las palabras "publicas" y "privadas", probablemente se podría asumir que es más fácil acceder a los miembros tipo `public`, ya que éstos pueden ser accedidos en cualquier parte (dentro de los miembros de clase y también fuera de ellos). Se debe declarar un método como `public` cuando se quiera que sea un portal a través del cual el mundo pueda acceder a los datos del objeto. Regrese a revisar la clase `Raton` y verifique que los tres métodos utilizan el modificador `public`. Cuando se quiera que un método ayude a realizar sólo una tarea local, debe declararse como tipo `private`, pero se dejará esa consideración hasta el capítulo 8.

Observe una vez más las declaraciones de variables de instancia de la clase `Raton`. Observe que las variables `edad` y `peso` son inicializadas a 0 y 1.0 respectivamente, pero que no se inicializa `tasaCrecimientoPorcentual`. Lo anterior se debe a que estamos satisfechos con `edad` = 0 y `peso` = 1.0 para todos los objetos `Raton` recién nacidos, aunque no lo estamos con un valor inicial predefinido para `ta-`

```
/*****************************************************************
* Raton.java
* Dean & Dean
*
* Este programa solicita al usuario adivinar un número seleccionado
* de manera aleatoria.
*****************************************************************/

public class Raton
{
  private int edad = 0;                    // edad del ratón en días
  private double peso = 1.0;               // peso del ratón en gramos
  private double tasaCrecimientoPorcentual; // incremento por día

  //*********************************************************

  // Este metodo asigna la tasa de crecimiento del raton.

  public void setTasaCrecimientoPorcentual(double tasaCrecimientoPorcentual)
  {
    this.tasaCrecimientoPorcentual = tasaCrecimientoPorcentual;
  } // fin setTasaCrecimientoPorcentual

  //*********************************************************

  // Este método simula un día de crecimiento para un ratón.

  public void crecer()
  {
    this.peso +=
      (.01 * this.tasaCrecimientoPorcentual * this.peso);
    this.edad++;
  } // fin crecer

  //*********************************************************

  // Este método imprime la edad y peso del ratón.

  public void desplegar()
  {
    System.out.printf("Edad = %d, peso = %.3f\n",
      this.edad, this.peso);
  } // fin desplegar
} // fin de la clase Raton
```

declaración de variables de instancia

parámetro

Para acceder a variables de instancia, utilizar este punto.

cuerpo del método

Figura 6.4 Clase `Raton`.

`saCrecimientoPorcentual`. Presumiblemente, se querrán utilizar diferentes valores para esta última variable para diferentes objetos tipo `Raton`. (Un ratón comiendo donas en un estudio podría tener una `tasadecrecimiento` más alta que en otro estudio donde el ratón está expuesto a humo de cigarro.)

Sin inicialización a la variable de instancia `tasaCrecimientoPorcentual`, ¿cómo se podría establecer la tasa de crecimiento para el objeto `Raton`? Se puede hacer que el objeto `Raton` llame al método `setTasaCrecimientoPorcentual` con un valor como tasa de crecimiento como argumento. Por ejemplo, he aquí cómo se puede establecer esta tasa de crecimiento en 10 (por ciento).

```
setTasaCrecimientoPorcentual(10);
```

Como se recordará del capítulo 5, los valores que aparecen dentro del paréntesis de las llamadas a los métodos se denominan *argumentos*. Por tanto, en este ejemplo, 10 es un argumento. El 10 se pasa a la variable `tasaCrecimientoPorcentual` en el encabezado del método `setTasaCrecimientoPorcentual`. Los valores de las variables pasadas dentro del paréntesis de la llamada al método se conocen como *parámetros*. Así, en el ejemplo mostrado en la figura 6.4, `tasaCrecimientoPorcentual` es un parámetro. Dentro del *cuerpo del método* `setTasaCrecimientoPorcentual` (el código que aparece entre la apertura y cierre de las llaves), el parámetro `tasaCrecimientoPorcentual` es asignado a la variable de instancia `tasaCrecimientoPorcentual`. He aquí la sentencia de asignación relevante:

```
this.tasaCrecimientoPorcentual = tasaCrecimientoPorcentual;
```

Observe el "`this` punto" en `this.tasaCrecimientoPorcentual`. El `this` punto es la manera en que se le dice al compilador de Java que la variable a la que nos estamos refiriendo es una variable de instancia. Puesto que la variable `tasaCrecimientoPorcentual` que está a la derecha no va antecedida por `this` punto, el compilador de Java sabe que `tasaCrecimientoPorcentual` se refiere al parámetro `tasaCrecimientoPorcentual`, no a la variable de instancia con el mismo nombre. En el método `setTasaCrecimientoPorcentual` de la figura 6.4, la variable de instancia y el parámetro tienen el mismo nombre. Ésta es una práctica común. No hay problema para distinguir entre las dos variables porque la variable de instancia utiliza el `this` punto y el parámetro no.

Ahora observe los métodos `desplegar` y `crecer` de la clase `Raton`. El método `desplegar` es sencillo: imprime la edad y peso de un ratón. El método `crecer` simula el aumento de peso en un día para un ratón. La fórmula de aumento de peso agrega un cierto porcentaje del peso al peso actual. Esto significa que el ratón seguirá creciendo cada día de su vida. Esto es simple, pero no es una representación muy segura sobre el aumento normal de peso. Se dejó de manera intencional la fórmula de aumento de peso así de simple para evitar empantanarse en matemáticas complicadas. En la sección final de este capítulo se proporcionan modelos de crecimiento más realistas.

Finalmente, eche un vistazo a cada uno de los comentarios de la clase `Raton`. Observe las descripciones en cada uno de ellos. El estilo adecuado sugiere que, sobre cada método, se debe tener una línea en blanco, una línea de asteriscos, una línea en blanco, una descripción del método, y otra línea en blanco. Las líneas en blanco y la de los asteriscos sirven para separar los métodos. Las descripciones de los métodos permiten a alguien que esté leyendo nuestro programa tener una idea de lo que se está haciendo en nuestro programa.

6.4 Clase controladora

¿Qué es un controlador?

Un *controlador* es un término de computación que se aplica a una pieza de software que ejecuta o "controla" algo más. Por ejemplo, un controlador de una impresora es un programa que se encarga de manejar una impresora. De la misma manera, una *clase controladora* o *clase driver* es una clase que se encarga de ejecutar otra clase.

En la figura 6.5 se presenta la clase `ControladorRaton`. Decimos que es `controladoraRaton` porque crea objetos `Raton` y los manipula. Por ejemplo, observe las `sentencias gus = new Raton()` y `jaq = new Raton()`. Este código crea los objetos `gus` y `jaq`[1] Además, observe el `código gus.setTasaCrecimientoPorcentual(tasaCrecimiento)`. Este `código` manipula el objeto `gus` al cambiar el valor de la `tasaCrecimientoPorcentual`.

Normalmente, una clase controladora consiste enteramente en una clase `main` y nada más. La clase controladora con su método `main` es el punto de arranque del programa. Llama a la clase controlada para crear objetos y manipularlos. La clase controlada, consciente de su trabajo, atiende los requerimientos de creación y manipulación de objetos. Por lo regular, la realización de estas tareas es el enfoque

[1] Padre de dos niñas de edad preescolar, el autor Jean Dean está inmerso en el mundo de Walt Disney. Gus y Jaq son dos ratones del cuento de *La Cenicienta*.

```
/*****************************************
* ControladorRaton.java
* Dean & Dean
*
* Éste es un controlador para la clase Raton.
*****************************************/

import java.util.Scanner;

public class ControladorRaton
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double tasaCrecimiento;
    Raton gus = new Raton();
    Raton jaq = new Raton();

    System.out.print("Introduzca % de tasa de crecimiento: ");
    tasaCrecimiento = stdIn.nextDouble();
    gus.setTasaCrecimientoPorcentual(tasaCrecimiento);
    jaq.setTasaCrecimientoPorcentual(tasaCrecimiento);
    gus.crecer();
    jaq.crecer();
    gus.crecer();
    gus.desplegar();
    jaq.desplegar();
  } // fin del main
} // fin de la clase ControladorRaton
```

la creación de dos objetos `Raton`

Figura 6.5 Clase `ControladorRaton` que controla a la clase `Raton` de la figura 6.4.

principal del programa, y sus implementaciones requieren de la mayor parte del código del programa. Así, las clases controladas normalmente existen (pero no siempre) más tiempo que las clases controladoras.

Las clases controladoras, como la clase `ControladorRaton`, están en archivos separados del archivo de las clases que controlan. Para hacerlas accesibles desde afuera del mundo, las clases controladoras deben ser tipo `public`. Cada clase tipo `public` debe ser almacenada en un archivo separado, cuyo nombre debe ser el mismo que el de la clase, así la clase `ControladorRaton` debe estar en un archivo llamado `ControladorRaton.java`. Para que el código de `ControladorRaton` pueda encontrar la clase `Raton`, ambas deben estar en el mismo directorio.[2]

Variables de referencia

En la clase `ControladorRaton` se crearon objetos `Raton`, y se hace referencia a los mismos como `gus` y `jaq`, donde `gus` y `jaq` son *variables de referencia*. El valor contenido en una variable de referencia es una "referencia" (de ahí el nombre de variable de referencia). De manera más exacta, una variable de referencia almacena la dirección en la que es almacenado el objeto en la memoria. Para una descripción gráfica de esta explicación, vea la figura 6.6. En la figura, las cajas que aparecen inmediatamente a la derecha de `gus` y `jaq` representan direcciones. Así, la cajita de la dirección de `gus` almacena la dirección del primer objeto.

[2] Se están simplificando las cosas, al solicitarle colocar ambas clases en el mismo directorio. De hecho, los archivos pueden estar en diferentes directorios, pero entonces se tendrían que agrupar las clases en paquetes. El apéndice 4 describe cómo agrupar las clases en paquetes.

Habla vernácula de la industria POO

La mayoría de los programadores de Java de la industria no utilizan el término variable de referencia. En lugar de ello, emplean el vocablo objeto. Esto crea una confusión entre los términos variables de referencia y objetos. Por ejemplo, en la clase `ControladorRaton` de la figura 6.5, la siguiente sentencia inicializa la referencia a la variable `gus`:

```
Raton gus = new Raton();
```

Aunque es una variable de referencia, la mayoría de los programadores de Java se referirán a `gus` como un objeto. A pesar de la práctica común de utilizar "objeto" como sustituto de "variable de referencia", es importante saber distinguir: un objeto almacena un grupo de datos, y una variable de referencia, la ubicación donde ese grupo de datos es almacenado en memoria. Entender la diferencia entre uno y otro término ayudará a entender el comportamiento del código de Java.

Declaración de una variable de referencia

Siempre se debe declarar una variable de referencia antes de poderla utilizar. Por ejemplo, para utilizar la variable tipo `int` llamada `cuenta`, primero se debe declararla de la siguiente manera:

```
int cuenta;
```

De la misma manera, para utilizar la variable de referencia `gus`, primero se debe declarar `gus` como a continuación:

```
Raton gus;
```

Como se puede ver, el proceso para declarar variable de referencia es un reflejo de la declaración de variables primitivas. La única diferencia es que en lugar de escribir un tipo primitivo a la izquierda (por ejemplo, `int`) para variables de referencia se escribe el nombre de la clase a la izquierda (por ejemplo, `Raton`).

Instancia y asignación de un valor a una variable de referencia

Como se sabe, el punto de una variable de referencia es para guardar una referencia a un objeto; pero antes de que se pueda almacenar una referencia en un objeto, se necesita tener el objeto; por lo que, veamos el proceso de creación de un objeto.

Para crear un objeto es necesario utilizar el operador `new`. Por ejemplo, para crear un objeto `Raton`, hay que especificar `new Raton()`. El operador `new` debe tener sentido cuando se da cuenta de que `new`

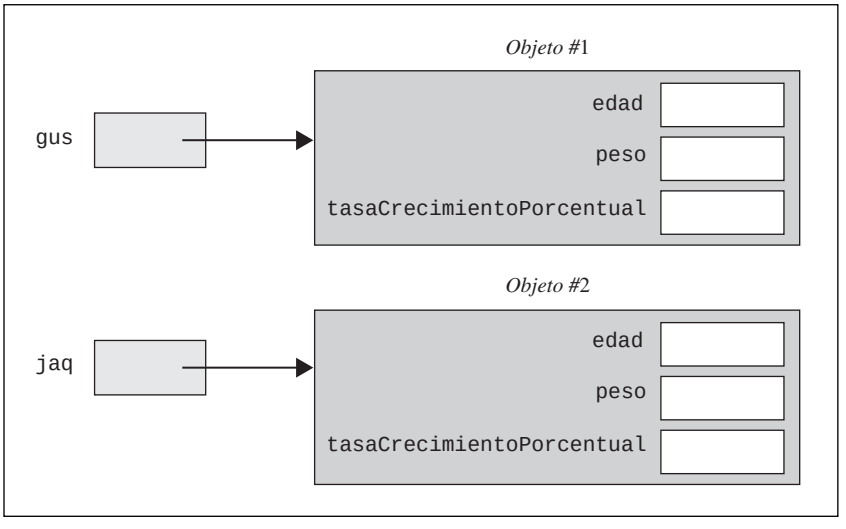


Figura 6.6 Variables de referencia y objetos para el programa `Raton` en las figuras 6.4 y 6.5. Las dos variables de referencia a la izquierda: `gus` y `jaq` contienen referencias que apuntan a los dos objetos a la derecha.

`Raton()` crea un nuevo objeto. El término formal para creación de un objeto es el de *instanciar* un objeto. Así, con `new Raton()` se instancia un objeto. El término "*instanciar*" es la forma verbalizada del sustantivo "*instancia*". Es un término de la jerga computacional para "crear una instancia de clase" o "crear un objeto".

Después de instanciar un objeto, normalmente se asignará éste a una variable de referencia. Por ejemplo, para asignar un objeto `Raton` a la variable de referencia `gus` se debe hacer lo siguiente:

```
gus = new Raton();
```

Después de la asignación, `gus` contiene una referencia al nuevo objeto creado `Raton`.

Revisemos. He aquí cómo se declaró una variable de referencia `gus`: se instanció un objeto tipo `Raton`, y se asignó la dirección de objeto a `gus`:

```
Raton gus;
gus = new Raton();
```

declaración

instancia y asignación

Ahora se verá cómo hacer lo mismo con sólo una sentencia:

```
Raton gus = new Raton();
```

inicialización

La sentencia anterior es lo que aparece en la clase `ControladorRaton` de la figura 6.5. Es una inicialización. Como se mencionó anteriormente, una inicialización se da cuando se declara una variable y se le asigna un valor, todo en la misma sentencia.

Llamada a un método

Después de que se instancia un objeto y se le asigna su variable de referencia, este método se puede llamar utilizando la sintaxis:

<*variable de referencia*>.<*nombre metodo*>(<*args separados por coma*>);

He aquí tres ejemplos de llamadas a métodos de instancia de la clase `ControladorRaton`:

```
gus.setTasaCrecimientoPorcentual(tasaCrecimiento);
gus.crecer();
gus.desplegar();
```

Observe cómo las tres llamadas a los métodos toman la sintaxis modelo. La primera llamada al método tiene un argumento y las siguientes dos llamadas a los métodos no tienen ninguno. Si se tuvieran métodos con dos parámetros, se les llamaría con dos parámetros separados por una coma.

Cuando un programa llama a un método, le pasa el control desde la sentencia de llamada a la primera sentencia de ejecución en la llamada al método. Por ejemplo, cuando el método `main` de la clase `ControladorRaton` llama al método `setTasaCrecimientoPorcentual` con `gus.tasaCrecimientoPorcentual(tasaCrecimiento)`, el control pasa a esta sentencia en el método `setTasaCrecimientoPorcentual` de la clase `Raton`.

```
this.tasaCrecimientoPorcentual = tasaCrecimientoPorcentual;
```

Volviendo a la figura 6.4 de la clase `Raton` se puede verificar que el método `setTasaCrecimientoPorcentual` contiene la sentencia anterior.

Después de que la última sentencia en cualquier método se ejecuta, el control vuelve a la llamada al método en el punto posterior a donde fue hecha la llamada. Para una explicación gráfica, vea la figura 6.7.

6.5 Objeto llamado, referencia `this`

Suponga que se tienen dos objetos que son instancias de la misma clase. Por ejemplo, `gus` y `jaq` hacen referencia a dos objetos que son instancias de la clase `Raton`. Suponga que se quiere que los dos objetos llamen al mismo método de instancia. Por ejemplo, se quiere que tanto `gus` como `jaq` llamen al método `setTasaCrecimientoPorcentual`. Para cada llamada al método, la máquina virtual de Java (JVM) necesita saber qué objeto actualizar (si `gus` llama a `setTasaCrecimientoPorcentual`, entonces la

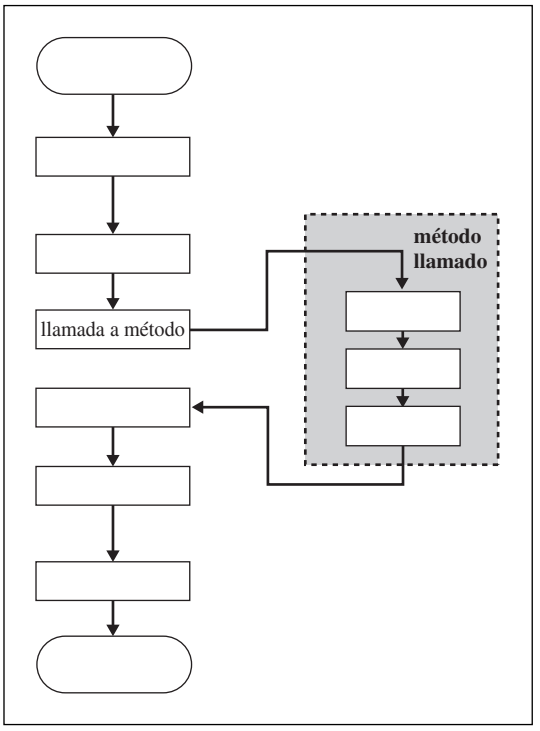


Figura 6.7 Llamada a un método.

JVM debe actualizar la `tasaCrecimientoPorcentual` de `gus`; si es `jaq` el que lo hace, entonces la JVM debe actualizar la `tasaCrecimientoPorcentual` de `jaq`). Esta sección describe de qué manera sabe la JVM qué objeto actualizar.

Llamada a un objeto

Como se mencionó en el capítulo 5, cuando un método de instancia es llamado, el método está asociado con el objeto que realiza la llamada. Se puede identificar el objeto llamador observando a la izquierda del punto en una sentencia de llamado a un método. ¿Puede usted identificar los objetos que realizan la llamada en el siguiente método `main`?

```
public static void main(String[] args)
{
  Scanner stdIn = new Scanner(System.in);
  double tasaCrecimiento;
  Raton gus = new Raton();

  System.out.print("Introduzca el % de tasa de crecimiento: ");
  tasaCrecimiento = stdIn.nextDouble();
  gus.setTasaCrecimientoPorcentual(tasaCrecimiento);
  gus.crecer();
  gus.desplegar();
} // fin de main
```

El objeto `gus` es el objeto llamador para estas sentencias.

```
gus.setTasaCrecimientoPorcentual(tasaCrecimiento);
gus.crecer();
gus.desplegar();
```

¿Existe algún otro objeto llamador? Sí. El objeto `stdIn` es un objeto que realiza una llamada en esta sentencia:

```
tasaCrecimiento = stdIn.nextDouble();
```

La referencia `this`

Es fácil identificar el objeto llamador cuando se está analizando una sentencia de llamada a un método; pero si se está dentro de ese método, ¿cómo se puede saber qué objeto fue el que realizó la llamada? Por ejemplo, cuando se está buscando en la definición de la clase `Raton` en la figura 6.4, ¿puede usted identificar el objeto que realiza la llamada de su método `crecer`? He aquí dicho método otra vez:

```
public void crecer()
{
  this.peso +=
    (0.01 * this.tasaCrecimientoPorcentual * this.peso);
  this.edad++;
} // fin crecer
```

El pronombre `this` (llamado *referencia `this`*) representa el objeto que realiza la llamada, pero no indica qué objeto es. Así, no se puede decir cuál es el objeto llamador, únicamente revisando el método que fue llamado. Se debe analizar lo que llamó a dicho objeto. De manera alternativa, si la sentencia que llamó a `crecer` fue `jaq.crecer()`, entonces `jaq` es el objeto que realiza la llamada. Como se verá cuando se realice el rastreo, se debe saber si el objeto `gus` o `jaq` es el objeto que realiza la llamada en este momento para actualizar el objeto adecuado. Dentro del método `crecer` observe las sentencias `this.peso` y `this.edad`. La referencia `this` nos recuerda que `peso` y `edad` son variables de instancia. Variables de instancia ¿en qué objetos? ¡En el objeto que realiza la llamada!

El método `setTasaCrecimientoPorcentual` de la figura 6.4 proporciona otro ejemplo. He aquí el método otra vez:

```
public void setTasaCrecimientoPorcentual(double tasaCrecimientoPorcentual)
{
  this.tasaCrecimientoPorcentual = tasaCrecimientoPorcentual;
} // fin setTasaCrecimientoPorcentual
```

La referencia `this` indica que la variable del lado izquierdo de esta única sentencia en el método es una variable de instancia en el objeto llamador. Como se indicó anteriormente, la referencia `this` en esta sentencia también ayuda al compilador y al usuario a distinguir la variable del lado izquierdo de la del lado derecho. Antes de la llegada de la POO, los lenguajes de cómputo no incluían la funcionalidad `this` punto. Entonces, la única forma en que el compilador y el usuario podían distinguir entre variables en diferentes lugares que se refirieran esencialmente a la misma cosa, era dándoles nombres similares, pero ligeramente diferentes.

La naturaleza *ad hoc* (caso especial) de los programadores de hace tiempo encontraron nombres ligeramente diferentes que hicieron un poco confusos los programas e incrementaron los errores de programación. La referencia `this` de Java proporcionó una forma estándar para hacer la distinción y mostrar la relación al mismo tiempo. Se puede utilizar exactamente el mismo nombre para mostrar la relación y después utilizar el `this` punto para hacer la distinción. Así no es necesario utilizar nombres ligeramente diferentes para tal propósito y además estamos en contra de esa práctica arcaica.

Para resaltar el significado y utilidad de la referencia `this` en Java se utilizará en todos los ejemplos de variables de instancia al final del siguiente capítulo, aunque esto no es necesario para realizar una distinción entre una variable de instancia y un parámetro. No hay castigo al utilizar `this` punto, y proporcionar un indicador inmediato a todos de que la variable es de instancia. Así, esto ayuda a explicar el programa; esto es, se proporciona autodocumentación de utilidad.

6.6 Variables de instancia

Durante un buen tiempo se ha expuesto el tema de las variables de instancia. Ahora se sabe que un objeto almacena sus datos en variables de instancia. Se sabe que los métodos de instancia acceden a las variables de instancia antecediéndolas con la referencia `this` (por ejemplo, `this.peso`). En esta sección se hablará de algunos detalles de las variables de instancia. De manera específica, se tratará el tema de valores por omisión y de la persistencia.

Valores por omisión para las variables de instancia

Como lo indica la definición de la palabra "omisión" (default), el *valor por omisión* de una variable es el que contiene ésta cuando no hay una asignación explícita a su valor inicial; esto es, cuando no hay una inicialización explícita. Existen diferentes tipos de valores por omisión para los diferentes tipos de variables.

Existen dos tipos enteros, que se han estudiado hasta este momento: `int` y `long`. A las variables de instancia de tipo entero (int) les es asignado el número cero por omisión; pero, en la clase `Raton`, observe que la variable `edad` es inicializada a 0:

```
private int edad = 0;  // edad del ratón en días
```

¿Por qué molestarse en una inicialización explícita? ¿No le sería asignado el valor de 0 por omisión, aun cuando "= 0" fuera omitido? La respuesta es sí, el programa hubiera funcionado de cualquier manera; pero es una lamentable práctica depender de los valores ocultos. Al asignar explícitamente valores a las variables se demuestra nuestro propósito, que es una forma de código autodocumentable.

Hay dos tipos de punto flotante: `float` y `double`. A las variables de instancia de tipo flotante se les asigna el valor de 0.0 por omisión. La clase `Raton` declara dos variables de instancia de tipo punto flotante: `peso` y `tasaCrecimientoPorcentual`:

```
private double peso = 1.0;                 // peso del ratón en gramos
private double tasaCrecimientoPorcentual;  // % incremento por día
```

En este caso, se inicializa la variable de instancia `peso` a 1.0, para que no se le asigne el valor por omisión. No se inicializa la variable `tasaCrecimientoPorcentual`, para que quede inicializada a 0.0 por omisión. Pero, ¿no se acaba de decir que hacer esto es una lamentable práctica? Sí, pero en este caso, no estamos dependiendo del valor por omisión. En la clase `ControladorRaton` se sustituye el valor `tasaCrecimientoPorcentual` con un valor hecho a la medida, llamando al método `setTasaCrecimientoPorcentual` como a continuación:

```
gus.setTasaCrecimientoPorcentual(TasaCrecimiento);
```

A las variables de instancia tipo `boolean` se les asigna el valor `false` por omisión. Por ejemplo, si se agrega una variable de instancia tipo `boolean` llamada `vacunado` a la clase Raton, `vacunado` tendría asignado el valor de `false` por omisión.

A las variables de instancia de tipo referencia se les asigna el valor `null` por omisión. Por ejemplo, si se agregara a `String` una variable llamada `alimento` a la clase `Raton`, `alimento` tendría un valor de `null` por omisión. Normalmente, una variable de referencia contiene la dirección de un objeto y esa dirección apunta a un objeto. Los diseñadores de Java agregaron `null` (*nulo*) al lenguaje como una forma de indicar que una variable de referencia apunta a nada. Así, por omisión, una variable de instancia de tipo referencia no apunta a nada.

He aquí un resumen de los valores por omisión de las variables de instancia:

Tipo de variable de instancia	Valor por omisión
entera	0
punto flotante	0.0
`boolean`	`false`
referencia	`null`

Persistencia de las variables de instancia

Ahora se considerará la *persistencia* de variables. La persistencia se refiere a cuánto tiempo sobrevive el valor de una variable antes de que sea eliminado. Las variables de instancia persisten para la duración de un objeto. Así, si un objeto realiza dos llamadas a métodos, la segunda llamada no restaura las variables de instancia del objeto a sus valores iniciales. En lugar de ello, las variables de instancia mantienen su valor de la llamada de un método al otro. Por ejemplo, en la clase `ControladorRaton` el objeto llama a `crecer` dos veces. En la primera llamada a `crecer`, la edad de `gus` se incrementa de 0 a 1; en la segunda, la edad del objeto empieza con 1 y se incrementa a 2. El objeto `gus` mantiene su valor de la primera llamada a `crecer` a la siguiente, porque `edad` es una variable de instancia.

6.7 Rastreo de un programa con POO

Para reforzar lo que se ha aprendido hasa ahora en este capítulo se realizará el rastreo de un programa Raton. ¿Recuerda el proceso de rastreo que se utilizó en los capítulos anteriores? Éste funcionó bien para programas con un solo método: el método `main`; pero para los programas con POO con múltiples clases y múltiples métodos, se requerirá rastrear en qué clase y en qué método estamos en un momento dados y qué objeto llamó a este método. Además, será necesario rastrear los parámetros y las variables de instancia. Esto requiere de una tabla de rastreo más elaborada.

Al rastrear el programa Raton se utilizará un controlador ligeramente diferente, la clase `ControladorRaton2`, que se muestra en la figura 6.8. En `ControladorRaton2` se retrasa la creación de instancias de ratones individuales y se les asigna sus tasas de crecimiento (llamando al método `setTasaCrecimientoPorcentual`) inmediatamente después de crear cada instancia. Éste es un mejor estilo, porque se asocia de manera más cercana a la realización de instancias de cada objeto con su asignación de tasa de crecimiento. Sin embargo, al cambiar el controlador, "accidentalmente" se olvidó llamar a `setTasaCrecimientoPorcentual` para `jaq`, el segundo ratón. El efecto de este error lógico puede ser visto en la salida: `jaq` no crece (después del primer día y todavía pesa 1 gramo); pero imaginemos que no se ha caído en la cuenta de la razón de esto. Recuerde, el rastreo es una herramienta útil cuando se requiere depurar un programa.

Utilice rastreo para encontrar la causa de un problema.

```
 1  /*****************************************************
 2  * ControladorRaton2.java
 3  * Dean & Dean
 4  *
 5  * Éste es un controlador para la clase Raton.
 6  *****************************************************/
 7
 8  import java.util.Scanner;
 9
10  public class ControladorRaton2
11  {
12    public static void main(String[] args)
13    {
14      Scanner stdIn = new Scanner(System.in);
15      double tasaCrecimiento;
16      Raton gus, jaq;
17
18      System.out.print("Introduzca % de tasa de crecimiento: ");
19      tasaCrecimiento = stdIn.nextDouble();
20      gus = new Raton();
21      gus.setTasaCrecimientoPorcentual(tasaCrecimiento);
22      gus.crecer();
23      gus.desplegar();
24      jaq = new Mouse();
25      jaq.crecer();
26      jaq.desplegar();
27    } // fin del main
28  } // fin de la clase ControladorRaton2
```

Esto declara variables de referencia, pero no las inicializa.

Intenta agrupar actividades de inicialización.

Hay un error de lógica aquí. De manera "accidental" se olvidó inicializar la tasa de crecimiento en `jaq`.

Sesión muestra:

```
Introduzca % de tasa de crecimiento: 10
Edad = 1, peso = 1.100
Edad = 1, peso = 1.000
```

`jaq` no crece. ¡Error!

Figura 6.8 Clase `ControladorRaton2` que controla a la clase `Raton` de la figura 6.9.

```
 1  /*************************************************************
 2  * Raton.java
 3  * Dean & Dean
 4  *
 5  * Este programa solicita al usuario adivinar un número seleccionado
    * de manera aleatoria.
 6  *************************************************************/
 7
 8  public class Raton
 9  {
10    private int edad = 0;                    // edad del ratón en días
11    private double peso = 1.0;               // peso del ratón en gramos
12    private double tasaCrecimientoPorcentual; // incremento por día
13
14    //*************************************************************
15
16    // Este método asigna la tasa de crecimiento del ratón.
17
18    public void setTasaCrecimientoPorcentual(double tasaCrecimientoPorcentual)
19    {
20      this.tasaCrecimientoPorcentual = tasaCrecimientoPorcentual;
21    } // fin setTasaCrecimientoPorcentual
22
23    //*************************************************************
24
25    // Este método simula un día de crecimiento para un ratón.
26
27    public void crecer()
28    {
29      this.peso +=
30        (.01 * this.tasaCrecimientoPorcentual * this.peso);
31      this.edad++;
32    } // fin crecer
33
34    //*************************************************************
35
36    // Este método imprime la edad y peso del ratón.
37
38    public void desplegar()
39    {
40      System.out.printf(
41        "Edad = %d, peso = %.3f\n", this.edad, this.peso);
42    } // fin desplegar
43  } // fin de la clase Raton
```

Figura 6.9 Clase Raton repetición de la figura 6.4.

Para ejecutar un rastreo, además del controlador, se requerirá del código de la clase controlada. Para su conveniencia, se repite el código original de la clase controlada `Raton` en la figura 6.9.

Organización del rastreo

La figura 6.10 muestra la organización. Al igual que en los rastreos de los capítulos anteriores, la entrada va en la parte superior izquierda. Al contrario de los rastreos en los capítulos anteriores, el encabezado bajo la entrada ahora requiere de más de una línea. La primera línea de los encabezados muestra los nombres de clase: `ControladorRaton2` y `Raton`. Bajo cada encabezado del nombre de clase hay un enca-

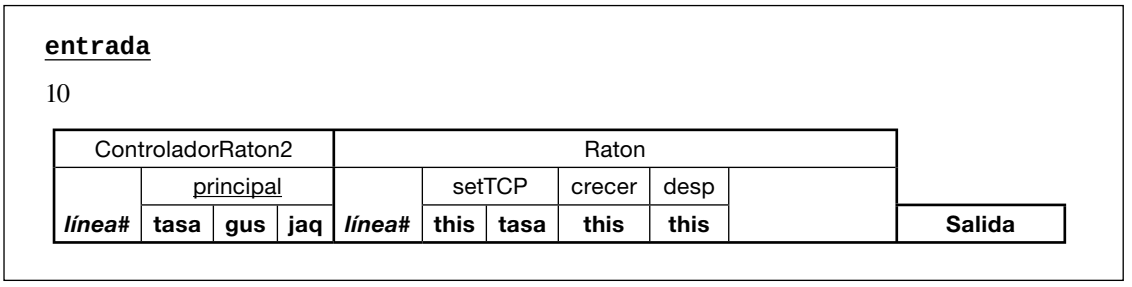


Figura 6.10 Organización del rastreo para el programa Raton.

bezado para cada uno de los métodos de clase. En la organización del rastreo, observe los encabezados de los métodos `setTasaCrecimientoPorcentual` y `desplegar` (para ahorrar espacio, se abrevian dichos métodos como setTCP y desp, respectivamente). Y bajo cada uno de los encabezados de los nombres de método hay un encabezado para cada variable local de los métodos y para sus parámetros.

Más tarde se hablará con más detalle acerca de las *variables locales*, pero por ahora basta con verificar que `tasaCrecimiento` (abreviada tasa en la organización de rastreo), `gus` y `jaq` son consideradas variables locales porque son declaradas y usadas "localmente" dentro de un método `main`. Esto es diferente de las variables de instancia `edad`, `peso` y `tasaCrecimientoPorcentual`, las cuales están declaradas fuera de todos los métodos, en la parte alta de la clase. Observe que `stdIn` es otra variable local dentro de `main`, pero que no hay necesidad de realizar rastreo sobre ésta porque ya ha instanciado desde una clase API, *Scanner.* No es necesario rastrear las clases API porque éstas ya han sido rastreadas y probadas por los camaradas de Sun. Podemos asumir que realizaron su trabajo adecuadamente.

Ahora se examinará el rastreo de la organización de los parámetros. El método `setTasaCrecimientoPorcentual` tiene dos parámetros: `tasaCrecimientoPorcentual`, abreviada como tasa en la organización del rastreo, y la referencia `this`, un parámetro implícito. Como se podrá recordar, la referencia `this` apunta al objeto que realiza la llamada. Para los métodos `setTasaCrecimientoPorcentual`, `crecer` y `desplegar`, se incluye una columna para `this` con la finalidad de que se pueda mantener un rastreo sobre el objeto que llamó al método.

Observe el área vacía bajo el encabezado `Raton`. Se llenará cuando se ejecute el rastreo.

Ejecución del rastreo

El utilizar el modelo de rastreo de la figura 6.10 como punto de inicio nos llevará a las secciones clave del rastreo mostrado en la figura 6.11. Se destacarán las partes de rastreo de la POO, ya que ésas son las partes nuevas para usted. Cuando se inicia un método, con encabezados de variables de métodos locales, se escriben valores iniciales para cada una de éstas. Se utilizan signos de interrogación para variables locales que no estén inicializadas. En las primeras tres líneas del rastreo en la figura 6.11, observe los signos ? para la variable no inicializada `tasaCrecimiento` (abreviada como tasa) de las variables local `gus` y `jaq`.

Cuando un objeto es instanciado, bajo el encabezado del nombre de una clase del objeto, proporcionar una columna "obj#", donde # es un número único. Bajo el encabezado obj#, proporcionar un encabezado de columna subrayado para cada una de las variables de instancias del objeto. Bajo los encabezados de variables de instancia, escribir valores iniciales para cada una de las variables de instancia. En el rastreo de la figura 6.11 observe las columnas de los encabezados obj1 y obj2, y sus subencabezados `peso`, `edad` y `tasaCrecimientoPorcentual` (abreviado como tasa). También observe los valores iniciales para las variables de instancia `peso`, `edad` y `tasaCrecimientoPorcentual`.

Cuando haya una asignación en una variable de referencia, escribir obj# bajo el encabezado de la columna de la variable de referencia, donde obj# concuerde con el obj# asociado en la porción de rastreo del objeto. Por ejemplo, en el rastreo de la figura 6.11 se creó un obj1 mientras se rastreaba la sentencia `gus = new Raton()`. Posteriormente, se colocó obj1 bajo el encabezado de columna de `gus`.

Cuando exista una llamada a un método, bajo el encabezado de columna `this` del método llamado, escribir la llamada al objeto obj# que realiza la llamada. En el rastreo de la figura 6.11 observe obj1 bajo el encabezado de `this` del método `setTasaCrecimientoPorcentual`. Si la llamada al método contiene un argumento, escribir el valor del argumento bajo el parámetro asociado del método llamado. En el

input

10

ControladorRaton2				Raton											
	main				setTCP		crecer	desp	obj1			obj2			
línea#	**tasa**	**gus**	**jaq**	*línea#*	**this**	**tasa**	**this**	**this**	**edad**	**peso**	**tasa**	**edad**	**peso**	**tasa**	**Salida**
15	?														
16		?	?												
18															Introduzca % de tasa de crecimiento:
19	*10.0*														
20															
				10					0						
				11						1.000					
				12							0.0				
20		obj1													
21					obj1	10.0									
				20							10.0				
22							obj1								
				29						1.100					
				31					1						
23								obj1							
				40											Edad = 1, peso = 1.100
24															
				10								0			
				11									1.000		
				12										0.0	
24			obj2												
25							obj2								
				29									1.000		
				31								1			
26								obj2							
				40											Edad = 1, peso = 1.000

Figura 6.11 Rastreo completo para el programa Raton.

rastreo, observe el 10 que se pasa en el encabezado del parámetro `tasaCrecimientoPorcentual` del método `setTasaCrecimientoPorcentual`. Dentro del método, si hay una referencia `this` encontrar el obj# bajo el encabezado de la columna `this` del método. Después, colocarse en el encabezado obj# encontrado y leer el valor actualizado de obj# según corresponda. En la clase `Raton` de la figura 6.9 observe que la referencia `this` en el cuerpo del método `this.tasaCrecimientoPorcentual` se refiere a obj1, de modo que `tasaCrecimientoPorcentual` es actualizada como corresponde.

Cuando termina el rastreo de un método se dibuja una línea horizontal bajo los valores de las variables del método para indicar el final del rastreo del método y para indicar que los valores en las variables locales del método desaparecen. Por ejemplo, en el rastreo la línea horizontal gruesa en Raton, línea #20 bajo el método setTCP indica el final del método `setTasaCrecimientoPorcentual` y significa que el valor de `tasaCrecimientoPorcentual` desaparece.

Practique. Ahora que se ha avanzado en las nuevas técnicas de rastreo para un programa en POO, le recomendamos regresar al modelo de rastreo de la figura 6.10 y hacer el rastreo completo por su cuenta. Ponga atención especial a lo que pasa cuando `gus` y `jaq` llaman al método `crecer`. Verifique los incrementos en peso de `gus` (como deberían ser) y en los incrementos y disminuciones de `jaq` (un error). Cuando haya terminado el rastreo, compare la respuesta con la de la figura 6.11.

La experiencia con la forma larga de rastreo utilizada en este libro facilitará el entendimiento de la depuración automática en lo que un *Ambiente de Desarrollo Integrado* (IDE) está indicando. Conforme se avance en el programa que se está ejecutando en modo depuración (debug) bajo el mando del depurador del IDE, cuando se encuentre una llamada a un método, se tendrán dos opciones: se puede ir paso a paso y revisar todas las sentencias en el método llamado, tal como se hace en el método llamado en la figura 6.11, o se puede avanzar "por proceso", y observe qué pasa después de que se ejecuta el método. En una actividad típica de depuración se tendrá una combinación de avance paso por paso y avance por proceso. Para el programa ejemplo que se ha estado considerando, la sesión muestra de la figura 6.8 indica que la simulación es adecuada para el primer objeto. El problema es con el segundo objeto. Así la solución apropiada sería avanzar paso a paso en las llamadas al método hasta la línea 23 en la clase `ControladorRaton2`. Después, al llegar a la línea 24 en la misma clase, avanzar por proceso en las llamadas a métodos con el cero que provoca el problema.

Rastree en papel la emulación de la depuración IDE.

6.8 Diagramas UML de clase

El método `crecer` de la clase `Raton` no es muy flexible: fuerza al controlador a llamar al método `crecer` de manera separada por cada día o a proporcionar un ciclo de tiempo para cada simulación de días múltiples. No es un buen estilo incluir ese tipo de cosas en un controlador. Es mejor incluir funcionalidad para múltiples días dentro de la clase controladora. En esta sección se hará justamente eso. Se presentará una clase revisada de `Raton` con un método `crecer` que maneja cualquier número de días, no sólo uno.

Organice.

Para especificar una segunda generación de la clase ratón (`Raton2`) y su clase controladora asociada (`ControladorRaton2`), se creará otro diagrama UML de clase. El diagrama que se presentó en la figura 6.3 es un diagrama UML de clase reducido. No incluía todas las características estándar. Esta vez, en la figura 6.12 se tiene un diagrama UML de clase que incluye todas las características estándar, más una característica extra.

El diagrama de clase de la figura 6.12 incluye cajas de diagrama de clase: un diagrama para la clase `ControladorRaton2` y otro diagrama para la clase `Raton2`. La clase `Raton2` tiene las mismas tres variables de instancia que la clase `Raton` original: `edad`, `peso` y `tasaCrecimientoPorcentual`. También tiene el mismo método `setTasaCrecimientoPorcentual`, pero los métodos `getEdad` y `getPeso` son nuevos y el método `crecer` está mejorado. El método `getEdad` devuelve la edad del ratón. Recuerde que la variable edad es privada, por lo que la única manera de que pueda ser leída en el exterior es utilizando un método `público`: el método `getEdad`. El método `getPeso` devuelve el peso

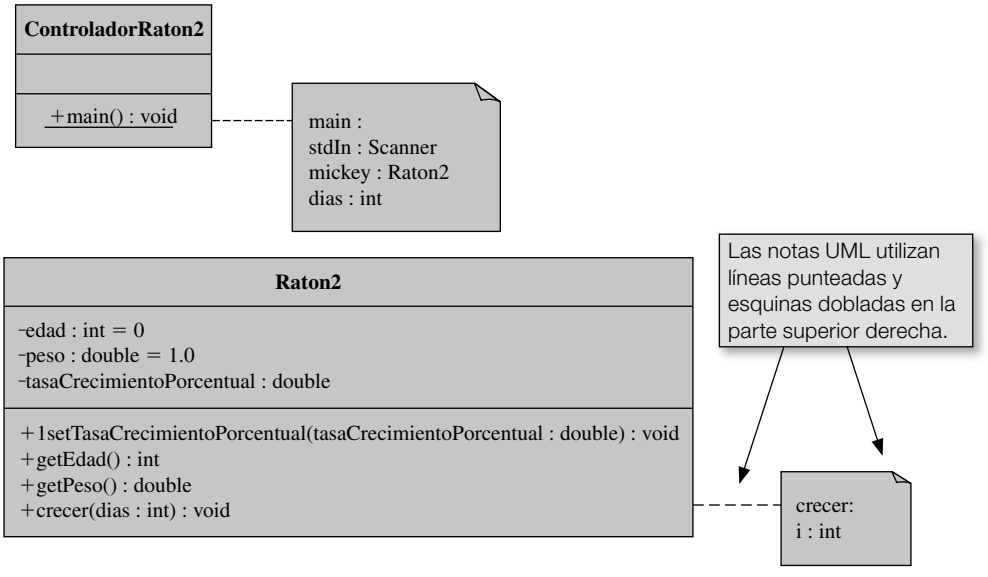


Figura 6.12 Un diagrama de clase UML para un programa Raton de segunda generación.

de un ratón. El método `crecer` simula el crecimiento de un ratón en un número de días específico. Observe el parámetro `dias`. El número de días se pasa en el parámetro `dias` y así es como el método sabe cuántos días tiene que simular.

He aquí algunas de las características de los diagramas estándar UML de clase no encontradas en la figura 6.3 y que sí aparecen en la figura 6.12:

- Para especificar la accesibilidad de un miembro, colocar un símbolo "−" para acceso tipo `private` o "+" para acceso tipo `public`. Las variables de instancia tienen prefijos "−" puesto que se desea que sean `privadas` y los métodos tienen prefijos "+" porque se desea que sean `públicas`.
- Para especificar inicialización, agregue "= <*valor*>" a cada declaración de variable que incluya inicialización. Por ejemplo, observe el "= 0" después de la especificación de la variable de instancia `edad`.
- Subraye el método `main` en el cuadro del diagrama de clase `ControladorRaton`, ya que el método `main` es declarada con el modificador `static`. Los estándares de UML sugieren que se deben subrayar todos los métodos y variables que sean declaradas con el modificador `static`. Como se aprendió en el capítulo 5, el modificador `static` indica un miembro de clase. Se estudiarán más los miembros de clase en el capítulo 9.
- Incluir un sufijo: "<*type*>" en cada método. Esto especifica el tipo de valor que el método devuelve. Todos los métodos en la clase Raton en la figura 6.4 devuelven `void` (nada), pero en el capítulo 5 se vieron muchos métodos de la clase API de Java que devuelven tipos como `int` y `double`, y sobre la implementación de los mismos se hablará más tarde en el presente capítulo.

La figura 6.12 también incluye una característica extra en los diagramas UML de clases. Tiene *notas* para dos de sus métodos: `crecer` y `main`. Las notas son representadas mediante rectángulos con las esquinas superiores derechas dobladas. ¿Por qué con esquinas dobladas? Porque se supone que dan la impresión de una pieza de papel con las esquinas dobladas, como indicación de nota en copia de papel. La inclusión de notas en diagramas UML de clases es totalmente opcional. Por lo regular no se utilizan, pero esta vez se emplearon porque se quiso demostrar cómo se pueden incluir variables locales en un diagrama UML de clases.

6.9 Variables locales

Una *variable local* es una variable que es declarada y utilizada "localmente" dentro de un método. Esto es diferente de las variables de instancia, que son declaradas en la parte superior de la clase, fuera de los métodos. Como podrá darse cuenta, todas las variables que se definieron en los capítulos anteriores eran variables locales y eran declaradas dentro del método `main`, por lo que eran variables locales dentro de dicho método. No se había presentado la necesidad de explicar el término "variable local" hasta ahora porque no había otros métodos además de `main` y la idea de una variable que fuera local a `main` no tendría mucho sentido; pero el contexto de la POO hace más significativo el concepto de una variable local.

Alcance

Una variable local tiene *alcance local*: puede utilizarse sólo a partir del punto en que es declarada y hasta el final del bloque de la variable. El *bloque* de una variable se establece por el par de llaves más cercanas que encierran la declaración de la variable. La mayor parte del tiempo se deben declarar variables locales de un método en la parte alta del cuerpo del método. El alcance de tal variable es por tanto el cuerpo entero del método.

Las variables de índice del ciclo `for` son locales, pero son especiales. Su alcance está determinado de una manera ligeramente diferente de lo que se acaba de señalar. Como se vio en el capítulo 4, normalmente debe declararse una variable índice para el ciclo `for` dentro del encabezado de un ciclo de este tipo. El alcance de tal variable es el encabezado del ciclo `for` más el cuerpo del mismo ciclo.

Los parámetros de un método usualmente no se consideran como variables locales, pero son muy similares a éstas en el sentido de que son declaradas y utilizadas "localmente" dentro del método. Al igual que con las variables locales, el alcance de los parámetros en un método está limitado únicamente al cuerpo de ese método.

Se complementará el tema del alcance haciendo una comparación del alcance local utilizado por las variables de instancia. Mientras las variables con alcance local pueden ser accedidas únicamente dentro de un método en particular, las de instancia pueden ser accedidas dentro de la clase a la cual pertenecen. Por otro lado, si una variable de instancia es declarada con el modificador de acceso `public`, entonces puede ser accedida desde fuera de la clase de la variable de instancia (con la ayuda de un objeto instanciado desde la clase de la variable de instancia).

Clase `ControladorRaton2`

Para ilustrar los principios de las variables locales se presenta el programa `Raton2` en las figuras 6.13 y 6.14. El código incluye número de líneas para facilitar el rastreo en un ejercicio al final del capítulo. El método `main` en la clase `ControladorRaton2` tiene tres variables locales: `stdIn`, `mickey` y `dias`. Éstas aparecen en la nota del diagrama UML de clase de la figura 6.12, y aparecen como declaraciones en el método `main` en la figura 6.13.

Ahora se examinará la clase `ControladorRaton2` de la figura 6.13. En la llamada al método `setTasaCrecimientoPorcentual` observe que se pasa una constante, 10, en lugar de una variable. Normalmente, se utilizarán variables para los argumentos, pero este ejemplo muestra que es válido utilizar constantes también. Después de establecer la tasa porcentual de crecimiento, se solicita al usuario que introduzca el número de días del crecimiento simulado, y después se pasa el valor de los `días` al método `crecer`. Después, se imprime la `edad` de `mickey` insertando las llamadas a los métodos `getEdad` y `getPeso` dentro de la sentencia `printf`.

Clase `Raton2`

Ahora se analizará la clase `Raton2` de la figura 6.14. ¿Existe alguna variable local ahí? Las variables `edad`, `peso` y `tasaCrecimientoPorcentual` son variables de instancia, no locales, porque son declaradas fuera de todos estos métodos en la parte superior de la clase. Dentro del método `crecer` se resalta este hecho precediendo cada una de estas variables de instancia con la referencia `this`. El mé-

```
1  /*******************************************************
2  * ControladorRaton2.java
3  * Dean & Dean
4  *
5  * Éste es un controlador para la clase Raton2.
6  *******************************************************/
7
8  import java.util.Scanner;
9
10 public class ControladorRaton2
11 {
12   public static void main(String[] args)
13   {
14     Scanner stdIn = new Scanner(System.in);
15     Raton2 mickey = new Raton2();
16     int dias;
17
18     mickey.setTasaCrecimientoPorcentual(10);
19     System.out.print("Introduzca el número de días a crecer: ");
20     dias = stdIn.nextInt();
21     mickey.crecer(dias);
22     System.out.printf("Edad = %d, peso = %.3f\n",
23       mickey.getEdad(), mickey.getPeso());
24   } // fin del main
25 } // fin de ControladadorRaton2
```

variables locales

Figura 6.13 Clase `ControladorRaton2` que controla a la clase `Raton2` de la figura 6.14.

```
1  /*****************************************************************
2  * Raton2.java
3  * Dean & Dean
4  *
5  * Esta clase modela un ratón para un programa de simulación de crecimiento.
6  *****************************************************************/
7
8  import java.util.Scanner;
9
10 public class Raton2
11 {
12   private int edad = 0;                        // edad en días
13   private double peso = 1.0;                   // peso en gramos
14   private double tasaCrecimientoPorcentual;  // % aumento diario de peso
15
16   //***************************************************************
17
18   public void setTasaCrecimientoPorcentual(double tasaCrecimientoPorcentual)
19   {
20     this.tasaCrecimientoPorcentual = tasaCrecimientoPorcentual;
21   } // fin de setTasaCrecimientoPorcentual
22
23   //***************************************************************
24
25   public int getEdad()
26   {
27     return this.edad;
28   }  // fin de getEdad
29
30   //***************************************************************
31
32   public double getPeso()
33   {
34     return this.peso;
35   } // fin de getPeso
36
37   //***************************************************************
38
39   public void crecer(int dias)
40   {
41     for (int i=0; i<dias; i++)
42     {
43       this.peso +=
44         (0.01 * this.tasaCrecimientoPorcentual * this.peso);
45     }
46     this.edad += dias;
47   } // fin de crecer
48 } // fin de clase Raton2
```

parámetro

parámetro

variable local

Figura 6.14 Clase `Raton2`.

todo `crecer` también incluye una variable local: la `i` en el ciclo `for`. Puesto que la `i` es declarada dentro del ciclo `for`, su alcance está limitado al bloque de dicho ciclo; por lo que sólo es posible leer y actualizar el valor de `i` dentro del ciclo `for`. Si se intenta acceder a la `i` fuera del ciclo se obtiene un error de compilación. El método `crecer` es similar al que se tenía en el anterior programa `Raton`; pero, en esta ocasión, se utiliza el ciclo `for` para simular varios días de crecimiento, en lugar de uno solo. El parámetro `dias` determina cuántas veces se repetirá el ciclo.

Antes se describieron los valores por omisión para las variables de instancia. Ahora se describirán estos valores para las variables locales. Las variables locales contienen *basura* por omisión. Basura significa que el valor de la variable se desconoce: es cualquier cosa que pueda estar en la memoria en el momento en que se crea la variable. Si un programa intenta acceder a una variable que contenga basura, el compilador generará un error de compilación. Por ejemplo, ¿qué pasaría si se eliminara la inicialización `= 0` del encabezado del ciclo `for` en el método `crecer` en la figura 6.14? En otras palabras, suponga que el ciclo `for` fuera reemplazado por esto:

```
for (int i; i
{
  this.peso +=
    (0.01 * this.tasaCrecimientoPorcentual * this.peso);
}
```

Puesto que a `i` nunca se le asigna cero, cuando se prueba la condición `i<dias`, contiene basura. Si intentara compilar el código con una sentencia como ésta, no compilaría, y el compilador reportaría el siguiente mensaje:

```
variable i might not have been initialized
(la variable i podría no haber sido inicializada)
```

Persistencia de variables locales

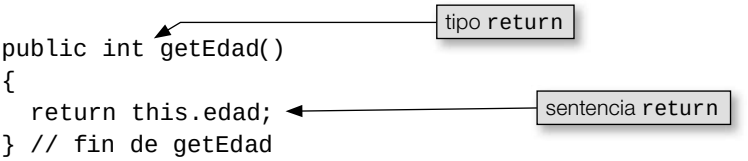
Bien, digamos que se inicializa una variable local. ¿Cuánto *persistirá*? Una variable local (o parámetro) persiste únicamente dentro de su alcance y sólo mientras dure el método (o ciclo `for`) en el que fue definida. La siguiente vez que el método (o ciclo `for`) es llamado, el valor de la variable local regresa al valor que obtiene cuando es inicializada. La línea horizontal de rastreo, que se presenta después de que el método finaliza, nos recuerda que la terminación del método convierte todas las variables locales del método en basura.

6.10 La sentencia `return`

Si revisa la clase `Raton` original en las figuras 6.4 y 6.10, notará que cada encabezado del método tiene un modificador `void` localizado a la izquierda del nombre del método. Eso significa que el método no devuelve ningún valor, y se dice que "el método devuelve un tipo `void`", o más sencillo, "es un método `void`". Pero recuerde del capítulo 5, que muchos métodos API de Java devuelven algún tipo de valor, y que en cada caso el tipo de valor devuelto es indicado por un tipo apropiado en el encabezado del método localizado a la izquierda del nombre del método.

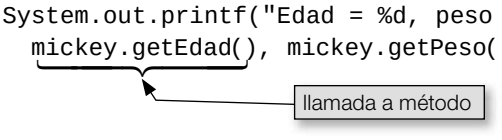
Devolución de un valor

Si se observa la clase `Raton2` de la figura 6.14 se verá que dos de los métodos devuelven un tipo diferente del de `void`. He aquí uno de esos métodos:




```
public int getEdad()
{
  return this.edad;
} // fin de getEdad
```

La sentencia `return` en este método siempre permite pasar un valor desde el método hasta el lugar en el que fue llamado el método. En este caso, el método `getEdad` devuelve el valor de `edad` a la sentencia `printf` de la clase `ControladorRaton2` de la figura 6.13. He aquí la sentencia una vez más:




```
System.out.printf("Edad = %d, peso = %.3f\n",
  mickey.getEdad(), mickey.getPeso());
```

En efecto, la JVM "asigna" el valor devuelto (this.edad) al método que realiza la llamada (mickey.getEdad()). Para ejecutar un rastreo mental, imagine que la llamada al método es revestida con el valor devuelto. Así, si la edad de Mickey es 2, entonces devuelve 2, y se puede reemplazar el método getEdad con el valor 2.

Siempre que el encabezado de un método sea diferente de void, el método debe devolver un valor a través de la sentencia return, y el tipo de valor debe concordar con el especificado en el encabezado del método. Por ejemplo, el encabezado del método getEdad especifica un número tipo int. La sentencia return dentro del método getEdad devuelve this.edad. En la figura 6.14, la variable de instancia edad fue declarada de tipo int, con lo que concuerda con el tipo devuelto por getEdad, por lo que todo está bien. Es adecuado tener una expresión después de la sentencia return; no se está limitado a tener una simple variable; pero la expresión debe evaluarse al tipo devuelto por el método. Por ejemplo, ¿sería válido lo siguiente?

```
return this.edad + 1;
```

Sí, porque this.edad + 1 se evalúa a un tipo int, y esto concuerda con el tipo devuelto por el método getEdad.

Cuando un método incluye una rama condicional (con una sentencia if o switch) es posible devolver el valor de más de un lugar en el método. En tales casos, todos los valores devueltos deben concordar con el tipo especificado en el encabezado del método.

Sentencia return vacía

Para métodos con un tipo return void es válido tener una sentencia return *vacía*. La sentencia return aparecería como a continuación:

```
return;
```

Las sentencias vacías return hacen lo que se esperaría. Terminan el método actual y provocan que el control regrese al módulo que realizó la llamada al punto que sigue inmediatamente después de la llamada al método. He aquí una pequeña variación del método anterior crecer que utiliza una sentencia return vacía:

```
public void crecer(int dias)
{
  int edadFinal = this.edad + dias;

  while (this.edad < edadFinal)
  {
    if (this.edad >= 100)
    {
      return;   <-- sentencia return vacía
    }
    this.peso +=
      .01 * this.tasaCrecimientoPorcentual * this.peso;
    this.edad++;
  }  // fin del while
} // fin de crecer
```

En esta variación del método crecer se recortó el proceso de envejecimiento a 100 días (después de la "adolescencia") verificando edad dentro del ciclo y saliendo cuando el valor sea mayor o igual a 100. Observe las sentencias return. Puesto que no se devuelve ningún tipo de valor, el encabezado de la sentencia debe especificar void como el tipo de valor return.

No es válido tener una sentencia return vacía y una no vacía dentro del mismo método. ¿Por qué? Las sentencias return vacías y no vacías devuelven tipos de datos diferentes (para una sentencia vacía se utiliza void y cualquier otro tipo para una sentencia return no vacía). No hay manera de especificar un tipo en el encabezado del método que de manera simultánea concuerde con dos tipos de datos devueltos.

La sentencia `return` vacía es una sentencia útil en la cual se proporciona una forma fácil de salir rápidamente de un método. Sin embargo, no proporciona una funcionalidad única. El código que utiliza una sentencia `return` vacía siempre podrá ser reemplazada por un código que carezca de sentencias `return`. Por ejemplo, he aquí una versión sin `return` de la versión anterior del método `crecer`.

```
public void crecer(int dias)
{
  int edadFinal = this.edad + dias;

  if (edadFinal > 100)
  {
    edadFinal = 100;
  }
  while (this.edad < edadFinal)
  {
    this.peso +=
      .01 * this.tasaCrecimientoPorcentual * this.peso;
    this.edad++;
  }  // fin del while
} // fin de crecer
```

Sentencias `return` dentro de un ciclo

A los programadores en la industria a menudo se les solicita mantener (arreglar y mejorar) el código de otras personas. Para hacer esto, a menudo tienen que examinar los ciclos y, de manera más específica, la terminación de las condiciones de éstos en el programa en el que están trabajando. Por tanto, es importante que las condiciones de terminación de ciclos sean claras. Normalmente, las condiciones de terminación de ciclos aparecen en la sección estándar de la condición del ciclo. Para los ciclos `while`, esto es el encabezado, para los ciclos `for`, el segundo componente del encabezado, y para los ciclos `do`, es al cierre. Sin embargo, una sentencia `return` dentro de un ciclo resulta en una condición de terminación del ciclo que no está en una ubicación estándar. Por ejemplo, en el método `crecer` de la página anterior, la sentencia `return` está dentro de una sentencia `if`, y la condición de terminación del ciclo está consecuentemente "oculta" en la condición de la sentencia `if`.

Con el interés de lograr el mantenimiento del código se deben establecer restricciones cuando se considere el uso de una sentencia `return` dentro de un ciclo. Con base en el contexto, si la inserción de sentencias `return` dentro de un ciclo mejora la claridad, entonces hay que sentirse con confianza para realizar dicha inserción. Sin embargo, si ello simplemente facilita la tarea de codificación y no le agrega claridad, entonces no debe insertarse. Así pues, ¿cuál implementación de `crecer` es mejor: la de la versión con el `return` vacío o la versión sin `return`? En general, se prefiere la versión sin `return` por razones de mantenimiento. Sin embargo, debido a que el código en ambos de los métodos `crecer` es tan simple, no hay mucha diferencia aquí.

6.11 Paso de argumentos

En la sección anterior se vio que cuando un método termina, la JVM de manera efectiva asigna el valor devuelto al método que realiza la llamada. Esta sección describe una transferencia similar en la otra dirección. Cuando un método es llamado, la JVM de manera efectiva asigna el valor de cada argumento a la sentencia que realiza la llamada al parámetro correspondiente en el método llamado.

Ejemplo

Examinemos el paso de argumentos analizando un ejemplo: otra versión del programa `Raton` llamado `Raton3`. He aquí el código para esta nueva versión del controlador:

```
public class ControladorRaton3
{
```

```
  public static void main(String[] args)
  {
    Raton3 minnie = new Raton3();
    int dias = 365;

    minnie.crecer(dias);
    System.out.println("Edad en # de días = " + dias);
  } // fin del main
} // fin de ControladadorRaton3
```

La JVM hace una copia del valor de `dias` y lo pasa al método `crecer`.

La clase `ControladorRaton3` llama al método `crecer` con un argumento llamado `dias`, cuyo valor parece ser 365. Después asigna este valor (365) al parámetro llamado `dias` en el método `crecer`. El siguiente código muestra lo que le pasa al parámetro `dias` dentro del método `crecer`:

```
public class Raton3
{
  private int edad = 0;                              // edad en días
  private double peso = 1.0;                         // peso en gramos
  private double tasaCrecimientoPorcentual = 10; // % aumento diario de peso
  public void crecer(int dias)
  {
    this.edad += dias;
    while (dias > 0)
    {
      this.peso +=
        .01 * this.tasaCrecimientoPorcentual * this.peso;
      dias--;
    }
  } // fin crecer
} // fin de la clase Raton3
```

La JVM asigna el valor pasado al parámetro `dias`.

El parámetro `dias` disminuye a 0.

Dentro de un método, los parámetros son tratados como variables locales. La única diferencia es que una variable local es inicializada dentro del método, mientras que un parámetro es inicializado por un argumento en la llamada al método. Como puede verse en el cuerpo del ciclo anterior, el parámetro `dias` disminuye hasta llegar a cero. ¿Qué sucedió con la variable `dias` en el método `main` de `ControladorRaton3`? Debido a que las dos variables `dias` son distintas, la variable `dias` en el método `main` no cambia con el parámetro `dias` en el método `crecer`. Así, cuando `ControladorRaton3` imprime su versión de `dias`, imprime un valor sin cambio de 365, como éste:

```
Edad en # de días = 365.
```

Paso por valor

Se dice que Java utiliza el *paso-por-valor* para su esquema de paso-de-argumentos. Como se muestra en la figura 6.15, el paso por valor significa que la JVM pasa una copia del valor del argumento (no el argumento mismo) al parámetro. El cambiar la copia no cambia el original.

En `ControladorRaton3` y `Raton3` observe que la llamada al argumento del método se denomina `dias` y el parámetro del método `crecer` es llamado `dias` también. ¿Es el parámetro la misma variable que el argumento? ¡No! Son variables encapsuladas de manera separada en bloques de código separados. Debido a que estas dos variables están en bloques de código separados, no hay conflicto, y es adecuado darles a ambas el mismo nombre. La utilización del mismo nombre es natural porque estas dos variables describen el mismo tipo de cosas. Ésa es una de las bondades del encapsulamiento. Los programas grandes serían una verdadera pesadilla si se prohibiera utilizar el mismo nombre en diferentes bloques de código.

Mismo nombre *versus* nombres diferentes para pares de argumentos-parámetros

La mayor parte del tiempo se querrá utilizar el mismo nombre para un par argumento-parámetro; pero hay que asegurarse de que la utilización de nombres distintos es válida y bastante común. Cuando es más

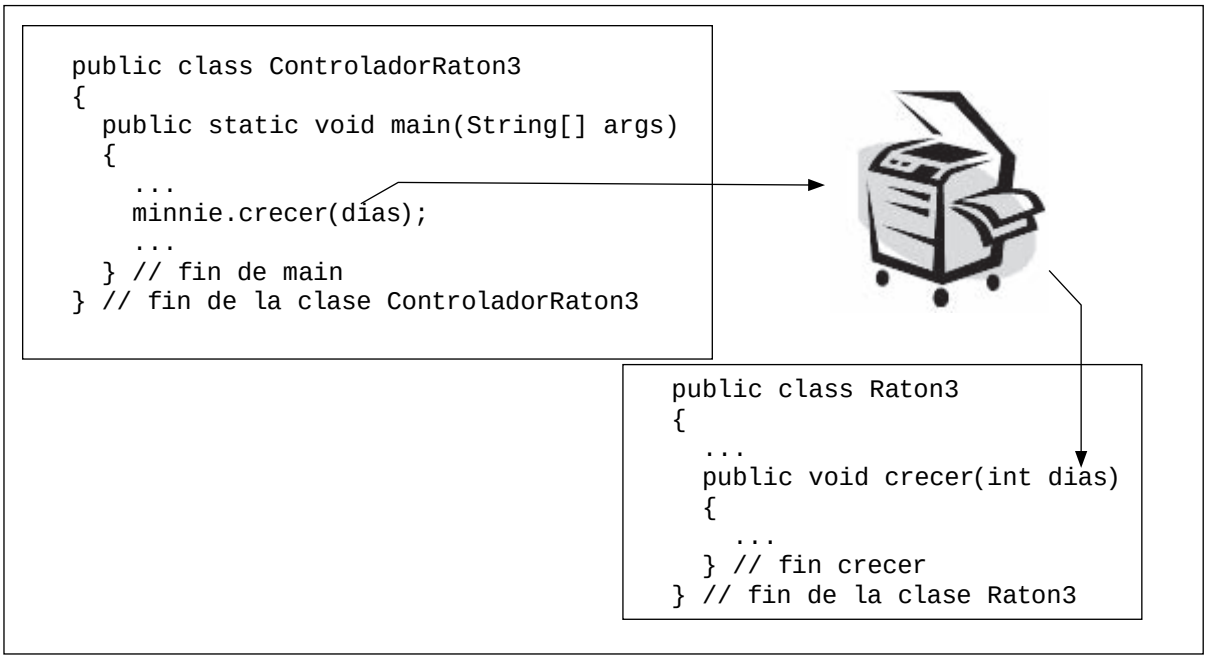


Figura 6.15 El paso-por-valor significa que una copia del valor del argumento va al parámetro correspondiente.

natural y razonable utilizar nombres diferentes para un par argumento/parámetro, entonces se recomienda utilizar nombres diferentes. El único requisito es que el tipo de argumento debe concordar con el del tipo del parámetro. Por ejemplo, en el programa Raton3, si `num` es una variable tipo `int`, entonces la siguiente llamada al método pasa de manera exitosa el valor de `num` al parámetro tipo `int dias`:

```
minnnie.crecer(num);
```

6.12 Métodos especializados: de acceso, de mutación y boolean

Ahora se hablará sobre algunos de los tipos comunes de métodos especializados. No se le solicitará que aprenda una nueva sintaxis, sólo se le solicitará que aplique lo que ha aprendido hasta ahora.

Métodos de acceso

Un método de *acceso* es un método que trae una parte de los datos almacenados de un objeto: típicamente datos `privados`. Observe los siguientes métodos `getEdad` y `getPeso` (tomados de la clase `Raton2` de la figura 6.14). Son métodos de acceso puesto que traen los valores de variables de instancia, `edad` y `peso`, respectivamente.

```
public int getEdad()
{
  return this.edad;
} // fin de getEdad

public double getPeso()
{
  return this.peso;
} // fin de getPeso
```

Como lo demuestran los métodos `getEdad` y `getPeso`, los métodos de acceso deben ser nombrados con el prefijo "*get*". Esto es porque los métodos de acceso son llamados a menudo *métodos get* (métodos de obtención).

Un método debe ejecutar una tarea. Debe ser escrito de tal manera que realice sólo la tarea que su nombre implique. Por ejemplo, un método `getEdad` simplemente debe devolver la variable de instancia `edad` y nada más. Se menciona esto porque existe a veces la tentación de brindar funcionalidad extra en un método y evitar tener que implementarla en otra parte. Una particularidad del *faux pas* (un término francés que significa error en etiqueta) es agregar una sentencia de impresión a un método que no requiere imprimir nada. Por ejemplo, un programador novato podría implementar el método `getEdad` de la siguiente manera:

```
public int getEdad()
{
  System.out.println("Edad = " + this.edad);
  return this.edad;
} // fin de getEdad
```

sentencia de impresión inapropiada.

Ese método `getEdad`, que toma en cuenta la sentencia no estándar de impresión, podría funcionar bien para el programa de un programador novato; pero, si después otro programador necesita trabajar con el programa y llamar al método `getEdad`, se sorprendería de encontrar una sentencia no estándar de impresión. El programador tendría entonces que: 1) acomodar la sentencia de impresión, o 2) eliminar el método `getEdad` y verificar el efecto dominó. Para evitar este escenario, es importante incluir sentencias de impresión en un método, únicamente si el objetivo de éste es imprimir algo.

La excepción a la regla anterior es que es aceptable y de ayuda el agregar sentencias de impresión a los métodos cuando se esté tratando de depurar un programa. Por ejemplo, si se piensa que hay algo mal en el método `getEdad` se podría querer agregar la sentencia de impresión para verificar las correcciones al valor `edad` antes de que éste sea devuelto por el método `getEdad`. Si se agregan esas sentencias de impresión para depurar, no hay que olvidar quitarlas más tarde, ya que el programa esté funcionando.

Depure con sentencias de impresión temporales.

Métodos de mutación

Un método de *mutación* es aquel que cambia o "muta" el estado de un objeto cambiando todos o parte de los datos de éste: típicamente datos `privados`. Por ejemplo, he aquí el método de mutación para establecer o cambiar la variable de instancia `tasaPorcentualCrecimiento` de un ratón:

```
public void setTasaPorcentualCrecimiento(double tasaPorcentualCrecimiento)
{
  this.tasaPorcentualCrecimiento = tasaPorcentualCrecimiento;
} // fin de setTasaPorcentualCrecimiento
```

Como se muestra en el método `setTasaPorcentualCrecimiento`, los métodos de mutación deben ser nombrados con el prefijo "set". Es por ello que este tipo de métodos se denominan *métodos set*.

Un método de acceso permite leer una variable de instancia de tipo `private`. Un método de mutación permite actualizar una variable de instancia `privada`. Si se proporciona una variable de instancia tipo `private` tanto con un método de acceso como con un simple método de mutación como el método `setTasaPorcentualCrecimiento` anterior, efectivamente convierte esa variable de instancia `privada` en una variable de instancia `public`, y elimina el encapsulamiento de esa variable. No hay mucho peligro al tener un único método de acceso, pero tener un simple método de mutación permite a un método externo introducir un valor no razonable que pueda producir una operación de programa errónea. Sin embargo, si se incluye una verificación de restricción y quizá corrige el código en los métodos de mutación, éstos pueden servir como *filtros* de datos que asignan únicamente datos adecuados a las variables de instancia `privadas`. Por ejemplo, he aquí un método de mutación `setTasaPorcentualCrecimiento` que filtra las tasas de crecimiento que son menores a −100 por ciento.

Utilice métodos de mutación para filtrar datos de entrada.

```
public void setTasaPorcentualCrecimiento(double tasaPorcentualCrecimiento)
{
  if (tasaPorcentualCrecimiento < -100)
  {
    System.out.println("Intentó asignar una tasa de crecimiento inválida.");
```

```
    }
    else
    {
      this.tasaPorcentualCrecimiento = tasaPorcentualCrecimiento;
    }
  } // fin de setTasaPorcentualCrecimiento
```

Los ejemplos de este libro ocasionalmente incluirán algún tipo de verificador de error de mutación para ilustrar esta función de filtrado, pero para disminuir el desorden, de manera usual se empleará la forma reducida.

Métodos boolean

Un *método boolean* verifica si hay alguna condición verdadera o falsa. Si la condición es verdadera, entonces devuelve `true` (verdadero); de lo contrario, devuelve `false` (falso). Para acomodar el valor `boolean` devuelto, los métodos tipo boolean siempre deben especificar el tipo que devuelven. El nombre de un método tipo boolean normalmente debe empezar con un "es". Por ejemplo, he aquí un método `esAdolescente` que determina si un objeto `Raton` es un adolescente comparando el valor de su `edad` con 100 días:

```
public boolean esAdolescente()
{
  if (this.age <= 100)
  {
    return true;
  }
  else
  {
    return false;
  }
} // fin de esAdolescente
```

Y a continuación, la manera en que el código podría simplificarse:

```
public boolean esAdolescente()
{
  return this.age <= 100;
} // fin de esAdolescente
```

Para mostrar cómo funciona la simplificación del método, se insertarán algunos valores de muestra; pero primero hay que concentrarse en la meta: cuando el valor de `edad` sea menor o igual a 100, se desea que el método devuelva `verdadero` para indicar adolescencia. Si `edad` es 50, ¿qué valor devolvería? `true` (porque la sentencia `this.age <= 100` se evalúa como `verdadera`). Si `edad` es 102, ¿qué valor devolvería? `false` (porque la sentencia `this.age <= 100` se evalúa como `falsa`). Asigne cualquier número a `edad` y verá que la función simplificada funciona correctamente. En otras palabras, el método simplificado `esAdolescente` devuelve `true` cuando `edad` es menor o igual a 100.

¿Le extraña la falta de paréntesis alrededor de la expresión de devolución de valor `return`? En sentencias que utilizan una condición (sentencia `if`, sentencia `while`, etc.), la condición debe encerrarse entre paréntesis. En la expresión de la sentencia `return`, el paréntesis es opcional. Ambas formas las encontrará en la industria: a veces se incluye el paréntesis y a veces se omite.

He aquí cómo puede utilizarse el método `esAdolescente` en un módulo de llamada:

```
Raton pinky = new Raton();
. . .
if (pinky.esAdolescente() == false)
{
  System.out.println("El crecimiento del ratón no" +
    " será simulado más tiempo: demasiado viejo.");
}
```

¿Sabe cómo simplificar la sentencia `if` anterior? He aquí sentencia `if` con una funcionalidad equivalente con una condición mejorada:

```
if (!pinky.esAdolescente())
{
  System.out.println("El crecimiento del ratón no" +
    " será simulado más tiempo: demasiado viejo.");
}
```

La meta es imprimir un mensaje de advertencia en el caso en que `pinky` sea un anciano (no adolescente). Si el método `esAdolescente` devuelve `false` (como indicación de un Pinky anciano), entonces la condición `if` es `verdadera` (`!false` se evalúa como `true`) y el programa imprime el mensaje de advertencia. Por otro lado, si `esAdolescente` devuelve `true` (como indicación de un joven Pinky), entonces la condición de la sentencia `if` es `falsa` (`!true` se evalúa como `false`) y el programa se salta el mensaje de advertencia.

A pesar de que la versión simplificada de la sentencia `if` podría ser difícil de entender en un principio, los programadores experimentados la preferirían. Bajo ese principio se sugiere el uso de ! en lugar de `==false` para situaciones similares.

6.13 Resolución de problemas con simulación (opcional)

En los ejemplos anteriores del ratón, para mantener la atención en los conceptos de POO, en lugar de los detalles de crecimiento del ratón, se utilizó una sencilla fórmula para el crecimiento. En esta sección se muestra cómo simular el crecimiento de tal forma que sea más cercano a aquel que ocurre en el mundo real. Después se muestra un simple truco que puede ser aplicado en muchos problemas de simulación para mejorar ampliamente la velocidad y seguridad del programa.

Antes, se modeló el crecimiento asumiendo que el peso agregado es proporcional al peso, como a continuación:

pesoAgregado = *fraccionTasaCrecimiento* × *peso*

Donde

fraccionTasaCrecimiento = .01 × *tasaPorcentualCrecimiento*

Esta clase de crecimiento hace que el peso se incremente exponencialmente y que la curva de crecimiento continúe en el tiempo, como se indica en la figura 6.16. Ésta es una buena aproximación para una planta o un animal jóvenes, donde la mayor parte de la energía proveniente de la ingesta de alimentos se convierte en nuevo crecimiento.

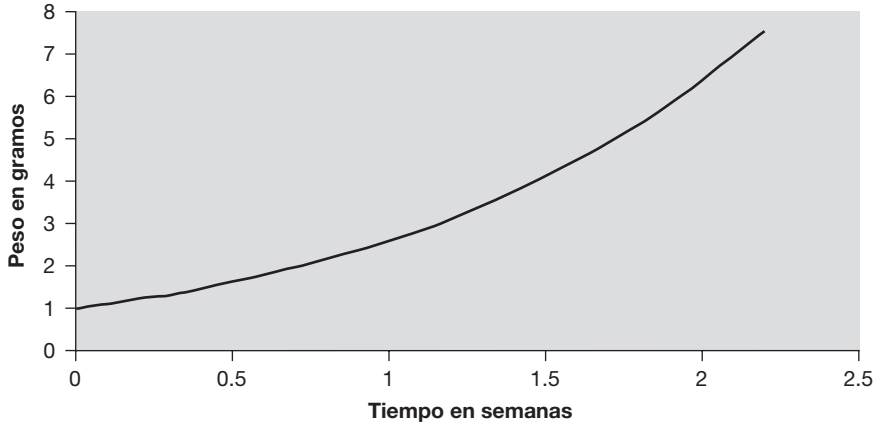


Figura 6.16 Crecimiento exponencial.

Maduración

Existe, no obstante, un problema con el modelo de crecimiento exponencial: ¡nada se mantiene en crecimiento por siempre! Después de un tiempo, el viejo tejido comienza a morir, y algunos de los nutrientes ingeridos deben ser utilizados para reemplazar al viejo tejido, en lugar de agregarlo. Esto hace más lento el crecimiento. Puesto que una fracción más grande de nutrientes ingeridos va reemplazándose, la línea recta de la curva de crecimiento comienza a inclinarse hacia el lado opuesto, y se aproxima a un máximo. La manera más sencilla de modificar la fórmula básica exponencial de crecimiento, para hacerla describir maduración, es multiplicarla por otro factor para obtener lo que se llama *ecuación logística*.

$$pesoAgregado = fraccionTasaCrecimiento \times peso \times \left(1.0 - \frac{peso}{pesoMaximo}\right)$$

Una rápida inspección a esta fórmula mejorada de crecimiento muestra que como `peso` se aproxima a `pesoMaximo`, la cantidad entre paréntesis a la derecha se aproxima a cero, y por tanto el peso agregado a la izquierda se aproxima a cero. En este punto, no hay más crecimiento. Esto proporciona una descripción razonable de un organismo alcanzando la madurez.

Las simulaciones computacionales confían en modelos matemáticos aproximados, como el modelo proporcionado por la ecuación logística anterior. Tales modelos de simulación son a veces buenos, a veces no tanto, y es difícil de saber qué tan buenos son comparándolos con datos reales. Pero para el problema actual de aumento de peso, se tiene el lujo de poder comparar el modelo de simulación con un modelo matemático exacto. He aquí una forma cercana a la solución matemática que determina el peso de alguna ocasión determinada:

$$peso = \frac{1.0}{\frac{1.0}{pesoMaximo} + e^{-(fraccionTasaCrecimiento \times ocasión + g_o)}}$$

Esta fórmula contiene una constante de crecimiento g_o, la cual es:

$$g_o = \log_e\left(\frac{pesoMin}{1.0 - \frac{pesoMin}{pesoMax}}\right)$$

La constante g_o puede encontrarse insertando los valores *pesoMin* y *pesoMax* en la segunda fórmula. Después se encuentra *peso* insertando g_o en la primera fórmula.

Simulación

Si se puede describir se puede simular.

Usualmente una solución exacta no está disponible, y la única manera de resolver un problema es con una simulación. Pero para este problema de aumento de peso, se tienen ambas cosas. Echemos un vistazo al programa que despliega a la vez, la solución exacta y la solución simulada juntas. Vea la clase `Crecimiento` en la figura 6.17.

La clase `Crecimiento` tiene tres variables de instancia, `tamañoInicial`, `tamañoFinal` y `fraccionTasaCrecimiento`, y tres métodos. El método `inicializar` inicializa las tres variables de instancia. El método `getTamaño` utiliza la solución matemática cercana a la fórmula proporcionada anteriormente. Devuelve el tamaño (por ejemplo, el peso actual del ratón) en un momento dado. Observe que este nombre de método inicia con "get", así parece el nombre de un método de acceso, y devuelve un valor tipo `double`, tal como lo hace el método `getPeso`. Pero esta clase no tiene una variable de instancia llamada "tamaño". He aquí un ejemplo de un método que no es realmente de acceso como los descritos en la sección 6.12 a pesar de que el nombre lo hace parecer como tal. El punto es: cualquier método puede devolver un valor, no simplemente como un método de acceso, y cualquier método puede tener cualquier nombre que parezca apropiado: `getTamaño` es simplemente el nombre más apropiado que se podría pensar para este método que calcula y devuelve un tamaño.

El método `getIncrementoTamaño` implementa un paso de simulación. Devuelve el cambio en tamaño entre la ocasión actual y la siguiente. Observe que los métodos `getTamaño` y `getIncremento-`

```
/*****************************************************************
* Crecimiento.java
* Dean & Dean
*
* Éste proporciona formas diferentes de calcular crecimiento.
*****************************************************************/

public class Crecimiento
{
  private double tamañoInicio;               // tamaño inicial
  private double tamañoFinal;                // tamaño maximo
  private double fraccionTasaCrecimiento;    // por unidad de tiempo

  //***************************************************************

  public void inicializar(double inicio, double fin, double factor)
  {
    this.tamañoInicio = inicio;
    this.tamañoFinal = fin;
    this.fraccionTasaCrecimiento = factor;
  } // fin de inicializar

  //***************************************************************

  public double getTamaño(double tiempo)
  {
    double g0 = Math.log(tamañoInicio / (1.0 - tamañoInicio / tamañoFinal));

    return 1.0 / (1.0 / tamañoFinal +
      Math.exp(-(fraccionTasaCrecimiento * tiempo + g0)));
  } // fin de getTamaño

  //***************************************************************

  public double getTamañoIncremento(double tamaño, double avanceTiempo)
  {
    return fraccionTasaCrecimiento *
      tamaño * (1.0 - tamaño / tamañoFinal) * avanceTiempo;
  } // fin de getTamañoIncremento
} // fin clase Crecimiento
```

Figura 6.17 Clase `Crecimiento` que implementa diferentes maneras de evaluar crecimiento.

`Tamaño` hacen cosas diferentes. El primero brinda la respuesta directamente. El segundo proporciona un valor de incremento el cual debe ser agregado a la respuesta anterior para obtener la siguiente respuesta.

Si está usted escribiendo su propia clase y quiere modelar el crecimiento de una de las entidades de la clase, podría copiar y pegar las variables y métodos de la clase `Crecimiento` en su clase. De manera alternativa, podría delegar el trabajo en un objeto de la clase `Crecimiento`, tal como lo delegaría en objetos de la clase `Scanner`. Para hacer esto, utilice `new` para instanciar un objeto de la clase `Crecimiento`, inicialícela con los datos relativos al crecimiento en su objeto y después solicite al objeto `Crecimiento` resolver el problema de crecimiento llamando al método `getTamaño` o `getIncrementoTamaño`. En su programa puede utilizar un código similar al del método `main` de la clase `ControladorCrecimiento` de la figura 6.18.

Esta clase controladora podría parecer imponente, pero no es difícil. Se inicia declarando e inicializando las variables locales, y esto incluye crear instancia e inicializar el objeto `Crecimiento`. Posteriormente se le solicita al usuario proporcionar un tiempo de incremento y el número total de incrementos. Por último,

se utiliza un ciclo `for` para imprimir el tiempo, la solución exacta, y la solución simulada para cada paso. Si se ejecuta el programa compuesto del código de las figuras 6.17 y 6.18 se obtendrá este resultado:

Sesión muestra:

```
Introduzca el incremento de tiempo: 1
Introduzca las unidades de tiempo total a simular: 15
         tamaño     tamaño
tiempo   exacto    simulado
 0.0       1.0        1.0
 1.0       2.6        2.0
 2.0       6.4        3.9
 3.0      13.6        7.3
 4.0      23.3       13.3
 5.0      31.7       22.2
 6.0      36.5       32.1
 7.0      38.6       38.4
```

```
/**************************************************************
* ControladorCrecimiento.java
* Dean & Dean
*
* Este programa compara las soluciones exacta y simulada para el crecimiento.
**************************************************************/

import java.util.Scanner;

public class ControladorCrecimiento
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double incTiempo;
    double tiempoMax;
    Crecimiento entidad = new Crecimiento();
    double tamañoInicio = 1.0;             // peso en gramos
    double tamañoFinal = 40.0;             // peso en gramos
    double fraccionTasaCrecimiento = 1.0;  // por unidad de tiempo
    double tamaño = tamañoInicio;

    entidad.inicializar(tamañoInicio, tamañoFinal, fraccionTasaCrecimiento);
    System.out.print("Introduzca el incremento de tiempo: ");
    incTiempo = stdIn.nextDouble();
    System.out.print("Introduzca las unidades de tiempo total a simular: ");
    tiempoMax = stdIn.nextDouble();
    System.out.println("tiempo   exacto   simulado");
    System.out.println("         tamaño     tamaño");

    for (double tiempo=0.0; tiempo<=tiempoMax; tiempo+=incTiempo)
    {
      System.out.printf("%4.1f%8.1f%8.1f\n",
        tiempo, entidad.getTamaño(tiempo), tamaño);
      tamaño += entidad.getTamañoIncremento(tamaño, incTiempo);
    } // fin del for
  } // fin del main
} // fin de la clase ControladorCrecimiento
```

Instancia de objeto `Crecimiento`.

Inicializa objeto `Crecimiento`.

Figura 6.18 Clase `ControladorCrecimiento` que utiliza la clase `Crecimiento` de la figura 6.17.

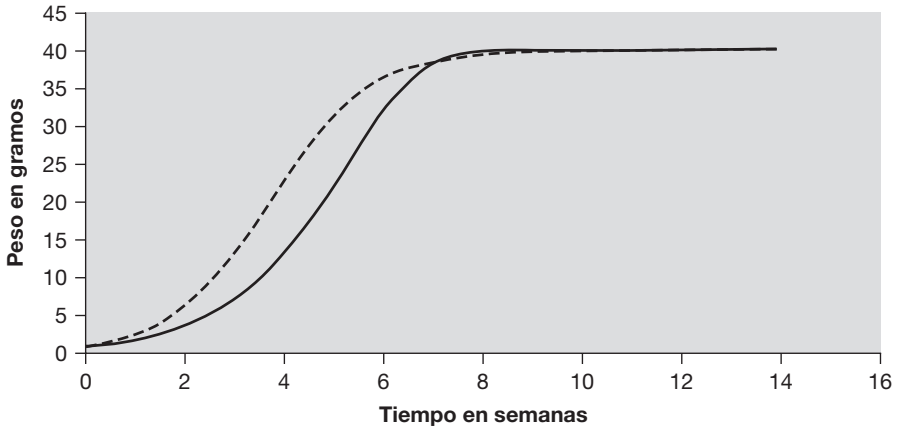


Figura 6.19 Solución simulada con el tiempo de incremento = 1 (línea sólida) comparada con la solución exacta (línea punteada).

```
 8.0       39.5       39.9
 9.0       39.8       40.0
10.0       39.9       40.0
11.0       40.0       40.0
12.0       40.0       40.0
13.0       40.0       40.0
14.0       40.0       40.0
15.0       40.0       40.0
```

La figura 6.19 muestra cómo se verían los datos en una gráfica de dos dimensiones. Por desgracia, la solución simulada no concuerda mucho con la solución exacta. No se eleva lo suficientemente rápido, y después se sobrepasa. La causa de este error es bastante sencilla. Cada incremento de tiempo está basado en el tamaño al inicio del crecimiento; pero conforme pasa el tiempo, el tamaño actual se incrementa, así para todos los datos, excepto para el primer instante en el incremento, el cálculo está utilizando datos viejos.

La forma más sencilla de resolver el problema de precisión es utilizar un ciclo de tiempo más pequeño. Con este algoritmo de simulación, el error es proporcional al tamaño del ciclo de tiempo. Si se recorta el ciclo de tiempo a la mitad, también se recorta el error a la mitad, si se divide el ciclo de tiempo entre 10, esto divide el error entre 10, etc. En la salida anterior, en cuatro semanas la solución exacta indica que el tamaño es de 23.3 gramos, pero la simulación dice que es sólo 13.3 gramos. Ése es un error de 23.3 – 13.3 = 10 gramos. Si se desea reducir este error a menos de un gramo, es necesario reducir el ciclo de tiempo por un factor de 10, aproximadamente.

Si se desconoce la solución exacta, ¿cómo saber cuál es el error? He aquí una regla general: si se quiere tener un error de menos de 1%, hay que asegurarse de que el tamaño que incrementa en cada ciclo de tiempo es siempre menos que 1% del tamaño promedio en dicho intervalo.

Este sencillo algoritmo funciona bien para problemas simples; pero si se tienen problemas difíciles, algunas cosas podrían ser sensibles a errores muy pequeños, y esto podría conducir a la realización de un largo número de pasos muy pequeños. Esto podría tomar más tiempo del que se esperaría. Hay también un problema más insidioso. Aun cuando un número de tipo `double` tiene precisión limitada, y cuando se procesan muchos números, los errores de redondeo se pueden acumular. En otras palabras, conforme se disminuyen los tamaños de los ciclos, los errores inicialmente disminuyen, pero en algún momento comenzarán a incrementarse también.

Precisión mejorada y eficiencia utilizando un algoritmo con un paso intermedio[3]

Elimine las tendencias.

Hay una manera más adecuada de mejorar la precisión. Está basada en un principio simple: En lugar de utilizar la(s) condición(es) (por ejemplo, el peso) al principio del inter-

[3] El nombre formal para este algoritmo es “método de segundo orden de Runge-Kutta”.

```
public double getTamañoIncremento2(double copiaTamaño, double cicloTiempo)
{
  copiaTamaño += getTamañoIncremento(copiaTamaño, 0.5 * cicloTiempo);
  return getTamañoIncremento(copiaTamaño, cicloTiempo);
} // fin de método getTamañoIncremento2
```

No se requiere prefijo, puesto que `getTamañoIncremento` y `getTamañoIncremento2` están en la misma clase.

Figura 6.20 Método que implementa el algoritmo de paso intermedio. Agregue este método al código en la figura 6.17 para mejorar la precisión y la eficiencia.

valo para estimar el (los) cambio(s) durante el intervalo, utilizar la(s) condición(es) en medio del intervalo para estimar el (los) cambio(s) durante el intervalo. Pero, ¿cómo conocer las condiciones en medio del intervalo si no se ha llegado ahí? ¡Realizar una "expedición avanzada"! En otras palabras, hacer un paso intermedio adelantado, y evaluar las condiciones ahí. Después volver al principio y utilizar la(s) condición(es) en el punto intermedio para determinar cuál(es) cambio(s) se realizará(n) por adelantado.

Al principio, esto podría sonar como una forma difícil de hacer una cosa sencilla. ¿Por qué no únicamente recortar el tamaño del ciclo a la mitad y tomar dos ciclos pequeños por adelantado? La respuesta cualitativa es: eso todavía deja una tendencia regular hacia los datos anteriores. La respuesta cuantitativa es: si se utiliza un algoritmo con un paso intermedio para la simulación, el tamaño del error es proporcional al cuadrado del tamaño del ciclo de tiempo. Esto significa que si se reduce el tamaño completo del ciclo de tiempo en un factor de 100, el error disminuirá en un factor de 10 000. En otras palabras, se puede obtener un factor de precisión extra de 100 incrementando el trabajo de la computadora por sólo un factor de 2.

Pero ¿qué trabajo se debe hacer? ¿Qué tan difícil es implementar un algoritmo con un paso intermedio? No mucho. Todo lo que se tiene que hacer es agregar un simple método. Específicamente, a la clase `Crecimiento` en la figura 6.17, sólo agregar el método `getTamañoIncremento2` mostrado en la figura 6.20.

¿Cómo funciona este método? Simplemente manda a llamar al método `getTamañoIncremento` dos veces. Observe que el parámetro `copiaTamaño` en la figura 6.20 es sólo una copia de la variable `tamaño` al principio del incremento de tiempo, y se va sólo medio paso adelante. Después se utiliza el valor devuelto para incrementar `copiaTamaño` al tamaño del punto medio. La segunda llamada a `getTamañoIncremento` utiliza el tamaño de este punto medio calculado y un paso de tiempo completo para determinar el cambio del principio al fin del intervalo completo.

En la definición del método `getTamañoIncremento2`, observe las llamadas a `getTamañoIncremento`. No hay variable de referencia punto prefijo a la izquierda de `getTamañoIncremento`. La razón es que si se llama a un método que está en la misma clase, entonces se puede llamar al método directamente, sin variable de referencia punto prefijo.

El trabajo requerido para modificar el controlador es sencillo. Todo lo que se tiene que hacer es cambiar el nombre del método anterior llamado al del nuevo. En este caso, todo lo que se tiene que hacer es cambiar la última sentencia en el controlador de la figura 6.18 por esto:

```
tamaño += entidad.getTamañoIncremento2(tamaño, cicloTiempo);
```

¡Este '2' agregado es la única diferencia!

La figura 6.21 muestra que el algoritmo mejorado produce un tamaño completo igual al tamaño utilizado en la figura 6.19. Esto toma tanto tiempo de cómputo como en la figura 6.19, pero es claramente más de dos veces tan bueno. Por ejemplo, en cuatro semanas el error es de sólo 1.5 gramos, en lugar de los 10 gramos anteriores.

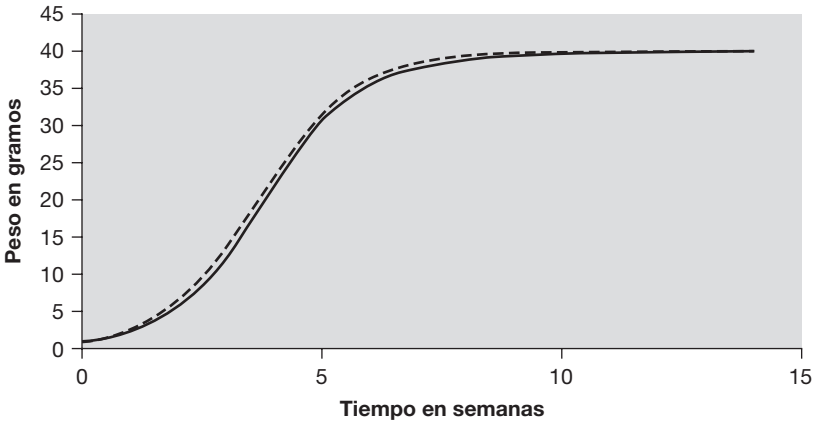


Figura 6.21 Solución con un paso intermedio simulado con incremento en tiempo = 1 (sólido) comparado con la solución exacta (punteada).

Resumen

- Un objeto es un grupo de datos relacionados que identifican la condición actual o *estado* del objeto más los métodos que describen el *comportamiento* del objeto.
- Los objetos son *instancias* de las clases que los definen. Una definición de clase especifica las variables de instancia que un objeto de dicha clase contiene, y define los métodos a que puede llamar ese objeto. Cada objeto contiene su propia copia de las variables de instancia que su clase define, y una variable de instancia dada por lo general tiene diferentes valores en distintos objetos.
- Utilice el modificador de acceso tipo `private` para especificar que una variable particular está encapsulada u oculta. Utilice el modificador de acceso `public` para hacer los métodos accesibles al mundo externo.
- Para hacer a una clase tan general como sea posible, contrólela desde el método `main` en una clase "controladora" separada. En el método `main` del controlador, declare una variable de referencia del tipo de clase controladora. Después, utilice la palabra reservada de Java `new` para instanciar un objeto de la clase controlada e inicializar la variable de referencia con el objeto de referencia devuelto por `new`.
- Utilice la palabra de referencia `this` para referirse al objeto que realiza la llamada desde dentro de los métodos del objeto. Utilice `this` para distinguir una variable de instancia de un parámetro con el mismo nombre de una variable local.
- Cuando se rastrea un programa orientado a objetos, es necesario mantener el rastro de la clase en la que está, en qué método se está, qué objeto llamó al método, los nombres de los parámetros y las variables locales, y los nombres de todas las variables de instancia en cada objeto.
- Un diagrama UML de clase tiene cuadros separados para el nombre de la clase, la descripción de las variables, y los encabezados de los métodos de clase. Utilice un prefijo "+" para `public` (públicas) y un prefijo "-" para `private` (privado). Especifique las variables y los tipos devueltos por el método y los valores iniciales diferentes de los que se tiene por omisión.
- Los valores por omisión de las variables de instancia son cero para números, `false` (falso) para los valores `boolean` y `null` para las referencias. Los valores de las variables de instancia persisten durante la vida de su objeto. Los valores por omisión de las variables locales son basura indefinida. Las variables locales y los parámetros persisten mientras sus métodos se ejecutan y, después de eso, sus valores son indefinidos.
- A menos que el tipo devuelto por una función sea de tipo `void`, cada ruta a través del método debe terminar con una sentencia que devuelva un valor del tipo del método.
- Un parámetro de un método debe tener el mismo tipo que el del argumento de la llamada al método. Lo que el método recibe es una copia de lo que está en el programa que realiza la llamada, por lo que al cambiar un parámetro en un método no cambia el valor del programa que realiza la llamada.
- Utilice métodos `setX` y `getX` para modificar y traer valores de la variable de instancia `private`. Incluya filtros en los métodos `setX` para proteger el programa de entradas erróneas. Utilice métodos

tipo `boolean isX` para devolver `true` (verdadero) o `false` (falso) dependiendo del valor de alguna condición.

- Opcionalmente, mejorar la velocidad de simulación y la precisión mediante el cálculo del siguiente incremento con valores determinados a la mitad entre los puntos de inicio y final de los incrementos.

Preguntas de revisión

§6.2 Introducción a la programación orientada a objetos

1. Una clase es una instancia de un objeto. (F/V)

2. ¿Cuántos objetos puede haber dentro de una simple clase?

§6.3 Primera clase de POO

3. Las variables de instancia de una clase deben ser declaradas fuera de todos ____, y todas las variables de instancia deben ubicarse en la _____.

4. Los métodos accesibles desde afuera de una clase son `públicos`, pero las variables de instancia (aun esas que un externo puede necesitar para cambiar o leer) son usualmente `privadas`. ¿Por qué?

§6.4 Clase controladora

5. ¿Dónde va el método `main`, en la clase controladora o en la clase controlada?

6. Cuando un programa tiene tanto clases controladoras como controladas, ¿dónde debe residir la mayor parte del código del programa?

7. ¿Cómo se trae el valor de una variable de instancia de tipo `private` desde adentro de un método `main`?

8. Una variable de referencia almacena la _____ de un objeto.

§6.5 Objeto llamado, referencia `this`

9. Un método de instancia podría contener una sentencia como `this.peso = 1.0`; pero si esa clase del método actualmente tiene cinco objetos instanciados, hay cinco variables diferentes llamadas `peso`. ¿Cómo se puede determinar cuál está obteniendo el nuevo valor?

§6.6 Variables de instancia

10. ¿Cuáles son los valores por omisión para las variables de instancia de tipo `int`, `double` y `boolean`?

11. En el programa Raton de las figuras 6.4 y 6.5, ¿cuál es la persistencia de la variable `edad` de `gus`?

§6.8 Diagramas UML de clase

12. Después de que un programa es escrito, un diagrama UML de clase proporciona una breve descripción de cada clase en el programa. Ayuda a otras personas a ver qué métodos están disponibles y qué argumentos se necesitan. Proporcione unas razones de por qué podría ser útil tener un diagrama de clase enfrente de uno mismo mientras se implementa la clase y se escriben sus métodos.

§6.9 Variables locales

13. Asuma que el método `main` en `ControladorRaton2` había iniciado de una manera más simple con sólo el `Raton mickey`. ¿Cuál sería el valor de `mickey` inmediatamente después de esta sentencia?

§6.10 La sentencia `return`

14. Usualmente, el uso de múltiples sentencias `return` conduce a un código más entendible. (F/V)

§6.11 Paso de argumentos

15. ¿En qué se parece el parámetro de un método a una variable local y en qué difieren?

16. ¿Cuál es la relación y diferencia entre el argumento de un método y el parámetro de un método?

§6.12 Métodos especializados: de acceso, de mutación y boolean

17. ¿Cuál es el prefijo estándar para un método de acceso?

18. ¿Cuál es el prefijo estándar para un método de mutación?

19. ¿Cuál es el prefijo estándar para un método boolean?

§6.13 Resolución de problemas mediante simulación (opcional)

20. Identifique dos formas generales para reducir el tamaño del error en una simulación. Para una precisión determinada, ¿qué forma es más eficiente?

Ejercicios

1. [Después de §6.2] Suponga que se le solicita modelar plantas utilizando POO. Para cada una de las siguientes entidades relacionadas con plantas, especifique el elemento más apropiado a utilizar para su implementación. Para cada entidad, seleccione entre variable de instancia, objeto, método o clase.
   a) alto de la planta
   b) secuencia de actividades que ocurren cuando una semilla germina
   c) una indicación de si la planta contiene un sistema vascular
   d) una planta individual
2. [Después de §6.3] ¿Cómo se encapsula una variable de instancia en Java?
3. [Después de §6.4] Describa la relación entre el método `main` y las clases controladora y controlada. Proporcione un ejemplo de una clase que corra por sí misma y que no necesite un controlador separado.
4. [Después de §6.4] Objetos envoltorio: las clases envoltorio, vistas en el capítulo 5, también proporcionan la habilidad de crear instancias de objetos que son versiones envoltorio de variables primitivas. Por ejemplo, para crear una versión envoltorio de un número `x` tipo `double`, se puede hacer esto:

```
double x = 55.0;
Double xEnvoltorio = new Double(x)
```

Esto crea una instancia de un objeto de instancia tipo `Double`, el cual es una versión envoltorio de la variable primitiva, `x`. Después asigna una referencia de ese objeto a la variable de referencia `xEnvoltorio`. La clase `Double` tiene un número de métodos preconstruidos que funcionan con objetos `Double`. Estos métodos pueden ser consultados en la documentación de Sun, en la clase `Double`. El siguiente programa ilustra algunos de estos métodos:

```
/*************************************************************
* Envoltorio.java
* Dean & Dean
*
* Este programa pone en práctica el uso de números envoltorio primitivos.
*************************************************************/

public class Envoltorio
{
  public static void main(String[] args)
  {
    double x = 44.5;
    double y = 44.5;
    Double xE = new Double(x);        // el objeto x envoltorio
    Double yE = new Double(y);        // el objeto y envoltorio

    System.out.println("objeto == objeto? " + (xE == yE));
    System.out.println("valor  == valor? " +
      (xE.doubleValue() == yE.doubleValue()));
    System.out.println(
      "object.equals(object)? " + xE.equals(yE));
    System.out.println("objeto.compareTo(objeto)? " +
      xE.compareTo(yE));

    yE = new Double(y + 3.0);
    System.out.println("object.compareTo(largerObject)? " +
      xE.compareTo(yE));

    yE = new Double(Double.NEGATIVE_INFINITY);
    System.out.println("-infinito esInfinito()? " +
      yE.isInfinite());
  } // fin del main
} // fin de la clase Envoltorio
```

Compile y ejecute este programa y despliegue la salida. Consulte acerca de la clase `Double` en la documentación de Sun y explique por qué cada una de las salidas resulta en la forma en que lo hace.

5. [Después de §6.4] Suponga que tiene una clase llamada `Pueblo` que describe la demografía de pueblos pequeños. Las estadísticas esenciales descritas por esta clase son `numeroDeAdultos` y `numeroDeNiños`. Estas estadísticas están encapsuladas y no son directamente accesibles desde afuera de la clase.

a) Escriba los siguientes métodos para la clase `Pueblo`:

i) Un método `inicializar` que establezca valores iniciales de variables de instancia. Asuma que `inicializar` comparte todos los datos de entrada que se requieren solicitar de un usuario.

ii) Un método `nacimientoSimulado` que simule el nacimiento de un niño.

iii) Un método `imprimirEstadisticas` que imprima las estadísticas escenciales.

b) Escriba un método `main` para una clase controladora separada que haga lo siguiente:

i) Cree un pueblo llamado `nuevoHogar`.

ii) Llame a `inicializar` para establecer valores iniciales de variables de instancia para `nuevoHogar`.

iii) Simule el nacimiento de un par de gemelos.

iv) Imprima las estadísticas escenciales de `nuevoHogar`.

6. [Después de §6.7] Dado este programa DiseñoPc:

```
 1 /*******************************************************
 2 * ControladorDiseñoPc.java
 3 * Dean & Dean
 4 *
 5 * Este programa pone en práctica el uso de la clase DiseñoPc.
 6 *******************************************************/
 7
 8 public class ControladorDiseñoPc
 9 {
10    public static void main(String[] args)
11    {
12      DiseñoPc miPc = new DiseñoPc();
13      miPc.asignarTamañoRam();
14      miPc.asignarTamañoDisco();
15      miPc.asignarProcesador();
16      miPc.calcularCosto();
17      miPc.imprimirEspecificacion();
18    } // fin del main
19 } // fin clase ControladorDiseñoPc
```

```
 1 /*********************************************************
 2 * DiseñoPc.java
 3 * Dean & Dean
 4 *
 5 * Esta clase obtiene las especificaciones para una PC.
 6 *********************************************************/
 7
 8 import java.util.Scanner;
 9
10 public class DiseñoPc
11 {
12   private long ramS tamañoRam = (long) 1000000000.0;
13   private long tamañoDisco;
14   private String procesador;
15   private double costo;
16
17   //*******************************************************
18
19   void asignarTamañoRam()
20   {
21     this.tamañoRam = (long) 2000000000.0;
22   } // fin asignarTamañoRam
23
```

```
24   //*******************************************************
25
26   void asignarTamañoDisco()
27   {
28     Scanner stdIn = new Scanner(System.in);
29     long tamañoDisco;
30     tamañoDisco = stdIn.nextLong();
31   } // fin asignarTamañoDisco
32
33   //*******************************************************
34
35   void asignarProcesador()
36   {
37     Scanner stdIn = new Scanner(System.in);
38     this.procesador = stdIn.nextLine();
39   } // fin asignarProcesador
40
41   //*******************************************************
42
43   void calculateCost()
44   {
45     this.costo = this.tamañoRam / 10000000.0 +
46       this.tamañoDisco / 100000000.0;
47     if (this.procesador.equals("Intel"))
48     {
49       this.costo += 400;
50     }
51     else
52     {
53       this.costo += 300;
54     }
55   } // fin calcularCosto
56
57   //*******************************************************
58
59   public void imprimirEspecificacion()
60   {
61     System.out.println("RAM = " + this.tamañoRam);
62     System.out.println("Tamaño del disco duro = " + this.tamañoDisco);
63     System.out.println("Procesador = " + this.procesador);
64     System.out.println("Costo = $" + this.costo);
65   } // fin de imprimirEspecificacion
66 } // fin de clase DiseñoPc
```

Utilice el siguiente modelo de rastreo del programa DiseñoPc. Observe que se han utilizado abreviaturas para mantener la anchura del modelo tan pequeña como sea posible. No olvide especificar valores por omisión e iniciales aun cuando no tengan impacto sobre el resultado final.

entrada

```
60000000000
Intel
```

Controlador		DiseñoPc											
	main		aTamRm	asigTamDsco		aProc	cCosto	impEsp	obj1				
línea#	**miPc**	*línea#*	**this**	**this**	**tamDisco**	**this**	**this**	**this**	**tamRam**	**tDsco**	**proc**	**costo**	**salida**

7. [Después de §6.8] La respuesta a esta pregunta no está en el libro: tendrá que buscarla en otra parte. ¿Quiénes son “los tres amigos” de UML?

8. [Después de §6.8] Construya un diagrama UML de clase para los archivos en un directorio de cómputo. El nombre de la clase debe ser `File`. Incluya los siguientes métodos: `public String getNombre()`, `public long longitud()` y `public boolean estaOculto()`. También incluya la variable de instancia asociada con el primero de estos métodos. Incluya una indicación de si el miembro es `public` o `private` y el tipo de valor o variable que devuelve. Existe ya una clase `File` como parte del lenguaje Java, y esta clase tiene muchos otros métodos, pero la documentación de la biblioteca API para esta clase no muestra variables de instancia, ¿significa esto que la clase no tiene variables de instancia?

9. [Después de §6.9] Si un objeto llama al mismo método en dos ocasiones por separado, en la segunda ejecución, las variables del método local comienzan con los valores que tenían al final de la primera ejecución de ese método. (F/V)

10. [Después de §6.9] Realice un rastreo del programa Raton2 mostrado en las figuras 6.13 y 6.14. Utilice el siguiente modelo de rastreo. Observe que se han utilizado abreviaturas para mantener la anchura del modelo tan pequeña como sea posible.

`input`

2

ControladorRaton2			Raton											
	main			setTCP		getEdad	getPeso	crecer			obj1			
línea#	**mickey**	**días**	***línea#***	**this**	**tasa**	**this**	**this**	**this**	**días**	**i**	**edad**	**peso**	**tasa**	**salida**

11. [Después de §6.11] El diagrama siguiente muestra los métodos del programa Raton2, con sus parámetros y variables locales con sangría, y el objeto del cual se creó una instancia, con sangría en sus variables de instancia. Su tarea es construir una línea de tiempo para cada método, variable local o parámetro, objeto y variable de instancia. En cada ocasión la línea debe mostrar la persistencia del elemento (cuando inicia y cuando termina) relativa a otros elementos. Para ayudarlo a empezar se proporcionan las líneas de tiempo para el método `main` y una de sus variables locales. Dibuje las otras líneas de tiempo, y muestre cómo se alinean entre ellas y entre las ya proporcionadas. (Asuma que los objetos y sus variables de instancia surgen simultáneamente.)

```
                                        tiempo →
métodos:
  main                  |------------------------|
    mickey               |----------------------|
    días
  setTasaCrecimientoPorcentual
  getEdad
  getPeso
  crecer
    días
    i
objeto:
  mickey
    edad
    peso
    TasaCrecimientoPorcentual
```

12. [Después de §6.12] Completar el siguiente esqueleto de la clase `ControladorIdEstudiante` reemplazando todas las seis ocurrencias de *<insertar-código-aquí>* con su propio código, de tal manera que el programa opere adecuadamente. Para detalles, lea los comentarios anteriores a las inserciones de *<insertar-código-aquí>*. Revise la clase `IdEstudiante`, que está después de `ControladorIdEstudiante`. Las dos clases están en archivos separados.

```
import java.util.Scanner;

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

```
    Scanner stdIn = new Scanner(System.in);
    IdEstudiante estudiante;
    String nombre;
    // Se crea una instancia del objeto IdEstudiante y se asigna a estudiante.
    <insertar-código-aquí>

    System.out.print("Introduzca el nombre del estudiante: ");
    nombre = stdIn.nextLine();

    // Asignar nombre al objeto estudiante.
    <insertar-código-aquí>

    System.out.print("Introduzca el id del estudiante: ");
    // En una sola línea, leer un int para el
    // valor id y asignarlo al objeto estudiante.
    <insertar-código-aquí>

    // Si es un id inválido ejecutar el ciclo.
    // (Utilizar método esValido en la condición del ciclo.)
    while (<insertar-código-aquí>)
    {
      System.out.print("Identificación de estudiante inválida,
      vuelva a introducirla: ");
      // En una sola línea, leer un int para el
      // valor id y asignarlo al objeto estudiante.
    <insertar-código-aquí>
    }

    System.out.println("\n" + nombre +
      ", tu nueva cuenta de correo electrónico es: \n" +
      <insertar-código-aquí>                  // obtener cuenta de correo.
  } // fin del main
} // fin de la clase ControladorIdEstudiante

public class IdEstudiante
{
  private String nombre;
  private int id;

  //*******************************************************

  public void setNombre(String n)
  {
    this.nombre = n;
  }

  public String getNombre()
  {
    return this.nombre;
  }

  public void setId(int id)
  {
    this.id = id;
  }
```

```
  public int getId()
  {
    return this.id;
  }

  //*******************************************************

  public String getCuentaCorreo()
  {
    // Incluir "" en la concatenación para convertir a cadena de caracteres.
    return "" + this.nombre.charAt(0) + this.id +
      "@pirate.park.edu";
  }

  //*******************************************************

  public boolean esValido()
  {
    return this.id >= 100000 && this.id <= 999999;
  }
} // fin de la clase IdEstudiante
```

13. [Después de §6.13] Construya un diagrama UML de clase para la clase `Crecimiento` en la figura 6.17, con el método `getIncrementoTamaño2` incluido en la figura 6.20.

Solución a las preguntas de revisión

1. Falso. Un objeto es una instancia de una clase.
2. Cualquier número, incluyendo cero.
3. Las variables de instancia de una clase deben ser declaradas fuera de todos los métodos, y todas las declaraciones de las variables de instancia deben ser ubicadas en la parte alta de la definición de la clase.
4. Las variables de instancia se declaran usualmente como tipo `private` para lograr la meta de encapsulamiento. Eso significa que los datos del objeto son más difíciles de acceder, y, consecuentemente, más difíciles de estropear. La única forma en que los datos pueden ser accedidos desde afuera de la clase es llamando a los métodos asociados tipo `public`.
5. El método `main` va en la clase controlada.
6. La mayoría del código de un programa debe estar en la clase controlada.
7. Para acceder a una variable de instancia tipo `private` desde dentro de un método `main`, se tiene que utilizar una variable de referencia de un objeto de instancia después llamar a un método de acceso. En otras palabras, utilizar esta sintaxis:
‹variable-de-referencia›.‹llamada-a-método-de-acceso›
8. Una variable de referencia almacena la ubicación en memoria de un objeto.
9. Volver a donde fue llamado el método, y verificar la variable de referencia que precede el nombre del método en ese punto. Esa variable de referencia es la que utiliza el método cuando se utiliza `this`.
10. Para una variable de la instancia de un objeto, los valores por omisión son: `int = 0`, `double = 0.0`, `boolean = false`.
11. La `edad` de `gus` es una variable de instancia. Las variables de instancia persisten durante la duración de un objeto en particular. Puesto que el objeto `gus` es declarado en `main`, `gus` y sus variables de instancia (incluyendo la `edad`) persisten durante la duración del método `main`.
12. Algunas razones para construir un diagrama UML de clase antes de escribir código:
a) Proporciona una lista de "cosas por hacer". Cuando esté en la lista de los detalles de escritura de un método, y preguntándose si ese método debería ejecutar una función en particular, el diagrama le recuerda qué otros métodos podrían ejecutar esa función.
b) Proporciona una lista completa de "partes" como las partes de una lista de un típico "kit" ensamblado por el usuario. Esta lista predefinida ayuda a evitar de manera accidental el generar nombres diferentes y conflictivos para variables y parámetros mientras se escribe el código.
c) Es un documento de trabajo que puede cambiar conforme se avanza en el mismo. El cambiar el diagrama UML de clase ayuda a identificar las modificaciones necesarias para el trabajo anterior.

13. Inmediatamente después de la sentencia `Raton mickey`, el valor de `mickey` será basura.
14. Falso. Normalmente, para que un método devuelva un valor, debe tenerse una sencilla sentencia return al final del método. Sin embargo, es también válido tener sentencias return en medio de un método. Esto podría ser muy apropiado en un método muy pequeño, donde un `return` sea inmediatamente obvio. Sin embargo, si el método es relativamente largo, un lector podría no notar un `return` interno. Con un método largo, es mejor tratar de arreglar las cosas para que haya un solo `return`, ubicado al final del método.
15. Tanto los parámetros como las variables locales tienen alcance de método como persistencia. El código dentro del método trata los parámetros como lo hace con las variables locales. El método inicializa las variables locales, mientras la llamada al método inicializa los parámetros.
16. Los argumentos y los parámetros son dos palabras diferentes para describir datos que se pasan en un método llamado. Un argumento es el nombre para los datos en la llamada al método, y un parámetro es el nombre del método para los mismos datos. Un parámetro es simplemente una copia del argumento de la llamada del método; sin embargo, si el método llamado cambia el valor de uno de sus parámetros, esto no altera el valor del argumento en el método llamado.
17. El prefijo estándar para un método de acceso es `get`.
18. El prefijo estándar para un método de mutación es `set`.
19. El prefijo estándar para un método boolean es `is`.
20. Para reducir el error en una simulación se puede reducir el tamaño del ciclo de tiempo o utilizar un algoritmo con punto intermedio. Para una determinada precisión, el algoritmo con un punto intermedio es más eficiente.

CAPÍTULO 7

Programación orientada a objetos: detalles adicionales

Objetivos

- Comprender mejor la relación entre una variable de referencia y un objeto.
- Comprender qué sucede cuando se asigna una referencia.
- Entender cómo Java recicla espacio de memoria.
- Aprender a comparar la igualdad de dos objetos diferentes.
- Aprender a intercambiar los datos en dos objetos diferentes.
- Entender cómo un parámetro de referencia puede realzar la transferencia de datos a/y desde un método llamado.
- Aprender a ejecutar una secuencia de llamadas a varios métodos en la misma sentencia.
- Aprender a crear variaciones alternativas para un método.
- Aprender a combinar la creación e inicialización en un constructor.
- Aprender a evitar la redundancia en el código anidando las llamadas a métodos y constructores.
- Aprender a realizar la partición de un problema grande en varios problemas pequeños con múltiples clases controladas.

Relación de temas

7.1 Introducción

En el capítulo 6 se aprendió a escribir programas de programación orientada a objetos (POO) utilizando bloques de construcción sencillos. En este capítulo se aprenderán conceptos de programas de POO más avanzados. En particular, se conocerán los detalles de lo que sucede detrás de escena cuando un programa instancia un objeto y almacena su dirección en una variable de referencia. Esto ayudará a apreciar y entender lo que pasa cuando un programa asigna una variable de referencia a otra.

Uno de los conceptos de la POO que se aprenderán en este capítulo es el de probar la igualdad de objetos. Es común comparar tipos primitivos por igualdad [por ejemplo, `if (marcadorEquipo1 == marcadorEquipo2)`], y de la misma manera, es común comparar referencias por igualdad. La comparación de referencias por igualdad requiere un poco de esfuerzo y en este capítulo se verá lo que implica ese esfuerzo. Algo más por aprender es lo que sucede detrás de escena cuando un programa pasa una referencia como argumento. Es importante conocer esto porque a menudo será necesario pasar referencias como argumentos.

Además de presentar conceptos de POO más avanzados, en este capítulo también se presentan aplicaciones más avanzadas de lo que ya se ha visto con relación a la POO. Por ejemplo, se aprenderá a llamar a varios métodos en sucesión, todos en una sola sentencia. Eso se denomina *encadenamiento de llamadas a métodos*, y puede conducir a un código más elegante y más compacto. También se aprenderá acerca de la *sobrecarga de métodos*. Esto es cuando se tienen diferentes versiones de un método y cada versión opera sobre diferentes clases de datos. Esto debe resultarle familiar porque lo vio en la clase `Math`. ¿Recuerda las dos versiones de los métodos `abs` de la clase `Math`? Una versión devuelve el valor absoluto de un dato tipo `double`, y la otra versión devuelve el valor absoluto de uno de tipo `int`.

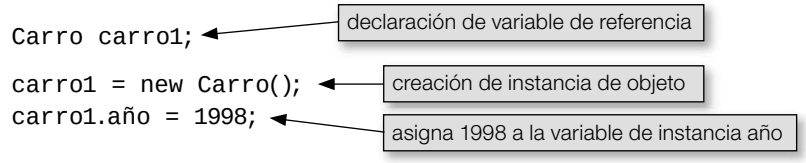
En el capítulo anterior se aprendió a instanciar un objeto en una sentencia (por ejemplo, `Raton gus = new Raton();`) y asignar un valor al objeto en una sentencia separada (por ejemplo, `gus.setTasaCrecimientoPorcentual(10);`). En este capítulo se aprenderá a combinar ambas tareas en una sola sentencia. Para hacerlo, se utilizará un tipo especial de método llamado *constructor*. Al igual que los métodos, los constructores se pueden sobrecargar utilizando distintos tipos de datos en las diferentes versiones de constructores; pero, al contrario de los métodos, los constructores están diseñados específicamente para crear e inicializar objetos.

En la sección final de resolución de problemas se aprenderá a partir un problema grande de programación en varios y sencillos problemas haciendo uso de múltiples clases controladas. Conforme se avance en este texto, el tamaño y complejidad de los problemas se incrementa de manera gradual, y se verán más y más ejemplos de programas con múltiples clases controladas.

7.2 Creación de objetos: un análisis detallado

Comencemos el capítulo con una mirada a los detalles de lo que sucede detrás de escena cuando un programa instancia un objeto y almacena su dirección en una variable de referencia. El tener un claro entendimiento de esto será útil cuando llegue el momento de entender otras operaciones de POO, y ayudará en otros esfuerzos de depuración.

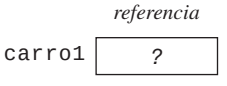
Considere el código de fragmentos siguiente:



Ahora se examinará este código con detalle, una sentencia a la vez.

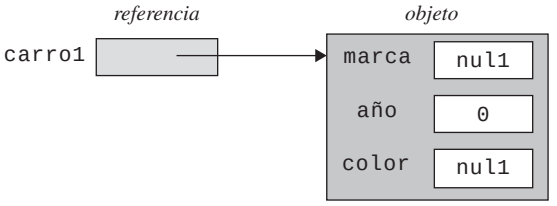
Sentencia 1:

La primera sentencia es una declaración de variables para la variable de referencia `carro1`: sólo la variable de referencia en sí, no un objeto. Después, la variable de referencia `carro1` contendrá la dirección de un objeto, pero, puesto que no se ha creado un objeto aún, no se almacena una dirección legítima todavía. ¿Cuál es el valor por omisión de una variable de referencia? Depende. Si la variable de referencia se define localmente dentro de un método (esto es, una variable local) entonces tiene basura inicialmente. Si está definida en la parte superior de la clase antes de la definición de métodos (esto es, una variable de instancia), entonces es inicializada a `null`. Puesto que la sentencia 1 no tiene un modificador de acceso (`private`), se asume que es una variable local. Por tanto, `carro1` contendrá basura por omisión, y eso es lo que el siguiente cuadro indica:



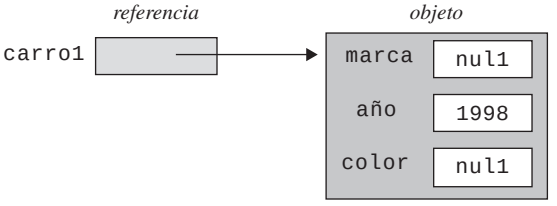
Sentencia 2:

El operador de la segunda sentencia asigna espacio en memoria para el nuevo objeto `Carro`. El operador de asignación asigna la dirección (ubicación en memoria) del espacio asignado a la variable de referencia `carro1`. No olvide esta operación. El olvidar instanciar es un error común del programador novato.



Sentencia 3:

La tercera sentencia utiliza el valor de la variable `carro1` (la dirección del objeto `Carro`) para encontrar un objeto `Carro` particular en memoria. Una vez que se encuentra el objeto `Carro` se le asigna 1998. De manera más específica, se asigna 1998 a la variable de instancia `año` del objeto `Carro`. Normalmente se utilizaría un método para asignar 1998 a la variable de instancia `año`. Con objeto de simplificar para lograr claridad se evitó la llamada al método asumiendo que la variable `año` es una variable de instancia `pública`.



7.3 Asignando una referencia

El resultado de asignar una variable de referencia a otra es que ambas se refieren al mismo objeto. ¿Por qué se refieren al mismo objeto? Puesto que las variables de referencia almacenan direcciones, se está de hecho asignando la dirección de la variable de referencia de la derecha a la del lado izquierdo. Así, después de la asignación, las dos variables de referencia contienen la misma dirección, y eso significa que se refieren al mismo objeto. Con ambas variables refiriéndose al mismo objeto, si un objeto se actualiza utilizando una de las variables de referencia se beneficiará (o sufrirá) de ese cambio cuando intente acceder al objeto. A veces, eso es lo que se desea, pero si no es así, entonces puede ser desconcertante.

Un ejemplo

Suponga que se desea crear dos objetos `Carro` que son iguales excepto en el color. El plan es instanciar el primer carro y utilizarlo como patrón para crear el segundo, y después actualizar la variable de instancia color de la segunda. ¿Cumplirá el siguiente código con esta meta?

```
Carro carroJohn = new Carro();
Carro carroStacy;
carroJohn.setMarca("Honda");
carroJohn.setAño(2003);
carroJohn.setColor("plata");
carroStacy = carroJohn;
carroStacy.setColor("durazno");
```

Esto hace que `stacyCar` se refiera al mismo objeto como `johnCar`.

El problema con el código anterior es que la sentencia `carroStacy = carroJohn;` provoca que las dos variables de referencia apunten al mismo objeto `Carro`. La figura 7.1*a* ilustra esto.

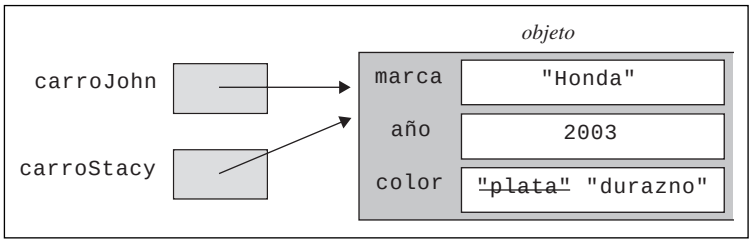


Figura 7.1*a* Efecto de asignación `carroStacy = carroJohn;`. Ambas variables de referencia se refieren exactamente al mismo objeto.

Más adelante se verá que este uso de *alias* (utilización de nombres diferentes para el mismo objeto) puede ser bastante útil, pero en este caso no es lo que se desea. En la última sentencia del fragmento del código anterior, cuando se utiliza el método `setColor` para cambiar el color de Stacy a "durazno", no se especifica el color para un nuevo carro, lo que se hace es repintar el color original del carro. La figura 7.1*a* muestra el resultado. ¡Oh oh… John podría no sentirse feliz de saber que su carro ha sido pintado color durazno!

Si se quiere hacer una copia de una variable de referencia, no se debe asignar la referencia a otra referencia. En lugar de esto, se debe instanciar un nuevo objeto a la segunda referencia y después asignar los dos objetos de las variables de instancia, uno a la vez. La figura 7.1*b* muestra esto.

Para ilustrar la estrategia mostrada en la figura 7.1*b*, se presenta el programa Carro en las figuras 7.2 y 7.3. El código incluye números de líneas para facilitar el rastreo en el ejercicio al final del capítulo. Observe el método `crearCopia` en la clase `Carro` de la figura 7.2. Como lo indica su nombre, éste es el método que se encarga de crear una copia del objeto `Carro`. El método `crearCopia` instancia un objeto `Carro` y asigna su referencia a una variable local llamada `carro`. Después copia cada uno de los valores de la variable de instancia del objeto llamado en variables de instancia del `carro`. Después devuelve `carro` al módulo llamado. Al devolver `carro`, devuelve una referencia al recientemente instanciado objeto `Carro`.

Ahora observe el controlador en la figura 7.3. Observe cómo `main` asigna el valor devuelto por `crearCopia` a `carroStacy`. Después de que `carroStacy` obtiene la referencia al recientemente creado objeto `Carro`, llama a `setColor` para cambiar el color del objeto `Carro`. Puesto que `carroJohn` y `carroStacy` se refieren a dos objetos separados, la llamada al método `carroStacy.setColor("durazno")` actualiza sólo el objeto `carroStacy`, no el objeto `carroJohn`. ¡Sí!

Siempre que un método termina, sus parámetros y variables locales se borran. En los rastreos que se realizan en el texto, se representa este borrado dibujando líneas gruesas debajo de todos los parámetros de terminación del método y de las variables locales. En el método `crearCarro` de la figura 7.2 hay una variable local, la variable de referencia, `carro`. Cuando el método `crearCarro` termina, la variable de referencia `carro` se elimina. Cuando una variable de referencia se borra, la referencia que alma-

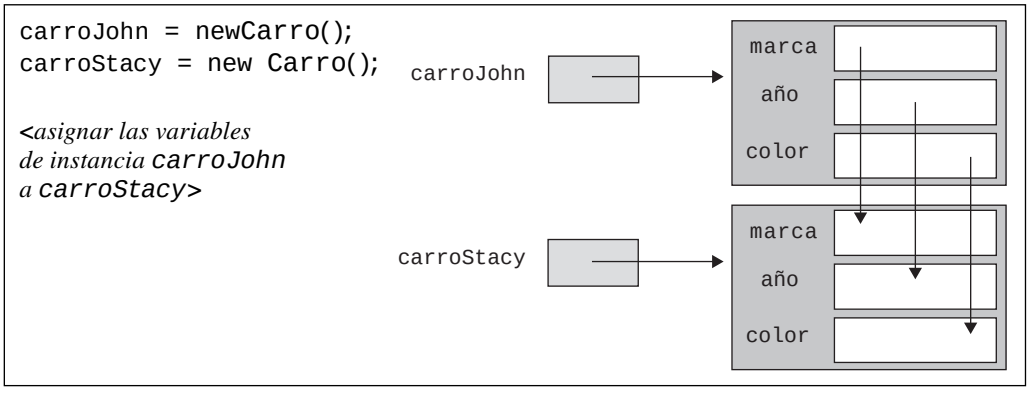


Figura 7.1*b* Efecto de instanciar dos objetos separados y copia de los valores de instancia del primer objeto en variables de instancia del segundo objeto.

```
 1  /*************************************************************
 2  * Carro.java
 3  * Dean & Dean
 4  *
 5  * Esta clase implementa una copia de la funcionalidad para un carro.
 6  *************************************************************/
 7
 8  public class Car
 9  {
10    private String marca;     // marca del carro
11    private int año;          // año de manufactura del carro
12    private String color;     // color primario del carro
13
14    //***********************************************************
15
16    public void setMarca(String marca)
17    {
18      this.marca = marca;
19    }
20
21    public void setAño(int año)
22    {
23      this.año = año;
24    }
25
26    public void setColor(String color)
27    {
28      this.color = color;
29    }
30
31    //***********************************************************
32
33    public Carro crearCopia()
34    {
35      Carro carro = new Carro();
36
37      carro.marca = this.marca;
38      carro.año = this.año;
39      carro.color = this.color;
40      return carro;
41    } // fin de crearCopia
42
43    //***********************************************************
44
45    public void desplegar()
46    {
47      System.out.printf("marca= %s\naño= %s\ncolor= %s\n",
48      this.marca, this.año, this.color);
49    } // fin de desplegar
50  } // fin de la clase Carro
```

Esto instancia un nuevo objeto.

Esto devuelve una referencia al nuevo objeto.

Figura 7.2 Clase Carro con método `crearCopia` que devuelve una referencia a la copia del objeto llamado.

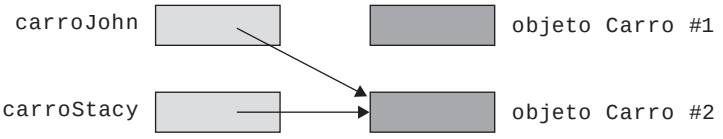
cena se pierde, y si esa referencia no es salvada en una variable separada, el programa no tendrá manera de encontrar el objeto al que se refería. En el método `crearCarro`, el valor de la variable de referencia `carro` no es salvado. Es devuelto al método `main` donde se asigna a `carroStacy`.

Objetos inaccesibles y recolección de basura

A veces se querrá instanciar un objeto temporal dentro de un método, utilizarlo con algún fin y después abandonar el objeto cuando el método termine. Otras veces se puede desear abandonar el objeto antes de que el método termine. Por ejemplo, suponga que en el método `main` de la figura 7.3, después de llamar a `crearCopia` y crear un nuevo objeto `Carro` para `carroStacy`, se quiere modelar el viejo carro de John destruido por el fuego, y Stacey se ofreció dejarlo ser el copropietario de su nuevo carro. Esto se podría representar uniendo la propiedad de un carro mediante la sentencia:

```
carroJohn = carroStacey;
```

El hacer esto reviste de la referencia anterior `carroJohn` al objeto `Carro` original de John, y ese objeto `Carro` se vuelve inaccesible al programa (abandonado), como el objeto #1 `Carro` en la siguiente imagen:




```
1  /*******************************************************
2  * ControladorCarro.java
3  * Dean & Dean
4  *
5  * Esta clase demuestra la copia de un objeto.
6  *******************************************************/
7
8  public class ControladorCarro
9  {
10   public static void main(String[] args)
11   {
12     Carro carroJohn = new Carro();
13     Carro carroStacy;
14
15     carroJohn.setMarca("Honda");
16     carroJohn.setAño(2003);
17     carroJohn.setColor("plata");
18     carroStacy = carroJohn.crearCopia();
19     carroStacy.setColor("durazno");
20     System.out.println("Carro de John:");
21     carroJohn.desplegar();
22     System.out.println("Carro de Stacy:");
23     carroStacy.desplegar();
24   } // fin de main
25 } // fin de clase ControladorCarro
```

Esto asigna la referencia devuelta a la variable de referencia en la llamada al método.

Salida:

```
Carro de John:
marca= Honda
año= 2003
año= plata
Carro de Stacy:
marca= Honda
año= 2003
color= durazno
```

Figura 7.3 Clase `CarDriver` que controla `Car` en la figura 7.2.

La pregunta es ¿cómo trata la máquina virtual de Java JVM a los objetos abandonados o inaccesibles? Los objetos inaccesibles no pueden participar en el programa, por lo que no es necesario mantenerlos. Llegan a ser "basura". De hecho, sería mala idea mantenerlos, porque esto puede conducir a atascar la memoria de la computadora. Una computadora tiene una cantidad finita de memoria, y cada pieza de basura utiliza algo de esa memoria; lo que significa que hay menos memoria disponible para nuevas tareas. Si se permite la acumulación de basura, al final acaparará todo el *espacio libre* en la memoria de una computadora (el espacio libre es la porción de memoria que no es utilizada). Si no hay espacio libre en la memoria, no hay espacio para nuevos objetos, y la computadora deja de trabajar (hasta que sea reinicializada).

Si a un objeto inaccesible se le permite persistir y utilizar espacio en la memoria de la computadora, eso se denomina *fuga de memoria*. Las fugas de memoria pueden ocurrir en programas de cómputo que asignan memoria durante la ejecución. Cuando en un lenguaje de cómputo es necesario que el programador haga algo específico para prevenir fugas de memoria, pero olvida hacerlo, surge un error muy desagradable: un error muy difícil de encontrar. Al crear el lenguaje Java, James Gosling y sus buenos amigos de Sun se dieron cuenta de que habían optado por hacer que el lenguaje mismo se encargara de este problema. ¿Cómo? Involucrándose en el asunto de la recolección de basura. No, no quiere decir que Dirk y Jenny lo hagan cuando recogen la basura de su contenedor cada martes, sino la *recolección de basura* de Java. De hecho, James Gosling no inventó la recolección de basura; ésta siempre ha existido. Pero Java es el primer lenguaje de programación popular que la incluye como un servicio estándar.

Entonces, ¿qué demonios es la recolección de basura? Es cuando un programa de recolección de basura busca objetos inaccesibles y recicla el espacio que ocupan indicando al sistema operativo que designe este espacio como espacio libre. Ese espacio podría no ser utilizado correctamente, y algún niño prodigio podría encontrar esos objetos abandonados hurgando en los botes de basura, haciendo uso del olfato de los perros, y buscando muebles, pero para propósitos prácticos, estos objetos se deben considerar como abandonados e irrecuperables.

La belleza de la recolección de basura en Java es que el programador no tiene que preocuparse acerca de ello, pues ocurre cuando es apropiado. ¿Y cuándo es apropiado? Cuando la computadora está ejecutando con poca memoria libre o cuando nada más está sucediendo, como cuando un programa está esperando una entrada a través del teclado. En ese punto, el sistema operativo despierta al amigo de la recolección de basura y le pide encargarse de esto.

7.4 Objetos de prueba para igualdad

En la sección anterior se ilustró la devolución de una referencia a un método. En esta sección se ilustra el paso de una referencia a un método para permitir leer los datos del objeto referenciado. Una de las aplicaciones más comunes de esto ocurre al probar la igualdad de dos objetos. Antes de analizar esta aplicación es apropiado observar la forma más sencilla de evaluar la igualdad.

El operador ==

El operador `==` funciona lo mismo para variables primitivas que para variables de referencia. Prueba si los valores almacenados en estas variables son lo mismo. Cuando se aplica a variables de referencia, el operador `==` devuelve `true` si y sólo si las dos variables de referencia se refieren al mismo objeto; esto es, las dos variables de referencia contienen la misma dirección y por tanto son alias del mismo objeto. Por ejemplo, ¿qué imprime el siguiente código?

```
Carro carro1 = new Carro();
Carro carro2 = carro1;

if (carro1 == carro2)
{
  System.out.println("el mismo");
}
else
{
  System.out.println("diferentes");
}
```

Imprime "el mismo" porque `carro1` y `carro2` contienen el mismo valor: la dirección del objeto solitario `Carro`; pero si se quiere ver dos objetos diferentes que tienen los mismos valores de variable de instancia, el operador `==` no es lo que se quiere. Por ejemplo, ¿qué imprime este código?

```
Carro carro1 = new Carro();
Carro carro2 = new Carro();

carro1.setColor("rojo");
carro2.setColor("rojo");
if (carro1 == carro2)
{
  System.out.println("el mismo");
}
else
{
  System.out.println("diferentes");
}
```

La expresión `carro1 == carro2` devuelve `false`, ¿por qué?

Este código imprime "diferentes" porque `carro1 == carro2` devuelve `false`. No importa que `carro1` y `carro2` contengan el mismo dato (rojo). El operador `==` no considera los datos del objeto; sólo verifica que las dos variables de referencia apunten al mismo objeto. En este caso, `carro1` y `carro2` se refieren a distintos objetos, con diferentes ubicaciones en memoria.

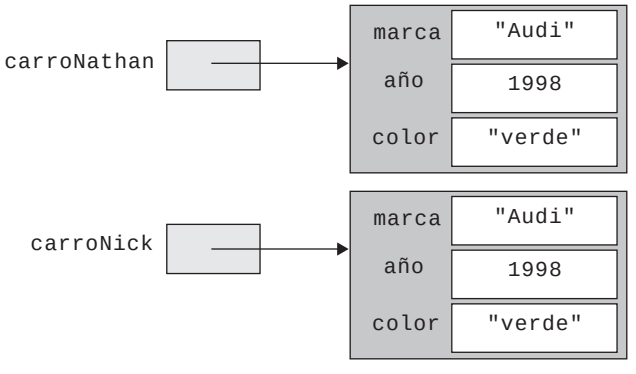
El método equals

Si se quiere ver dos objetos diferentes que tienen las mismas características, es necesario comparar los contenidos de éstos en lugar de sólo ver si las dos variables de referencia apuntan al mismo objeto. Para hacer esto, es necesario un método `equals` en la definición de la clase del objeto que compare las variables de instancia de dos objetos. Tener el método `equals` es muy común pues con frecuencia se quiere probar dos objetos para ver si tienen las mismas características. Para las clases API de Java, utilice los métodos `equals` construidos. Por ejemplo, al comparar los contenidos de dos cadenas de caracteres, conviene llamar al método `equals` de la clase `String`. Para clases que usted mismo implemente, adopte el hábito de escribir su propio método `equals`.

Un método `equals` es una utilidad práctica.

Un ejemplo

El siguiente diagrama muestra dos objetos con idénticos valores de variable de instancia. Al comparar `carroNathan` con `carroNick` mediante el operador `==` se obtiene `false`, porque las dos variables de referencia apuntan a diferentes objetos. Sin embargo, comparar nuevamente estas variables con el método `equals` genera `true`, porque un método estándar `equals` compara valores de variables de instancia, y estos objetos tienen idénticos valores de variables de instancia.



El programa Carro2 de las figuras 7.4 y 7.5 ilustra este ejemplo. La clase de la figura 7.5 define un método `equals`, y la clase `ControladorCarro2` de la figura 7.4 llama al método `equals` mientras compara dos objetos `Carro2`. Como sucede comúnmente con las llamadas al método `equals`, la lla-

```
/*************************************************************
* ControladorCarro2.java
* Dean & Dean
*
* Esta clase demuestra el controlador de la clase Carro2.
*************************************************************/

public class ControladorCarro2
{
  public static void main(String[] args)
  {
    Carro carroNathan = new Carro2();
    Carro carroNick = new Carro2();

    carroNathan.setMarca("Audi");
    carroNathan.setAño(1998);
    carroNathan.setColor("verde");
    carroNick.setMarca("Audi");
    carroNick.setAño(1998);
    carroNick.setColor("verde");
    if (carroNathan.equals(carroNick))
    {
      System.out.println("Los carros tienen características idénticas.");
    }
  } // fin de main
} // fin de clase ControladorCarro2
```

Observe cómo la llamada al método `equals` es insertada en una condición `if`.

Figura 7.4 Clase `controladorCarro2` que controla la clase `Carro2` de la figura 7.5.

mada al método `equals` de la figura 7.4 está insertada en la condición de una sentencia `if`. Esto debe tener sentido cuando nos damos cuenta de que una sentencia de condición `if` se debe evaluar como `true` o `false` (verdadera o falsa), y que un método `equals`, por tanto, se evaluará como tal. Típicamente, un método `equals` se evalúa como `true` si las variables de instancia en los dos objetos contienen los mismos valores en los datos, de lo contrario se evalúan como `false`. Para el programa `Carro2`, el método `equals` se evalúa como `true` si `carroNathan` contiene los mismos datos (`marca`, `año` y `color`) que `carroNick`. La figura 7.4 muestra que ambos objetos tienen asignados los mismos datos. Por tanto, el método `equals` devuelve `true` y el programa imprime "Los carros tienen características idénticas".

En la llamada al método `equals` observe cómo la primera variable de referencia a `Carro2`, `carroNathan` aparece a la izquierda del .`equals` y que la segunda variable de referencia `carroNick` aparece dentro del paréntesis. Por tanto, `carroNathan` es el objeto que realiza la llamada, y `carroNick` es un argumento. Esto sucede infinidad de veces cuando se utilizan dos variables de referencia con una llamada a un método: una variable de referencia será el objeto que realice la llamada y el otro será el argumento.

Ahora se examinará la definición del método `equals` de la figura 7.5. Primero, observe el encabezado del método `equals`. ¿Por qué devuelve un tipo `boolean`? Porque el tipo devuelto debe concordar con el valor del tipo devuelto, y ese método siempre devuelve un valor tipo boolean (ya sea `verdadero` o `falso`). También observe que el tipo del parámetro de `otroCarro` es `Carro2`. Eso tiene sentido cuando se analiza el método `equals` en la figura 7.4 donde se muestra que el argumento que se pasa en el método `equals` es `carroNick`, y que `carroNick` es una variable de referencia de tipo `Carro2`.

Bien, ahora es tiempo de examinar el cuerpo del método `equals`. Observe que hay una sola sentencia: la sentencia `return`. El valor devuelto debe ser tipo `boolean`, por lo que la expresión después de la palabra `return` se debe evaluar como `true` o `false`. Esta expresión es una unión de tres subexpresiones de tipo `boolean`. Para que la expresión completa sea `verdadera`, las tres subexpresiones deben ser `true`.

Cada subexpresión verifica si una variable de instancia en particular tiene el mismo valor que el objeto que realiza la llamada y que el objeto pasado como parámetro. Por ejemplo, para verificar si la varia-

```
/*****************************************************************
* Carro2.java
* Dean & Dean
*
* Esta clase implementa la funcionalidad equals para un carro.
*****************************************************************/

public class Carro2
{
  private String marca;
  private int año;
  private String color;

  //*************************************************************

  public void setMarca(String marca)
  {
    this.marca = marca;
  }

  public void setAño(int año)
  {
    this.año = año;
  }

  public void setColor(String color)
  {
    this.color = color;
  }

  //*************************************************************

  // Este método verifica si dos carros contienen los mismos datos.

  public boolean equals(Carro2 otroCarro)
  {
    return this.marca.equals(otroCarro.marca) &&
           this.año == otroCarro.año &&
           this.color.equals(otroCarro.color);
  } // fin de equals
} // fin de la clase Carro2
```

Esto compara todas las variables de instancia.

Figura 7.5 Clase `Carro2` con método `equals`.

ble de instancia `año` tiene el mismo valor en el objeto que realiza la llamada que en el del objeto pasado como parámetro, se hace lo siguiente:

```
this.año == otroCarro.año
```

En este caso, se utiliza el operador `==` para verificar la igualdad. Esto funciona bien para la variable de instancia `año`, porque `año` es de tipo `int`; pero las variables de instancia `marca` y `color` son de tipo `String`, y el operador `==` es un anatema para las cadenas de texto. En esos casos debe utilizarse el método `equals`. Así, para verificar si la variable de instancia `marca` tiene el mismo valor en el objeto que realiza la llamada que en el objeto pasado como parámetro, se hace lo siguiente:

```
this.marca.equals(otroCarro.marca)
```

Mmmm… ¿Le parece extraño utilizar el método `equals` de la clase `String` dentro del método `equals` de la clase `Carro2`? Esto es perfectamente adecuado: al compilador no le importa si dos méto-

dos tienen el mismo nombre siempre y cuando provengan de clases diferentes. ¡Eso es parte de la belleza del encapsulamiento!

¿Puede pensar en otra forma de escribir el cuerpo del método `equals` de la clase `Carro2`? Se podría haber utilizado esa expresión `boolean` a la derecha de la palabra reservada `return` como la condición de una sentencia `if` y después agregar `return true` en la cláusula `if` y `return false` en la cláusula `else`; pero eso habría sido una forma más difícil y más larga de hacer lo mismo (y probablemente más confuso también), porque hubiera requerido más paréntesis. Aunque la sentencia `return` de la figura 7.5 podría parecer a primera vista un nido de rata Cerberano,[1] que la mayoría de los programadores veteranos consideran más elegante.

Suponga que se quiere que los colores en mayúscula se consideren como iguales a sus equivalentes en minúscula. En otras palabras, se quiere que Ford plata 2005 sea considerado igual a Ford Plata 2005. ¿Cómo se debe cambiar el código para manejar esto? Utilice `equalsIgnoreCase` en lugar de `equals` para comparar las cadenas de caracteres de colores.

```
this.color.equalsIgnoreCase(otroCarro.color)
```

Esto demuestra que se puede hacer que el método `equals` devuelva `true` cuando haya una igualdad sólo aproximada, donde "aproximada" se puede definir como se desee. Se discutirá el método `equals` con más detalle en el capítulo 13.

7.5 Paso de referencias como argumentos

Por ahora usted debe sentirse suficientemente conforme con el concepto de paso de un argumento en un método. Se ha cubierto todo lo que requiere saber acerca del paso de tipos primitivos como argumentos. Pero todavía requiere saber un poco más acerca de las referencias como argumentos. Por ejemplo, en la figura 7.4 se pasó la referencia `carroNick` como argumento en el método `equals`. El método `equals` asignó la referencia `carroNick` a su parámetro `otroCarro`, y después utilizó el parámetro `otroCarro` para leer los datos del objeto. En ese ejemplo, se utilizó la referencia pasada para leer los datos de un objeto. Ahora se utilizará la referencia pasada para actualizar los datos de un objeto.

Suponga que se pasa una variable de referencia a un método, y dentro del método se actualizan las variables de instancia de una variable de referencia. ¿Qué sucede? Recuerde que una variable de referencia contiene la dirección de un objeto, no el objeto mismo. Entonces, al pasar un argumento de variable de referencia a un método, una copia de la dirección del objeto (no una copia del objeto mismo) se pasa al método y se almacena en el parámetro del método. Puesto que el parámetro y el argumento contienen el mismo valor en la dirección, ambos apuntan al mismo objeto. Así, si las variables de instancia se actualizan, entonces se actualizan simultáneamente las variables de instancia del argumento en el módulo que realiza la llamada. Éste es un caso donde el uso de alias (utilizar dos nombres para el mismo objeto) es realmente sencillo.

Ejemplo de intercambio de personas

Ahora se verá si se entiende todo lo relacionado con el paso de referencia poniendo en contexto un programa completo. Observe el programa Persona en las figuras 7.6 y 7.7. El programa Persona intercambia los nombres de dos objetos `Persona`. Como se muestra en el método `main` de la figura 7.6, la variable de referencia `persona1` comienza con el nombre "Jonathan" y la variable de referencia `persona2` empieza con el nombre "Benji". Después de llamar al método `intercambiaPersona`, `persona1` tiene el nombre "Benji" y `persona2` tiene el nombre "Jonathan". El método `intercambiaPersona` intercambia los nombres aprovechándose del fenómeno del que se ha hablado anteriormente: si una variable de referencia se pasa a un método, entonces el parámetro y el argumento se refieren al mismo objeto, y una actualización en uno de ellos significa una actualización en el otro también. La línea inferior

[1] Probablemente usted ya sabe lo que es un "nido de ratas", es un desorden. Pero ¿qué es "Cerberano"? En la mitología griega, Cerbero era un perro, un monstruo de tres cabezas que cuidaba la entrada de Hades (el mundo de los muertos). Se dice que la sentencia `return` podría parecer un nido de ratas Cerberano porque es complicado y tiene tres partes. ¿Qué preferiría usted encontrar en un callejón oscuro: una criatura canina de tres cabezas o una sentencia `return` complicada?

```
/*********************************************************
* ControladorPersona.java
* Dean & Dean
*
* Esta clase es un controlador demostración para una clase persona.
*********************************************************/

public class ControladorPersona
{
  public static void main(String[] args)
  {
    Persona persona1 = new Persona();
    Persona persona2 = new Persona();

    persona1.setNombre("Jonathan");
    persona2.setNombre("Benji");
    System.out.println(persona1.getNombre() + ", " +
      persona2.getNombre());

    persona1.intercambiarPersona(persona2);
    System.out.println(persona1.getNombre() + ", " +
      persona2.getNombre());
  } // fin de main
} // fin de la clase ControladorPersona
```

Este argumento permite al método que realiza la llamada modificar el objeto referenciado.

Salida:

```
Jonathan, Benji
Benji, Jonathan
```

Figura 7.6 Controlador para programa que implementa el intercambio pasando una referencia a un método.

muestra que cuando se pasa una referencia a un método, se permite que el método modifique el objeto referenciado.

Algoritmo de intercambio de propósito general

¿Cómo intercambiar dos valores?

Antes de adentrarse en el código del programa Persona es bueno adentrarse en el algoritmo de intercambio de propósito general. Intercambiar dos valores es un requerimiento de programación muy común, por lo que hay que asegurarse de que se ha entendido lo que hay que hacer.

Suponga que se le ha solicitado proporcionar un algoritmo que intercambie el contenido de dos variables, `x` y `y`. Para cumplir con este propósito de manera más concreta, se le ha proporcionado el siguiente esqueleto de un algoritmo. Reemplazar <*Inserte código de intercambio aquí.*> con el seudocódigo apropiado de tal manera que el algoritmo imprima `x=8, y=3`.

```
x ← 3
y ← 8
<Inserte código de intercambio aquí.>
imprimir "x = " + x + ", y = " + y
```

Observe que el esqueleto del algoritmo utiliza la versión formal del seudocódigo introducido cerca del final del capítulo 2. La versión formal del seudocódigo es más compacta y más cercana a Java que la versión informal. Por ejemplo, en lugar de decir "asignar `x` = 3", la versión formal del seudocódigo utiliza una flecha hacia la izquierda y dice "`x` ← 3". Sentimos que en este punto con tantos capítulos de Java en su bolsillo, la versión formal de seudocódigo es preferible a la informal por su consistencia.

¿Funcionaría el siguiente código? ¿Intercambiaría los contenidos de `x` y `y` de manera correcta?

```
y ← x
x ← y
```

```
/*****************************************************************
* Persona.java
* Dean & Dean
*
* Esta clase almacena, trae e intercambia el nombre de una persona.
*****************************************************************/

public class Persona
{
  private String nombre;

  //*************************************************************

  public void setNombre(String nombre)
  {
    this.nombre = nombre;
  }

  public String getNombre()
  {
    return this.nombre;
  }

  //*************************************************************

  // Este método intercambia los nombres de dos objetos Persona.

  public void intercambiarPersona(Persona otraPersona)
  {
    String temp;

    temp = otraPersona.nombre;
    otraPersona.nombre = this.nombre;     } ← algoritmo de intercambio
    this.nombre = temp;
  } // fin intercambiarPersona
} // fin de la clase Persona
```

Figura 7.7 Clase `Persona`, que implementa en un método el intercambio del paso de una referencia.

La primera sentencia pone el valor original de `x` en `y`. La segunda sentencia intenta poner el valor original de `y` en `x`. Desafortunadamente, la segunda sentencia no funciona porque el valor original de `y` se ha perdido (se sobrescribió por `x` en la primera sentencia). Si usted insertó el código anterior en el algoritmo anterior, este último imprimirá:

```
x = 3, y = 3
```

El intercambio requiere una variable temporal.

¡Eso no es lo que se quiere! El truco es guardar el valor de `y` antes de intercambiarlo con el valor de `x`. ¿Cómo se guarda? Utilizando una variable como ésta:

```
temp ← y
y ← x
x ← temp
```

Ejemplo de intercambio de persona: continuación

Ahora se revisará la clase `Persona` de la figura 7.7. En particular, se examinará cómo el método `intercambiarPersona` implementa un algoritmo de intercambio. Los elementos intercambiados son los nombres del objeto que se pasa y el nombre del objeto llamado. Se accede al objeto pasado a través

del parámetro `otraPersona`. Observe cómo se accede al nombre del objeto pasado mediante `otraPersona.nombre`. Y observe cómo se accede al nombre del objeto llamado con `this.nombre`. Y, finalmente, observe cómo se utiliza una variable local `temp` como medio de almacenamiento temporal para `otraPersona.nombre`.

7.6 Encadenamiento de llamadas a métodos

Hasta este punto, usted debe sentirse muy cómodo con la llamada a métodos. Ahora es tiempo de ir un paso más adelante. En esta sección, se aprenderá a llamar a varios métodos en sucesión, todo con una sentencia. Eso es lo que se llama *encadenamiento de llamadas a métodos*, y puede conducir a un código más compacto y más elegante.

Si se observan las figuras 7.3 y 7.4 se verán varias instancias donde se llama a varios métodos uno después de otro, y se utiliza una sentencia separada para cada llamada a método sucesivo, como el fragmento de código de la figura 7.4:

```
carroNathan.setMarca("Audi");
carroNathan.setAño(1998);
```

¿No sería más agradable poder encadenar las llamadas a los métodos juntos como a continuación?

```
carroNathan.setMarca("Audi").setAño(1998);
```

El encadenamiento de llamadas a métodos es una opción, no un requisito; entonces, ¿por qué utilizarlo? Porque con frecuencia puede conducir a un código más elegante: más compacto y fácil de entender.

Ahora se verá un encadenamiento de llamadas a métodos en el contexto de un programa completo. Observe el encadenamiento de llamadas a métodos (indicados por una llamada) en la clase `ControladorCarro3` de la figura 7.8. La precedencia de izquierda a derecha se aplica aquí, por lo que `carro.setMarca` se ejecuta primero. El método `setMarca` devuelve la llamada al objeto, que es el objeto `carro` a la izquierda de `carro.setMarca`. El objeto `carro` devuelto después se utiliza para llamar al método `setAño`. El método `setAño` llama al método `Imprimirlo` de una manera similar.

El encadenamiento de una llamada a método no funciona por omisión. Si se quiere habilitar el encadenamiento de llamadas a métodos de la misma clase, es necesario alguno de los siguientes elementos en la definición de cada método:

1. La última línea en el cuerpo del método debe devolver el objeto llamado especificando `return this;`
2. En el encabezado del método, el tipo devuelto debe ser el nombre de clase del método.

```
/**************************************************************
* ControladorCarro3.java
* Dean & Dean
*
* Esta unidad controla Carro3 para ilustrar el encadenamiento de
* llamadas a métodos.
**************************************************************/

public class ControladorCarro3
{
  public static void main(String[] args)
  {
    Carro3 carro = new Carro3();

    carro.setMarca("Honda").setAño(1998).Imprimelo();
  } // fin de main
} // fin de clase ControladorCarro3
```

Utilice puntos para encadenar juntos los métodos llamados.

Figura 7.8 Programa controlador `Carro3` que ilustra el encadenamiento de llamadas a métodos.

Esos elementos se implementaron en la clase `Carro3` en la figura 7.9. Verifique que los métodos `setMarca` y `setAño` estén habilitados propiamente para realizar el encadenamiento de llamadas a métodos. De manera específica, verifique que 1) la última línea en cada cuerpo del método sea `return this;` y 2) que en cada encabezado del método el tipo devuelto sea el nombre de la clase, `Carro3`.

Cuando se termina un método con la sentencia `return this;` se hace lo posible por utilizar el mismo objeto que llama al siguiente método en la cadena. Sin embargo, también se pueden encadenar métodos llamados por distintos tipos de objetos. Basta con arreglar la cadena de tal manera que el tipo de referencia devuelto en cada método precedente concuerde con la clase de cada uno de los siguientes métodos. Así, en general, para hacer un método encadenable, hay que hacer lo siguiente:

1. En el encabezado del método, especificar el tipo devuelto como la clase de un siguiente método potencial.
2. Terminar el cuerpo del método con:

   `return` <*referencia-al-objeto-que-llama-al-siguiente-método*>`;`

He aquí un ejemplo familiar que ilustra el encadenamiento de dos métodos definidos en el API de Java.

```
ch = stdIn.nextLine().charAt(0);
```

La variable `stdIn` es una referencia a un objeto de la clase `Scanner`. Llama al método `nextLine` de esa clase, la cual devuelve una referencia a un objeto de la clase `String`. Después, ese objeto llama al método `charAt` de la clase `String`, que devuelve un carácter.

```
/*********************************************************
* Carro3.java
* Dean & Dean
*
* Esta clase ilustra métodos que pueden ser encadenados.
*********************************************************/

public class Carro3
{
  private String marca;
  private int año;

  //*******************************************************

  public Carro3 setMarca(String marca)
  {
    this.marca = marca;
    return this;
  } // end setMake

  public Carro3 setAño(int año)
  {
    this.año = año;
    return this;
  } // end setYear

  //*******************************************************

  public void Imprimelo()
  {
    System.out.println(marca + ", " + año);
  } // fin de Imprimelo
} // fin de la clase Carro3
```

El tipo devuelto es el mismo que el nombre de la clase.

Devuelve el objeto llamado.

Figura 7.9 Clase `Carro3`.

7.7 Métodos sobrecargados

Hasta este punto, todos los métodos que se definieron para una clase dada tenían nombres únicos. Pero si usted recuerda los métodos API de Java presentados en el capítulo 5, sabrá que había varios ejemplos donde se utilizó el mismo nombre (`abs`, `max`, `min`) para identificar más de un método en la misma clase (la clase `Math`). En esta sección se muestra cómo hacer esto en las clases que se escriban.

¿Qué son los métodos sobrecargados?

Los *métodos sobrecargados* son dos o más métodos en la misma clase que utilizan el mismo nombre. Puesto que utilizan el mismo nombre, el compilador necesita algo más aparte del nombre con el fin de poder distinguirlos. ¡Los parámetros entran al rescate! Para distinguir dos métodos sobrecargados, éstos se definen con un número de parámetro diferente o con tipos de parámetros diferentes. La combinación del nombre del método, el número de sus parámetros, y los tipos de sus parámetros se denomina *firma* del método. Cada método diferente tiene su firma diferente. ¿Podrían utilizarse las siguientes tres líneas para tres métodos `encontrarMaximo` sobrecargados?

```
int encontrarMaximo(int a, int b, int c)
double encontrarMaximo(double a, double b, double c)
double encontrarMaximo(double a, double b, double c, double d)
```

Sí, es una sobrecarga válida del nombre de método `encontrarMaximo` porque cada encabezado es distinguible en términos de número y tipos de parámetros. ¿Qué sucede con las siguientes dos líneas?, ¿podría el método `encontrarPromedio` sobrecargarse de la siguiente forma?

```
int encontrarPromedio(int a, int b, int c)
double encontrarPromedio(int x, int y, int z)
```

No. Éstos no son métodos distinguibles porque tienen la misma firma: el mismo nombre de método y el mismo número y tipos de parámetros. Puesto que estos dos métodos no son distinguibles, si se intenta incluir estos dos encabezados de métodos en la misma clase, el compilador pensará que se está tratando de definir el mismo método dos veces, y eso lo irritará. Hay que estar preparado para volver a escuchar el mensaje de error de compilación "duplicate definition" (definición duplicada).

Observe que el encabezado del método `encontrarPromedio` anterior tiene dos tipos de valor devuelto diferentes. Se podría pensar que el diferente tipo de retorno indica firmas distintas, lo cual es falso. El tipo de retorno distinto no es parte de la firma, por lo que no se pueden utilizar sólo tipos de retorno diferentes para distinguir métodos sobrecargados.

Beneficio de los métodos sobrecargados

¿Cuándo utilizar métodos sobrecargados? Cuando se tenga la necesidad de ejecutar esencialmente la misma tarea con diferentes parámetros. Por ejemplo, los métodos asociados con los encabezados `encontrarMaximo` ejecutan esencialmente la misma tarea básica: calculan el valor máximo de una lista dada de números; pero ejecutan la tarea con diferentes parámetros. Dada esa situación, los métodos sobrecargados se ajustan perfectamente a esto.

Observe que el uso de los métodos sobrecargados no es nunca un requerimiento absoluto. Como alternativa se pueden utilizar siempre diferentes métodos. Entonces, ¿por qué son mejores los encabezados anteriores del método `encontrarMaximo` que los que se transcriben a continuación?

```
int encontrarMaximoDe3Ints(int a, int b, int c)
double encontrarMaximoDe3Doubles(double a, double b, double c)
double encontrarMaximoDe4Doubles(double a, double b, double c, double d)
```

Como lo sugieren estos ejemplos, utilizar diferentes nombres es cansado y no tiene sentido. Con un solo nombre de método, el nombre puede ser simple. Como programador, ¿no preferiría usted utilizar y recordar un simple nombre en lugar de varios nombres largos?

Un ejemplo

Observe la clase de la figura 7.10. Utiliza los métodos sobrecargados `setAltura`. Ambos métodos asignan un parámetro `altura` a la variable de instancia `altura`. La diferencia es la técnica para asignar las unidades de altura. El primer método asigna automáticamente código duro "cm" (centímetros) a la variable de instancia `unidades`. El segundo método asigna un parámetro unidades especificado por el usuario a la variable de instancia `unidades`, mientras que el primer método requiere sólo un parámetro `altura`. Los dos métodos ejecutan más o menos la misma tarea, con una ligera variación. Es por eso que se quiere utilizar el mismo nombre y "sobrecargarlo".

Ahora observe el controlador de la figura 7.11 y sus dos llamadas al método `setAltura`. Para cada llamada al método, ¿puede usted decir cuál de los dos métodos sobrecargados es llamado? La primera llamada al método `setAltura(72.0, "pgs")` en la figura 7.11, llama al segundo método `setAltura` del método 7.10 porque los dos argumentos en la llamada al método concuerdan con los dos parámetros en el encabezado del segundo método. La segunda llamada al método `setAltura(180.0)` de la figura 7.11 llama al primer método `setAltura` de la figura 7.10 porque el argumento en la llamada al método concuerda con un parámetro en el primer encabezado del método.

```
/**********************************************************
* Altura.java
* Dean & Dean
*
* Esta clase almacena e imprime valores de altura.
**********************************************************/

class Altura
{
  double altura;    // altura de una persona
  String unidades;  // como cm para centímetros

  //********************************************************

  public void setAltura(double altura)
  {
    this.altura = altura;
    this.unidades = "cm";
  }

  //********************************************************

  public void setAltura(double altura, String unidades)
  {
    this.altura = altura;
    this.unidades = unidades;
  }

  //********************************************************

  public void imprimir()
  {
    System.out.println(this.altura + " " + this.unidades);
  }
} // fin de la clase Altura
```

Figura 7.10 Clase `Altura` con métodos sobrecargados.

```
/*************************************************************
* ControladorAltura.java
* Dean & Dean
*
* Esta clase es un controlador demostración para una clase Altura.
*************************************************************/

public class ControladorAltura
{
  public static void main(String[] args)
  {
    Altura miAltura = new Altura();

    miAltura.setAltura(72.0, "in");
    miAltura.imprimir();
    miAltura.setAltura(180.0);
    miAltura.imprimir();
  } // fin de main
} // fin de clase ControladorAltura
```

Figura 7.11 Clase `ControladorAltura` que controla a la clase `Altura` de la figura 7.10.

Llamada a un método sobrecargado dentro de un método sobrecargado

Suponga que se han sobrecargado métodos y que se quiere que uno de ésos llame a otro de los sobrecargados. La figura 7.12 proporciona un ejemplo que muestra cómo hacer eso. El método `setAltura` de la figura 7.12 es una versión alternativa del método `setAltura` de un parámetro de la figura 7.10. Observe cómo llama al método de dos parámetros `setAltura`.

La segunda llamada adicional al método hace al programa un poco menos eficiente, pero se podría considerar un poco más elegante porque elimina la redundancia en el código. En la figura 7.10 `this.altura = altura;` aparece en ambos métodos, y eso es redundancia de código, a pesar de ser redundancia de código trivial.

¿Por qué no hay variable de referencia punto a la izquierda del método `setAltura` en el cuerpo del método en la figura 7.12? Porque si se está en un método de instancia y se llama a otro método que está en la misma clase, la variable de referencia punto prefijo no es necesaria. Y en este caso, los dos métodos sobrecargados `setAltura` son métodos de instancia y por tanto están en la misma clase.

Con variables que no son de referencia punto prefijo en las llamadas al método `setAltura(altura, "cm");` de la figura 7.12, se podría pensar que esas llamadas no tienen objeto llamante. De hecho, hay un objeto implícito que realiza la llamada; es el mismo objeto que llamó al método actual. Pregunta de repaso: ¿cómo se puede acceder al método actual del objeto que realiza la llamada? Utilizando la referencia `this`. Si se quiere que `this` sea explícita se puede agregar la llamada al método `setAltura` en la figura 7.12, como a continuación:

```
this.setAltura(altura, "cm");
```

Se apunta a esta alternativa de sintaxis no porque se quiera que se utilice, sino porque se quiere tener una visión más clara de los detalles de la llamada a objetos.

```
public void setAltura(double altura)
{
  setAltura(altura, "cm");
}
```

No poner una variable de referencia punto prefijo aquí.

Figura 7.12 Ejemplo de método que llama a otro método en la misma clase. Esto ayuda a evitar duplicación de detalles de código y posibles inconsistencias internas.

Evolución del programa

La habilidad de sobrecargar el nombre de un método promueve una evolución elegante porque corresponde con la manera en que el lenguaje natural regularmente sobrecarga significados de las palabras. Por ejemplo, la primera versión de su programa podría definir sólo la versión del método `setAltura` de un solo parámetro. Entonces, cuando se decide mejorar el programa, es más fácil para los usuarios existentes si se minimizan las nuevas cosas que han aprendido. En este caso, se les permite ya sea utilizar el método original o pasar al método mejorado. Cuando se quiera utilizar el método mejorado, todo lo que se tiene que recordar es el nombre original del método y agregar el segundo argumento para unidades a la llamada al método. Eso es una variación casi obvia, y es más fácil de recordar que un nombre de método diferente. Es mucho más fácil que forzar el aprendizaje de un nuevo nombre de método: lo cual sería un costo necesario de actualización si no existiera la sobrecarga.

Mantenga la simplicidad reutilizando nombres buenos.

7.8 Constructores

Hasta este punto, se han utilizado métodos de mutación para asignar valores a las variables de instancia en objetos recientemente instanciados. Eso funciona bien, pero es necesario tener y llamar a un método de mutación para cada variable de instancia. Como alternativa, se podría utilizar un simple método para inicializar todas las variables de instancia de un objeto tan pronto como sea posible después de que se crea un objeto. Por ejemplo, en este capítulo, en la figura 7.2 en la clase `Carro`, en lugar de definir tres métodos de mutación, se podría definir un simple método `iniciarCarro` para inicializar objetos `Carro`. Después se podría usar un código como el siguiente:

```
Carro carroAlex = new Carro();
carroAlex.iniciarCarro("Porsche", 2006, "beige");
```

Este fragmento de código utiliza una sentencia para asignar espacio a un nuevo objeto, y utiliza otra sentencia para inicializar las variables de instancia de ese objeto. Puesto que la creación de instancias y la inicialización de objetos son tan comunes, ¿no sería mejor si hubiera una simple sentencia que pudiera manejar ambas operaciones? Sí hay tal sentencia y ésta es:

```
Carro carroAlex = new Carro("Porsche", 2006, "beige");
```

Esto unifica la creación de un objeto y la inicialización de sus variables de instancia en sólo una llamada. Garantiza que las variables de instancia de un objeto se inicialicen tan pronto como el objeto sea creado. El código que sigue a la palabra `new` debe recordarle a usted la llamada a un método. Ambos, el código y la llamada a un método, consisten en una palabra definida por el programador (`Carro` en este caso) y después un paréntesis alrededor de los elementos. Se puede pensar en ese código como en una llamada a un método especial, pero es tan especial que tiene su propio nombre. Es utilizado para construir objetos, por lo que se le denomina *constructor.*

¿Qué es un constructor?

Un constructor es una entidad semejante a un método que es llamado automáticamente cuando se instancia un objeto. La creación de la instancia del objeto anterior `Carro("Porsche", 2006, "beige")` llama a un constructor llamado `Carro` que tiene tres parámetros: un `String`, un `int` y un `String`. He aquí un ejemplo de ese constructor:

```
public Carro(String m, int y, String c)
{
  this.marca = m;
  this.año = y;
  this.color = c;
}
```

Como se puede ver, este constructor simplemente asigna los valores de los parámetros pasados a sus variables de instancia correspondientes. Después de que el constructor se ejecuta, la JVM devuelve la dirección del nuevo objeto instanciado e inicializado al lugar donde fue llamado el constructor. En la

declaración anterior `Carro carroAlex = new Carro("Porsche", 2006, "beige")`, la dirección del objeto instanciado se asigna a la variable de referencia `carroAlex`.

Existen muchos detalles de construcción que se deben conocer antes de completar un programa ejemplo. El nombre de un constructor debe ser el mismo que la clase a la cual está asociado. Así, el constructor de una clase `Carro` debe ser `Carro`, con "C" mayúscula.

En el encabezado del método se debe incluir el tipo devuelto, por lo que se podría esperar que se tuviera este requisito para el encabezado de un constructor; pero no. Los tipos devueltos no se utilizan en los encabezados de los constructores[2] porque la llamada a un constructor (con `new`) devuelve automáticamente una referencia al objeto que construye y el tipo de este objeto siempre lo especifica el nombre del constructor mismo. Sólo se requiere especificar `public` a la izquierda y después escribir el nombre de la clase (que es el nombre del constructor).

Un ejemplo

Ahora se examinará un ejemplo de un programa completo que utiliza un constructor. Vea el programa Carro4 de las figuras 7.13 y 7.14. En la figura 7.13 observe que se ponen constructores encima del método `getMarca`. En todas las definiciones de clase, es un buen estilo poner los constructores sobre los métodos.

Colocación del constructor por omisión de Java

En cualquier momento que se instancia un objeto (con `new`), debe haber un constructor concordante. Esto es, el número y tipos de argumentos en el constructor deben concordar con el número y tipos de paráme-

[2] Si se intenta definir un constructor con una especificación del tipo devuelto, el compilador no lo reconocerá como un constructor y pensará, por el contrario, que se trata de un método.

```
/**************************************************
* Carro4.java
* Dean & Dean
*
* Esta clase almacena y trae los datos de un carro.
**************************************************/

public class Carro4
{
  private String marca;   // marca del carro
  private int año;        // año de manufactura del carro
  private String color;   // color primario del carro

  //*********************************************

  public Carro4(String m, int y, String c)
  {
    this.marca = m;
    this.año = y;
    this.color = c;
  } // end constructor

  //*********************************************

  public String getMarca()
  {
    return this.marca;
  } // fin getMarca
} // fin clase Carro4
```

definición del constructor

Figura 7.13 Clase `Carro4`, que tiene un constructor.

```
/*************************************************************
* ControladorCarro4.java
* Dean & Dean
*
* Esta clase es una demostración del controlador para la clase Carro4.
*************************************************************/

public class ControladorCarro4
{
  public static void main(String[] args)
  {
    Carro4 carroAlex = new Carro4("Porsche", 2006, "beige");
    Carro4 carroLatisha = new Carro4("Saturn", 2002, "red");

    System.out.println(carroAlex.getMarca());
  } // fin de main
} // fin de clase ControladorCarro4
```

llamadas a contructor

Salida:

```
Porsche
```

Figura 7.14 Clase `ControladorCarro4`, que controla a la clase `Carro4` de la figura 7.13.

tros en el constructor definido. Pero hasta hace poco se han instanciado objetos sin un constructor explícito. Entonces, ¿están mal los ejemplos? No. En todos ellos se utilizó un *constructor default* con cero parámetros que el compilador de Java proporciona de manera automática si y sólo si no hay un constructor explícitamente definido. El programa Empleado en las figuras 7.15*a* y 7.15*b* ilustra el uso del constructor de Java por omisión con cero parámetros.

En la figura 7.15*a* observe cómo el código de `main` `new Empleado()` llama a un constructor sin ningún parámetro; pero la figura 7.15*b* no define un constructor de este tipo. No hay problema; puesto que no hay otros constructores, el compilador de Java proporciona el constructor sin parámetros, y lo hace concordar con la llamada al constructor sin argumentos `new Empleado()`.

Observe que tan pronto como se define cualquier clase de constructor para una clase, el constructor por omisión de Java se vuelve indisponible. Por lo que si la clase contiene la definición de un constructor explícito y si `main` incluye una llamada a un constructor con cero argumentos, entonces ese constructor se debe incluir en la definición de la clase.

Observe el programa Empleado2 en las figuras 7.16*a* y 7.17*b*. La clase controlada en la figura 7.16*a* compila de manera exitosa, pero la controladora en la figura 7.16*b* genera un error de compilación. Como en la figura 7.15*a*, el código controlador de la figura 7.16*b* llama al constructor sin parámetros. Si funcionó antes, ¿por qué no funciona esta vez? Ahora, la clase controlada en la figura 7.16*a* define de manera explícita al constructor, por lo que Java no proporciona un constructor por omisión sin parámetros. Y sin ese constructor, el compilador protesta por no haber un constructor que concuerde con la llamada

```
public class ControladorEmpleado
{
  public static void main(String[] args)
  {
    Empleado emp = new Empleado();

    emp.leerNombre();
  } // fin de main
} // fin de la clase ControladorEmpleado
```

llamada al constructor sin parámetros

Figura 7.15*a* Controlador para el programa Empleado.

```
import java.util.Scanner;

public class Empleado
{
  private String nombre;

  //**************************************

  public void leerNombre()
  {
    Scanner stdIn = new Scanner(System.in);

    System.out.print("Nombre: ");
    this.nombre = stdIn.nextLine();
  } // fin de leerNombre
} // fin de la clase Empleado
```

Figura 7.15*b* Clase controlada para el programa Empleado. Éste trabaja aunque no se haya definido un constructor de manera explícita porque el compilador de Java proporciona un constructor con cero parámetros que concuerda.

al constructor sin parámetros. ¿Cómo arreglar el programa Empleado2 para eliminar este error? Agregue el siguiente constructor `Empleado2` sin parámetros a la clase `Empleado2`:

```
public Empleado2()
{ }
```

Éste es un ejemplo de un *constructor maniquí.* Se le denomina así porque no hace otra cosa que satisfacer al compilador. Observe cómo los paréntesis de llave están en una línea con un espacio en blanco entre ellas. Eso es un asunto de estilo. Escribir un constructor maniquí como ése, provoca que las llaves vacías

```
import java.util.Scanner;

public class Empleado2
{
  private String nombre;

  //**************************************

  public Employee2(String n)
  {
    this.nombre = n;
  } // fin del constructor

  //**************************************

  public void leerNombre()
  {
    Scanner stdIn = new Scanner(System.in);

    System.out.print("Nombre: ");
    this.nombre = stdIn.nextLine();
  } // fin de leerNombre
} // fin de la clase Empleado2
```

Figura 7.16*a* Clase controlada para el programa Empleado2.

```
public class ControladorEmpleado2
{
  public static void main(String[] args)
  {
    Empleado2 mesera = new Empleado2("Wen-Jung Hsin");
    Empleado2 azafata = new Empleado2();

    azafata.leerNombre();
  } // fin de main
} // fin de la clase ControladorEmpleado2
```

La llamada al constructor sin parámetros genera un error de compilación.

Figura 7.16*b* Controlador para el programa Empleado2.

sean más prominentes y que muestren claramente el intento del programador de hacer del constructor un constructor maniquí.

Inicialización de constantes nombradas

Si se incluye el modificador `final` en la declaración de una variable de instancia, esa "variable" se convierte en una constante nombrada. Siempre que se utilice `final`, es un buen estilo escribir el nombre de la variable en mayúsculas. En el capítulo 3 se utilizó `final` y mayúsculas para declarar e inicializar constantes nombradas como esto:

```
final double PUNTO_CONGELAMIENTO = 32.0;
```

En este punto, todas las declaraciones de constantes nombradas estaban dentro de un método (el método `main`). Cuando una constante nombrada se define dentro de un método, se llama *constante nombrada local*, y su alcance se limita a ese método. Si se quiere un atributo que sea constante a lo largo de un objeto particular, será necesaria otra clase de constante nombrada, una *constante de instancia*. Se declara esta clase de constante nombrada al principio de la clase, pero normalmente no se inicializa en la declaración. En lugar de eso, se inicializa en un constructor. Esto permite inicializar constantes de instancia con valores diferentes para objetos diferentes. Por tanto, una constante de instancia puede representar un atributo cuyo valor varía de un objeto a otro, pero que permanece constante a lo largo de la vida de un objeto en particular. Representa un atributo inalienable de ese objeto, un atributo que permanentemente distingue ese objeto de todos los otros objetos en la misma clase. Debido a que el modificador `final` evita que una constante nombrada se cambie después de que se inicializa, es seguro hacer una constante de instancia `pública`. Esto hace fácilmente determinar el valor de los atributos permanentes de un objeto. Basta con utilizar esta sintaxis:

<*variable de referencia*>.<*constante de instancia*>

Por ejemplo, en lugar de tratar el nombre de un empleado como una variable de instancia, tal como se hizo en las clases `Empleado` y `Empleado2`, se puede tratar como una constante de instancia, como en la clase `Empleado3` de la figura 7.17*a*.

Observe que la clase `Empleado3` no incluye un constructor con cero parámetros. ¿Por qué no se incluye aquí? Porque se quiere forzar el uso de un constructor de un solo parámetro para asegurarse de que `NOMBRE` sea inicializado con un valor diferente que resulte apropiado para cada objeto. Para manejar la clase `Empleado3` se puede utilizar algo como lo que se tiene en la figura 7.17*b*. Observe cómo el modificador `public` en la constante de instancia de la figura 7.17*a* hace posible acceder a este valor constante directamente de otra clase.

Elegancia

Observe que el uso de los constructores definidos por el programador no es nunca un requerimiento absoluto. Aunque fuera en contra de su naturaleza, se podrían inicializar constantes de instancia cuando se declaren. Y siempre se puede instanciar un objeto con paréntesis vacío y después llamar a un método de inicialización para inicializar variables de instancia, tal como se hizo antes. Entonces, ¿por qué molestarse

```
/*********************************************************
* Empleado3.java
* Dean & Dean
*
* Este programa otorga un nombre de empleado permanente.
*********************************************************/

import java.util.Scanner;

public class Empleado3
{
  Scanner stdIn = new Scanner(System.in);
  public final String NOMBRE;

  //*****************************************************

  public Empleado3(String nombre)
  {
    this.NOMBRE = nombre;
  } // fin del constructor
} // fin de la clase Empleado3
```

declaración de constante de instancia

inicialización de constante de instancia

Figura 7.17*a* Clase `Empleado3`, que utiliza una constante de instancia.

en utilizar constructores definidos por el programador? Si se quiere distinguir constantes de instancia, se debe inicializarlas en un constructor: el compilador no permitirá hacerlo en un método. Siempre que se requiera inicializar las variables de instancia de un objeto, es más elegante hacerlo con el constructor que instancia el objeto. El constructor de manera íntima une constantes y variables de instancia con la creación del objeto. Los constructores simplifican las cosas evitando un paso de inicialización separado, y no se requieren nombres separados para ellos porque utilizan el nombre de la clase. ¡Bravo, constructores!

```
/*********************************************************
* ControladorEmpleado3.java
* Dean & Dean
*
* Ésta instancia un objeto e imprime un atributo permanente.
*********************************************************/

import java.util.Scanner;

public class ControladorEmpleado3
{
  public static void main(String[] args)
  {
    Empleado3 mesera = new Empleado3("Angie Klein");

    System.out.println(mesera.NOMBRE);
  } // fin de main
} // fin de la clase ControladorEmpleado3
```

acceso directo a la constante de instancia

Salida:

```
Angie Klein
```

Figura 7.17*b* Controla la clase `Empleado3` de la figura 7.17*a*.

7.9 Constructores sobrecargados

Sobrecargar un constructor es como sobrecargar un método. La sobrecarga de los constructores ocurre cuando hay dos o más constructores con el mismo nombre y diferentes parámetros. Los constructores sobrecargados son muy comunes (más comunes que los métodos sobrecargados). Es por ello que con frecuencia se querrá poder crear objetos con diferentes cantidades de inicialización. A veces, se querrá pasar valores iniciales al constructor. Otras veces querrá abstenerse de hacerlo, y dejar la asignación de valores para más tarde. Para permitir ambos escenarios, son necesarios constructores sobrecargados: un constructor con parámetros y otro sin ellos.

Un ejemplo

Suponga que se desea implementar la clase `Fraccion`, que almacena el numerador y denominador para una fracción determinada. La clase `Fraccion` también almacena el cociente, que se genera al dividir el numerador entre el denominador. Normalmente, se desea instanciar la clase `Fraccion` mediante el paso de los argumentos numerador y denominador a un constructor `Fraccion` de dos parámetros. Pero para un número completo, se deseará instanciar una clase `Fraccion` pasando sólo un argumento (el número entero) a un constructor de esta clase, en lugar de pasar dos argumentos. Por ejemplo, para instanciar un número 3 como objeto `Fraccion`, se querrá pasar sólo un 3 al constructor, en lugar de 3 como numerador y 1 como denominador. Para manejar instancias de dos parámetros de `Fraccion`, así como de uno solo, se requieren constructores sobrecargados. Una forma de resolver este problema es escribir un controlador que muestre cómo se desea utilizar la solución. Con esto en mente, se presenta un controlador en la figura 7.18 que ilustra cómo la clase propuesta `Fraccion` y sus constructores sobrecargados se pueden utilizar. El código del controlador incluye los números de líneas para facilitar su rastreo.

Asuma que dentro de la clase `Fraccion`, el `numerador` y `denominador` son variables de instancia de tipo `int` y que `cociente` es una variable de instancia de tipo `double`. El constructor de dos parámetros debe parecerse al siguiente:

```
public Fraccion(int n, int d)
{
  this.numerador = n;
  this.denominador = d;
  this.cociente = (double) this.numerador / this.denominador;
}
```

¿Por qué la conversión (`double`)? Sin ella, se obtendría una división entera y se truncarían los valores fraccionales. La conversión realiza el cambio de `numerador` en `double`, el numerador `double` promueve la variable de instancia `denominador` a `double`, la división de punto flotante ocurre, y se preservan los valores fraccionales. La conversión a `double` también proporciona una respuesta más elegante si el denominador es cero. La división entre cero provoca que el programa colisione; pero la división de punto flotante entre cero sí está permitida. En lugar de fallar, el programa imprime "Infinity" (Infinito) si el numerador es positivo, o "-Infinity" si el numerador es negativo.

Hágalo robusto.

Para un número entero como 3, se podría llamar al constructor con dos parámetros teniendo a 3 como primer argumento y a 1 como el segundo. Pero se quiere que la clase `Fraccion` sea amigable. Se quiere que tenga otro constructor (sobrecargado) con un solo parámetro. Este constructor de un solo parámetro se podría parecer al que se muestra a continuación:

```
public Fraccion(int n)
{
  this.numerador = n;
  this.denominador = 1;
  this.cociente = (double) this.numerador;
}
```

```
 1  /*******************************************************
 2  * ControladorFraccion.java
 3  * Dean & Dean
 4  *
 5  * Esta clase controladora hace uso de la clase Fraccion.
 6  *******************************************************/
 7
 8  public class ControladorFraccion
 9  {
10    public static void main(String[] args)
11    {
12      Fraccion a = new Fraccion(3, 4);
13      Fraccion b = new Fraccion(3);
14
15      a.Imprimelo();
16      b.Imprimelo();
17    } // fin de main
18  } // fin de clase ControladorFraccion
```

llama a constructores sobrecargados

Sesión muestra:

```
3 / 4 = 0.75
3 / 1 = 3.0
```

Figura 7.18 Clase `ControladorFraccion`, que controla la clase `Fraccion` de la figura 7.19.

Llamada a un constructor desde otro constructor

Evite duplicar código.

Los dos constructores anteriores contienen código duplicado. La duplicación hace los programas más largos. De manera más importante, posibilitan la inconsistencia. Antes se utilizaron métodos sobrecargados para evitar este tipo de peligro. En lugar de repetir código como en la figura 7.10, en la figura 7.12 se insertó una llamada a un método previamente escrito que ya tuvo el código que se quería. Eso mismo se hace con los constructores, esto es, se puede llamar a un constructor previamente escrito a partir de otro constructor. Las llamadas a los constructores son diferentes desde las llamadas a los métodos en los que se utiliza la palabra reservada `new`, la cual le pide a la JVM asignar espacio en memoria para un nuevo objeto. Dentro del constructor original se podría utilizar el operador `new` para llamar a otro constructor. Pero eso crearía un objeto separado del objeto original; y la mayoría de las veces, eso no es lo que se quiere. Normalmente, cuando se llama a un constructor sobrecargado, se quiere trabajar con el objeto original, no con un nuevo objeto aparte.

Para evitar crear un objeto separado, los diseñadores de Java descubrieron una sintaxis especial que permite al constructor llamar a uno de sus constructores socios sobrecargados, de tal manera que el objeto original se utilice. He aquí la sintaxis:

`this(`<*argumentos-para-constructor-meta*>`);`

Una llamada a un constructor `this`(<*argumentos-para-constructor-meta*>) puede aparecer sólo en una definición de un constructor, y debe aparecer en la primera sentencia de la definición del constructor. Eso significa que no se puede utilizar la sintaxis `this` para llamar a un constructor desde adentro de la definición de un método. También significa que se puede tener sólo una llamada a tal constructor en la definición de un constructor porque sólo una sentencia de llamada podría ser la primera sentencia en la definición de un constructor.

Ahora revise la clase `Fraccion` en la figura 7.19, que tiene tres variables de instancia: `numerador`, `denominador` y `cociente`. La variable de instancia `cociente` contiene el resultado en punto flotante de dividir el numerador entre el denominador. El primer constructor es como el constructor de dos parámetros que se escribió antes. Pero el segundo constructor es más corto. En lugar de repetir el código que aparece en el primer constructor, llama al primer constructor con el comando `this(...)`.

Suponga que durante el desarrollo del programa, se decide con propósitos de depuración imprimir "En constructor de 1 parámetro" desde dentro del constructor de un parámetro de la clase `Fraccion`.

```
 1  /*************************************************************
 2  * Fraccion.java
 3  * Dean & Dean
 4  *
 5  * Esta clase almacena e imprime fracciones.
 6  *************************************************************/
 7
 8  public class Fraccion
 9  {
10    private int numerador;
11    private int denominador;
12    private double cociente;
13
14    //***********************************************************
15
16    public Fraccion(int n)
17    {
18      this(n, 1);
19    }
20
21  //***********************************************************
22
23    public Fraccion(int n, int d)
24    {
25      this.numerador = n;
26      this.denominador = d;
27      this.cociente = (double) this.numerador / this.denominador;
28    }
29
30    //***********************************************************
31
32    public void Imprimelo()
33    {
34      System.out.Imprimelo(this.numerador + " / " +
35        this.denominador + " = " + this.cociente);
36    } // fin de Imprimelo
37  } // fin de clase Fraccion
```

Esta sentencia llama al otro constructor.

Figura 7.19 Clase `Fraccion` con constructores sobrecargados.

¿Dónde se pondría esa sentencia de impresión? Puesto que la llamada al constructor `this(n, 1)` debe ser la primera sentencia en la definición del constructor, se tendría que poner la primera sentencia debajo de la llamada al constructor.

Rastreo con constructores

La figura 7.20 muestra un rastreo del programa Fraccion. En la siguiente explicación sobre ello, será necesario referirse activamente no sólo a la figura de rastreo, sino también a la clase `ControladorFraccion` (figura 7.18) y a la clase `Fraccion` (figura 7.19). Observe cómo la línea 12 en la clase `ControladorFraccion` pasa 3 y 4 al constructor de dos parámetros. Los números 3 y 4 se asignan a los parámetros *n* y *d*. Como parte de la funcionalidad del constructor de las líneas 10-12 de la clase `Fraccion`, se ejecutan e inicializan las variables de instancia `Fraccion` con sus valores por omisión. Entonces las líneas 25-27 sobrescriben esos valores inicializados. Volviendo a `ControladorFraccion`, `new` devuelve una referencia a un objeto (obj1) a la variable de referencia `a`. Entonces en la línea 13, el controlador pasa 3 al constructor de un parámetro. Después de una asignación de parámetro y la inicialización de variables, en la línea 18 de la clase `Fraccion` se pasa 3 y 1 al constructor de dos parámetros. Después el constructor de dos parámetros sobrescribe las variables de instancia, el flujo de con-

ControladorFraccion			Fraccion											
	main			Fraccion		Fraccion		obj1			obj2			
línea#	a	b	*línea#*	n	d	n	this	num	den	coc	num	den	coc	salida
12				3	4									
			10					0						
			11						0					
			12							0.00				
			25					3						
			26						4					
			27							0.75				
12	obj1													
13						3								
			10								0			
			11									0		
			12										0.00	
			18	3	1									
			25								3			
			26									1		
			27										3.00	
13		obj2												
15							obj1							
			34											3 / 4 = 0.75
16							obj2							
			34											3 / 1 = 3.00

Figura 7.20 Rastreo de programa Fraccion en las figuras 7.18 y 7.19.

trol vuelve al constructor de 1 parámetro, y vuelve a `ControladorFraccion`, donde `new` devuelve una referencia a un objeto (obj2) a la variable de referencia, variable `b`. Finalmente, en las líneas 15 y 16, el controlador imprime los dos resultados.

7.10 Resolución de problemas con diversas clases

Se inició de una manera simple y gradualmente se ha añadido complejidad. De los capítulos 1 a 5 se mostraron programas que contienen sólo una clase y un método (el método `main`). En los capítulos 6 y 7 se han visto programas que contienen dos clases: 1) una clase controladora, que contiene un sencillo método `main`, y 2) una clase controlada, que contiene típicamente varios métodos.

Hasta ahora se ha utilizado sólo una clase controlada para mantener simples las cosas, pero en el mundo real será necesario tener más de una clase controlada. Esto porque la mayoría de los programas del mundo real son heterogéneos (contienen mezclas de diferentes cosas). Para cada tipo de cosa, es apropiado tener una clase diferente. Tener más de una clase controlada le permite dividir un problema en varios problemas más simples. Eso hace posible estudiar un tipo de cosa a la vez. Cuando se ha terminado ésta, se puede continuar con otra. De esta forma, paso a paso, se puede construir gradualmente un programa largo.

No es un gran trabajo controlar más de una clase controlada desde un simple controlador. De hecho, esto ya se vio antes en el capítulo 5 cuando la sentencia en un sencillo método `main` llamó a los métodos desde más de una clase envoltorio, como `Integer` y `Double`. La única cosa que hay que recordar es que cuando se está compilando el controlador, el compilador debe poder encontrar todas las clases controladas. Si son clases preconstruidas deben ser parte del paquete `java.lang` o se deben importar. Si son las clases que se escriben, deben estar en el mismo directorio que el controlador.[3]

[3] Es posible poner sus propias clases en sus propios paquetes en directorios separados e importarlas como se importan las clases preconstruidas. Puede aprender acerca de esto, en el apéndice 4. Sin embargo, si sus clases controladas están en el mismo directorio de su clase controladora no es necesario colocarlas en un paquete e importarlas, y se asume que éste es el caso a lo largo del libro.

Ejemplo: controlador de apertura de la puerta de la cochera

A manera de ejemplo, suponga que se quiere escribir un programa que modela la operación de un control de apertura de la puerta de una cochera. Un sistema típico contiene cuatro componentes de control: un botón de presión, un interruptor de cierre, un interruptor de apertura y un controlador. El botón de presión comienza el movimiento de la puerta, o si la puerta se está moviendo y no ha alcanzado el punto final de su trayecto, el botón de presión detiene el movimiento. Siempre que la puerta para, su trayectoria cambia a la dirección contraria a la que iba. El interruptor de subida realiza la apertura abriendo sus contactos y deteniendo la puerta en su límite superior. Los contactos del interruptor superior se cierran otra vez cuando la puerta vuelve hacia abajo. El interruptor inferior detiene la trayectoria hacia abajo abriendo sus contactos y parando el movimiento de la puerta en su límite inferior. Los contactos del interruptor inferior se cierran otra vez cuando la puerta comienza a subir.

El controlador interpreta la información de los diferentes interruptores y opera el motor que sube y baja la puerta. El sistema tiene cuatro diferentes estados: puerta detenida después de ir hacia abajo, que se denominará estado #0; puerta hacia arriba, estado #1; puerta detenida después de ir hacia arriba, estado #2; puerta hacia abajo, estado #3.

He aquí el tipo de cosas que se desea que haga el programa:

Sesión muestra:

```
Puerta inicialmente hacia abajo.
Introduzca el número de operación: 8
Introduzca 'b' para botón o 'f' para interruptor final: f
Ahora detenida. Introduzca 'b': b
Botón presionado. Moviendo hacia arriba.
Introduzca 'b' para botón o 'f' para interruptor final: f
Límite superior alcanzado. La puerta está arriba.
Introduzca 'b' para botón o 'f' para interruptor final: b
Botón presionado. Puerta moviendo hacia abajo.
Introduzca 'b' para botón o 'f' para interruptor final: b
Botón presionado. Puerta detenida por el botón.
Introduzca 'b' para botón o 'f' para interruptor final: f
Ahora detenida. Introduzca 'b': b
Botón presionado. Moviendo hacia arriba.
Introduzca 'b' para botón o 'f' para interruptor final: f
Límite superior alcanzado. La puerta está hasta arriba.
Introduzca 'b' para botón o 'f' para interruptor final: b
Botón presionado. Puerta moviendo hacia abajo.
Introduzca 'b' para botón o 'f' para interruptor final: f
Límite inferior alcanzado. La puerta está abajo.
```

Utilice una clase separada para cada tipo de tarea.

Ya que se ha descrito el problema y que se ha dicho lo que se quiere que haga el programa, se analizará el problema para ver cómo organizar el programa.

Los interruptores inferiores y superiores son automáticos y el botón de contacto contiene un transmisor de radio. Pero con un punto de vista desde el modelado, se puede pensar en el botón de presión como automático, como en los interruptores finales. Así los dos interruptores y el botón de contacto pueden ser tres instancias de una cosa llamada "switch". Esto sugiere que se escriba una clase `Switch` y se construyan tres objetos de ello: un `switchSuperior`, un `switchInferior` y un `boton`.

Aunque los tres interruptores son similares uno a otro, todos son diferentes del controlador que comparte información de ellos y hace cosas para cambiar el estado de la puerta. Así tiene sentido utilizar una clase separada para el controlador: una clase `ControladorPuertaCochera`. Se hará que el controlador construya un objeto de ella: un `control`.

Existe también la puerta, que es lo que importa realmente. Se puede pensar en la puerta en el sentido amplio como otro componente, o se puede pensar en ella en el sentido amplio como el sistema: el `Sis-`

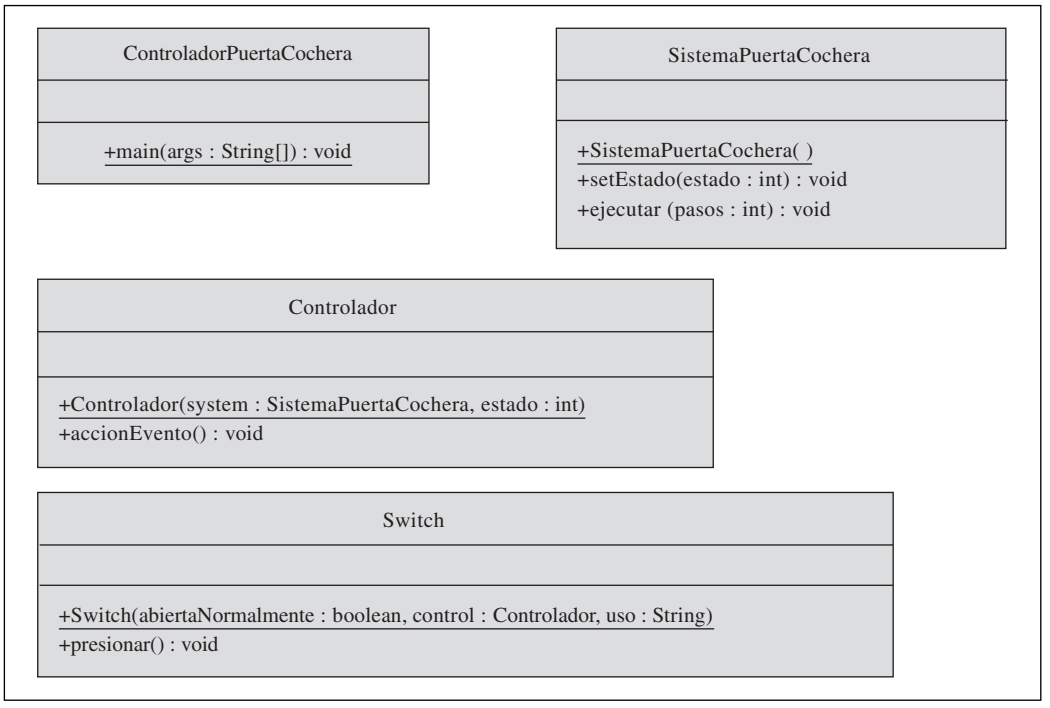


Figura 7.21 Diagrama UML genérico para el programa Puerta de cochera.

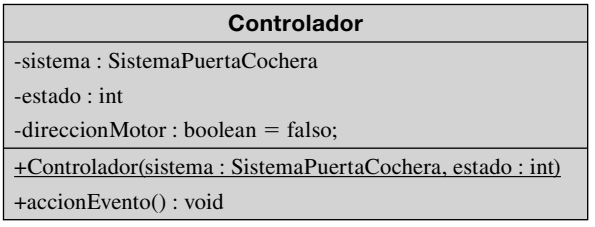
`temaPuertaCochera`. Un sistema es un objeto que contiene otros objetos (sus componentes), y que los conoce. En el presente ejemplo, el objeto `SistemaPuertaCochera` contiene la puerta, el controlador que mueve la puerta y los tres interruptores que envían las señales al controlador.

La figura 7.21 muestra un diagrama de clases genérico del lenguaje de modelado unificado (UML) para el programa. Este diagrama genérico muestra métodos `públicos`, pero no variables o constantes de instancia. Cada una de las clases controladas contiene un simple constructor. Observe que cada constructor está subrayado. Eso atiende a los estándares de UML, los cuales sugieren subrayar todos los constructores.

De manera usual, la clase `SistemaPuertaCochera` tiene sólo un método `main`. Puesto que se planea realizar la mayor parte de las actividades en la clase `SistemaPuertaCochera` en lugar de hacerlo en la clase controladora, ese método `main` puede ser muy sencillo. Como se puede ver en la figura 7.22, el controlador realiza sólo dos cosas: construye un objeto `SistemaPuertaCochera` y corre y prueba ese objeto.

Observe que este controlador no construye uno de los componentes del sistema, sino que delega este trabajo en el constructor `SistemaPuertaCochera`. Puesto que la construcción del componente depende de lo que parezcan los componentes, se observarán esos componentes a continuación. La figura 7.23 contiene el código para el controlador.

Como se anotó en el código de la figura 7.23 se tiene la seguridad de la necesidad de variables de instancia. Éstas expanden el diagrama UML de clases de la clase `Controlador` de la siguiente manera:



```
/************************************************************
* ControladorPuertaCochera.java
* Dean & Dean
*
* Este programa simula la instalación y prueba.
************************************************************/

import java.util.Scanner;

public class ControladorPuertaCochera
{
  public static void main(String[] args)
  {
    SistemaPuertaCochera sistema;
    Scanner stdIn = new Scanner(System.in);

    // Instala sistema
    sistema = new SistemaPuertaCochera();

    // Prueba sistema
    System.out.print("Introduzca el número de operaciones: ");
    sistema.ejecutar(stdIn.nextInt());
  } // fin de main
} // fin de clase ControladorPuertaCochera
```

Figura 7.22 Programa para Controlador de puerta de cochera.

La variable de instancia `sistema` le da al controlador una referencia al sistema que la incluye. Se querrá que el constructor inicialice o reinicialice las variables de instancia `sistema`, `estado` y `direccionMotor`.

Las variables de instancia `estado` y `direccionMotor` representan transmisiones electromecánicas o flip flops de estado sólido (elementos de memoria electrónica primitivos). Estos elementos de memoria primitiva mantienen el rastro del estado actual del controlador: su modo de operación actual. En los controladores físicos, estos elementos de memoria primitiva determinan lo que hace el controlador. El método `accionEvento` cambia los valores de estos elementos de memoria principal cuando es llamado por un interruptor (switch). Cada vez que el método es llamado, el estado se incrementa en uno, *módulo cuatro*. Modular cuatro significa que el valor de `estado` entra en un ciclo de cuatro valores como esto: 0, 1, 2, 3, 0, 1, 2, etc. También cuando el motor se para, se va en dirección contraria. Esto es cuando el `estado` cambia a un número par (0 o 2), la variable `direccionMotor` realiza el cambio al valor de tipo `boolean` opuesto.

La figura 7.24 contiene el código que define los cambios. Cada interruptor está caracterizado por una constante de instancia nombrada; `NORMALMENTE_ABIERTA`. `NORMALMENTE_ABIERTA` es una propiedad del interruptor de contacto. Si `NORMALMENTE_ABIERTA` es `true`, cuando se presiona el interruptor, sus contactos se cierran. Así cuando se presionen ya sea uno o los dos interruptores de límite, los contactos se abren y esto detiene el flujo actual al motor. La variable de instancia, `uso`, indica cómo está unido el interruptor al sistema. La variable de instancia `Controlador` es una referencia al controlador a la cual está unido el interruptor. (Si existen dos sistemas de puertas de cochera, un interruptor en particular podría estar asociado con cualquier controlador del sistema.) El constructor inicializa estos tres valores. El método `presionar` imprime un mensaje que identifica el interruptor que fue presionado. Después llama al método `accionEvento` de `control` en la clase `Controlador`.

La constante de instancia y las variables de instancia de la figura 7.24 expanden el diagrama UML de clase para la clase `Switch` a esto:

```
/*****************************************************************
* Controlador.java
* Dean & Dean
*
* Esta clase modela el controlador con sensores.
*****************************************************************/

public class Controlador
{
  private SistemaPuertaCochera sistema;
  private int estado;    // 0=abajo, 1=hacia arriba, 2=arriba, 3=hacia abajo
  private boolean direccionMotor = false;         // true = go up

  //***************************************************************

  public Controlador(SistemaPuertaCochera sistema, int estado)
  {
    this.sistema = sistema;
    this.estado = estado;
    if (estado < 2)
    {
      this.direccionMotor = true;
    }
  } // fin de constructor

  //***************************************************************

  public void accionEvento()
  {
    this.estado++;
    this.estado %= 4;
    if (this.estado % 2 == 0)
    {
      this.direccionMotor = !this.direccionMotor;
    }
    sistema.setEstado(this.estado);
  } // fin accionEvento
} // fin de la clase Controlador
```

Figura 7.23 Clase `Controlador` para el programa Puerta de cochera.

Switch
+NORMALMENTE_ABIERTA : boolean +uso : String -control : Controlador
+Switch(normalmenteAbierto : boolean, control : Controlador, uso : String) +presionar() : void

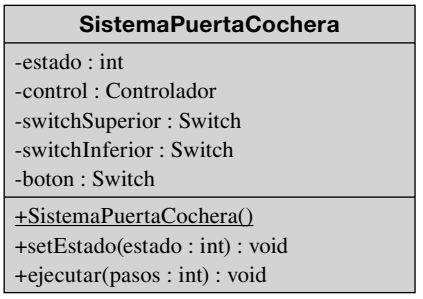
Ahora estamos listos para la clase `SistemaPuertaCochera` mostrada en las figuras 7.25*a* y 7.25*b*. En la figura 7.25*a* se declara una variable de instancia llamada `estado`. Después se declaran las cuatro variables de referencia que se refieren a los cuatro componentes en el sistema. El constructor instancia todos los objetos componentes e inicializa todas las variables de referencia con referencias a esos objetos componentes. Inicializa `estado` a 0, correspondiente a la posición de puerta hacia abajo y llama al constructor de `Controlador`, le pasa el valor de `estado` al nuevo objeto `control` para sincronizar el estado de ese objeto con el estado del sistema completo. El método `setEstado` proporciona una

```
/*********************************************************************
* Switch.java
* Dean & Dean
*
* Esta clase modela interruptores.
*********************************************************************/

import java.util.Scanner;

public class Switch
{
  public final boolean NORMALMENTE_ABIERTO;  // presion hace conexion
  public String uso;                         // papel en el sistema
  private Controlador control;

  //*******************************************************************

  public Switch(
    boolean normalmenteAbierto, Controlador control, String uso)
  {
    this.NORMALMENTE_ABIERTO = normalmenteAbierto;
    this.control = control;
    this.uso = uso;
  } // fin de constructor

  //*******************************************************************

  public void presionar()
  {
    System.out.print(this.uso + " interruptor presionado. ");
    control.accionEvento();
  } // fin de presion
} // fin de la clase Switch
```

Figura 7.24 Clase `Switch` para el programa Puerta de cochera.

forma para que el objeto subordinado `control` mantenga sincronizado el estado del objeto con el estado del controlador inmediatamente después de que éste realiza la acción que cambia el estado.

Las variables de instancia en la figura 7.25*a* expanden el diagrama UML de clase de la clase `SistemaPuertaCochera` a esto:



La figura 7.25*b* contiene el resto de la clase `SistemaPuertaCochera`. Todo esto no es más que un enorme método, el método `ejecutar`, el cual describe la operación de apertura de la puerta de la cochera: el proceso del sistema. Es un ciclo grande que toma un número específico de pasos. En cada paso, el usuario especifica uno de los dos tipos de eventos, ya sea la presión de un botón o la llegada a un límite

```
/*******************************************************************
* SistemaPuertaCochera.java
* Dean & Dean
*
* Esta clase representa la puerta de una cochera.
*******************************************************************/

import java.util.Scanner;

public class SistemaPuertaCochera
{
  private int estado;          // 0=abajo, 1=hacia arriba, 2=arriba, 3=hacia abajo
  private Controlador control;
  private Switch switchSuperior;    // interruptor de limite superior
  private Switch switchInferior;    // interruptor de limite inferior
  private Switch boton;             // boton de presion electronico

  //*****************************************************************

  public SistemaPuertaCochera()
  {
    this.estado = 0;
    System.out.println("Puerta inicialmente hacia abajo.");
    this.control = new Controlador(this, this.estado);
    this.switchSuperior =
      new Switch(false, this.control, "Limite superior");
    this.switchInferior =
      new Switch(false, this.control, "Limite inferior");
    this.boton = new Switch(true, this.control, "Boton");
  } // end constructor

  //*****************************************************************

  public void setEstado(int estado)
  {
    this.estado = estado;
  }

  //*****************************************************************
```

Figura 7.25*a* Clase `SistemaPuertaCochera` para el programa Puerta de cochera, parte A.

del trayecto. Un ciclo grande utiliza el valor de entrada para imprimir un mensaje apropiado y quizá solicita que se vuelvan a introducir datos. El delegar este detalle en una clase subordinada es más elegante que tratar de manejar todo en el método `main` de un controlador.

Resumen

- Cuando se declara una variable de referencia, la JVM destina espacio en memoria para almacenar una referencia a un objeto. En este punto, no hay memoria destinada para el objeto mismo.
- La asignación de una variable de referencia a otra no crea un clon del objeto. Simplemente hace que ambas variables de referencia se refieran al mismo objeto y le da a dicho objeto un nombre alterno: un alias.
- Para crear un objeto separado se debe utilizar el operador `new`. Para hacer que el segundo objeto sea como el primero, se copian los valores de las variables de instancia del primer objeto en las variables de instancia del segundo objeto.
- Un método puede devolver una variedad de datos originados en un método, devolviendo una referencia a un objeto internamente instanciado que contenga datos.

```
  public void ejecutar(int pasos)
  {
    Scanner stdIn = new Scanner(System.in);
    char entrada;
    boolean correcto = false;

    for (int paso=0; paso<paso; paso++)
    {
      System.out.print(
        "Introduzca 'b' para botón o 'f' para interruptor final: ");
      do
      {
        entrada = stdIn.nextLine().charAt(0);
        if (entrada == 'b')
        {
          button.presionar();
          switch (estado)
          {
            case 0: case 2:
              System.out.println("Botón presionado.");
              break;
            case 1:
              System.out.println("Moviendo hacia arriba.");
              break;
            case 3:
              System.out.println("Moviendo hacia abajo.");
          } // fin de switch
          correcto = true;
        }
        else
        {
          switch (estado)
          {
            case 1:
              switchSuperior.presionar();
              System.out.println("La puerta está arriba.");
              correcto = true;
              break;
            case 3:
              switchInferior.presionar();
              System.out.println("La puerta está abajo.");
              correcto = true;
              break;
            default:
              System.out.print("Ahora detenida. Introduzca 'b': ");
              correcto = false;
          } // fin de switch
        } // fin de if
      } while (!correcto);
    } // fin del for
  } // fin de ejecutar
} // fin de la clase SistemaPuertaCochera
```

Figura 7.25*b* Clase `SistemaPuertaCochera` para el programa Puerta de cochera, parte B.

- El programa de recolección de basura de Java busca objetos inaccesibles y recicla el espacio que éstos ocupan pidiendo al sistema operativo que designe el espacio de los mismos como espacio libre.
- Si se comparan dos referencias de objetos con el operador `==`, el resultado es `true` (*verdadero*) si y sólo si las referencias apuntan al mismo objeto.

- Para verificar si dos objetos similares contienen datos similares, se debe escribir un método `equals` que compare de manera individual los valores de las respectivas variables de instancia.
- Para intercambiar los valores de dos variables, es necesario almacenar uno de los valores de las variables en una variable temporal.
- Si se pasa una referencia como argumento, y las variables de instancia del parámetro de la referencia son actualizadas, entonces se actualiza simultáneamente a las variables de instancia del argumento de referencia en el módulo que realiza la llamada.
- Si un método devuelve una referencia a un objeto, se puede utilizar lo que fue devuelto para llamar a otro método en la misma sentencia. Eso es lo que se conoce como encadenamiento de llamadas a métodos.
- Para hacer más entendible un programa, se puede *sobrecargar* el nombre de un método utilizando el mismo nombre otra vez en una definición diferente del método que tenga diferente secuencia de tipo de parámetros. La combinación de nombre de método, número de parámetros y tipo de parámetros es lo que se denomina *firma*.
- Un constructor permite inicializar, de manera separada, variables de instancia para cada objeto. El nombre de un constructor es el mismo que el nombre de la clase, y no se especifica tipo de valor que devuelve.
- Para que funcione la llamada a un constructor, debe haber una definición de constructor que concuerde, esto es, una definición con la misma firma.
- Si se define un constructor, el constructor por omisión sin parámetros desaparece.
- Utilice un constructor para inicializar constantes de instancia, las cuales representan atributos permanentes de objetos individuales.
- Para llamar a un constructor sobrecargado desde dentro de un constructor, utilice como primera sentencia lo siguiente: `this`(<*argumentos-del-constructor*>).
- Divida un problema largo en un conjunto de problemas más simples utilizando clases controladas.

Preguntas de revisión

§7.2 Creación de objetos: un análisis detallado

1. La sentencia

```
Carro carro;
```

destina espacio en memoria para un objeto. (F/V)

2. ¿Qué hace el operador `new`?

§7.3 Asignación de una referencia

3. La asignación de una variable de referencia copia las variables de instancia del objeto del lado derecho a las variables de instancia del objeto del lado izquierdo. (F/V)

4. ¿Qué es la fuga de memoria?

§7.4 Prueba de objetos para igualdad

5. Considere el siguiente fragmento de código:

```
boolean igual;
Carro carroX = new Carro();
Carro carroY = carroX;
igual = (carroX == carroY);
```

¿Cuál es el valor final para `igual`?

6. ¿Qué tipo devuelve el método `equals`?

7. Por convención, se utiliza el nombre `equals` para métodos que ejecutan cierta clase de evaluación. ¿Cuál es la diferencia entre la evaluación ejecutada por un método `equals` y el operador `==`?

§7.5 Paso de referencias como argumentos

8. Cuando se pasa una referencia a un método se permite modificar el objeto referenciado. (C/F)

§7.6 Encadenamiento de llamada a métodos

9. ¿Cuáles son las dos cosas que se deben incluir en la definición de un método para que pueda ser llamado como parte de una sentencia de encadenamiento de llamadas a métodos?

§7.7 Sobrecarga de métodos

10. ¿Cómo se llama al hecho de tener dos o más métodos con el mismo nombre en la misma clase?

11. Si se quiere que el objeto actual llame a un método diferente en la misma clase en la que se está en ese momento, la llamada al método es fácil: simplemente se llama al método directamente, sin el prefijo de la variable de referencia. (F/V)

§7.8 Constructores

12. ¿Cuál es el tipo devuelto por un constructor?

13. El nombre de un constructor debe ser exactamente el mismo que el nombre de la clase. (F/V)

14. Las convenciones estándar de codificación sugieren que se pongan definiciones de constructores después de las definiciones de todos los métodos. (F/V)

§7.9 Constructores sobrecargados

15. Si el código fuente de una clase contiene un solo constructor de un parámetro, el constructor es sobrecargado porque este constructor de un solo parámetro tiene el mismo nombre que el constructor por omisión sin parámetros. (F/V)

16. Suponga que tiene una clase con dos constructores. ¿Cuáles son las reglas para llamar a un constructor desde el otro constructor?

§7.10 Resolución de problemas con múltiples clases controladas

17. Un problema se puede dividir en varios problemas pequeños utilizando muchas clases controladas. (F/V)

18. ¿Cómo se le da a un objeto componente una referencia a su contenedor o a otro componente en el mismo contenedor?

19. ¿Qué se hace para lograr que un objeto contenga de manera lógica a otro objeto?

Ejercicios

1. [Después de §7.2] Dada una clase `Carro` con estas dos variables de instancia:

```
String marca;
int año;
```

Describa todas las operaciones que ocurren cuando la siguiente sentencia se ejecuta:

```
Carro carroCaiden = new Carro();
```

2. [Después de §7.3] Rastree el programa `Carro` mostrado en las figuras 7.2 y 7.3. Utilice el siguiente modelo de rastreo. Observe que se han utilizado abreviaturas para mantener el rastreo tan pequeño como sea posible.

ControladorCarro			Carro																		
	main			setMarca		setAño		setColor		hacerCopia		desp	obj1			obj1					
línea#	**carroJ**	**carroS**	***línea#***	**this**	**marca**	**this**	**año**	**this**	**color**	**this**	**carro**	**this**	**marca**	**año**	**color**	**marca**	**año**	**color**	**salida**		

3. [Después de §7.3] ¿Qué es la recolección de basura?

4. [Después de §7.5] Suponga que una clase `Computadora` contiene entre otras variables de instancia, una variable tipo String llamada `discoDuro`. Complete el siguiente método `intercambiarDiscoDuro` que realiza el intercambio del valor del objeto disco duro llamado con el valor de la unidad de disco duro pasado como parámetro.

```
public void intercambiarDiscoDuro(Computadora otraComp)
{
   <insertar código aquí>
} // fin de intercambiarDiscoDuro
```

5. [Después de §7.5] Normalmente, se le da a cada objeto un nombre único asignando su dirección a sólo una variable de referencia. La asignación del valor a una variable de referencia a otra variable de referencia crea dos nombres diferentes para la misma cosa, lo cual es ambiguo. Identifique una situación donde este tipo de asignación es útil, a pesar de la ambigüedad.

6. [Después de §7.6] Dado este programa especificación de automóvil:

```
1  /**************************************************************
2  * ControladorOpcionesAuto.java
3  * Dean & Dean
4  *
5  * Ésta para ejercicios de la clase AutoOptions.
6  **************************************************************/
7
8  import java.util.Scanner;
9
10 public class ControladorOpcionesAuto
11 {
12   public static void main(String[] args)
13   {
14     Scanner stdIn = new Scanner(System.in);
15     String serial;
16     OpcionesAuto auto = new OpcionesAuto();
17
18     System.out.print("Introduzca el numero de serie: ");
19     serial = stdIn.nextLine();
20     auto.specifyEngine(auto.setSerial(serial).
21       specifyFrame().specifyBody().isTight());
22     auto.especificarTransmision();
23     auto.imprimirOpciones();
24   } // fin de main
25 } // fin de la clase ControladorOpcionesAuto
```

```
1  /**************************************************************
2  * OpcionesAuto.java
3  * Dean & Dean
4  *
5  * Esta clase para opciones "custom" automóviles
6  **************************************************************/
7
8  import java.util.Scanner;
9
10 public class OpcionesAuto
11 {
12   private String serial;          // numero de serie de auto
13   private char bastidor = 'x';    // tipo de bastidor: A,B
14   private String carroceria = ""; // estilo de carroceria:
                                       2-puertas,4-puertas
15   private int hp = 0;             // caballos de fuerza del motor: 85,
                                       115, 165
16
17   // transmission: false = manual, true = automatica
18   private boolean automatico = false;
19
20   //*************************************************************
21
22   public OpcionesAuto setSerial(String serial)
23   {
24     this.serial = serial;
25     return this;
26   } // fin de set Serial
27
28   //*************************************************************
29
30   public AutoOptions especificarBastidor()
31   {
```

```
32      Scanner stdIn = new Scanner(System.in);
33
34      while (this.bastidor != 'A' && this.bastidor != 'B')
35      {
36        System.out.print("Introduzca bastidor (A o B): ");
37        this.bastidor = stdIn.nextLine().charAt(0);
38      } // fin del while
39      return this;
40    } // fin de especificarBastidor
41
42    //*************************************************************
43
44    public OpcionesAuto especificarCarroceria()
45    {
46      Scanner stdIn = new Scanner(System.in);
47
48      while (!this.carroceria.equals("2-puertas")
49        && !this.carroceria.equals("4-puertas"))
50      {
51        System.out.print(
52          "Introduzca (2-puertas o 4-puertas): ");
53        this.carroceria = stdIn.nextLine();
54      } // fin del while
55      return this;
56    } // fin de especificarCarroceria
57
58    //*******************************************************
59
60    public boolean esAjustado()
61    {
62      boolean ajustado = false;
63
64      if (this.bastidor == 'A' && this.carroceria.equals("4-puertas"))
65      {
66        ajustado = true;
67      }
68      return ajustado;
69    } // fin de esAjustado
70
71    //*********************************************************
72
73    public void especificarMotor(boolean ajustado)
74    {
75      Scanner stdIn = new Scanner(System.in);
76
77      if (ajustado)
78      {
79        while (this.cf != 85 && this.cf != 115)
80        {
81          System.out.print("Introduzca CF (85 o 115): ");
82          this.cf = stdIn.nextInt();
83        } // fin del while
84      }
85      else
86      {
87        while (this.cf != 85 && this.cf != 115 && this.cf != 165)
88        {
89          System.out.print("Introduzca CF (85, 115, 165): ");
90          this.cf = stdIn.nextInt();
```

```
91          } // fin del while
92        } // fin del else de ajustado
93        stdIn.nextLine(); // flush \r\n after nextInt
94      } // fin de especificarMotor
95
96      //*******************************************************
97
98      public void especificarTransmision()
99      {
100       Scanner stdIn = new Scanner(System.in);
101
102       System.out.print("Automatico (s/n?): ");
103       if (stdIn.nextLine().charAt(0) == 'y')
104       {
105         this.automatico = true;
106       }
107     } // fin de especificarTransmision
108
109     //*******************************************************
110
111     public void imprimirOpciones()
112     {
113       System.out.printf("# serial %s\n%s bastidor\n%s\n%-3d CF\n",
114         this.serial, this.bastidor, this.carroceria, this.cf);
115       if (automatico)
116       {
117         System.out.println(" automatico");
118       }
119       else
120       {
121         System.out.println("manual 4 velocidades");
122       }
123     } // fin de imprimirOpciones
124   } // fin clase OpcionesAuto
```

Utilice el siguiente modelo de rastreo del programa OpcionesAuto. Observe que se utilizan abreviaturas para mantener el ancho del modelo del rastreo tan pequeño como sea posible.

entrada

X142R
A
4-puertas
165
115
Y

ControladorOpcionesAuto			OpcionesAuto																	
	main			setMarca		set Marca	spec Body	isTight		specEngine		spec Trans	hacer Copia	obj1						
línea#	**ser**	**auto**	***línea#***	**this**	**ser**	**this**	**this**	**this**	**tight**	**this**	**tight**	**this**	**this**	**ser**	**frm**	**body**	**hp**	**auto**	**salida**

7. [Después de §7.6] En los siguientes esqueletos de las clases `Tiempo` y `ControladorTiempo`, reemplace las líneas en cursivas <*insertar...*> con su propio código, de tal manera que el programa opere correctamente. De manera más específica:

a) En la clase `ControladorTiempo` proporcione una definición de método para `setHoras`, de tal manera que ese método pueda ser llamado como parte del encadenamiento de llamadas a métodos.

b) En la clase `ControladorTiempo` proporcione una sencilla sentencia que encadene las llamadas a los métodos `setHoras`, `setMinutos`, `setSegundos` e `imprimeHora`. Utilice valores razonables para los argumentos en las llamadas a los métodos. Si se pasa 8 a `setHoras`, 59 a `setMinutos` y 0 a `setSegundos`, entonces la sentencia de encadenamiento de llamadas a métodos debe imprimir lo siguiente:

```
   08:59:00

public class Tiempo
{
  private int horas;
  private int minutos;
  private int segundos;

  //*********************************************************

  <insertar la definición del método setHoras aquí>

  public Tiempo setMinutos(int minutos)
  {
    this.minutos = minutos;
    return this;
  } // fin setMinutos

  public Tiempo setSegundos(int segundos)
  {
    this.segundos = segundos;
    return this;
  } // fin setSegundos

  //*********************************************************

  public void imprimeHora()
  {
    System.out.printf("%02d:%02d:%02d\n", horas, minutos, segundos);
  } // fin imprimeHora
} // fin de clase Tiempo

public class ControladorTiempo
{
  public static void main(String[] args)
  {
    Tiempo tiempo = new Tiempo();
    <insertar sentencia de encadenamiento de llamada a métodos aquí>
  }
} // fin de clase ControladorTiempo
```

8. [Después de §7.7]

a) Modifique el método de dos parámetros `setAltura` de la figura 7.10 para hacerlo probar su parámetro `unidades` y verificar si `unidades` es igual a alguno de los siguientes símbolos: “m”, “cm”, “mm”, “in” o “ft”. Si es igual a uno de éstos, establezca las variables de instancia y devuelva `true`. Si no es igual a alguno de éstos, devuelva `false`.

b) ¿*Requiere* esta modificación algún cambio a algún programa que llame al método de dos parámetros `setAltura`? ¿Por qué sí o por qué no?

c) Escriba una sentencia que llame al método modificado y utilice la información devuelta para imprimir un mensaje de error “Error: unidades no reconocidas” si el argumento de las unidades no es alguno de los valores permitidos.

9. [Después de §7.8] Proporcione un constructor estándar de tres parámetros para una clase llamada `articuloJoyeria`. La clase contiene tres variables de instancia: `descripción`, `precio` y `cantActual`. El constructor simplemente asigna sus tres parámetros a las tres variables de instancia.

10. [Después de §7.9] Constructores sobrecargados:

a) Agregue un par de constructores a la clase `Altura` que implemente las inicializaciones proporcionados por las dos operaciones `setAltura` de la figura 7.11. Minimice el número total de sentencias haciendo que el constructor de un parámetro llame al método `setAltura` de un parámetro y haciendo que el constructor de dos parámetros llame al método `setAltura` de dos parámetros.

b) Proporcione un método `main` replanteado para la clase `ControladorAltura` de tal manera que ese método `main` utilice uno de los dos constructores de la parte *a*) para generar esta salida:

```
6.0 ft
```

11. [Después de §7.9] Constructores sobrecargados:
Asuma que la clase `Altura` de la figura 7.10 contiene sólo un método `setAltura`: la versión de dos parámetros. Escriba dos constructores para esta clase `Altura`, uno con un argumento (`double altura`), y otra con dos argumentos (`double altura` y `String unidades`). Para el constructor de un argumento, utilice por omisión "m" para las `unidades`.

No duplique ningún código interno. Esto es, haga que el constructor de un parámetro transfiera el control al constructor de dos parámetros, y haga que el constructor de dos parámetros transfiera el control al método de dos parámetros `setAltura`.

12. [Después de §7.9] Asuma que las siguientes dos clases son compiladas y se corren, ¿cuál sería la salida?

```
public class ControladorAutoSilly
{
  public static void main(String[] args)
  {
    ClaseSilly sc = new ClaseSilly();
    sc.desplegar();
  }
} // fin de clase ControladorAutoSilly

public class ClaseSilly
{
  private int x = 10;

  public ClaseSilly()
  {
    this(20);
    System.out.println(this.x);
  }

  public ClaseSilly(int x)
  {
    System.out.println(this.x);
    System.out.println(x);
    this.x = 30;
    x = 40;
  }

  public void desplegar()
  {
    int x = 50;
    desplegar(x);
    System.out.println(x);
  }

  public void desplegar(int x)
  {
    x += 10;
    System.out.println(x);
  }
} // fin de clase ClaseSilly
```

Solución a las preguntas de revisión

1. Falso. Sólo se asigna memoria para variables de referencia.
2. El operador `new` asigna memoria a un objeto, y devuelve la dirección donde ese objeto es almacenado en memoria.
3. Falso. La asignación de una variable de referencia a otra variable de referencia provoca que la dirección de la variable del lado derecho sea asignada a la variable del lado izquierdo, y eso provoca que ambas variables de referencia se refieran al mismo objeto.
4. Una fuga de memoria es cuando a un objeto inaccesible se le permite persistir y utilizar el espacio en la memoria de la computadora.

5. El valor final de `misma` es `true`.
6. El tipo devuelto de un método `equals` es `boolean`.
7. El operador == compara los valores de dos variables del mismo tipo. Si las variables son de referencia, == compara su dirección para verificar si se refieren al mismo objeto. Un método `equals` de manera típica compara los valores de todas las variables de instancia en el objeto referido por su parámetro con los valores de las variables de instancia en el objeto que lo llamó. El método `equals` devuelve verdadero sólo si todas las variables de instancia correspondientes tienen los mismos valores.
8. Cierto. La referencia da el acceso al método a la referencia del objeto.
9. Para que un método sea llamado como parte de una sentencia de encadenamiento de llamada a métodos, se deben incluir las siguientes cosas:
   - Dentro del cuerpo del método, especificar `return` <*variable de referencia*>;
   - Dentro del encabezado del método, especificar la clase asociada de la variable de referencia como el tipo devuelto.
10. Si se tienen dos o más métodos con el mismo nombre en la misma clase, se denominan métodos sobrecargados.
11. Cierto.
12. Un constructor no devuelve nada y no utiliza la sentencia `return`, pero cuando se llama a un constructor, `new` devuelve una referencia al objeto construido.
13. Cierto.
14. Falso. Las convenciones estándar de codificación sugieren que se pongan las definiciones de los constructores antes de todas las definiciones de los métodos.
15. Falso. Hay un solo constructor porque si una clase contiene un constructor definido por el programador, entonces el compilador no proporciona un constructor por omisión.
16. Utilice esta sintaxis:

    ```
    this(<argumentos-para-un-constructor-meta>);
    ```

17. Cierto.
18. Para el componente en cuestión, se declara una variable de referencia para el contendor u otro objeto que se quiere conocer. Después, cuando se instancia el componente en cuestión, se pasa a su constructor una referencia al contenedor u otro componente, y se hace que este constructor inicialice la variable de referencia de la instancia correspondiente.
19. En la definición de la clase contenedora, se declara una instancia de una variable de referencia para cada componente prospecto. En el constructor del contenedor, se instancia cada componente y se le asigna una referencia a la instancia de la variable de referencia correspondiente.

CAPÍTULO 8

Ingeniería de software

Objetivos

- Desarrollar un buen estilo de codificación.
- Aprender a simplificar algoritmos complicados mediante el encapsulamiento de tareas subordinadas.
- Distinguir el uso de variables de instancia y variables locales.
- Aprender cuándo y cómo usar una estrategia de diseño arriba-abajo.
- Aprender cuándo y cómo usar una estrategia de diseño ascendente.
- Decidir el uso de software escrito previamente siempre que sea factible.
- Reconocer la importancia de realizar prototipos.
- Desarrollar el hábito de realizar pruebas a menudo y en forma exhaustiva.
- Evitar el uso innecesario del prefijo `this`.

Relación de temas

8.1 Introducción

En los capítulos 6 y 7 se abordó principalmente la "ciencia" de la programación Java: cómo declarar objetos, definir clases, definir métodos, etc. En este capítulo se considera más la "práctica" de la programación Java: cómo diseñar y desarrollar un programa, y cómo hacerlo fácil de leer. La práctica de la programación se resume en forma elegante en la expresión *ingeniería de software*, donde por ingeniería de software se entiende:[1]

[1] Definición tomada de la Norma 610.12 del Institute of Electrical and Electronics Engineers (IEEE).

1. La aplicación de un método sistemático, disciplinado y cuantificable para el desarrollo, operación y mantenimiento de software, es decir, la aplicación de la ingeniería al software.
2. El estudio de métodos como en el punto 1.

El capítulo empieza con un estudio profundo de las convenciones de estilo para codificar, que ayudan a hacer más legibles los programas. Se mostrará cómo dividir una gran tarea en un conjunto de tareas más pequeñas al delegar algo del trabajo de un método en otros métodos. Se analiza el encapsulamiento, una de las piedras angulares del diseño POO idóneo. Luego, se describen estrategias alternativas de diseño: arriba-abajo, ascendente y basada en casos. A medida que se trabaja con algo, la comprensión del tema mejora, y se sugiere planificar el rediseño continuo de forma más elaborada en un proceso evolutivo denominado *mejoramiento iterativo.* Se recalca que se obtiene más satisfacción y el producto es mejor si se realizan pruebas exhaustivas y frecuentes durante el proceso. Para facilitar las pruebas modulares, se mostrará la forma en que es posible incluir un método `main` en cada clase. Hasta el momento se ha utilizado bastante el `this` para recalcar que cada ejecución de un método de instancia está ligada en forma única con un objeto particular, aunque cerca del fin del capítulo se mostrará cómo racionalizar el código al omitir el `this` cuando no hay ambigüedad. En una sección opcional final se mostrará la forma en que es posible utilizar gráficas simples para elaborar una herramienta organizacional práctica denominada tarjetas CRC.

8.2 Convenciones de estilo para codificar

A continuación se presentan algunas directrices sobre el estilo para codificar. Ya antes se han mencionado e ilustrado muchas de estas directrices, de modo que en gran parte el contenido de esta sección es un repaso. Más adelante, en la medida en que avancemos en la descripción de Java, se proporcionarán más directrices. Para consultar una lista completa de todas las directrices sobre el estilo para codificar proporcionadas en este libro, es necesario leer el apéndice 5: "Convenciones Java de estilo para codificar". Principalmente, estas convenciones son un subconjunto simplificado de las convenciones sobre el estilo presentadas en el sitio en la red Sun's Java Conventions Web.[2] Si el lector tiene alguna pregunta de estilo no tratada en el apéndice 5, consulte el sitio de Sun mencionado.

Se sabe que hay algunas cuestiones de estilo en las que existe un desacuerdo legítimo respecto a la mejor forma de hacer las cosas. Hay muchas normas distintas. Sun intenta escoger las mejores convenciones de entre todas las de uso común. Aquí se intenta hacer lo mismo. Si usted lee este libro como parte de un curso y su profesor está en desacuerdo con las convenciones de estilo del libro o con las convenciones de estilo de Sun, por favor siga las directrices de su profesor. Una cosa en particular que su profesor pudiera requerir es un formato especial para la clase y documentación del método. Muchos programadores profesionales de Java usan la herramienta `javadoc` de Java como ayuda para la clase y documentación del método. La herramienta `javadoc` obtiene documentación especialmente comentada del código fuente y la presenta en un reporte claramente organizado. Sun usa la herramienta `javadoc` para producir su documentación de la biblioteca API. Para conocer más detalles, consulte el apéndice 6.

Las convenciones sobre el estilo para codificar se ilustrarán pidiendo al estudiante que consulte el programa Student en la figura 8.1 y en las figuras 8.2*a* y 8.2*b*. Este programa es una versión modificada del programa Student en la parte posterior del apéndice "Convenciones Java de estilo para codificar".

Sección del prólogo

Observe el texto delimitado en las partes superiores de las figuras 8.1 y 8.2*a*. Se denomina *prólogo.* Incluya una sección del prólogo en la parte superior de cada archivo. El prólogo contiene los siguientes elementos en orden:

- línea de asteriscos
- nombre de archivo
- nombre(s) del programador
- línea en blanco con un asterisco

[2] http://java.sun.com/docs/codeconv

- descripción
- línea de asteriscos
- línea en blanco

El prólogo se encierra en un comentario `/*. . . */` y para que se vea como una caja, se inserta un asterisco y un espacio frente al nombre del archivo, el nombre del programador, la línea en blanco y las líneas de descripción.

Constantes con nombre y variables de instancia

Haga una línea en blanco, una línea de asteriscos y otra línea en blanco después del bloque de sentencias que declara y/o inicializa todas las constantes con nombre y las variables de instancia.

Descripciones del método

Observe las descripciones arriba de uno de los constructores en la figura 8.2*a* y los métodos en la figura 8.2*b*. Arriba de cada método, los elementos se escriben en el orden siguiente:

- línea en blanco
- línea de asteriscos
- línea en blanco
- descripción
- línea en blanco

Para métodos evidentes cortos es correcto omitir la descripción del método. Entre constructores cortos y entre métodos cortos de acceso y reguladores (*mutator*), también es correcto omitir la línea de asteriscos.

Líneas en blanco

En general, las líneas en blanco se usan para separar trozos lógicos del código. En la clase `StudentDriver` en la figura 8.1, observe las líneas en blanco:

- entre la sección del prólogo y la definición de la clase.
- justo después de las declaraciones de las variables locales de un método.

```
/*******************************************************
* StudentDriver.java
* Dean & Dean
*
* Esta clase actúa como controlador de la clase Estudiante.
*******************************************************/

public class StudentDriver
{
  public static void main(String[] args)
  {
    Estudiante s1;  // primer estudiante
    Estudiante s2;  // segundo estudiante

    s1 = new Student();
    s1.setFirst("Adeeb");
    s1.setLast("Jarrah");
    s2 = new Student("Heejoo", "Chun");
    s2.printFullName();
  } // end main
} // end class StudentDriver
```

Figura 8.1 Clase `StudentDriver`.

```
/*********************************************************************
* Student.java
* Dean & Dean
*
* Esta clase se encarga del procesamiento del nombre de un estudiante.
*********************************************************************/

import java.util.Scanner;

public class Student
{
  private String first = ""; // nombre del estudiante
  private String last = "";  // apellido del estudiante

  //*******************************************************************

  public Student()
  { }

  // Este constructor verifica que cada nombre empiece
  // con una mayúscula y siga con minúsculas.

  public Student(String first, String last)
  {
    setFirst(first);
    setLast(last);
  }

  //*******************************************************************
```

Figura 8.2*a* Clase `Student`, parte A.

Aunque no se muestra en el programa Student, para métodos largos resulta conveniente insertar líneas en blanco entre trozos de código separados lógicamente dentro del método. Asimismo, cuando una línea de comentario aparece en el cuerpo del código, conviene contar con espacio en blanco arriba del comentario, para que éste sea más visible.

Nombres con sentido

Use nombres con sentido para sus clases y variables. Por ejemplo, `Student` es un nombre idóneo para la clase en las figuras 8.2*a* y 8.2*b* porque la clase modela un estudiante. En forma semejante, `setName` podría ser un buen nombre para un método regulador (*mutator*) que establece las variables de instancia `first` y `last` al nombre y apellido de un estudiante, y `getLast` sería un buen nombre para un método de acceso que regrese el apellido.

Paréntesis de llave y sangrado

Como se muestra en la figura 8.1 y en las figuras 8.2*a* y 8.2*b*, se abre un paréntesis de llave ({) inmediatamente abajo de la primera letra en la línea precedente. Se sangra todo lo que está lógicamente dentro de los paréntesis de llave. Una vez que se acaba con un bloque (es decir, cuando se está listo para cerrar los paréntesis de llave) se cuelga el párrafo, de modo que las llaves de apertura y de cierre de un bloque particular estén alineadas. Al seguir este esquema de sangrado y colgado, los paréntesis de llave correspondientes siempre se alinean en la misma columna. Por ejemplo, observe cómo ambas llaves de apertura y cierre de la clase `Student` están en la misma columna.

Nuestra recomendación sobre dónde colocar el paréntesis de llave de apertura ({) difiere de la recomendación de Sun, en que esta llave debe colocarse al final de la línea previa, como esto:

```
  // Este método verifica que el nombre empiece con una mayúscula
  // y a partir de ahí contenga minúsculas.

  public void setFirst(String first)
  {
    // [A-Z][a-z]* es una expresión regular. Consulte la clase API Pattern.
    si (first.coincide("[A-Z][a-z]*"))
    {
      this.first = first;
    }
    else
    {
      System.out.println(first + " es un nombre inválido.\n" +
        "Los nombres deben empezar con una mayúscula y tener" +
        " minúsculas a partir de ahí.");
    }
  } // end setFirst

  //*****************************************************************

  // Este método verifica que el apellido empiece con una mayúscula
  // y a partir de ahí tenga minúsculas.

  public void setLast(String last)
  {
    // [A-Z][a-z]* es una expresión regular. Consulte la clase API Pattern.
    si (last.coincide("[A-Z][a-z]*"))
    {
      this.last = last;
    }
    else
    {
      System.out.println(last + " es un nombre inválido.\n" +
        "Los nombres deben empezar con una mayúscula y tener" +
        " minúsculas a partir de ahí.");
    }
  } // end setLast

  //*****************************************************************

  // Imprimir el nombre y el apellido del estudiante.

  public void printFullName()
  {
    System.out.println(this.first + " " + this.last);
  } // end printFullName
} // end class Student
```

Figura 8.2*b* Clase `Student`, parte B.

```
public void setName(String first, String last) {
  this.first = first;
  this.last = last;
}
```

Ésta es una de las pocas ocasiones en que nuestra recomendación difiere de la recomendación de Sun. Muchos programadores siguen nuestra recomendación, ya que proporciona mejor visualización de los paréntе-

sis de llave del bloque del código definido por los paréntesis de llave. Sin embargo, colocar la llave de apertura al final de la línea previa hace un poco más denso el bloque y si usted, su profesor o su jefe desean que esta llave esté al final de la línea previa, les deseamos lo mejor si quieren seguir esta convención.

Debe ser consistente en el sangrado. Cualquier ancho de sangrado entre dos y cinco es aceptable siempre y cuando haya consistencia en todo el programa. En este libro se usan dos espacios porque el ancho de las páginas del libro es menor que el de las pantallas de las computadoras, y no es recomendable quedarse sin espacio para programas con bastantes anidamientos.

Muchos programadores novatos utilizan el sangrado en forma errónea. No lo hacen cuando deben hacerlo o lo hacen cuando no deben, o usan anchos inconsistentes para los sangrados. Esto origina programas poco profesionales y difíciles de leer. Algunos programadores novatos posponen la introducción de los sangrados hasta el final, después de haber terminado la depuración. ¡Gran error! Es necesario usar un sangrado idóneo a medida que se elabora el programa. Esto es bastante sencillo pues en realidad sólo hay dos reglas que recordar:

1. Usar paréntesis de llave para rodear un bloque de código que esté lógicamente dentro de otra cosa.
2. Sangrar el código que está dentro de los paréntesis de llave.

Hay una excepción para la primera regla:

> El código a continuación de una cláusula `case` de una sentencia `switch` se considera como lógicamente dentro de la cláusula `case`, pero no se usan llaves.

Declaraciones de variables

Como se muestra en el método `main` de la figura 8.1, coloque todas las declaraciones de variables locales en la parte superior del método (aun cuando el compilador no lo requiera). Excepción: a menos que requiera que una variable de iteración del ciclo `for` persista más allá del final del ciclo `for`, declárela en el campo de inicialización del encabezado del ciclo `for`.

Normalmente, especifique sólo una declaración de variables por línea. Excepción: si varias variables con significados evidentes están relacionadas, es correcto agruparlas en una línea.

Incluya un comentario para todas las variables cuyo significado no es evidente. Por ejemplo, las declaraciones de variables locales crípticas en el método `main` en la figura 8.1 definitivamente requieren comentarios, y también se proporcionan comentarios para las declaraciones de las variables de instancia en la figura 8.2*a*. Observe la manera en que están alineados estos comentarios: sus `//` están en la misma columna. En general, si usted tiene comentarios que aparecen en el lado derecho de varias líneas próximas entre sí, intente alinearlos.

Ajuste de línea o línea de recapitulación

Si hay una sentencia demasiado larga que no cabe en una línea, sepárela en uno o más puntos de ruptura naturales en la sentencia. Por ejemplo, observe dónde se separó la sentencia larga de impresión en los métodos `setFirst` y `setLast` en la figura 8.2*b*. Los siguientes puntos se consideran puntos de ruptura naturales:

- justo después de abrir paréntesis
- después de un operador de concatenación
- después de una coma que separa parámetros
- en los espacios en blanco en las expresiones

Después de un punto de ruptura en una sentencia larga, la parte restante de la sentencia debe sangrarse en la línea siguiente. En la figura 8.2*b* observe la forma en que se sangró la continuación de las líneas con el mismo ancho estándar de dos espacios que se usa para todos los demás sangrados.

En lugar de simplemente sangrar la continuación de las líneas con el ancho de sangrado estándar, algunos programadores prefieren alinear estas continuaciones de las líneas con una entidad paralela en la línea previa. Por ejemplo, en la sentencia de impresión mencionada, alinearían la continuación de la línea con `first` como sigue:

```
System.out.println(first +
      " es un nombre inválido.\n" +
      " Los nombres deben empezar con una" +
      " mayúscula y tener minúsculas" +
      " a partir de ahí.");
```

En nuestra opinión, el código anterior está demasiado lejos a la derecha y se corta de manera innecesaria. Ésta es la razón de que hayamos preferido mantenerlo simple y sólo sangrar con el ancho normal de sangrado.

Paréntesis de llave que abarcan una sentencia

Para una sentencia de ciclo o una sentencia `if` que incluye sólo un subordinado, es legal omitir los paréntesis de llave que abarcan la sentencia. Por ejemplo, en el método `setFirst` de la figura 8.2*b*, la sentencia if-else podría escribirse como sigue:

```
if (first.matches("[A-Z][a-z]*"))
  this.first = first;
else
  System.out.println(first + " es un nombre inválido.\n" +
    "Los nombres deben empezar con una mayúscula y tener minúsculas" +
    " a partir de ahí.");
```

No obstante, nos gusta usar paréntesis de llave para todas las sentencias de ciclo y las sentencias `if`, inclusive si sólo hay una sentencia encerrada entre llaves. ¿Por qué?

- Los paréntesis de llave son un recurso visual para recordar que hay que sangrar.
- Los paréntesis de llave impiden cometer errores lógicos en caso de que después se agregue código que supuestamente ya debe estar dentro de la sentencia del ciclo o dentro de la sentencia `if`.

El segundo punto puede comprenderse mediante un ejemplo. Suponga que un programa contiene el siguiente código:

```
if (person1.isFriendly())
  System.out.println("Hi there!");
```

Suponga que un programador desea agregar una segunda sentencia de impresión (“¿Cómo estás?”) para un objeto amistoso `person1`. Un programador descuidado haría algo como:

```
if (person1.isFriendly())
  System.out.println("Hi there!");
  System.out.println("How are you?");
```

Puesto que la segunda sentencia de impresión no está entre llaves, se ejecuta sin tomar en cuenta si `person1` es amistoso. Aún así, se insiste en preguntar a alguien no amistoso: “¿Cómo estás?” Como respuesta podría obtenerse un ceño fruncido.

Por otra parte, si el programa hubiese seguido nuestras directrices de estilo, el código original se vería como sigue:

```
if (person1.isFriendly())
{
  System.out.println("Hi there!");
}
```

Entonces, si el programador desea agregar una segunda sentencia de impresión (“¿Cómo estás?”) para un objeto amistoso `person1`, sería más difícil cometer un error. Inclusive un programador descuidado tal vez codificaría correctamente la segunda sentencia de impresión como sigue:

```
if (person1.isFriendly())
{
  System.out.println("Hi there!");
  System.out.println("How are you?");
}
```

En el análisis anterior se afirmó "nos gusta usar paréntesis de llave para todas las sentencias del ciclo y las sentencias `if`". De manera más formal, lo anterior se plantea como: nos gusta usar un *bloque* para todas las sentencias de ciclo y las sentencias `if`. Un bloque es un conjunto de sentencias escritas entre paréntesis de llave.

Comentarios

Como se muestra en la figura 8.1 y en las figuras 8.2*a* y 8.2*b*, para todos los bloques, salvo para los más pequeños, se incluye un comentario después del paréntesis de llave de cierre para especificar que el bloque se está cerrando. Por ejemplo, en la figura 8.2*b* observe esta llave de cierre de línea para el método `setFirst`:

```
} // end setFirst
```

¿Por qué está bien hacer esto? Porque alguien que lea el programa puede identificar rápidamente el bloque que está terminando sin tener que desplazarse hasta la parte superior del bloque para saber qué ocurre. Es válido omitir el cierre de los comentarios entre llaves para bloques cortos de menos de aproximadamente cinco líneas. Para bloques cortos, resulta fácil decir a qué bloque está asociada la llave de cierre, de modo que el comentario final simplemente agrega más desorden.

Es necesario incluir comentarios para segmentos del código que pudieran no resultar evidentes para un programador típico de Java. En la figura 8.2*b* observe este comentario que aparece en las partes superiores de los cuerpos de los métodos `setFirst` y `setLast`:

```
// [A-Z][a-z]* is a regular expression. See API Pattern class.
```

Dirige al lector a más información.

Este comentario es útil porque la siguiente sentencia es más oscura que todo. El comentario debe explicar directamente o ayudar al programador a encontrar más información sobre el tema, o ambas cosas. Un comentario como este que refiere a una fuente autorizada es especialmente importante siempre que el código implementa algo misterioso: una definición arbitraria como la "expresión regular" anterior, una fórmula con coeficientes empíricos o una expresión matemática misteriosa.

Siempre que un comentario es tan largo que no cabe en la parte derecha de la línea que está explicándose, es necesario colocarlo una o más líneas arriba. El signo `//` debe sangrarse igual que la línea descrita. Si un comentario se coloca sobre una línea arriba del comentario, asegúrese de que arriba del comentario hay suficiente espacio. En los métodos `setFirst` y `setLast` de la figura 8.2*b*, arriba de los comentarios hay suficiente espacio en blanco porque ocurre que las líneas anteriores son llaves de apertura para los cuerpos de sus métodos respectivos. En otros casos, es necesario insertar toda una línea en blanco arriba del comentario. Es opcional insertar una línea en blanco abajo del comentario.

No deben insertarse comentarios individuales que simplemente replantean lo ya indicado por el código. Por ejemplo, para la primera sentencia de asignación en el método `main` de la figura 8.1, este comentario tiene una cobertura exagerada:

```
s1 = new Student();  // instantiate a Student object
```

Desarrollar programas legibles es una habilidad importante y una forma de arte. Tener demasiados comentarios no es aconsejable porque lleva a programas difíciles de comprender. Pero tener demasiados comentarios también es desaconsejable porque ocasionan programas confusos, difíciles de leer. Hay una acción de equilibrio semejante para las líneas en blanco. Tener demasiadas líneas en blanco es inconveniente porque así se llega a programas difíciles de comprender. Pero tener demasiadas líneas en blanco también es inconveniente porque así se llega a programas con demasiado espacio muerto.

Espacios en blanco

Como se muestra en la figura 8.1 y en las figuras 8.2*a* y 8.2*b*, incluya espacios en blanco:

- después de los asteriscos simples en el prólogo
- antes y después de todos los operadores (excepto en los operadores dentro del encabezado de un ciclo `for`)
- entre un paréntesis de llave de cierre y las `//` para su comentario asociado

- después del signo `//` para todos los comentarios
- después de las palabras clave `if`, `while` y `switch`

Por otra parte, no incluya espacios en blanco:

- entre la llamada a un método y su paréntesis de apertura
- dentro de cada una de las tres componentes en el encabezado de un ciclo `for`

Esta última cuestión puede comprenderse mediante un ejemplo. A continuación se presenta un encabezado de ciclo `for` escrito en forma elegante:

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

Observe que alrededor de los operadores = y < no hay espacios. ¿Por qué es bueno hacer esto? Porque el encabezado del ciclo `for` es intrínsecamente complicado. Para moderar esa complejidad, se agregan recursos visuales a fin de compartir el encabezado del ciclo `for`. En forma más específica, cada sección se consolida (no hay espacios dentro de cada sección) y se inserta un espacio después de cada punto y coma para mantener separadas las tres secciones.

Agrupación de constructores, reguladores (*mutators*) y de acceso

Omita las descripciones de métodos cortos y evidentes. Por ejemplo, los reguladores (*mutators*) y de acceso son cortos y evidentes, por lo que es necesario omitir sus descripciones. Algunas veces, los constructores son cortos y evidentes, pero no siempre. Si un constructor simplemente asigna valores paramétricos a variables de instancia asociadas, entonces es corto, por lo que debe omitirse su descripción. Si, por otra parte, un constructor realiza validación de entrada no evidente sobre valores introducidos por el usuario antes de asignarlos a variables de instancia asociadas, entonces es necesario incluir una descripción para el constructor.

En aras de agrupar cosas semejantes, se recomienda omitir la línea de asteriscos entre reguladores (*mutators*) y accesores, así como entre constructores evidentes. En el supuesto de que una clase contiene dos constructores cortos evidentes, varios métodos reguladores (*mutators*) y de acceso, así como otros dos métodos cortos y evidentes, a continuación se presenta el marco de referencia para una clase así:

```
<class-heading>
{
    <instance-variable-declarations>
    //*********************************************************
    <constructor-definition>
    <constructor-definition>
    //*********************************************************
    <mutator-definition>
    <mutator-definition>
    <accessor-definition>
    <accessor-definition>
    //*********************************************************
    <method-definition>
    //*********************************************************
    <method-definition>
}
```

En este caso, no hay descripciones para los constructores, los accesores o los reguladores (*mutators*). Arriba del primer regulador (*mutator*) hay una línea de asteriscos, pero no arriba de los reguladores (*mutators*) o accesores ulteriores. Estas omisiones hacen que el programa sea más legible al agrupar términos semejantes.

8.3 Métodos de ayuda

En los cuatro primeros capítulos, esencialmente se resolvieron todos los problemas abordados en apenas un módulo: el método `main` en una clase. Sin embargo, a medida que los problemas crecen, es más necesario separarlos en subproblemas, cada uno de tamaño manejable. Esto comenzó a hacerse en el capítulo 5, cuando el método `main` solicitó ayuda al llamar algunos de los métodos Java API. Luego, en los capítulos 6 y 7, el programa se separó en dos clases: una controladora, que contenía el método `main`, y una clase controlada, que contenía todos los demás métodos. Al final del capítulo 7 se presentó la idea de varias clases controladas, dos o más clases, cada una de las cuales contenía otros métodos. Luego, parte de la partición provino de la separación del programa en dos o más clases, y parte de la partición provino al definir múltiples métodos en cada clase. Lo anterior permitió que el método `main` en la clase controladora delegase la mayor parte de su trabajo en métodos de otras clases.

En un sentido amplio, podría decirse que todos los demás métodos llamados por el código en el método `main` son "métodos de ayuda": ayudan al método `main` a hacer su trabajo. En otras palabras, en un sentido amplio, cualquier método que es llamado por otro, es un método de ayuda: el método llamado ayuda al método que llama. El método que llama es un *cliente*, y el método llamado (el método de ayuda en sentido amplio) es un *servidor.*

La definición de método de ayuda se puede reducir a: un método llamado que está en la misma clase que el método que llama. En la sección previa, el constructor `Student` en la figura 8.2*a* llama a dos de los métodos en la misma clase: `setFirst` y `setLast`, en la figura 8.2*b.* Presumiblemente, estos reguladores (*mutators*) fueron escritos para permitir que el usuario modifique las variables de instancia en un objeto después que el objeto fue inicializado originalmente. Pero una vez que se escribe su código, ¿por qué no reusarlo? Al incluir llamadas a estos dos métodos comunes en el constructor, se evita la duplicación del código en los métodos llamados. Debido a que cada uno de los métodos reguladores (*mutators*), `setFirst` y `setLast`, incluyen una cantidad importante de código para comprobar errores que ayudan a que el constructor haga su trabajo, esta organización ayuda a dividir el problema en trozos más pequeños.

La definición de método de ayuda puede reducirse aún más. Hasta el momento, todos los métodos cubiertos han utilizado el modificador de acceso `public`. Estos métodos `public` forman parte de la *interfaz* de la clase, ya que son responsables de la comunicación entre los datos de un objeto y el mundo exterior. Algunas veces, quiere crearse un método que no forme parte de la interfaz: que en lugar de ello simplemente soporte la operación de otros métodos dentro de su propia clase. Este tipo especial de método, que está en la misma clase y cuenta con un modificador de acceso `private`, a menudo se denomina *método de ayuda*.

Por ejemplo, suponga que se le solicita escribir un programa que maneje entradas de orden para camisetas de un uniforme de deportes. Para cada orden de camiseta, el programa debe mostrar al usuario un color primario para la camiseta y el color de sus ribetes. Para cada selección de color, el programa debe efectuar la misma validación de entrada. Debe comprobar que el color introducido tiene uno de tres valores: b, r o a, por blanco, rojo o amarillo. Este código de validación de entrada no es trivial. Es responsable de:

- Solicitar al usuario un color de entrada.
- Comprobar si la entrada es válida.
- Repetir la solicitud si la entrada es inválida.
- Convertir la entrada de color de un solo carácter en un valor de color de toda una palabra.

Estas cuatro tareas constituyen un grupo coherente de actividades. En consecuencia, resulta lógico encapsularlas (agruparlas juntas) en un módulo por separado. El hecho de que el constructor `Shirt` debe realizar este grupo coherente de actividades dos veces por separado es una razón adicional para encapsularlas en un módulo por separado.

Así, en lugar de repetir el código completo para estas cuatro tareas en el constructor cada vez que se requiere la selección de color, este código de color debe colocarse en un método de ayuda por separado y luego llamarlo siempre que se necesite la selección de color. Estudie el programa `Shirt` y la sesión de muestra en las figuras 8.3, 8.4*a* y 8.4*b*, especialmente el constructor `public`, `Shirt` y el método de ayuda `private`. Observe cómo el constructor llama dos veces al método `selectColor`. En este caso particular (y en la sección previa la clase `Student`), las llamadas al método de ayuda son de un constructor. Un método de ayuda también puede llamarse desde cualquier método común y corriente en la misma clase.

Hay dos beneficios de usar métodos de ayuda:

Primero, al mover algunos detalles de los métodos `public` a los métodos `private`, se obtienen métodos `public` más racionalizados. Esto conduce a los métodos `public`, cuya funcionalidad fundamental es más evidente. A su vez, lo anterior conduce a que mejore la legibilidad del programa.

Segundo, la utilización de métodos de ayuda puede reducir la redundancia al codificar. ¿Cómo así? Suponga que una tarea particular (como la validación de colores de entrada) debe efectuarse en varios sitios dentro de un programa. Con un método de ayuda, el código de la tarea aparece sólo una vez en el programa, y cada vez que es necesario efectuar la tarea, se llama al método de ayuda. Por otra parte, sin métodos de ayuda, siempre que es necesario efectuar la tarea, el código completo de la tarea debe repetirse cada vez que se efectúa la tarea.

Observe que en la figura 8.4*a*, el método `selectColor` se llama sin ningún prefijo de variable de referencia:

```
this.primary = selectColor("primary");
```

¿Por qué no hay prefijo de punto en la variable de referencia? (Si usted se encuentra en un constructor o en un método de instancia, en ese caso), y desea que el objeto actual llame a otro método que esté en

```
/*****************************************
* ShirtDriver.java
* Dean & Dean
*
* Este es un controlador de la clase Shirt.
*****************************************/

public class ShirtDriver
{
  public static void main(String[] args)
  {
    Shirt shirt = new Shirt();

    System.out.println();
    shirt.display();
  } // end main
} // end ShirtDriver
```

Sesión muestra:

```
Enter person's name: Corneal Conn
Enter shirt's primary color (w, r, y): m
Enter shirt's primary color (w, r, y): r
Enter shirt's trim color (w, r, y): w

Corneal Conn's shirt:
red with white trim
```

Figura 8.3 Clase `ShirtDriver` y asociación muestra.

```
/*******************************************************
* Shirt.java
* Dean & Dean
*
* Esta clase almacena y exhibe opciones de color
* para una camiseta deportiva.
*******************************************************/

import java.util.Scanner;

public class Shirt
{
  private String name;      // person's name
  private String primary;   // shirt's primary color
  private String trim;      // shirt's trim color

  //*****************************************************

  public Shirt()
  {
    Scanner stdIn = new Scanner(System.in);

    System.out.print("Enter person's name: ");
    this.name = stdIn.nextLine();

    this.primary = selectColor("primary");
    this.trim = selectColor("trim");
  } // end constructor

  //*****************************************************

  public void display()
  {
    System.out.println(this.name + "'s shirt:\n" +
    this.primary + " with " + this.trim + " trim");
  } // end display

  //*****************************************************
```

Aquí no es necesaria una variable de referencia prefijo punto.

Figura 8.4*a* Clase `Shirt`, parte A.

la misma clase, el prefijo de punto en la variable de referencia es innecesario. Puesto que el método constructor y el `selectColor` están en la misma clase, no se requiere el prefijo de punto en la variable de referencia.

8.4 Encapsulamiento (con variables de instancia y variables locales)

Se dice que un programa muestra encapsulamiento si sus datos están ocultos; es decir, si es difícil acceder a los datos desde el "mundo exterior". ¿Por qué el encapsulamiento es algo bueno? Puesto que el mundo exterior no es capaz de acceder directamente a los datos encapsulados, para el mundo exterior es más difícil desordenar las cosas.

Directrices para implementar el encapsulamiento

Para implementar el encapsulamiento hay dos técnicas:

```
      El modificador de acceso private se usa para un método de ayuda.
  // El método de ayuda pide e ingresa la selección hecha por el usuario

  private String selectColor(String colorType)
  {
    Scanner stdIn = new Scanner(System.in);
    String color; // chosen color, first a letter, then a word

    do
    {
      System.out.print("Enter shirt's " + colorType +
        " color (w, r, y): ");
      color = stdIn.nextLine();
    } while (!color.equals("w") && !color.equals("r") &&
             !color.equals("y"));

    switch (color.charAt(0))
    {
      case 'w':
        color = "white";
        break;
      case 'r':
        color = "red";
        break;
      case 'y':
        color = "yellow";
    } // end switch

    return color;
  } // end selectColor
} // end class Shirt
```

Figura 8.4*b* Clase `Shirt`, parte B, método de ayuda `selectColor`.

- Primero, un problema se separa en clases por separado, donde cada clase define un conjunto de datos encapsulados que describen el estado actual de un objeto en esa clase. Este objeto de estado de los datos se encapsula usando el modificador de acceso `private` para cada uno de estos datos. Como ya se sabe, los datos de estado actual de un objeto se denominan variables de instancia.
- Segundo, las tareas de una clase se descomponen en métodos por separado, donde cada método contiene un conjunto de datos encapsulados adicionales necesarios para realizar su trabajo. Como ya se sabe, los datos de un método se denominan variables locales.

El hecho de declarar variables de instancia dentro de una clase es una forma de encapsulamiento, y declarar variables locales dentro de un método es otra forma de encapsulamiento. ¿Cuál es la forma más fuerte de encapsulamiento (la más oculta)? Todos los métodos de instancia tienen acceso a todas las variables de instancia definidas en la misma clase. Por otra parte, sólo el método actual tiene acceso a una de sus variables locales. En consecuencia, una variable local está más encapsulada que una variable de instancia. Así, para promover encapsulamiento, deben usarse variables de instancia en lugar de variables locales siempre que sea posible.

Al escribir un método, a menudo se necesitan más datos que los proporcionados por las variables de instancia actuales. Luego, surge la pregunta: ¿cómo deben almacenarse los datos?, ¿en otra variable de instancia?, ¿lógicamente? Intente resistir el apremio de agregar otra variable de instancia. Las variables de instancia deben usarse sólo para almacenar atributos fundamentales de los objetos de la clase, no para almacenar detalles adicionales. Si puede almacenar los datos localmente, proceda a hacerlo. Así

aumenta el objetivo del encapsulamiento. Por lo regular, cuando se piensa en almacenar localmente los datos, se piensa en una variable local declarada dentro del cuerpo del método. Tenga en cuenta que los datos son otra forma de almacenar localmente los datos. Recuerde que un parámetro es declarado en el encabezado de un método, lo cual indica que posee alcance local.

Variables locales *versus* variables de instancia en la clase `Shirt`

A continuación se analizará el papel que tiene la filosofía anterior en la clase `Shirt`. Los atributos fundamentales de una camiseta son su nombre, su color primario y el color de sus ribetes. Ésta es la base de la declaración de las tres variables de instancia declaradas en la figura 8.4*a*:

```
private String name;       // person's name
private String primary;    // shirt's primary color
private String trim;       // shirt's trim color
```

Ahora se considerarán las otras variables necesarias a medida que se escriban métodos clase. Todas estas otras variables están asociadas de alguna manera con el método `selectColor` en la figura 8.4*b*. Es necesario transferir datos en ambas direcciones entre la llamada del constructor `Shirt` y el método llamado `selectColor`.

Primero considere la transferencia de datos hacia el método `selectColor`. Si se requiere un color primario de la camiseta, entonces `selectColor` debe imprimir el mensaje:

```
Enter shirt's primary color (w, r, b):
```

Es necesario transferir datos hacia el método `selectColor` que le indiquen cuál solicitud debe imprimir.

```
Enter shirt's trim color (w, r, b):
```

Estos datos pueden transferirse al declarar otra variable de instancia denominada `colorType`, hacer que el constructor `Shirt` escriba un valor para esta variable de instancia y luego hacer que el método `selectColor` lea el valor de esta variable de instancia. Sin embargo, esto podría ser una mala práctica, puesto que rompería el encapsulamiento dentro del método `selectColor` y agregaría trozos confusos a nuestra elegante y limpia lista de atributos del objeto. La manera idónea de implementar esta comunicación de método a método es como se hizo, con una transferencia argumento/parámetro.

Segundo, considere la transferencia de datos fuera del método `selectColor`. También es necesario transferir datos del método `selectColor` al constructor `Shirt`. Estos datos son la representación en cadena del color seleccionado. A continuación se presentan tres maneras aceptables para transferir datos de regreso al método que llama:

1. Si sólo hay un valor de retorno, es posible regresarlo al módulo que llama como un valor `return`.
2. Si hay más de un valor de retorno, tales valores pueden ensamblarse en un objeto, crear ese objeto en el método de ayuda y regresar una referencia a ese "objeto de comunicación" creado localmente.
3. Es posible pasar en las referencias del método de ayuda a "objetos de comunicación" instanciados en el módulo que llama y usar el código en el método de ayuda para escribir en tales objetos.

Se necesita transferir los datos al método `selectColor` si se quiere imprimir. También es posible transferir datos de regreso al módulo que llama al declarar otras variables de instancia, hacer que el método de ayuda escriba valores en ellas y hacer que el módulo que llama las lea después que el método de ayuda termine su ejecución. Sin embargo, ésta sería una práctica deficiente, puesto que se rompería el encapsulamiento y agregaría trozos confusos a nuestra elegante y limpia lista de atributos del objeto. La manera idónea de implementar esta comunicación de método a método es como se hizo, con una valor `return`. En este caso, el valor `return` es una referencia a un objeto `String`.

La clase `Shirt` tiene otra variable a considerar, la referencia `StdIn` a un objeto de comunicación del teclado. Este objeto particular se usa tanto en el método constructor que llama como en el método de ayuda llamado, y es instanciado dos veces: una vez en cada uno de estos dos módulos. Resulta tentador intentar evitar la doble instanciación al hacer que `StdIn` sea una variable de instancia. Y lo anterior "funciona". Sin embargo, nos oponemos a ello, ya que resulta evidente que `StdIn` no es un atributo fundamental de esta clase de objetos. ¡No es una variable que describe el estado de una camiseta! En una versión posterior del programa, quizá sería aconsejable modificar el método de entrada desde el teclado a alguna otra cosa, como un archivo de datos, que se describe en el capítulo 15. Inclusive, el lector podría

intentar usar un método de entrada para el nombre y otro método de entrada distinto para las otras variables de estado. Así, no es necesario cambiar `StdIn`, y tal vez podría modificarlo en varias formas para métodos distintos. Declararlas como locales también hace locales las modificaciones ulteriores, y es una mejor práctica de diseño.

Un argumento que se usa para no hacer local una variable es "quizás algún día necesitemos un enfoque más amplio". Si el lector tiene un plan específico que realmente requiera el enfoque más amplio que se propone, está bien. Pero si justamente "quizás algún día" no proporciona un enfoque más amplio sino hasta que "algún día" llega realmente. Entonces, en ese momento, es necesario modificar el programa a fin de incrementar su enfoque sólo cuando sea verdaderamente necesario.

8.5 Filosofía de diseño

En las siguientes secciones se analizan estrategias alternativas para resolver problemas. Se trata de estrategias, en plural porque no hay una simple estrategia que pueda aplicarse para resolver todos los problemas. Si sólo hubiera una estrategia universal, la programación sería fácil y cualquiera podría hacerla. Pero no es fácil. Ésta es la razón de que los buenos programadores sean solicitados y ganen salarios aceptables.

Enfoque simplista al diseño

A continuación se proporciona una receta simple para diseñar cosas:

1. Imagine lo que desea hacer.
2. Imagine cómo hacerlo.
3. Hágalo.
4. Pruébelo.

A primera vista, lo anterior parece algo de sentido común. Pero realmente sólo funciona para problemas bastante simples: problemas en los que todo es fácil y no se requiere ninguna receta. ¿Qué tiene de malo esta receta?

Primero, si un problema es difícil, resulta difícil saber cómo será su solución. A menudo se requiere experiencia incluso para saber qué se desea hacer. La mayor parte de los clientes reconocen este hecho y son suficientemente flexibles para aceptar un intervalo de soluciones posibles. Los clientes desean evitar la imposición de especificaciones arbitrarias que les provocaría la pérdida de oportunidades no costosas o incurrir en faltas costosas. Con los problemas difíciles, la gente desea mantener abiertas sus opciones.

Segundo, la mayor parte de los problemas cuenta con varias formas alternativas de resolverlos. Se requiere algo de experiencia para determinar la mejor forma de resolver un problema difícil. Para problemas muy difíciles es imposible conocer exactamente "cómo hacerlo" hasta que se ha hecho.

Tercero, cuando se "ha hecho", es necesario reconocer que no es perfecto. Habrá errores ocultos. Se descubrirá una mejor manera de hacerlo. El cliente descubrirá que hubiera sido mejor solicitar algo distinto. Y entonces será necesario hacerlo de nuevo.

Cuarto, si la prueba de algo complicado se pospone hasta el final, es casi seguro que se fracasará. La propuesta puede superar su prueba "final", aunque probablemente fracasará en realizar su última tarea, porque una sola prueba final es incapaz de detectar todos los problemas.

Así, ¿cómo es posible manejar estas dificultades?

1. Desarrollar y mantener un punto de equilibrio sensible entre especificaciones estrictas y flexibilidad.
2. Realizar pruebas continuas a todos los niveles. Esto es de ayuda para identificar pronto problemas cuando es fácil resolverlos, y proporciona valoración objetiva del grado de avance. Suponga que usted es responsable de un gran proyecto de programación y pregunta a sus programadores cómo van las cosas. Usted no desea que le respondan simplemente "bien". Usted quiere que le demuestren cómo ejecutan las pruebas que muestran lo que su código actual realmente hace.

Realización de pruebas

Se ha dicho que, en promedio, los programadores experimentados cometen un error por cada ocho o 10 líneas de código.[3] ¡Uf! Son bastantes errores. Con esta elevada incidencia de errores, esperamos que el lector esté convencido respecto a la importancia de realizar pruebas.

La realización de pruebas tiene tres aspectos:

Compruebe primero las cuestiones más evidentes.

- Primero, someta su programa a valores de entrada típicos. Si su programa no funciona con valores de entrada típicos, entonces está en serios problemas. Los colaboradores y los usuarios finales pueden poner en duda su competencia si su programa genera repuestas incorrectas para los casos típicos.
- Segundo, someta su programa a valores de entrada que estén en los límites de la aceptación. Este tipo de pruebas a menudo revela problemas sutiles que no surgen sino hasta después, cuando son mucho más difíciles de resolver.
- Tercero, someta su programa a valores de entrada inválidos. En respuesta a un valor de entrada inválido, su programa debe imprimir un mensaje amistoso para el usuario que identifique el problema y solicite que el usuario vuelva a intentarlo.

Muchas personas suponen que las pruebas se hacen una vez que el producto está terminado. Esta idea es desafortunada, ya que una sola prueba al final de la producción de un artículo complicado casi siempre es inútil. Si el producto fracasa en dicha prueba, puede resultar difícil determinar la razón del fracaso. Si la reparación requiere muchos cambios, entonces quizá se desperdició bastante trabajo. Si el producto no fracasa después de una sola prueba final, entonces usted puede tener una falsa sensación de seguridad, de que todo está en orden cuando no es así. Superar una sola prueba final puede en realidad ser peor que no superarla, ya que la aprobación es la motivación para liberar el producto. Resulta mucho más costoso resolver un problema una vez que el producto ha sido liberado. (¡Ray sabe sobre esto!) En conclusión, no espere hasta el final para iniciar la realización de pruebas. Pruebe su programa en forma regular a lo largo de todo el proceso de desarrollo.

Los programadores novatos algunas veces piensan que "no es científico" tener una idea preconcebida del resultado de una prueba antes de realizarla. Esto es un error. Es importante tener una buena idea de cuál puede ser el resultado de la prueba antes de realizarla. Antes de oprimir el botón "ejecutar", ¡diga en voz alta cuál cree que será el resultado de la prueba! Esto mejora su oportunidad de reconocer un error.

La realización de pruebas lo mantiene en el terreno de juego. En el desarrollo de cualquier programa, es necesario intercalar la realización de pruebas y la codificación a fin de tener una retroalimentación más expedita. Si un programador experimentado comete un error en ocho de cada 10 líneas de codificación, ¡un nuevo programador sabe que debe realizar una prueba cada cuatro o cinco líneas de nueva codificación! Así sería más fácil detectar errores y reducir el nivel de estrés. Mientras más a menudo se realicen pruebas, más retroalimentación positiva se obtiene, esto ayuda a mantener una actitud optimista: proporciona un sentimiento de "calidez". La realización de pruebas a menudo hace de la programación una experiencia más placentera.

No existe ninguna forma práctica para comprobar todos los aspectos de un sistema complicado cuando se le considera sólo desde el exterior. La realización de pruebas debe realizarse en cada componente y en cualquier combinación de componentes, a todos los niveles. Como se verá en un análisis ulterior, la realización de pruebas requiere esencialmente de algún tipo de codificación de pruebas adicional. Algunas veces se trata de un regulador extra para algo que sólo se usa en un entorno de prueba y no en un entorno de ejecución real. En efecto, es un trabajo adicional, aunque bien vale la pena hacerlo. Escribir y usar un código de prueba ahorra tiempo a largo plazo, y se obtiene un mejor producto final.

8.6 Diseño arriba-abajo

La metodología dominante de diseño para sistemas de alto rendimiento es la estrategia del diseño arriba-abajo. Este diseño requiere que el diseñador considere primero la primera gran imagen; es decir, la "parte superior", o "arriba". Después de terminar el diseño arriba, el diseñador trabaja en el diseño en el si-

[3] Por supuesto, nosotros (John y Ray) jamás cometemos errores. ☺

guiente nivel inferior. El proceso de diseño continúa de esta manera iterativa hasta que se alcanza el nivel inferior (el nivel más detallado).

Para un proyecto de programación orientado a objetos, el diseño arriba-abajo significa iniciar con una descripción del problema y trabajar hacia una solución usando estas directrices:

1. Tomar una decisión sobre las clases necesarias. Normalmente, es necesario incluir una clase reguladora como una de las clases. Para determinar las otras clases, el problema debe plantearse en términos de sus objetos componentes. Para cada tipo único de objeto debe especificarse una clase. Con sistemas grandes que cuentan con muchas clases, el diseño arriba-abajo difiere la identificación de clases identificadas, ya que identificar este tipo de clases constituye en sí un detalle.
2. Para cada clase, de entre sus variables de instancia es necesario decidir cuáles deben ser variables de instancia que identifican atributos de objetos. La clase reguladora no debe tener ninguna variable de instancia.
3. Para cada clase es necesario decidir sobre sus métodos `public`. La clase reguladora debe contener sólo un método `public`: `main`.
4. Para cada método `public` es necesario implementar de manera arriba-abajo. Considere que cada método `public` es un método "superior". Si está implicado equitativamente y es posible separarlo en subtareas, debe llamar a los métodos de ayuda `private` para efectuar el trabajo de las subtareas. Se debe terminar escribiendo los métodos superiores antes de empezar a escribir los métodos de ayuda como *stubs* (talones). Un talón es un método falso que actúa como marcador de posición de un método real. Típicamente, el cuerpo de un talón consta de una sentencia de impresión que muestra algo como "En el método x, los parámetros = a, b, c" donde x es el nombre del método y a, b y c son valores de argumentos ya pasados. Más adelante, en esta sección, se presentará un ejemplo.

Debe comprobarse de inmediato.

5. Probar y depurar el programa. El talón sugerido imprime mensajes que ayudan a seguir las acciones del programa.
6. Sustituya los métodos de talones uno por uno con métodos de ayuda completamente implementados. Al cabo de cada sustitución, vuelva a probar y depurar el programa.

Algunas veces, el diseño arriba-abajo se denomina *refinamiento paso por paso.* El término paso por paso se usa porque la metodología alienta a los programadores a implementar soluciones en una forma iterativa donde cada "paso" de la solución constituye una versión refinada del paso previo de la solución. Después de implementar las tareas del nivel superior, el programador regresa y refina la solución al implementar tareas en los siguientes niveles inferiores.

Beneficios de utilizar diseño arriba-abajo

En el diseño arriba-abajo, el diseñador no se preocupa inicialmente de los detalles de la implementación de detallar las subtareas. El diseñador se concentra primero en la "gran perspectiva". Debido a lo anterior, el diseño arriba-abajo es aceptable porque hace que un proyecto avance en la dirección adecuada. Esto ayuda a que el programa completado coincida con las especificaciones originales.

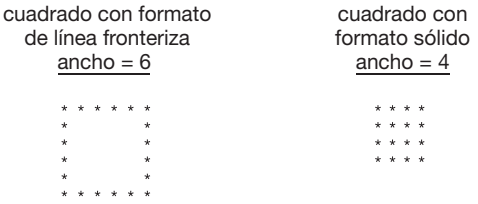
El diseño arriba-abajo es particularmente apropiado para un proyecto que implica muchos programadores. Su temprano énfasis en la perspectiva global obliga a los programadores a coincidir en metas comunes. Este énfasis a lo organizacional promueve la coherencia y evita la diversificación del mismo. La metodología de diseño arriba-abajo facilita un control de gestión estricto.

Ejemplo del programa cuadrado: versión del primer corte

A continuación se aplicará la metodología de diseño arriba-abajo a un ejemplo sencillo. Se implementará la clase `Cuadrado` de modo que cada objeto `Cuadrado` pueda:

Conciba una forma para resolver el problema.

- Inicializar el ancho del cuadrado.
- Calcular y devolver su área.
- Dibujarse a sí mismo con asteriscos usando una línea fronteriza de asteriscos o un patrón sólido de asteriscos. Cada vez que se traza el cuadrado se pregunta al usuario si desea un formato de línea fronteriza o un formato sólido, como se muestra a continuación:



Al usar las directrices de diseño arriba-abajo anteriores, como primer paso se debe decidir sobre las clases. En este ejemplo sencillo resulta fácil identificar a todas las clases justo al principio: `ControladorCuadrado` y `Cuadrado`. El siguiente paso consiste en decidir sobre las variables de instancia. Estas variables deben ser un conjunto mínimo definitivo de propiedades del objeto: variables de estado. Para especificar un cuadrado todo lo que se requiere es un dato. El dato que suele usarse es el ancho, de modo que como única variable de instancia se usará el ancho (`width`).

Pero ¿qué hay respecto al área del cuadrado? El área es una propiedad, aunque es una función simple del ancho: el área es igual al ancho al cuadrado. Puesto que resulta fácil calcular el área a partir del ancho, sería redundante incluir el área como otra variable de estado. En principio, es posible usar `area` como única variable de estado y calcular `ancho` como la raíz cuadrada de `area` cada que se requiera el `ancho`. Pero calcular la raíz cuadrada es más difícil que calcular el cuadrado, y a menudo se termina con un valor no entero para `ancho`, lo cual sería difícil de mostrar en el formato prescrito de asteriscos. Así, para este problema, la mejor estrategia es usar `ancho` como única variable de instancia.

Identifique las variables de estado.

¿Y qué hay respecto al carácter sólido del cuadrado? Esta opción es conceptual. Si se desea entender el carácter sólido del cuadrado como una propiedad inherente de los objetos de la clase `Cuadrado`, resulta idóneo crear otra variable de instancia como `sólido booleano` (*boolean solid*). Por otra parte, si el carácter sólido del cuadrado se entiende simplemente como una opción a mostrar temporalmente, entonces este carácter no debe tener la condición de variable de estado, y no debe ser una variable de instancia. Para este ejemplo, se ha elegido entender el carácter sólido del cuadrado simplemente como una opción a mostrar temporalmente, de modo que no se incluye como otra variable de instancia.

Volviendo a las directrices de diseño arriba-abajo, se observa que el siguiente paso consiste en decidir sobre los métodos públicos (`public`). La descripción del problema a menudo determina qué requiere ser `public`. A continuación es lo que se necesita:

- un constructor que establezca el ancho del cuadrado
- `setArea`: calcula el área del cuadrado
- `dibujar`: muestra el cuadrado con asteriscos usando asteriscos como línea fronteriza o un patrón sólido de asteriscos.

Ahora se retrocederá un poco para considerar lo que se ha hecho hasta ahora. Vea la figura 8.5. Muestra un diagrama de la clase UML de primer corte de nuestras clases de solución, variables de instancia y los métodos constructor y `public`.

El siguiente paso en el proceso de diseño arriba-abajo consiste en implementar el método `main` en la clase de nivel superior. Esta implementación se muestra en la figura 8.6. El código en `main` incluye las llamadas al constructor `Cuadrado` y los métodos identificados en la figura 8.5, pero aún no dice nada sobre la forma en que se implementan estos miembros de la clase `Cuadrado`.

El siguiente paso consiste en implementar los métodos públicos en la clase `Cuadrado`. Esta implementación se muestra en la figura 8.7*a*. Los métodos constructor y `getArea` son directos, por lo que no requieren explicación. Sin embargo, observe que el "get" en `getArea` hace que este método parezca un accesor que simplemente recupera una variable de instancia.

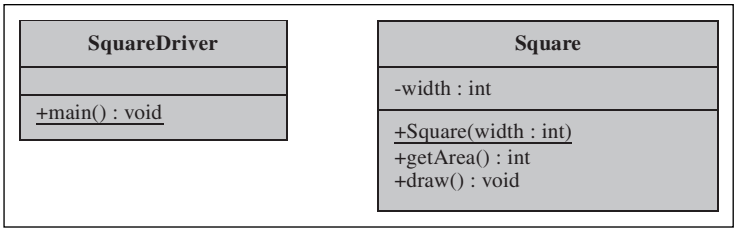


Figura 8.5 Diagramas UML de la clase Cuadrado: versión del primer corte.

```
/*************************************************************
* SquareDriver.java
* Dean & Dean
*
* Este es el controlador de la clase Square.
*************************************************************/

import java.util.Scanner;

public class SquareDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Square square;

    System.out.print("Enter width of desired square: ");
    square = new Square(stdIn.nextInt());
    System.out.println("Area = " + square.getArea());
    square.draw();
  } // end main
} // end class SquareDriver
```

Figura 8.6 Clase `SquareDriver`.

¿Es correcto crear esta falsa impresión? Sí, porque la variable de instancia es `privada` y por tanto es oculta a la vista del público. De hecho, como ya se observó, ¡en realidad hubiera podido usarse `area` como la única variable de instancia! Un usuario de una clase no tiene por qué saber cómo se implementó. No es necesario preocuparse de la implementación cuando se elige el nombre de un método. Lo que importa es el efecto, y `getArea` describe con toda precisión el efecto de llamar a ese método.

El método `draw` pide al usuario escoger un formato fronterizo o uno sólido para la presentación del cuadrado. Ahora resulta evidente que este método no es trivial. Las llamadas a los métodos `drawBorderSquare` y `drawSolidSquare` son ejemplos de subtareas que es necesario dividir en métodos de ayuda por separado.

Talones

El diseño arriba-abajo indica implementar los métodos de ayuda inicialmente como talones. Para el programa Cuadrado, eso significa implementar `drawBorderSquare` y `drawSolidSquare` como talones. Observe los talones en la figura 8.7*b*.

Como quizás el lector pueda inferir de los ejemplos, un talón no es de mucha utilidad. Su objetivo principal es satisfacer al compilador, de modo que el programa sea capaz de compilar y ejecutar. Su segundo objetivo es proporcionar resultados que confirmen que el método fue llamado y (donde corresponda) mostrar los valores que fueron procesados por ese método. Cuando se ejecuta el programa Square con talón, se obtiene una de las sesiones siguientes:

```
Enter width of desired square: 5
Area = 25.0
Print with (b)order or (s)olid? b
In drawBorderSquare
```

o esta sesión de muestra:

```
Enter width of desired square: 5
Area = 25.0
Print with (b)order or (s)olid? s
In drawSolidSquare
```

Pruebe una cosa a la vez.

El uso de talones permite a los programadores probar sus programas implementados parcialmente para determinar si su comportamiento es correcto luego del nivel del talón. También facilita la depuración. Después de compilar y ejecutar el programa exitosamente con talones, éstos deben sustituirse método por método por un código actual. A medida que se sustituye cada talón, el programa debe probarse y depurarse. Si se presenta alguna dificultad, es fácil detectarla, puesto que se sabe que probablemente se ubica en el método más recientemente sustituido.

Ejemplo del programa cuadrado: versión del segundo corte

El siguiente paso en el proceso del diseño arriba-abajo es sustituir las implementaciones de los métodos de ayuda talones por implementaciones reales. Hay dos métodos de ayuda con los cuales trabajar: `drawBorderSquare` y `drawSolidSquare`.

```
/*************************************************************
* Square.java
* Dean & Dean
*
* Esta clase administra cuadrados.
*************************************************************/

import java.util.Scanner;

public class Square
{
  private int width;

  //*********************************************************

  public Square(int width)
  {
    this.width = width;
  }

  //*********************************************************

  public int getArea()
  {
    return this.width * this.width;
  }

  //*********************************************************

  public void draw()
  {
    Scanner stdIn = new Scanner(System.in);

    System.out.print("Print with (b)order or (s)olid? ");
    if (stdIn.nextLine().charAt(0) == 'b')
    {
      drawBorderSquare();
    }
    else
    {
      drawSolidSquare();
    }
  } // end draw
```

Figura 8.7*a* Clase `Square`: versión de primer corte, parte A.

```
  //*********************************************************

  private void drawBorderSquare()                  // un TALÓN.
  {
    System.out.println("In drawBorderSquare");
  }

  //*********************************************************

  private void drawSolidSquare()                   // un TALÓN.
  {
    System.out.println("In drawSolidSquare");
  }
} // end class Square
```

Figura 8.7*b* Clase `square`: versión de primer corte, parte B.

Se empezará con el método de ayuda `drawBorderSquare`. Imprime una línea horizontal de asteriscos, imprime los lados del cuadrado y luego imprime otra línea horizontal de asteriscos. El pseudocódigo de este algoritmo es el siguiente:

```
drawBorderSquare method
  draw horizontal line of asterisks
  draw sides
  draw horizontal line of asterisks
```

Las tres sentencias de dibujo `drawBorderSquare` representan tareas no triviales. Así, cuando el seudocódigo `drawBorderSquare` se traduce en un método Java, para cada una de las subtareas de dibujo se usan llamadas a métodos:

```
private void drawBorderSquare()
{
  drawHorizontalLine();
  drawSides();
  drawHorizontalLine();
} // end drawBorderSquare
```

A continuación se considerará el método de ayuda `drawSolidSquare`. Imprime una serie de líneas horizontales de asteriscos. El pseudocódigo de este algoritmo es el siguiente:

```
drawSolidSquare method
  for (int i=0; i<square's width; i++)
    draw horizontal line of asterisks
```

De nuevo, la sentencia de dibujo representa una tarea no trivial. Por tanto, cuando el pseudocódigo `drawSolidSquare` se traduce en un método Java, se usa una llamada repetida del método para la subtarea dibujar:

```
private void drawSolidSquare()
{
  for (int i=0; i<this.width; i++)
  {
    drawHorizontalLine();
  }
} // end drawSolidSquare
```

Observe que ambos métodos: `drawBorderSquare` y `drawSolidSquare` llaman al mismo método de ayuda `drawHorizontalLine`. La capacidad de poder compartir el método `drawHorizontalLine` es una recompensa justa por el hecho de usar métodos de ayuda, y es un buen ejemplo de este principio general:

> Si uno o más métodos realizan la misma tarea, la codificación redundante debe evitarse haciendo que estos métodos llamen a un método de ayuda compartido que realice la subtarea.

Al escribir el código final para los métodos `drawBorderSquare` y `drawSolidSquare` y escribir el código talón para los métodos `drawHorizontalLine` y `drawSides` se completa el código para la versión del segundo corte del programa. Cuando se ejecuta con sentencias de impresión idóneas en los dos métodos talón, `drawHorizontalLine` y `drawSides`, la versión del segundo corte produce una de las siguientes sesiones muestra:

```
Enter width of desired square: 5
Area = 25.0
Print with (b)order or (s)olid? b
In drawHorizontalLine
In drawSides
In drawHorizontalLine
```

o esta sesión muestra:

```
Enter width of desired square: 5
Area = 25.0
Print with (b)order or (s)olid? s
In drawHorizontalLine
In drawHorizontalLine
In drawHorizontalLine
In drawHorizontalLine
In drawHorizontalLine
```

Ejemplo del programa Cuadrado: versión final

Mantenga al corriente la información.

Para facilitar la gestión, una buena idea es formalizar el diseño del programa en varios puntos durante el proceso de diseño. La formalización suele asumir la forma de diagramas de clase UML. Contar con diagramas de clase UML actualizados resulta útil para asegurar la coherencia del programa. Por lo menos, los diagramas de clase UML actualizados aseguran que todos los miembros de un proyecto usen las mismas clases, variables de instancia y encabezados de métodos. Vea la figura 8.8.

En la figura 8.8 se muestra el diagrama de la clase UML de todo el programa Cuadrado completo. Es el mismo que el diagrama de la clase UML, excepto que se han agregado métodos de ayuda.

La versión del segundo corte del programa Square contiene implementaciones de talón para los métodos `drawHorizontalLine` y `drawSides`. Ahora, es necesario sustituir estos métodos de talón por

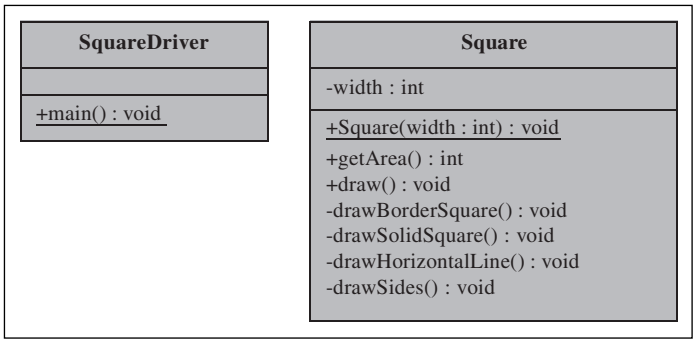


Figura 8.8 Diagrama del programa UML Cuadrado: versión final.

métodos reales. Las figuras 8.9*a* y 8.9*b* contienen la versión final de nuestra clase `Square`. Los únicos puntos nuevos son los métodos `drawHorizontalLine` y `drawSides`, que son directos. Se solicita que el lector estudie sus propias implementaciones en la figura 8.9*b*.

Inconveniente del diseño arriba-abajo

Casi todos los proyectos que diseña el ser humano necesariamente deben incluir alguna forma de pensamiento arriba-abajo. Sin embargo, el verdadero diseño arriba-abajo presenta algunos efectos colaterales indeseables. Uno es que los módulos subordinados tienden a permanecer especializados. Un ejemplo particular bien conocido y muy notable de cómo esta forma de pensar puede llevar a una especialización excesiva es el caso de los ceniceros de $660 del Pentágono. El Pentágono (cuartel general del Departa-

```
/*********************************************************
* Square.java
* Dean & Dean
*
* Esta clase administra cuadrados.
*********************************************************/

import java.util.Scanner;

public class Square
{
  private int width;

  //*******************************************************

  public Square(int width)
  {
    this.width = width;
  }

  //*******************************************************

  public double getArea()
  {
    return this.width * this.width;
  }

  //*******************************************************

  public void draw()
  {
    Scanner stdIn = new Scanner(System.in);

    System.out.print("Print with (b)order or (s)olid? ");
    if (stdIn.nextLine().charAt(0) == 'b')
    {
      drawBorderSquare();
    }
    else
    {
      drawSolidSquare();
    }
  } // end draw
```

Figura 8.9*a* Clase `Square`: versión final (copia exacta de la figura 8.7*a*).

```
  //*****************************************************

  private void drawBorderSquare()
  {
    drawHorizontalLine();
    drawSides();
    drawHorizontalLine();
  } // end drawBorderSquare

  //*****************************************************

  private void drawSolidSquare()
  {
    for (int i=0; i<this.width; i++)
    {
      drawHorizontalLine();
    }
  } // end drawSolidSquare

  //*****************************************************

  private void drawHorizontalLine()
  {
    for (int i=0; i<this.width; i++)
    {
      System.out.print("*");
    }
    System.out.println();
  } // end drawHorizontalLine

  //*****************************************************

  private void drawSides()
  {
    for (int i=1; i<(this.width-1); i++)
    {
      System.out.print("*");
      for (int j=1; j<(this.width-1); j++)
      {
        System.out.print(" ");
      }
      System.out.println("*");
    }
  } // end drawSides
} // end class Square
```

Figura 8.9*b* Clase `Square`: versión final, parte B (versión desarrollada de la figura 8.7*b*).

mento de Defensa de Estados Unidos) contaba con los servicios de un gran contratista militar que manufacturaba ceniceros para usar en el Pentágono. Puesto que la compatibilidad es importante para muchos componentes militares, los militares requieren una adherencia fidedigna a sus especificaciones, y los contratistas desarrollan de forma natural procedimientos y actitudes que promueven el cumplimiento de este requisito. No obstante, algunas veces puede haber demasiado de algo bueno. Los ceniceros se ajustaban perfectamente a sus especificaciones, pero el precio de cada uno era de $660. El diseño arriba-abajo llegó a un extremo ridículo. Aun cuando algunas de las especificaciones de nivel superior pueden no ser convencionales, tal vez el contratista siguió el procedimiento operativo normal e intentó cumplirlas a la

perfección. Cita hipotética del gerente de mercadotecnia del contratista: "Lo especificado no concuerda con nada de lo disponible, de modo que fue necesario hacerlo manualmente en la tienda de máquinas."

Quizá el lector piense que esta historia es interesante, pero ¿cuál es la relación de los ceniceros de $660 con la programación? La filosofía arriba-abajo puede llevar a prácticas de desarrollo ineficientes. En el caso extremo, esta filosofía condujo al contratista a dedicar enormes esfuerzos en el diseño y manufactura de algo tan simple como un cenicero. En general, la filosofía arriba-abajo puede motivar a las personas a "reinventar la rueda", lo cual incrementa el costo global del producto. También reduce la confiabilidad del producto final. ¿Por qué? Porque cuando todo se reinventa o es nuevo, no hay una historia pasada y depuración en que confiar.

8.7 Diseño ascendente

A continuación se revisará el opuesto del diseño arriba-abajo: el diseño ascendente. Este diseño implementa primero tareas específicas de bajo nivel. Para aplicar el diseño ascendente al programa Cuadrado, primero podría implementarse un método `drawSolidSquare`. Luego, podría implementarse un método `drawBorderSquare`. Al terminar estos métodos de bajo nivel, podrían implementarse métodos de nivel superior, que están a cargo de realizar tareas más generales, como un método `draw` para dibujar cualquier tipo de cuadrado; uno sólido o uno delineado.

A medida que se implementa cada componente del programa, es necesario probarlo de inmediato con un controlador adecuado para ese componente en particular. Entonces no se requiere ningún talón, puesto que métodos de nivel inferior ya probados están a disposición para ser llamados por cualquier método de alto nivel que se esté probando actualmente.

Para programas simples como muchos de los que aparecen a lo largo de todo este libro, el diseño ascendente constituye una estrategia idónea a utilizar porque permite enfocarse rápidamente en la esencia de cualquier problema que actualmente sea el más crítico, y permite diferir los detalles de la presentación. Para mencionar un ejemplo de diseño ascendente, considere cualquier programa de este libro en el que se presenta una clase controlada antes de presentar un controlador para esa clase. Siempre que se haga lo anterior, se está usando una presentación ascendente, y se extiende una invitación a reflexionar sobre el programa que se está describiendo.

El diseño ascendente también facilita mucho más el uso de software escrito previamente, como el del Java API y el descrito antes en el capítulo 5. El Java API es una fuente particularmente buena de software escrito previamente porque su código 1) está optimizado para alta velocidad y bajo consumo de memoria y 2) es altamente confiable porque durante años ha sido sometido a pruebas y depuraciones. Está bien usar Java API, aunque para aprender a usarlo se requiere tiempo. Para aprender sobre Java API, consulte el sitio http://java.sun.com/javase/6/docs/api/. Ahí encontrará varias formas de considerar las cosas. A continuación se presentan dos técnicas:

1. Intente adivinar el nombre de una clase que parezca idónea. Use la barra de desplazamiento en el marco de clases para buscar el nombre de clase adivinado. Hay alrededor de 4 000 clases, de modo que encontrar una clase particular requiere un ratón de buen comportamiento (recomendamos una dieta idónea y ejercicio para que el ratón funcione a la perfección). Cuando encuentre el nombre de una clase que parezca promisorio, haga clic en él y lea sobre sus constantes y métodos públicos.
2. Las clases relacionadas están agrupadas entre sí en alrededor de 166 paquetes. Use la barra de desplazamiento en el marco de los paquetes para encontrar uno que parezca promisorio. Haga clic en el paquete y busque en sus clases. De nuevo, cuando encuentre el nombre de una clase que parezca promisorio, haga clic en él y lea sobre sus constantes y métodos públicos.

El uso de software escrito previamente para sus módulos de bajo nivel reduce el tiempo de desarrollo y el costo del proyecto. También mejora la calidad del producto, porque quizá las partes previamente escritas del programa ya han sido probadas y depuradas exhaustivamente. Así como ocurre en el caso del código Java API, a menudo se encuentra que el software de bajo nivel previamente escrito es bastante flexible, ya que se diseñó para una amplia gama de aplicaciones. Esta flexibilidad intrínseca de bajo nivel facilita la ampliación de las capacidades del programa cuando sea necesario actualizarlo en el futuro. El uso de software previamente escrito puede facilitar el desarrollo paralelo. Si varios programadores quieren usar

un módulo subordinado común, pueden hacerlo de forma independiente. No es necesario que coordinen sus esfuerzos, porque el diseño del módulo ya está establecido y es estable.

Trabaje primero en el problema más crítico.

Otro beneficio del diseño ascendente es que proporciona libertad para la implementación de tareas en el orden más conveniente. Si hay interés particular en cuanto a si un cálculo particular es factible, resulta importante empezar a trabajar en dicho cálculo lo más pronto posible. Con el diseño ascendente no es necesario esperar la dirección en cuestión: hay que atacarla de inmediato. Así, es posible determinar lo más pronto posible si la preocupación es un obstáculo. En forma semejante, si hay alguna tarea de bajo nivel que requiera bastante tiempo para su terminación, entonces el diseño ascendente permite empezar a trabajar en ello de inmediato y evitar cuellos de botella potenciales.

No obstante, hay varias desventajas de usar diseño ascendente. En comparación con el diseño arriba-abajo, el diseño ascendente proporciona menos estructura y orientación. A menudo resulta difícil saber dónde empezar, y puesto que es difícil predecir el desarrollo, los proyectos con diseño ascendente son difíciles de manipular. En particular, a los programadores, con menos orientación inherente, les resulta difícil mantener sus programas en la pista. Como resultado, los programadores podrían dedicar cantidades importantes de tiempo a trabajar en un código que quizá no sea relevante para el programa final. Otra desventaja de usar diseño ascendente es que puede conducir a dificultades al hacer que el producto final se ajuste precisamente a las especificaciones del diseño. El diseño arriba-abajo facilita el ajuste al contar con especificaciones detalladas al principio. Con el diseño ascendente, a las especificaciones sólo se les considera de manera superficial al principio.

Entonces, ¿cuándo usar diseño ascendente? Cuando es posible usar una cantidad sustancial de software de bajo nivel escrito y probado previamente, el diseño ascendente facilita el diseño alrededor de este software de modo que se ajuste de forma natural al programa completo. Cuando es posible usar una cantidad sustancial de software escrito previamente que esté abierto para su inspección y ya diseñado también para ajustarse (como el software Java API),[4] el diseño ascendente promueve de manera simultánea alta calidad y bajo costo. Cuando los detalles de bajo nivel son críticos, el diseño ascendente motiva a tratar primero los problemas difíciles: proporciona mucho tiempo para solucionarlos. Así, el diseño ascendente puede ayudar a minimizar el tiempo de entrega.

Un ejemplo conocido del diseño ascendente de software es el primer desarrollo del sistema operativo de Microsoft Windows. La versión original de Windows estaba incorporada en la parte superior del ya existente y exitoso sistema operativo DOS.[5] La siguiente versión importante de Windows se incorporó en la parte superior de un nuevo núcleo de software de bajo nivel denominado "NT" (de Nueva Tecnología). Es importante observar que el componente del código fuente en estos casos siempre estaba abierto a y bajo control de los creadores del sistema, ya que era propiedad de la misma compañía.[6]

8.8 Diseño basado en casos

Hay otra forma fundamental para resolver problemas y diseñar cosas. La gente común lo hace todo el tiempo de manera cotidiana. En lugar de ejecutar una serie de pasos formales de diseño ascendente o diseño arriba-abajo, se busca un problema ya resuelto parecido al problema a la mano. Luego, se imagina la manera en que se resolvió ese problema y la solución se modifica para que se ajuste al problema que se desea resolver. Este enfoque es holístico. Empieza con la solución completa y "ajusta" esa solución a diferentes aplicaciones.

---

[4] Aunque hemos alentado a que el lector considere el software API de Java como completamente encapsulado, Sun no preserva el secreto del código fuente del API de Java. Puede ser descargado y puesto a disposición para su inspección por programadores de Java.

[5] El conjunto de comandos que es posible introducir en una ventana de comandos de Microsoft Windows, esencialmente son comandos DOS, constituyen legado de la IBM PC que proviene de inicios de la década de 1980.

[6] En principio, es posible construir sistemas de software a partir de componentes que son programas *Commercial-Off-The-Shelf* (COTS) de diferentes compañías. Esta estrategia puede usarse para evitar "reinventar la rueda" en gran forma, y minimiza nuevo código al "pegamento" que constituyen las componentes de interfaces. Sin embargo, se requiere más tiempo para escribir este código pegamento que para escribir un código normal. Además, puesto que (en general) el desarrollador del sistema no tiene acceso al componente del código fuente y carece de controles de la evolución de los componentes, el proceso de desarrollo es relativamente arriesgado, y el programa compuesto resultante es relativamente frágil. El diseño de sistemas basado en COTS posee una metodología característica que rebasa el alcance de este texto.

Si el lector tiene acceso al código fuente y el derecho de copiarlo o modificarlo y luego redistribuirlo en un nuevo contexto, entonces puede modificar un programa existente o partes significativas del código existente. Algunas veces, el código que desea tomar prestado es uno que usted mismo escribió para una aplicación diferente. Ese código merece su consideración, ya que usted está estrechamente relacionado con lo que hace y la forma en que lo hace. Por ejemplo, muchos de los proyectos de este libro se diseñaron para mostrarle cómo resolver una amplia gama de problemas del mundo real. Puede usar los algoritmos presentados en las asignaciones de proyectos para generar un código Java que resuelve versiones particulares de esos problemas. Una vez que ha escrito el código es completamente libre de modificarlo y reutilizarlo en cualquier otro contexto para resolver otras variantes de esos problemas.

A menudo, el código que uno quiere usar lo escribió otra persona. Usar ese código, ¿constituye un robo o un plagio? Podría ser. Si el código está registrado y no cuenta con la autorización para usarlo, no debería intentarlo. Sin embargo, el lector podría tener la autorización para usarlo. Siempre que se use un código escrito por otra persona, es necesario reconocer e identificar su fuente.

Hay un cuerpo creciente de lo que se denomina software "libre",[7] que depura y mantiene un cuerpo selecto de expertos, y está disponible para que la gente lo utilice y modifique según sus necesidades, en el supuesto de que se respeten ciertas reglas razonables. Básicamente, estas reglas son: reconocer la fuente y no intentar beneficiarse comercializando el código original. Algunas veces este software es un código de bajo nivel que puede usarse como el software Java API. Pero también a veces se trata de un programa completo que puede adaptarse al problema en el que se está trabajando.

8.9 Mejoramiento iterativo

A menudo es necesario empezar a trabajar en un problema para comprender cómo resolverlo. Esto lleva a un proceso de diseño cuya naturaleza suele ser iterativa. En la primera iteración, se implementa una solución esencial del problema. En la siguiente iteración, se agregan características y se implementa una solución mejorada. Se siguen agregando características y repitiendo el proceso de diseño hasta que se ha implementado una solución que hace todo lo que se desea. Este proceso repetitivo se denomina *mejoramiento iterativo*.

Elaboración de prototipos: Un primer paso opcional

Un *prototipo* es una implementación "esencial" o "rudimentaria", o quizás una "simulación" falsificada de un programa prospectivo. Debido al alcance limitado de los prototipos, los desarrolladores pueden producir prototipos relativamente rápido y presentarlos a los clientes muy pronto en el proceso de desarrollo.

Asegúrese de estar resolviendo el problema correcto.

Un prototipo ayuda a los usuarios finales a tener una idea temprana de qué uso podría darse al programa, mucho antes que concluya. Ayuda a los clientes a suministrar retroalimentación pronta que mejore la calidad de la especificación del producto. Así, la elaboración de prototipos constituye una ayuda invaluable para la primera parte del proceso de diseño ascendente. Sin los prototipos, siempre se corre el riesgo de resolver el problema equivocado. Inclusive si el problema se resuelve con mucha elegancia, si se trata del problema equivocado, entonces todo el esfuerzo es un desperdicio de tiempo.

Hay dos formas básicas para producir un prototipo. Una consiste en escribir una versión muy limitada del programa final en Java. Puesto que un prototipo debe ser relativamente simple, puede aplicarse el método de diseño que parezca más fácil. La otra forma consiste en usar una aplicación de computadora que proporcione representaciones agradables que simulen la interfaz del usuario del programa final para datos particulares "enlatados" o un rango estrecho de entradas del usuario.

La elaboración de prototipos puede ser una herramienta de comunicación valiosa, aunque debe usarse con cuidado. Suponga que se elabora un prototipo, se le muestra al cliente y éste dice: "Me agrada. Déme una copia para poder empezar a usarlo mañana." ¡No haga esto! Si su prototipo es una iteración

[7] Consulte el sitio http://www.fsf.org. La Fundación Free Software está "dedicada a promover los derechos de los usuarios de computadora a usar, estudiar, modificar y redistribuir programas de computadora". Dos ejemplos conocidos de este tipo de software son el sistema operativo GNU/Linux (GNU es el acrónimo de "Gnu's Not Unix") y el software Apache que subyace en la mayor parte de servidores de la red (http://www.apache.org).

temprana de una secuencia de iteraciones planificadas, pliéguese a lo que aprenda de la reacción del cliente, y proceda a la siguiente iteración como estaba planeado originalmente. Si su prototipo es justo una representación visual elaborada con componentes dispares, resista a la tentación de desarrollar ese prototipo hasta un producto terminado. Esto es tentador porque el lector podría pensar que así reduce tiempo de desarrollo. No obstante, agregar parches a algo improvisado y simulado suele producir resultados desastrosos, difíciles de mantener y actualizar. Al final, es necesario volver a escribir enormes cantidades del código, y la confusión asociada puede destruir el programa. Resulta mejor pensar en este prototipo como nada más un recurso de comunicación que provoca retroalimentación que mejora la especificación del producto.

Iteración

La primera iteración normal del diseño, o la iteración después de un prototipo opcional, debe ser una simple adaptación de algún programa ya existente o una implementación esencial desarrollada con la estrategia arriba-abajo o con la estrategia ascendente. Las iteraciones ulteriores pueden continuar utilizando o no la misma estrategia de desarrollo.

Ajuste la estrategia de diseño para manipular la necesidad actual más importante con recursos disponibles en ese momento.

¿Cómo decidir cuál estrategia usar para cada iteración? Seleccione la estrategia que manipule mejor su necesidad o interés más importante:

- Si su necesidad actual más importante es comprender lo que el cliente desea, es necesario elaborar un prototipo.
- Si su interés actual más importante consiste en el tiempo de entrega, intente usar una adaptación de software existente.
- Si su interés actual más importante es la posibilidad de implementar alguna función particular, use la estrategia ascendente para implementar esa función lo más pronto posible.
- Si sus necesidades más importantes son confiabilidad y bajo costo, use software escrito previamente junto con diseño ascendente.
- Si su mayor interés es el rendimiento global y control de gestión, use la estrategia arriba-abajo.

Un famoso ejemplo de diseño iterado es el programa espacial de la NASA del hombre en la Luna. El presidente Kennedy estaba pensando en la estrategia arriba-abajo cuando anunció el programa. No obstante, la primera implementación fue un prototipo. Usar una versión modificada del cohete Atlas ICBM, "Proyecto Mercurio", disparó a un hombre a cientos de millas hacia el océano Atlántico.

Las iteraciones subsecuentes del Proyecto Mercurio usaron un enfoque ascendente para poner en órbita terrestre a los astronautas. Luego, la NASA sustituyó el cohete propulsor Atlas por un cohete Titán ICBM más nuevo y grande, que puso en órbita terrestre a varias personas en varias operaciones del "Proyecto Géminis".

La siguiente iteración de la NASA fue un diseño arriba-abajo denominado "Proyecto Apolo". Originalmente, este proyecto consideraba la utilización de un cohete propulsor gigantesco denominado Nova. Después de trabajar cierto tiempo en este diseño, la NASA percibió que un cohete propulsor mucho más pequeño (denominado Saturno) bastaba si un vehículo de alunizaje más pequeño se desprendía de la nave matriz al orbitar la Luna, y el módulo de retorno del vehículo de alunizaje se desprendía de su mecanismo descendente.

El Proyecto Apolo fue un diseño arriba-abajo, optimizado por los requerimientos de la NASA, más que por una adaptación ascendente del equipo militar existente. Al final, el plan arriba-abajo que implicaba a Nova se desechó y fue sustituido por un plan arriba-abajo radicalmente distinto. Esta secuencia de desarrollo aparentemente errática constituye un gran ejemplo de diseño exitoso en el mundo real. La historia de software con éxito es la misma. Diferentes ciclos de diseño a menudo recalcan estrategias de diseño diferentes, y algunas veces hay cambios esenciales.

Mantenimiento

Una vez que un programa se ha desarrollado y puesto en operación, podría pensarse que no hay más necesidad de trabajar en él. No es así. En el mundo real, si un programa es útil, a los programadores suele pedírseles que lo *mantengan* después que fue puesto en operación. En promedio, 80% del trabajo en un programa exitoso se realiza después que el programa se pone en operación. El mantenimiento consiste en

efectuar depuraciones y realizar mejoras. El mantenimiento es mucho más fácil si al inicio y a lo largo del programa se aplican buenas prácticas de software. En primer lugar, esto incluye la redacción elegante del código preservando la elegancia al hacer modificaciones, y contar con documentación completa y bien organizada.

Recuerde que la documentación es más que sólo comentarios para los programadores que leen el código fuente. La documentación también incluye información de interfaz para programadores que desean utilizar clases ya compiladas. En el apéndice 6 se muestra cómo incrustar información de interfaz en su código fuente de modo que `javadoc` pueda leerlo y sea posible presentarlo como documentación de Sun del Java API. La documentación también incluye información para personas que no son programadores en absoluto, pero que requieren utilizar un programa terminado. Este tipo de documentación debe estar más orientado al usuario que a los resultados de `javadoc`.

Si el lector es responsable del mantenimiento de algún programa existente, a continuación se muestran algunas reglas prácticas:

1. Respete a su predecesor. No modifique nada del código que usted considere erróneo sino hasta que haya dedicado tanto tiempo a pensar en esta cuestión como algún otro programador (o usted mismo) dedicó para crearlo en primer lugar. Puede haber una razón importante para hacer algo de cierta forma, inclusive si hay algún problema en el modo de hacerlo, de modo que el lector debe comprender esa razón antes de hacer cambios.
2. Respete a su sucesor. Siempre que el lector tiene problemas para imaginar la sección del código que está haciendo, después de comprender a fondo el problema, debe fijar el código y la documentación, de modo que sea más fácil pensar la próxima vez.
3. Mantenga un banco "estándar" de datos de entrada de prueba (así como los datos de salida correspondientes) y úselo para comprobar que los cambios que usted ha realizado sólo afectan al problema que está intentando resolver y no tienen ningún otro efecto indeseable que pueda extenderse a todo el programa.

8.10 Método controlador de fusión en una clase controlada

Es legal incluir un método `main` en cualquier clase. La figura 8.10 contiene un programa Time simple que incluye su propio método `main`.

Hasta ahora, cada uno de los programas POO se ha dividido en clases por separado: una clase controladora y una o más clases controladas. Resulta más fácil comprender el concepto de un objeto si está asociado con una clase mientras el código que lo instancia está asociado con otra clase. Las clases controladas y las clases controladoras tienen papeles distintos. Una clase controlada describe una cosa que está siendo modelada. Por ejemplo, en nuestros programas `Ratón`, la clase `Ratón` describe un ratón. Una clase controladora contiene un método `main`, y controla la clase `Ratón` por separado. En nuestros programas `Ratón`, la clase `MouseDriver` instancia los objetos `Ratón` y realiza acciones sobre estos objetos. El uso de dos o más clases adopta el hábito de colocar tipos distintos de cosas en módulos diferentes

Aunque aquí se continuarán utilizando clases por separado para la mayor parte de nuestros programas, para programas breves que no hacen mucho sino demostrar un concepto, algunas veces `main` se fusionará en la clase que implementa el resto del programa. Es una cuestión de conveniencia: hay un archivo menos que crear y hay menos código que introducir.

Haga que cada clase cuente con un método de prueba integrado.

En un gran programa que tiene una clase controladora a cargo de un gran número de clases controladas, algunas veces resulta práctico insertar un método `main` adicional en una o en todas las clases controladas. El método `main` adicional en una clase controlada sirve como un probador local para el código en esa clase. Siempre que se realice un cambio en el código de una clase particular, el método `main` local puede usarse para probar directamente la clase. Es fácil. Simplemente ejecute la clase de interés y el JVM automáticamente usa el método `main` de la clase. Una vez que se han comprobado los cambios realizados localmente, es posible proceder a ejecutar el controlador en un módulo de orden superior para probar más programas o todos los programas. No es necesario eliminar los métodos `main` locales. Es posible dejarlos simplemente para futuras pruebas locales o para una demostración de las características de cada clase particular. Una vez que se ejecuta la clase controladora global del programa, el

```
/*****************************************************************
* Time.java
* Dean & Dean
*
* Esta clase almacena la hora en forma de horas, minutos y
* segundos. Imprime la hora usando un formato militar.
*****************************************************************/

public class Time
{
  private int hours, minutes, seconds;

  //***************************************************************

  public Time(int h, int m, int s)
  {
    this.hours = h;
    this.minutes = m;
    this.seconds = s;
  }

  //***************************************************************

  public void printIt()
  {
    System.out.printf("%02d:%02d:%02d\n",
      hours, minutes, seconds);
  } // end printIt

  //***************************************************************

  public static void main(String[] args)
  {
    Time time = new Time(3, 59, 0);
    time.printIt();
  } // end main
} // end class Time
```

Éste es un controlador para el resto del código en esta clase.

Figura 8.10 Clase `hora` con un método controlador `main`.

JVM usa automáticamente el método `main` en esa clase controladora e ignora cualquier otro método `main` que pudiera estar en otras clases en el programa.

Así, es posible agregar un método `main` a cualquier clase, de modo que la clase pueda ejecutarse directamente y actuar como su propio controlador. Cuando un programa con varias clases contiene varios métodos `main` (no más de uno por clase), el método `main` particular que se utiliza es el que está en la clase actual cuando empieza la ejecución.

8.11 Acceso de variables de instancia sin la utilización del `this`

Ya desde hace rato se ha usado el `this` para acceder a las variables de instancia del objeto que llama desde un método. A continuación se presenta una explicación formal de cómo utilizar el `this`:

> Use el `this` dentro de un método de instancia o un constructor para acceder a las variables de instancia del objeto que llama.
> La referencia `this` distingue las variables de instancia de otras variables (como variables locales y parámetros) que tienen la misma identificación.

Sin embargo, en caso de que no haya ambigüedad en la identificación, se puede omitir el prefijo `this` cuando se accede a una variable de instancia.

El código en la figura 8.11 tiene varios sitios en los cuales bien vale mencionar el prefijo `this`. Es adecuado omitir el `this` en la sentencia en el método `setAge`, porque el nombre de la variable de instancia es diferente al nombre del parámetro. No es correcto omitir el `this` en la sentencia en el método `setWeight`, ya que la semejanza en los nombres de la variable de instancia y del parámetro podría originar una ambigüedad. Es permisible omitir el `this` en la sentencia en el método `print`, ya que no hay ambigüedad en los nombres.

Algunas veces un método de instancia es llamado por un objeto y tiene un parámetro que se refiere a un objeto diferente en la misma clase. El método `String equals` es un ejemplo conocido de esta si-

```
/*************************************************************
* MouseShortcut.java
* Dean & Dean
*
* Esta clase ilustra usos y omisiones del this.
*************************************************************/

public class MouseShortcut
{
  private int age;            // age in days
  private double weight;      // weight in grams

  //*******************************************************

  public MouseShortcut(int age, double weight)
  {
    setAge(age);
    setWeight(weight);
  } // end constructor

  //*******************************************************

  public void setAge(int a)
  {
    age = a;
  } // end setAge

  //*******************************************************

  public void setWeight(double weight)
  {
    this.weight = weight;
  } // end setWeight

  //*******************************************************

  public void print()
  {
    System.out.println("age = " + age +
      ", weight = " + weight);
  } // end print
} // end class MouseShortcut
```

Es correcto omitir el `this` antes de la variable de instancia `age`, ya que es diferente del parámetro, `a`.

No es correcto omitir el `this` antes de la variable de instancia `weight`, ya que es igual al parámetro, `weight`.

Es correcto omitir el `this` antes de las variables de instancia `age` y `weight`.

Figura 8.11 La clase `MouseShortcut` ilustra el uso y la omisión del `this`.

tuación. Dentro de este método hay un código que requiere referirse a dos objetos diferentes, el objeto que llama y el objeto al que se refiere el parámetro.

La forma más segura y comprensible de referirse a estos dos objetos consiste en utilizar el prefijo `this` para referirse al objeto que llama y el prefijo de referencia para referirse al otro objeto. Sin embargo, está bien omitir el `this` cuando se hace referencia al objeto que llama, y este hecho se observa bastante a menudo. Hace más compacto el código.

8.12 Resolución de problemas con el API de la clase `Calendar` (opcional)

Aunque los libros de texto (incluyendo éste) piden escribir programas pequeños que manipulan horas y fechas, si el lector está realmente interesado en estos temas, es como agitar el avispero de diferentes bases de números, diferentes duraciones del mes, años bisiestos, horas diarias de ahorro de luz solar, diferentes husos horarios y muchas convenciones de formateo. Para un trabajo serio sobre la hora y fecha, debe usarse el software previamente escrito de Java API. Desafortunadamente, no siempre es fácil encontrar la clase Java correcta. Éste es un caso idóneo, ya que casi todos los métodos en las clases evidentes `Time` y `Date` son obsoletos. Por lo general, en lugar de las clases anteriores, debe usarse la clase `Calendar`. La figura 8.12 muestra un ejemplo en que se aplican algunos de los métodos en la clase `Calendar`.

No reinvente la rueda.

La clase `Calendar` está en el paquete `java.util`. Para incluirla en su programa, puede usar esta sentencia importante:

```
import java.util.Calendar;
```

Sin embargo, puesto que la clase `Calendar` está en el mismo paquete que la clase `Scanner`, que también requiere este programa, es más fácil hacer que ambas clases estén disponibles a la vez con esta importante sentencia "comodín":

```
import java.util.*;
```

En la primera declaración, el programa carga `StdIn` con una referencia a una instancia de la clase `Scanner`. En la segunda declaración, el programa carga `Time` con una referencia a una instancia de la clase `Calendar`. No obstante, observe que el programa crea el objeto `Calendar` de manera extraña. Por una razón, que se explicará en el capítulo 13, no es posible usar simplemente `new Calendar ()` directamente. En lugar de ello, es necesario utilizar el método `getInstance`. Si este método se consulta en la documentación Java API para la clase `Calendar`, se verá que este método cuenta con un modificador `static`, por lo que es un método de clase. ¿Cómo se invoca un método de clase? Piense en retrospectiva la forma en que fueron invocados los métodos `Math-class` en el capítulo 5. En lugar de usar una variable de instancia antes del nombre del método, se usa el nombre de la clase. ¿Cómo funciona `getInstance`? Se supone que no se sabe, ya que es un módulo encapsulado, aunque probablemente instancia un objeto `Calendar`, lo inicializa en la hora actual y luego regresa una referencia a ese objeto. Aunque ésta no es la manera normal de instanciar objetos nuevos, funciona. El Java API incluye varios ejemplos de este tipo indirecto de construcción de programa.

Para el resto del programa, es posible olvidarse de cómo se creó el objeto `Time` y usarlo como se desee con cualquier otro objeto para llamar métodos de instancia en su propia clase. La primera sentencia usa el método `CalendargetTime` para recuperar la información sobre la hora, y luego imprime todo como se muestra en la primera línea de la sesión de muestra.

Las dos sentencias siguientes usan la referencia al objeto con métodos `get` a fin de recuperar dos valores particulares de variables de instancia. Pero ¡momento! Hay algo maravillosamente extraño sobre estos dos métodos `get`. No son métodos por separado como lo serían `getDayOfYear` y `getHour`. Ambos son el mismo método: uno de ellos denominado simplemente `get`. En lugar de usar el nombre del método para identificar la variable de instancia que será recuperada, los diseñadores de esta clase decidieron usar un valor de parámetro `int` para identificar esa variable. No es necesario saber cómo se implementó el método, ya que está encapsulado, aunque es posible hacer una conjetura plausible para arrojar luz sobre lo que hace. Por ejemplo, el parámetro `get` podría ser un

Use el número de identificación en el argumento para seleccionar una de muchas variables semejantes.

índice switch que conduce el flujo de control hacia un caso particular, donde hay un código que regresa el valor de la variable de instancia que corresponde a ese número de índice.

El problema al usar un número de índice para identificar una de muchas variables de instancia es que enteros simples no conllevan mucho significado. Sin embargo, se conoce una solución para este problema. Todo lo que debe hacerse es transformar cada uno de tales números de índice en una constante con nombre. Luego, para el argumento del método a distinguir, es necesario usar constantes con nombre en lugar de números. Ésta es la forma en que la clase Calendar implementa su método genérico get. Y resulta por lo menos tan fácil para un usuario recordar un método get con argumentos de constantes con nombres diferentes como recordar distintos nombres del método get.

```
/*********************************************************************
* CalendarDemo.java
* Dean & Dean
*
* Este programa ilustra el uso de la clase Calendar.
*********************************************************************/

import java.util.*;                  // para Scanner y Calendar

public class CalendarDemo
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Calendar time = Calendar.getInstance();     // inicialmente ahora
    int day;                                    // día del año
    int hour;                                   // hora del día

    System.out.println(time.getTime());
    day = time.get(time.DAY_OF_YEAR);
    hour = time.get(time.HOUR_OF_DAY);
    System.out.println("day of year= " + day);
    System.out.println("hour of day= " + hour);

    System.out.print("Enter number of days to add: ");
    day += stdIn.nextInt();
    System.out.print("Enter number of hours to add: ");
    hour += stdIn.nextInt();

    time.set(time.DAY_OF_YEAR, day);
    time.set(time.HOUR_OF_DAY, hour);
    System.out.println(time.getTime());
  } // end main
} // end class CalendarDemo
```

Los parámetros son códigos int que especifican el tipo de información deseada.

Sesión muestra:

```
Mon Sep 24 16:42:27 CDT 2007
day of year= 267
hour of day= 16
Enter number of days to add: 8
Enter number of hours to add: 13
Wed Oct 03 05:42:27 CDT 2007
```

Figura 8.12 Programa de demostración para la clase Calendar.

Armado con esta idea, ahora el lector debe poder visualizar lo que está haciendo el resto del código en nuestro programa `CalendarDemo`. Obtiene el día actual del año y la hora actual del día. Luego agrega un número de días introducido por el usuario al día actual y un número de horas introducido por el usuario a la hora actual. A continuación usa el método genérico `set` de `Calendar` (que probablemente funcione como el método genérico `set` de `Calendar`) para modificar las variables de instancia del objeto para el día del año y la hora. Por último, imprime la hora modificada.

La clase `Calendar` ilustra en forma elegante el valor de usar software escrito previamente. Realmente es más sencillo aprender a usar esta clase que escribir un programa que hace lo que hace. Además, el código de otras personas algunas veces ilustra técnicas que pueden ser aplicables al código que escribe el lector. Sin embargo, la clase `Calendar` también ilustra los tipos de penalizaciones asociadas con el uso de software escrito previamente. La penalización más elevada suele ser el tiempo dedicado a localizar e imaginar lo que está disponible. Otra penalización es que lo que se encuentra puede no corresponder exactamente a sus necesidades inmediatas, por lo que quizá sea necesario proporcionar código adicional para adaptar el software escrito previamente a su programa actual. Estas penalizaciones motivan a que muchos programadores digan "¡Hey! lo voy a escribir yo mismo". Algunas veces esto es lo que hay que hacer, aunque a largo plazo tomará la delantera si se da tiempo para aprender lo que otros ya han desarrollado.

8.13 Apartado GUI: resolución de problemas mediante tarjetas CRC (opcional)

Cuando se inicia un nuevo diseño, a menudo hay un periodo de rascarse la cabeza cuando se está intentando imaginar cuáles deben ser las clases y qué deben hacer. En la sección 8.6 se presentó una receta formal arriba-abajo, aunque algunas veces simplemente se requiere dar vueltas a algo o durante un tiempo tener reuniones sin orden del día donde los participantes hagan sugerencias para resolver uno o varios asuntos.

Explore sus opciones.

Inclusive cuando ocurre lo anterior, sigue siendo de utilidad escribir las cosas. A fin de contar con una estructura mínima para esta actividad informal, hace varios años los especialistas en computación Kent Beck y Ward Cunningham[8] sugirieron usar tarjetas para archivos pasadas de moda de 3" × 5", así como lápiz y goma de borrar. Su idea era asignar una tarjeta a cada clase propuesta, con tres tipos de información en cada tarjeta: 1) En la parte superior, escribir el nombre de una clase. 2) Abajo y a la izquierda, escribir una lista de frases verbales activas que describieran lo que debía realizar la clase. 3) Abajo y a la derecha, escribir una lista de otras clases con las que interactúa la clase actual: ya sea activamente como cliente o pasivamente como un servidor. El acrónimo CRC ayuda a recordar los tipos de información que debe contener cada tarjeta. La primera 'C' representa "Clase"; la 'R', "Responsabilidad"; y la última 'C', "Colaboración".

Cuando varias personas participan en una reunión sin orden del día para hacer sugerencias sobre cómo resolver uno o varios asuntos, quizás el mejor medio a utilizar sean lápices, gomas para borrar y unas cuantas tarjetas blancas. Pero cuando el diseñador es una sola persona podría ser más divertido utilizar ventanas pequeñas en la computadora. El programa que se presenta en la figura 8.13 establece tarjetas CRC simuladas en la pantalla de la computadora, de modo que simplemente puede hacerse lo anterior.

Este programa importa el paquete `javax.swing` a fin de contar con acceso a tres clases en el Java API: `JFrame`, `JTextArea` y `JSplitPane`. En un método `main`, reiteradamente solicita al usuario otra clase hasta que una entrada "q" indica que es tiempo de salir. Luego que el usuario introduce el nombre de cada clase, el programa instancia una pequeña ventana `JFrame` que representa una tarjeta CRC. El constructor `JFrame` inserta automáticamente el texto, "Class: <*classname*>" en el encabezado de esa ventana y, por tanto, implementa la primera "C" en CRC. Luego, el programa instancia dos "paneles" `JTextArea`, que actúan como pequeños cojinetes borrables, sobre los cuales es posible escribir cualquier texto en cualquier parte. El constructor de los dos `JTextArea` pide automáticamente que se escriba "RESPONSIBILITIES:" y "COLLABORATORS:", respectivamente, en la primera línea de cada

[8] Procedimientos de la Conferencia OOPSLA '89.

uno de estos dos paneles `JTextArea`. Luego, el programa instancia un `JSplitPane` con una especificación `HORIZONTAL_SPLIT` que divide la ventana en dos "aberturas" lado a lado separadas por una separación vertical móvil. Los dos últimos parámetros `JSplitPane` pegan los paneles individuales `JTextArea` en estas dos aberturas.

El método `setSize` solicita que los tamaños de la ventana semejen una tarjeta para archivo de 3" × 5". El método `add` solicita agregar a la ventana el panel separado. El método `setLocationByPlatform` indica a la computadora compensar cada tarjeta adicional de modo que sea posible continuar observando títulos y bordes de tarjetas creadas previamente a medida que se "apilan" en su escritorio. El método `setVisible` hace visible cada tarjeta nueva. El método `toFront` las mueve hacia el frente de la pantalla. El método `setDividerLocation` coloca al divisor `JSplitPane` a dos terceras partes a la derecha, a fin de proporcionar el doble de espacio para el texto "responsibilities", así como para el

```
/**************************************************************
* CRCCard.java
* Dean & Dean
*
* Este programa crea una nueva presentación de tarjetas CRC.
**************************************************************/

import java.util.Scanner;
import javax.swing.*;  // for JFrame, JTextArea, & JSplitFrame

public class CRCCard
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String input;

    System.out.print("Enter class name or 'q' to quit: ");
    input = stdIn.nextLine();
    while (!input.equalsIgnoreCase("q"))
    {
      JFrame frame = new JFrame("Class: " + input);
      JTextArea responsibilities =
        new JTextArea("RESPONSIBILITIES:\n");
      JTextArea collaborators =
        new JTextArea("COLLABORATORS:\n");
      JSplitPane splitPane =
        new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
        responsibilities, collaborators);

      frame.setSize(350, 210);
      frame.add(splitPane);
      frame.setLocationByPlatform(true);
      frame.setVisible(true);
      frame.toFront();
      splitPane.setDividerLocation(0.67);

      System.out.print("Enter class name or 'q' to quit: ");
      input = stdIn.nextLine();
    } // end while
  } // end main
} // end class CRCCard
```

Crea una nueva ventana.

Crea dos contenedores.

Coloca los contenedores en paneles por separado.

Coloca los paneles en ventanas.

Figura 8.13 Programa que coloca tarjetas interactivas CRC en la pantalla de la computadora.

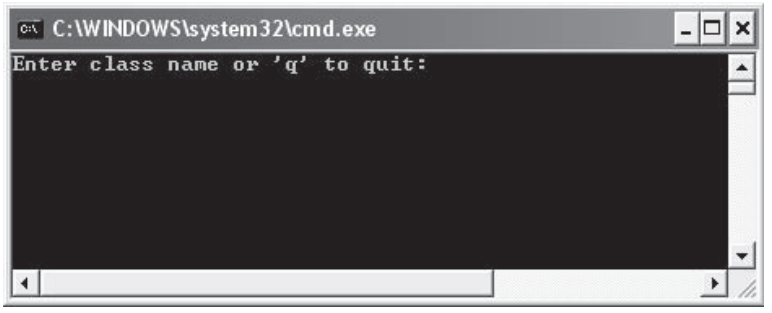


Figura 8.14 Presentación de un comando Prompt para el programa CRCCards.

texto "collaboration". Las dimensiones especificadas de la ventana y la ubicación del panel divisorio constituyen apenas escenarios iniciales, y si el lector encuentra que requiere de más espacio, podrá cambiarlo de manera interactiva en la pantalla de la computadora en cualquier instante mientras se ejecute el programa.

Cuando el programa se ejecuta en un entorno de Windows se obtiene una ventana de comando Prompt con una solicitud que pide el nombre de una clase, como la que se muestra en la figura 8.14.

Una vez que se ha introducido el nombre de una clase, aparece una ventana adicional. Ésta es la primera tarjeta CRC. Si la ventana de comando Prompt ahora se encuentra abajo de la nueva tarjeta, arrastre hacia abajo y a la derecha la ventana de comando Prompt para sacarla de en medio. Luego, mueva el cursor hacia la nueva tarjeta CRC y proporcione la información adicional, como la entrada "ejecutar el programa" en el panel RESPONSIBILITIES y en la entrada "GarageDoorSystem" en el panel COLLABORATORS en la figura 8.15.

Regrese al comando Prompt e introduzca otro nombre de clase y así sucesivamente hasta que haya creado todas las tarjetas CRC que necesite. Las tarjetas deben acomodarse de manera automática en una "pila" que parece algo así como las cuatro tarjetas en la parte superior izquierda de la pantalla de computadora que se muestra en la figura 8.16.

Luego, antes de introducir una "q" en la ventana de comando Prompt, es posible reducirla a un ícono y jugar con cuatro tarjetas CRC. Luego, arrastre lo anterior a cualquier parte de la pantalla a fin de formar jerarquías o agrupamientos lógicos.

En cualquiera de las tarjetas es posible modificar cualquiera de los textos en los dos paneles. Si el lector decide que una de las clases no es conveniente, puede hacer clic en su propio recuadro X a fin de eliminarla, reactivar la ventana de comando Prompt y crear más tarjetas CRC nuevas con nombres de clases distintos. Una vez que ha terminado es necesario usar Ctrl-PrtScr para imprimir la pantalla a fin de no olvidar las ideas del lector, y luego introducir "q" en la ventana de comando Prompt para terminar el programa.

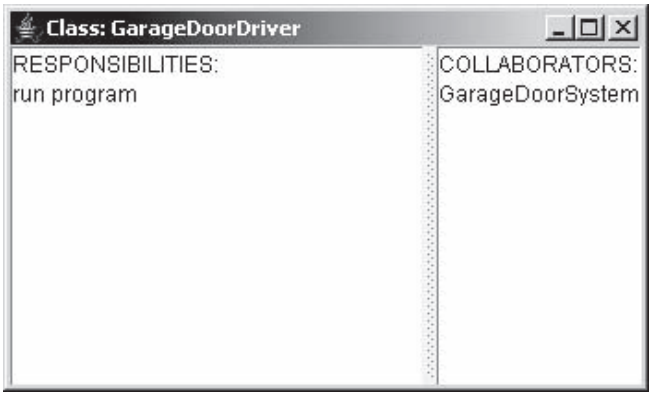


Figura 8.15 Primera tarjeta CRC después de las entradas del usuario.

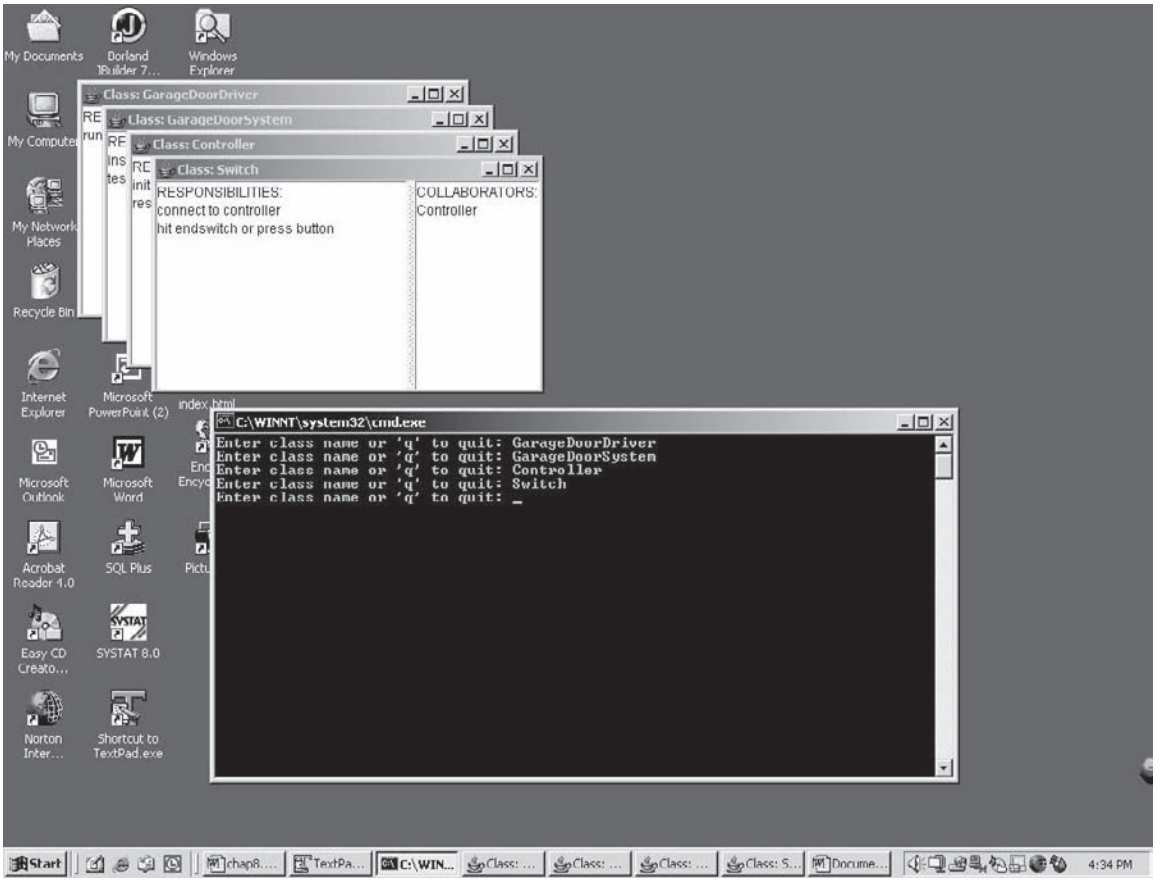


Figura 8.16 Lo que podría verse después de crear cuatro tarjetas CRC.

Resumen

- Inicie cada clase con un prólogo. Incluya el nombre del programa, del autor o los autores y una breve descripción de lo que hace la clase.
- Proporcione un comentario descriptivo arriba o después de cualquier código que algún programador experimentado de Java no pudiera comprender.
- Use nombres con sentido para todo. No sea críptico.
- Encierre entre paréntesis de llave bloques lógicos de código. La abertura y el cierre de las llaves deben estar en la misma columna como el inicio de la línea que precede al paréntesis de llave de apertura.
- Escriba un comentario final `// end` <*nombre del bloque*> después de cerrar paréntesis de llave para mejorar la legibilidad.
- Declare cada variable al inicio de su clase o método, o en el encabezado de su ciclo `for`. Normalmente, use una línea por variable y siga cada declaración con un comentario descriptivo idóneo.
- Use métodos de ayuda subordinados para simplificar métodos grandes y reducir redundancia en el código. Haga los métodos de ayuda `private` a fin de minimizar el abarrotamiento y la interfaz de la clase.
- Use variables de instancia sólo para atributos del objeto (información de estado). Use variables locales y parámetros de entrada para hacer cálculos dentro de un método y transferir datos a un método. Use valores de retorno y/o parámetros de referencia de entrada para transferir datos fuera de un método.
- Planifique probar a menudo y exhaustivamente el software que desarrolle a medida que avance. Incluya casos típicos, límite y absurdos.
- El diseño arriba-abajo es idóneo para proyectos grandes que cuentan con objetivos bien comprendidos. Proceda de lo general a lo específico, usando talones para posponer la implementación de métodos subordinados.

- El diseño ascendente permite asignar prioridades a los detalles críticos. Adopta la reutilización de software existente, con lo cual se reduce el costo de desarrollo y se mejora la confiabilidad del sistema. Sin embargo, esta metodología hace difíciles de administrar los proyectos grandes.
- Espere pasar por varias iteraciones de diseño. Use prototipos para ayudar a los consumidores a comprender mejor lo que quieren, evitando al mismo tiempo la trampa de intentar convertir un prototipo burdo directamente en un producto final. En cada iteración ulterior, seleccione la estrategia de diseño que atienda de la mejor manera la necesidad o el interés actual más importante. Un programa exitoso requiere mantenimiento constante, y usted puede facilitar esto si mantiene y mejora la elegancia a medida que el programa cambia y crece.
- Para facilitar las pruebas modulares, cuente con un método `main` en cada clase.
- En caso de no haber ambigüedad, puede omitir el prefijo `this` cuando acceda a un miembro de instancia.

Preguntas de revisión

§8.2 Convenciones de estilo de codificación

1. Es necesario evitar la inserción de líneas en blanco entre secciones diferentes del código (ya que hacerlo conduce a desperdicio de papel cuando se imprime el programa). (F/C)

2. Enumere en orden los siete puntos que se recomienda incluir en el prólogo de un archivo.

3. Cuando se agrega un comentario a una declaración de una variable el comentario siempre debe comenzar con un espacio después del final de la declaración. (F/C)

4. Para obtener lo máximo en cada línea del código, siempre es necesario romper una línea larga en el punto determinado por el editor de texto o IDE. (F/C)

5. Para un `if` o un `while` que tiene una sola sentencia en su cuerpo, los paréntesis de llave para el cuerpo son opcionales. El compilador no los requiere, pero un estilo depurado sugiere su inclusión. Proporcione al menos una razón de que es una buena idea escribir el cuerpo de una sentencia simple entre paréntesis de llave.

6. ¿Qué está mal con el estilo de descripción de clase que termina como se muestra a continuación?

```
      }
   }
}
```

¿Qué cambiaría el lector?

7. ¿Qué usaría el lector para separar grandes "paquetes" de código?

8. Escriba "sí" o "no" en cada uno de las siguientes afirmaciones para indicar si el hecho de incluir un espacio en blanco es un buen estilo o no.

- Después de los asteriscos simples en el prólogo.
- Entre la llamada a un método y su paréntesis de apertura.
- En el interior de cada uno de los tres componentes en el encabezado de un ciclo `for`.
- Después de dos punto y coma en el encabezado de un ciclo `for`.
- Entre un paréntesis de llave de cierre y los signos `//` de su comentario asociado.
- Después de los signos `//` de todos los comentarios.
- Después de las palabras clave `if`, `while` y `switch`.

§8.3 Métodos de ayuda

9. ¿Cuál de las siguientes razones es legítima para crear un método de ayuda?

a) Quiere que el método esté oculto para el mundo exterior.
b) Tiene un método largo y complicado, que desea partir en varios módulos más pequeños.
c) Su clase contiene dos o más métodos donde algo del código es el mismo en ambos métodos.
d) Todas las anteriores.

10. ¿La interfaz de una clase incluye los nombres de los métodos `private`?

§8.4 Encapsulamiento (con variables de instancia y variables locales)

11. En interés del encapsulamiento, use variables locales en lugar de variables de instancia siempre que sea posible. (F/C)

12. Si un método modifica una variable de instancia particular, y si un programa llama al método en dos ocasiones por separado, se garantiza que el valor de la variable de instancia al inicio de la segunda llamada al método tiene el mismo valor que tenía al final de la llamada del primer método. (F/C)

§8.5 Filosofía de diseño

13. Puesto que algo de su código preliminar podría cambiar en el transcurso del desarrollo, no desperdicie tiempo en realizar pruebas sino hasta que todo esté terminado. (F/C)

14. Cuando está probando un programa, es importante no estar prejuiciado sobre las expectativas de lo que podría obtenerse como resultado. (F/C)

§8.6 Diseño arriba-abajo

15. La metodología de diseño arriba-abajo es buena porque:

a) Mantiene a todo mundo enfocado en un objetivo común. (F/C)

b) Evita "reinventar la rueda". (F/C)

c) Mantiene informada a la gerencia. (F/C)

d) Minimiza las posibilidades de resolver un problema equivocado. (F/C)

e) Minimiza el costo global. (F/C)

f) Da por resultado el menor número de errores detectados. (F/C)

16. En un proceso de diseño arriba-abajo, ¿cuál sería su primera decisión? ¿Las clases o los métodos `public`?

§8.7 Diseño ascendente

17. ¿Cuándo debe usarse el diseño ascendente?

§8.9 Mejoramiento iterativo

18. Si un prototipo tiene éxito, ¿qué tentación debe resistir el lector?

19. Una vez que se elige una metodología de diseño particular, es necesario usarla a lo largo de todo el proceso de diseño, y no permitir que otras metodologías "contaminen" el proceso originalmente seleccionado. (F/V)

8.10 Método controlador de fusión en una clase controlada

20. El lector puede controlar cualquier clase desde un método `main` dentro de esa clase, y puede retener ese método `main` para realizar pruebas futuras de esa clase, inclusive cuando esa clase esté normalmente controlada desde otra clase en un programa más grande.

Ejercicios

1. [Después de §8.2]. Describa la manera de declarar variables que se ajustan a un buen estilo. Incluya la descripción de cuándo y cómo incluir comentarios asociados.

2. [Después de §8.2]. Corrija el estilo de la siguiente definición de clase.

```
/*Environment.java This class models the world's environment.
It was written by Dean & Dean and it compiles so it must be OK*/
public class Environment{//instance variables
private double sustainableProduction;private double
initialResources;private double currentResources;private
double yieldFactor = 2.0;public void setSustainableProduction
(double production){this.sustainableProduction = production;}
// Set pre-industrial mineral and fossil resources
public void setInitialResources(double resources){this.
initialResources=resources;}
// Initialize remaining mineral and fossil resources
public void setCurrentResources(double resources){this.
currentResources = resources;}
// Fetch remaining mineral and fossil resources
public double getCurrentResources(){return this.
currentResources;}/*Compute annual combination of renewable
and non-renewable environmental production*/public double
produce(double populationFraction,double extractionExpense){
double extraction;extraction=this.yieldFactor*
extractionExpense*(this.currentResources/this.
initialResources);this.currentResources-= extraction;return
extraction+populationFraction*this.sustainableProduction;}}
```

3. [Después de §8.3]. Dado el siguiente programa de diseño de una camiseta, que es el mismo que el programa Camiseta en las figuras 8.3 y 8.4, salvo por una ligera modificación en `main`:

```
/************************************************************
* ShirtDriver.java
* Dean & Dean
*
* This is a driver for the Shirt class.
************************************************************/

public class ShirtDriver
{
  public static void main(String[] args)
  {
    Shirt shirt1 = new Shirt();
    Shirt shirt2 = new Shirt();

    System.out.println();
    shirt1.display();
    shirt2.display();
  } // end main
} // end ShirtDriver
```

```
/************************************************************
* Shirt.java
* Dean & Dean
*
* This class stores and displays color choices for
* a sports-uniform shirt.
************************************************************/

import java.util.Scanner;

public class Shirt
{
  private String name;      // person's name
  private String primary;   // shirt's primary color
  private String trim;      // shirt's trim color

  //*********************************************************

  public Shirt()
  {
    Scanner stdIn = new Scanner(System.in);
    System.out.print("Enter person's name: ");
    this.name = stdIn.nextLine();

    this.primary = selectColor("primary");
    this.trim = selectColor("trim");
  } // end constructor

  //*********************************************************

  public void display()
  {
    System.out.println(this.name + "'s shirt:\n" +
      this.primary + " with " + this.trim + " trim");
  } // end display

  //*********************************************************

  // Helping method prompts for and inputs user's selection
```

```
40
41    private String selectColor(String colorType)
42    {
43      Scanner stdIn = new Scanner(System.in);
44      String color; // chosen color, first a letter, then word
45
46      do
47      {
48        System.out.print("Enter shirt's " + colorType +
49          " color (w, r, y): ");
50        color = stdIn.nextLine();
51      } while (!color.equals("w") && !color.equals("r") &&
52               !color.equals("y"));
53
54      switch (color.charAt(0))
55      {
56        case 'w':
57          color = "white";
58          break;
59        case 'r':
60          color = "red";
61          break;
62        case 'y':
63          color = "yellow";
64      } // end switch
65
66      return color;
67    } // end selectColor
68  } // end class Shirt
```

Rastree el programa de diseño de una camiseta usando la forma corta o la forma larga. Para ayudarlo a comenzar, a continuación se muestra la disposición del rastreo, incluyendo la entrada. Para la forma corta no es necesaria la columna línea #.

input

Corneal
r
w
Jill
w
y

ShirtDriver			Shirt												
	main			Shirt	display	selectColor			obj1			obj2			
line#	**sh1**	**sh2**	***line#***	**this**	**this**	**this**	**cType**	**color**	**name**	**prim**	**trim**	**name**	**prim**	**trim**	**output**

4. [Después de §8.3]. Suponga que la clase `GarageDoorSystem` en las figuras 7.25*a* y 7.25*b* tiene otra variable de instancia:

```
public final String SYSTEM_ID;
```

Vuelva a escribir el constructor `GarageDoorSystem` de modo que pueda llamar a un método de ayuda denominado `initialize`, que solicita al usuario proporcionar un nombre para `SYSTEM_ID`. En ese método, también pregunta al usuario si la posición inicial debe estar up. Luego, use la entrada del usuario para inicializar la variable `state`, suponiendo que los únicos estados iniciales posibles son down (0) o up (2).

5. [Después de §8.4]. Este ejercicio demuestra la utilización de un parámetro de referencia para pasar datos de vuelta al método que llama. Suponga que desea que una clase `Car5` incluya un método con este encabezado:

```
public boolean copyTo(Car5 newCar)
```

Se supone que este método debe ser llamado por un objeto existente `Car5` con un argumento a un nuevo objeto `Car5`. Si cualquiera de las variables de instancia `car` que llaman no ha sido inicializada, entonces el método deseado no debe intentar modificar ninguno de los valores de las variables de instancia `car`, y el

método debe regresar `false`. En caso contrario, el método debe copiar todos los valores de las variables de instancia `car` que llaman en un nuevo `car` y devolver `true`. A continuación se muestra un controlador que ilustra el uso:

```
/*************************************************************
* Car5Driver.java
* Dean & Dean
*
* This class is a demonstration driver for the Car5 class.
*************************************************************/

public class Car5Driver
{
  public static void main(String[] args)
  {
    Car5 annaCar = new Car5();
    Car5 nickCar = new Car5();

    System.out.println(annaCar.copyTo(nickCar));
    annaCar = new Car5("Porsche", 2006, "beige");
    System.out.println(annaCar.copyTo(nickCar));
  } // end main
} // end class Car5Driver
```

Output:

```
false
true
```

Escriba el código para el método `CopyTo` deseado.

6. [Después de §8.5]. Recomendamos aplicar pruebas a menudo, aun si esto significa crear un código de prueba especial que no se use en el programa final. ¿Por qué habría de ser útil ahorrarse este código de prueba especial?

7. [Después de §8.6]. En el supuesto de que será llamado por el método `draw` en la clase `Square` en la figura 8.7*a*, escriba un método `drawSolidSquare` que pida al usuario el carácter de impresión y dibuje todo el cuadrado sólido deseado por sí mismo, sin llamar a ningún método `drawHorizontalLine` por separado.

Sample session:

```
Enter width of desired square: 5
Area = 25
Print with (b)order or (s)olid? s
Enter character to use: #
#####
#####
#####
#####
#####
```

8. [Después de §8.6]. En el supuesto de que será llamado por el método `draw` en la clase `Square` en la figura 8.7*a*, escriba un método `drawBorderSquare` que pida al usuario dos caracteres para utilizar un cuadrado delimitado, un carácter para la línea fronteriza y otro carácter para el espacio de en medio. Observe que usar el mismo carácter para la línea fronteriza y el espacio de en medio hace que este método trace un cuadrado sólido, de modo que este método hace redundante al método `drawSolidSquare`, aunque este método requiere más interacción de parte del usuario.

Sample session:

```
Enter width of desired square: 5
Area = 25
Print with (b)order or (s)olid? b
Enter character for border: B
```

```
Enter character for middle: m
BBBBB
BmmmB
BmmmB
BmmmB
BBBBB
```

9. [Después de §8.6]. La figura 8.2*b* contiene dos condiciones de sentencias `if` que contienen lo que se conoce como expresiones regulares. Como se indica, éstas se encuentran explicadas en la clase `Pattern` del Java API. Se pretende que este ejercicio sea de ayuda para obtener un mejor conocimiento de las expresiones regulares Java y su uso. Utilice su propia documentación Java API en la clase `Pattern` para responder las siguientes preguntas:

a) ¿Cuál es el significado de la expresión regular `"[A-Z] [a-z]*"` que aparece en la figura 8.2*b*?

b) ¿Cuál es la expresión regular para una cadena de caracteres que empieza con una 'Z' y que contiene cualquier número de caracteres adicionales de cualquier tipo excepto un espacio o un tabulador?

c) ¿Cuál es la expresión regular para una cadena que representa un número telefónico de larga distancia en Estados Unidos (tres dígitos, una raya o un espacio, tres dígitos, una raya o un espacio y cuatro dígitos)?

10. [Después de §8.6]. Defina "refinamiento paso por paso".

11. [Después de §8.6]. Escriba talones para todos los constructores y métodos en la clase `Student` de las figuras 8.2*a* y 8.2*b*. Cada talón debe imprimir el nombre del método seguido por los valores iniciales (ya pasados) de todos los parámetros, como esta muestra de salida:

```
in Student
in setFirst, first= Adeeb
in setLast, last= Jarrah
in Student, first= Heejoo, last= Chun
in printFullName
```

12. [Después de §8.7]. Escriba un método genérico `drawRow` con este encabezado:

```
private void drawRow(int startCol, int endCol)
```

`startCol` y `endCol` son los números de columna de los límites izquierdo y derecho, respectivamente. Luego, modifique el método `draw` de la clase `Square` en la figura 8.7*a* para dibujar ya sea un cuadrado sólido o un triángulo sólido cuya altura y ancho sean iguales al ancho del ancho de entrada de un cuadrado contenedor. ¿Qué ocurre ahora con el área? ¿Es un valor redundante o es un atributo legítimo del objeto? Modifique en forma consecuente las variables de instancia y el método `getArea`. Luego controle su clase `Square` modificada (denomínela `Square2`) con un `Square2Driver` cuyo método `main` se parezca a:

```
public static void main(String[] args)
{
  Scanner stdIn = new Scanner(System.in);
  Square2 square;

  System.out.print("Enter width of square container: ");
  square = new Square2(stdIn.nextInt());
  square.draw();
  System.out.println("Area = " + square.getArea());
} // end main
```

Sample session:

```
Enter width of square container: 5
Print (s)quare or (t)riangle? t
*
**
***
****
*****
Area = 15
```

13. [Después de §8.6]. Escriba un prototipo del programa `Square`, usando una sola clase denominada `Square-Prototype` con un solo método, `main`. Escriba la cantidad mínima de código necesario para generar la

salida prescrita sólo para el caso más simple de un cuadrado sólido. La sesión de muestra debe verse exactamente como si fuese para el programa final descrito en las figuras 8.6, 8.9*a* y 8.9*b*, si el usuario selecciona la opción (s)ólido. No obstante, si el usuario selecciona la opción (b)orde, el prototipo debe responder imprimiendo "No implementada".

14. [Después de §8.9]. Cuando se diseña algo, es necesario seleccionar la metodología de diseño que sea la más conveniente para tratar el mayor interés del diseño actual. (F/V)

15. [Después de §8.10]. Escriba un programa controlador por separado que ejecute la clase `Time` mostrada en la figura 8.10 y fije la hora en 17 horas, 30 minutos y cero segundos. Suponga que el método `main` que aparece en la figura 8.10 aún sigue ahí.

16. [Después de §8.11]. Vuelva a escribir la clase `Car` en la figura 7.2 para eliminar el uso del `this`.

17. [Después de §8.12]. La clase Java API `Calendar` contiene un método denominado `getTimeMillisec` que permite recuperar la hora absoluta (en milisegundos) en que fue creado cualquier objeto `Calendar`. Como se indicó en la sección 8.12, este objeto puede obtenerse llamando al método de la clase `getInstance`. Esta capacidad puede usarse para evaluar el tiempo de ejecución de cualquier trozo del código. Todo lo que debe hacerse es crear un objeto `Calendar` antes del inicio de la prueba del código, crear otro objeto `Calendar` justo después que termine el código, e imprimir la diferencia en las horas de estos dos objetos. Para demostrar esta capacidad, escriba un programa corto denominado `TestRunTime` que pide al usuario un número de iteraciones deseadas, `num`. Luego, haga que mida el tiempo de ejecución para un bucle de iteraciones `num` que ejecute la sentencia simple:

```
Math.cos(0.01 * i);
```

La variable `i` es la variable de conteo del ciclo.

Soluciones a las preguntas de revisión

1. Falso. La legibilidad es un atributo importante de un buen código de computación. Para ahorrar papel de impresión, imprima en ambos lados de la página y/o use un tipo de fuente más pequeño.

2. Los siete elementos a incluir en el prólogo de un archivo son los siguientes:
- línea de asteriscos
- nombre de archivo
- nombre(s) del programador
- línea en blanco con un asterisco
- descripción
- línea de asteriscos
- línea en blanco

3. Falso. Así se obtendría espacio máximo para cada comentario, aunque los buenos programadores hacen los comentarios de inicio de declaración una línea tras otra, e intentan hacer los comentarios de la declaración lo suficientemente breves para evitar ajuste de líneas o líneas de recapitulación.

4. Falso. Asuma el control y rompa una nueva línea en el sitio más lógico o en los sitios más lógicos.

5. Aun cuando no es necesario, una buena idea consiste en escribir el cuerpo de una sentencia simple `if` o `while` entre paréntesis de llave.
- Los paréntesis de llave son un mecanismo visual para recordar el sangrado.
- Los paréntesis de llave son de ayuda para evitar errores lógicos en caso de que más tarde se agrande el código.

6. A menos que un bloque sea muy corto, puede no resultar demasiado evidente cuál bloque es terminado por un paréntesis de llave particular. Resulta una buena práctica terminar todos los bloques excepto los más cortos con un comentario; por ejemplo,

```
      } // end if
    } // end main
  } // end class Whatever
```

7. Para separar grandes "paquetes" de código, use líneas en blanco.

8. Sí significa incluir un espacio en blanco. No significa no hacerlo.
- Sí, después de los asteriscos simples en el prólogo.
- No, no entre la llamada a un método y su paréntesis de apertura.
- No, no en el interior de cada una de las tres componentes en el encabezado de un ciclo `for`.
- Sí, después de dos punto y coma en el encabezado de un ciclo `for`.
- Sí, entre un paréntesis de llave de cierre y los signos `//` de su comentario asociado.

- Sí, después de los signos `//` de todos los comentarios.
- Sí, después de las palabras clave `if`, `while` y `switch`.

9. ***d*)** Todas las anteriores.

10. No, la interfaz no describe los nombres de los métodos `private`.

11. Cierto. En general, en la medida de lo posible es necesario intentar mantener como locales las cosas, y una forma de lograr esto consiste en usar variables locales en lugar de variables de instancia. Las variables de instancia deben reservarse para atributos que describen el estado de un objeto.

12. Falso. Es cierto que una variable de instancia persiste a lo largo de la vida de un objeto, y si la segunda llamada al mismo método fuese justo después de la primera llamada a ese método, entonces el valor final de la variable de instancia en la primera llamada sería la misma que el valor inicial en la segunda llamada al método. Sin embargo, es posible que otro método pueda modificar el valor de la variable de instancia entre las dos llamadas al método en cuestión.

13. Falso. Es necesario realizar pruebas a lo largo de todo el proceso de desarrollo.

14. Falso. Es importante tener una idea clara de lo que se espera ver antes de realizar una prueba, de modo que tenga la mejor posibilidad de identificar discrepancias posibles una vez que ocurran.

15. La metodología de diseño arriba-abajo es la mejor porque:

***a*)** Cierto.

***b*)** Falso. Algunas veces obliga a las personas a "reinventar la rueda".

***c*)** Cierto.

***d*)** Falso. Si el lector está preocupado porque se encuentra resolviendo un problema equivocado, debe acudir a la realización de prototipos.

***e*)** Falso. Para minimizar el costo, el diseño debe organizarse a efectos de reutilizar componentes existentes.

***f*)** Falso. Para maximizar la confiabilidad, el diseño debe organizarse a efectos de reutilizar componentes existentes.

16. En el diseño arriba-abajo, primero se decide sobre las clases, y luego sobre los métodos `public`.

17. El diseño ascendente debe usarse cuando su programa pueda usar una cantidad importante de código escrito previamente, o cuando los detalles de bajo nivel son críticos y requieren atención inmediata.

18. Si un prototipo tiene éxito, resulta importante resistir la tentación de continuar el desarrollo al enredarse con ese prototipo.

19. Falso. Muchos problemas requieren beneficiarse de más de una metodología de diseño. Resulta una buena idea adoptar una metodología durante un ciclo de diseño (planificación, implementación, prueba y evaluación), aunque podría ser necesario cambiar a otra metodología en la siguiente iteración del ciclo.

20. Cierto. El método `main` particular usado es el actual cuando se inicia la ejecución.

CAPÍTULO 9

Clases con miembros de clase

Objetivos

- Aprender cómo y cuándo usar variables de clase.
- Aprender cómo escribir métodos de clase y cuándo usarlos.
- Aprender cómo y cuándo usar constantes de clase.
- Practicar algunos de los métodos de diseño sugeridos en el capítulo 8.
- Opcionalmente, aprender cómo elaborar una lista ligada de objetos y acceder a éstos a través de métodos clase.

Relación de temas

9.1 Introducción
9.2 Variables de clase
9.3 Métodos de clase
9.4 Constantes nombradas
9.5 Escritura de su propia clase Utility
9.6 Utilización de miembros de clase en conjunción con miembros de instancia
9.7 Resolución de problemas con miembros de clase y miembros de instancia en una clase de listas ligadas (opcional)

9.1 Introducción

Cuando el lector piensa en una solución orientada a objetos, ¿qué se imagina? Con base en lo que ha aprendido hasta ahora, debe ver objetos por separado, cada uno con su propio conjunto de datos y comportamientos (variables de instancia y métodos de instancia, respectivamente). Esta imagen es válida, aunque el lector debe estar al corriente de que además de datos y comportamientos, que son específicos de objetos individuales, también es posible tener datos y comportamientos que relacionan toda una clase. Puesto que estos datos y comportamientos se relacionan con toda una clase, se denominan *variables de clase* y *métodos de clase*, respectivamente.

Se considerará un ejemplo. Suponga que usted es responsable de mantener la pista de los videos YouTube. Necesita instanciar un objeto YouTube para cada objeto YouTube, y dentro de cada objeto debe almacenar atributos como el filmador, la duración del video y el archivo mismo del video. Debe almacenar estos atributos en variables de instancia porque están asociados con objetos YouTube individuales. También, usted debe almacenar atributos como el número de videos y el video más popular. Debe almacenar estos atributos en variables de clase porque están relacionados con la colección de objetos YouTube como un todo.

Como otro ejemplo, piense en la clase `Math`. Sus miembros, como `Math.round` y `Math.PI`, son miembros de clase porque están asociados con la clase `Math` como un todo. En el capítulo 5 aprendió cómo acceder a los miembros de la clase `Math` y usarlos. En este capítulo, usted aprenderá a implementar sus propios miembros de clase.

Este capítulo empieza mostrándole cómo implementar sus propias variables de clase. Luego se mostrará cómo implementar métodos de clase, y las variables de clase se usarán dentro de estos métodos. A continuación se abordarán las constantes de clase, que son variables de clase que usan el modificador `final`. Más adelante se presenta una *clase de utilidad*: una clase con funcionalidad de objetivo general que otras clases pueden usar fácilmente. En la parte final del capítulo, el lector verá juntos diferentes miembros de instancia y miembros de clase en dos programas completos. El segundo de estos programas es más que un simple ejemplo con miembros de instancia y de clase. Implementa una importante estructura de datos denominada *lista ligada*, que permite crear de manera dinámica un número arbitrariamente grande de objetos ligados entre sí.

9.2 Variables de clase

El lector ya sabe que las variables de clase están asociadas a una clase como un todo. En esta sección, el lector aprenderá más sobre las variables de clase, como la forma de declararlas, cuándo usarlas, cuáles son sus valores por defecto y cuál es su alcance. En la siguiente sección se presentan ejemplos de cómo utilizar variables de clase desde métodos de clase.

Sintaxis para la declaración de variables de clase

Para que una variable sea una variable de clase, utilice el modificador `static` en su declaración. Este modificador es la razón de que muchos programadores usen el término "variable estática" cuando se refieren a variables de clase. En forma semejante, puesto que las constantes de clase y los métodos de clase también usan el modificador `static`, muchos programadores usan los términos constante estática y método estático. Aquí se usan los términos variable de clase, constante de clase y método de clase puesto que son los que se usan en Sun.

A continuación se muestra la sintaxis para una sentencia de declaración de una variable de clase:

<*private-or-public*> `static` <*type*> <*variable-name*>`;`

Y he aquí un ejemplo:

```
private static int mouseCount;  // total number of mouse objects
```

Las variables de clase, ¿deben ser `públicas` o `privadas`? La filosofía en esta cuestión es la misma que en las variables de instancia. Puesto que siempre es posible escribir/obtener métodos de clase `públicos`, no se requiere ninguna variable de clase `pública` más de lo que se requieren variables de instancia `públicas`. Es mejor preservar las variables como `privadas` en la medida de lo posible a fin de tener control sobre la forma en que es posible acceder a ellas. En consecuencia, además de hacer `privadas` las variables de instancia, también debe hacer `privadas` las variables de clase.

¿Por qué el término "estáticas"?

Como sabe el lector, cuando la máquina virtual Java (Java Virtual Machine: JVM) detecta el nuevo operador en un programa, instancia un objeto para la clase especificada. Al hacerlo, asigna espacio de la memoria a todas las variables de instancia del objeto. Luego, el recolector de basura puede liberar (retirar) ese espacio de la memoria antes que el programa se detenga si todas las referencias a ese espacio desaparecen. Este tipo de administración de la memoria, realizado durante la ejecución del programa, se denomina *asignación dinámica*. Las variables de clase son diferentes. La JVM asigna espacio a una variable de clase cuando se inicia el programa, y este espacio de la variable de clase permanece asignado en tanto dura la ejecución del programa. Este tipo de administración de la memoria, realizado durante la ejecución del programa, se denomina asignación estática. Ésta es la razón de que las variables de clase se denominen `estáticas`.

Ejemplos de variables de clase

Como sabe el lector, cada uso de `new` crea una copia por separado de todas las variables de instancia para cada objeto. Las variables de clase son diferentes. Para una clase particular, sólo hay una copia de cada variable de clase, y todos los objetos comparten esta simple copia. Así, es necesario usar variables de clase para describir propiedades de los objetos de una clase que deben compartir todos los objetos. Por ejemplo, considere otra vez el problema de simular el crecimiento de un ratón. En los programas ratón

previos, se siguió la pista de los datos pertinentes a cada ratón individual: una tasa de crecimiento del ratón, una edad del ratón y un peso del ratón. Para un programa de simulación más útil, quizá sea conveniente seguir la pista de datos del grupo y de datos de entorno comunes. Por ejemplo:

`mouseCount` podría seguir la pista del número total de ratones.
`youngestMouse` podría seguir la pista del último ratón que ha nacido.
`averageLifeSpan` podría seguir la pista de la duración media de la vida de todos los ratones.
`simulationDuration` podría limitar el número de iteraciones de simulación.
`researcher` podría identificar a una persona responsable de un experimento realizado en el grupo de ratones.
`noiseOn` podría indicar la presencia o ausencia de un ruido estresante escuchado por todos los ratones.

Si el lector usó variables de instancia para `mouseCount`, `averageLifeSpan`, etc., entonces cada objeto ratón individual debe contar con su propia copia de los datos. Entonces, si en total hay cien ratones, cada uno de estos cien ratones debe almacenar el valor 100 en su propia variable `mouseCount`, el valor medio de la duración de la vida en su propia variable `averageLifeSpan` y así sucesivamente. Esto significa que cada vez que un nuevo ratón nace, muere o envejece un año, es necesario actualizar 100 copias por separado de `mouseCount`, `averageLifeSpan`, etc., todas con exactamente la misma información. ¡Qué desperdicio de esfuerzo! ¿Por qué no hacer lo anterior una sola vez y dejar que cada quien escriba y lea los mismos datos comunes? Si `mouseCount`, `averageLifeSpan`, etc., son variables de clase, todos los objetos ratón pueden escribirse en y leerse desde un solo registro para cada una de estas piezas de información. Un forastero puede acceder a estas propiedades de clase simplemente al escribir como prefijo el nombre de la clase en un método de clase idóneo. No es necesario ni aconsejable recorrer toda una instancia particular para llegar a este tipo de información.

Las declaraciones de las variables de clase en la clase `Mouse` mejorada se vería como el código en la figura 9.1. En la figura, ¿sorprende al lector que el tipo de `youngestMouse` sea el nombre de la clase en la que está definido? ¿Esto significa que hay un ratón dentro de un ratón? ¡No! El modificador `static` en la declaración `youngestMouse` significa que `youngestMouse` es una variable de clase. Como tal, es una propiedad de la colección de todos los ratones. Más específicamente, identifica el objeto ratón que se ha instanciado más recientemente. En la siguiente sección, `youngestMouse` se presenta en el contexto de un programa completo, y el lector verá cómo se actualiza cada vez que hay una instanciación de un objeto ratón.

Valores por defecto

Las variables de clase usan los mismos valores por defecto que las variables de instancia:

Tipo de variable de clase	Valor por defecto
entera	0
de punto flotante	0.0
`booleana`	`falso`
de referencia	`nulo`

```
public class Mouse
{
  private static int mouseCount;
  private static Mouse4 youngestMouse;
  private static double averageLifeSpan = 18;  // meses
  private static int simulationDuration = 730; // días
  private static String researcher;
  private static boolean noiseOn;
  ...
```

Se permiten las inicializaciones.

atributos del entorno

Figura 9.1 Declaraciones de variables de clase en una clase `Mouse` mejorada.

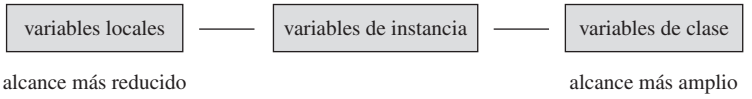
Se concluye que los valores por defecto para las variables de clase de la figura 9.1 son:

```
mouseCount = 0
youngestMouse = null
averageLifeSpan = 0.0
simulationDuration = 0
researcher = null
noiseOn = false
```

Presumiblemente, durante su ejecución, el programa actualiza `mouseCount`, `youngestMouse` y `averageLifeSpan`. Los valores por defecto de `averageLifeSpan` y `simulationDuration` son cero, así como el de `mouseCount`, aunque en la figura 9.1 los valores por defecto no son válidos porque las declaraciones incluyen inicializaciones. Aun cuando sea de esperar que el programa vuelva a calcular `averageLifeSpan`, ésta se inicializa para contar con información sobre lo que se considera un valor razonable. También se inicializa `simulationDuration` (a 730), inclusive si se espera que el programa reasigne `simulationDuration` con un valor introducido por el usuario. Quizás el programa pide al usuario que introduzca el número de días a simular. Con el código idóneo, el usuario podría recibir la invitación para introducir −1 para obtener una simulación "estándar" de 730 días.

Alcance

A continuación se compararán las variables de clase, las variables de instancia y las variables locales en términos de sus alcances. Es posible acceder a una variable de clase desde cualquier sitio dentro de su clase. Más específicamente, lo anterior significa que es posible acceder a las variables de clase desde métodos de instancia, así como desde métodos de clase. Esto contrasta con las variables de instancia, a las que sólo es posible acceder desde métodos de instancia. Así, las variables de clase tienen un alcance más amplio que las variables de instancia. Por otra parte, las variables locales tienen un alcance más reducido que las variables de instancia. A éstas sólo es posible acceder dentro de un método particular. Éste es el continuo del alcance:



El hecho de tener un alcance más reducido para variables locales podría parecer algo malo porque es menos "poderoso", aunque en realidad es algo bueno. ¿Por qué? Un alcance más reducido es igual a más encapsulamiento, y como se aprendió en el capítulo 6, el encapsulamiento significa que se es menos vulnerable a modificaciones inapropiadas. Por otra parte, a las variables de clase, con su alcance amplio y falta de encapsulamiento, es posible acceder y actualizarlas desde muchos sitios diferentes, lo cual hace que los programas sean difíciles de comprender y depurar. Tener un alcance amplio es necesario algunas veces, aunque en general debe intentar evitar un alcance más amplio. Se sugiere al lector que prefiera variables locales en lugar de variables de instancia, y variables de instancia en lugar de variables de clase.

9.3 Métodos de clase

Los métodos de clase, como las variables de clase, se relacionan con la clase como un todo, y no se relacionan con objetos individuales. Como tales, si el lector necesita realizar una tarea que implique la clase como un todo, entonces debe implementar y usar un método de clase. En el capítulo 5 se usaron métodos de clase definidos en la clase API `Math` de Java; por ejemplo, `Math.round` y `Math.sqrt`. Ahora, el lector aprenderá a escribir sus propios métodos de clase. Los métodos de clase a menudo acceden a variables de clase, y al escribir sus propios métodos de clase, el lector obtiene una oportunidad de ver cómo acceder a variables de clase que ha definido.

Método de clase Syntax

Vea la clase `Mouse4` en la figura 9.2. En particular, observe el método `printMouseCount`. Trata con información de la clase a escala, de modo que resulta idóneo hacerlo un método de clase. Más específica-

mente, imprime el valor de `mouseCount`, donde `mouseCount` es una variable de clase que sigue la pista del número total de objetos ratón.

Para declarar un método de clase, use la siguiente sintaxis para el inicio del método:

<*private-or-public*> `static` <*return-type*> <*method-name*>(<*parameters*>)

Observe cómo el método `printMouseCount` en la figura 9.2 obedece este patrón sintáctico.

Normalmente, para acceder a un miembro de clase es necesario escribir como prefijo el miembro de clase con el nombre de la clase del miembro de clase y luego un punto. Por ejemplo, dentro del método `printMouseCount` y el constructor `Mouse4`, observe cómo es posible acceder a `mouseCount` y `youngestMouse` con prefijos `Mouse4` punto: `Mouse4.mouseCount` y `Mouse4.youngestMouse`. También, dentro del método `main` observe cómo el método de clase `printMouseCount` es llamado con `Mouse4.printMouseCount( )`. Prefijar un miembro de clase con el nombre de su clase y luego un punto debe parecer conocido. El lector ya hizo esto con miembros de la clase `Math` desde hace tiempo (por ejemplo, `Math.round( )`, `Math.PI`).

No olvide que cuando se accede a un miembro de clase no siempre es obligatorio usar el prefijo punto del nombre de la clase. Al acceder a un miembro de clase, es posible omitir el prefijo punto del nombre de la clase si el miembro de clase está en la misma clase que la clase a partir de la cual se está intentando el acceso. Así, en la clase `Mouse4`, puesto que todos los accesos a los miembros de clase y los miembros de clase mismos están en la misma clase `Mouse4`, es posible omitir todos los prefijos punto del nombre de la clase. Pero si el programa estuviese escrito de modo que `main` aparece en una clase controladora por separado, entonces no sería posible omitir la clase `Mouse4` punto desde la llamada `main` a `printMouseCount`.

```
public class Mouse4
{
  private static int mouseCount;
  private static Mouse4 youngestMouse;
  private int age;

  public Mouse4()
  {
    Mouse4.mouseCount++;
    Mouse4.youngestMouse = this;
  }

  public static void printMouseCount()
  {
    System.out.println("Total mice = " + Mouse4.mouseCount);
  }

  public void olderByOneDay()
  {
    this.age++;
  }

  //*******************************************************

  public static void main(String[] args)
  {
    Mouse4 pinky = new Mouse4();
    pinky.olderByOneDay();
    Mouse4.printMouseCount();
  }
} // end class Mouse4
```

variables de clase

especifica un método de clase

Normalmente, para acceder a una variable de clase, es necesario escribir el prefijo <*class-name*> dot.

Normalmente, para acceder a un método de clase, es necesario escribir el prefijo <*class-name*> dot.

Figura 9.2 Programa mouse simple que ilustra ideas sobre miembros de clase.

Aunque a menudo es legal omitir el prefijo punto del nombre de la clase, aquí se tiene una ligera preferencia por incluirlo siempre porque constituye una forma de autodocumentación. Alerta a quien lee el código sobre el hecho de que el miembro al que se ha accedido es especial, trata con información de la clase a escala.

Llamando a un método de instancia desde un método de clase

Si se está en un método de clase, se comete un error de compilación si se intenta acceder directamente a un miembro de instancia. Para acceder a un miembro de instancia, primero debe tenerse un objeto, y luego acceder al miembro de instancia del objeto al anteponerlo (*prefacing*) a la variable de referencia del objeto. A menudo, la variable de referencia se denomina objeto que llama. ¿Le suena conocido? El método `main` es un método de clase (el encabezado de `main` incluye al modificador `static`), y ya desde algún tiempo el lector ha estado llamando métodos de instancia desde `main`. Sin embargo, siempre que se haga lo anterior, primero debe instanciarse un objeto y asignar la referencia del objeto a una variable de referencia. Luego se llama un método de instancia al prefijarlo con la variable de referencia y un punto. El método `main` en la figura 9.2 muestra de qué se está hablando:

```
public static void main(String[] args)
{
  Mouse4 pinky = new Mouse4();
  pinky.olderByOneDay();
  Mouse4.printMouseCount();
}
```

El prefijo punto de la variable de referencia es necesario cuando un método de instancia se llama desde un método de clase.

Si se intenta acceder directamente un método de instancia directamente desde un método de clase, aparece un mensaje de error como éste:

`Ningún <`*nombre del método*`> no estático puede referirse desde un contexto static`

Este mensaje de error es muy común (quizás el lector ya lo ha visto bastantes veces) porque es fácil olvidar prefijar las llamadas de los métodos de instancia con una variable de referencia. Cuando los programadores veteranos ven este mensaje, saben qué hacer; se aseguran de prefijar la llamada del método de instancia con una variable de referencia del objeto que llama. Pero cuando programadores novatos ven el mensaje de error, a menudo agravan el error al intentar "arreglar" de manera inadecuada su causa.

Más específicamente, cuando se confronta con el mensaje de error del método no estático, un programador novato a menudo cambia el método de instancia ofensivo por un método de clase al insertar `static` al inicio del método. (En el programa `Mouse4`, `olderByOneDay` podría cambiarse a un método de clase.) Luego, obtiene el mensaje de error del miembro no estático para cualquier variable de instancia dentro del método. Después, agravan todavía más el problema al cambiar las variables de instancia del método a variables de clase. (En el programa Mouse4, la variable edad de `olderByOneDay` podría cambiarse a una variable de clase.) Una vez hecho ese cambio, el programa compila exitosamente y el programador novato está feliz como gallo, listo para enfrentar al siguiente dragón. Desafortunadamente, este tipo de solución conduce a un problema más grave que un error de compilación. Lleva a un error lógico.

Como sabe el lector, si un miembro de clase se relaciona con un objeto en lugar de hacerlo con la clase como un todo, es necesario hacerlo un miembro de instancia. Si hace lo anterior como se ha descrito, y "repara" un desperfecto al cambiar un miembro de instancia a un miembro de clase, puede lograr que su programa compile y funcione. Y si tiene sólo un objeto, inclusive tal vez su programa produzca un resultado válido. Pero si tiene más de un objeto, ahora o en el futuro, entonces con variables de clase, los objetos compartirán los mismos datos. Si se modifican los datos de un objeto, simultáneamente se cambian los datos de todos los demás objetos, lo cual normalmente es incorrecto.

Al margen: acceso a un miembro de clase desde un método de instancia o un constructor

Aunque no es posible acceder directamente a un miembro de instancia desde un método de clase, sí es posible acceder a un miembro de clase desde un método de instancia. Además, es posible acceder a una

variable de clase desde un constructor, lo cual se ilustra en la figura 9.2. El código relevante se repite a continuación para su conveniencia. Muestra cómo las variables de clase `mouseCount` y `youngestMouse` se actualizan automáticamente con cada nueva instanciación. Observe cómo la referencia del `this` asigna el ratón más recientemente instanciado del constructor a la variable `youngestMouse`.

```
public Mouse4()
{
  Mouse4.mouseCount++;
  Mouse4.youngestMouse = this;
}
```

Cuándo utilizar métodos de clase

¿Cuándo un método debe hacerse un método de clase? La respuesta general es "cuando sea necesario realizar una tarea que implique a la clase como un todo". Pero es necesario ser más específico. A continuación se presentan situaciones en las que los métodos de clase son idóneos:

1. Si se tiene un método que usa variables de clase y/o llama a métodos de clase, entonces es un buen candidato a ser un método de clase. Por ejemplo, `printMouseCount` en la figura 9.2 es un método de clase porque imprime la variable de clase `mouseCount`. Advertencia: si además de acceder a miembros de clase el método también accede a miembros de instancia, entonces el método debe ser un método de instancia, no uno de clase.
2. Si es necesario llamar a un método inclusive cuando no haya objetos de la clase del método, entonces es necesario convertirlo en un método de clase. Por ejemplo, durante una simulación de población de ratones, podría llamarse a `printMouseCount` cuando no hay objetos ratón (tal vez todos han muerto). Puesto que se trata de un método de clase, se hace como sigue, sin necesidad de un objeto que llama:

   ```
   Mouse4.printMouseCount();
   ```

3. El método `main` es el punto de partida para todos los programas y, como tal, se ejecuta antes de la instanciación de cualquier objeto. Para dar paso a esta funcionalidad, es necesario hacer del método `main` un método de clase. Si el método `main` es más bien largo y se decide dividirlo con métodos de ayuda, entonces estos métodos (en el supuesto de que no implican miembros de instancia) también deben ser métodos de clase. Al hacerlos métodos de clase, es más fácil que `main` los llame.
4. Si se tiene un método de propósito general autosuficiente, debe transformarse en un método de clase. Por autosuficiente se entiende que el método no está relacionado con ningún objeto particular. Estos métodos se denominan *métodos de utilidad*. El lector ya conoce ejemplos de estos métodos, como `Math.round` y `Math.sqrt` en la clase `Math`. En la sección 9.5, el lector aprenderá a escribir sus propios métodos de utilidad.

9.4 Constantes nombradas

El uso de nombres en lugar de valores difíciles de codificar hace que un programa sea más de código libre. Cuando un valor constante se requiere en más de un sitio en el bloque del código, establecer el valor en un lugar al inicio de ese bloque minimiza la posibilidad de inconsistencias. En Java, es posible definir constantes nombradas en varios niveles de escala.

Constantes locales nombradas: repaso del capítulo 3

En el nivel más microscópico, es posible definir constantes locales nombradas. Volviendo a la figura 3.5 del capítulo 3, se definieron dos constantes locales nombradas: `FREEZING_POINT` y `CONVERSION_FACTOR`, para autodocumentar la fórmula de conversión de grados Fahrenheit a grados Celsius en un programa simple que no hacía nada más que una conversión de temperaturas. En términos generales, este tipo de actividad se incrusta en algún programa más grande al colocarlo en algún método de ayuda como se muestra a continuación:

```
private double fahrenheitToCelsius(double fahrenheit)
{
  final double FREEZING_POINT = 32.0;
  final double CONVERSION_FACTOR = 5.0 / 9.0;

  return CONVERSION _ FACTOR * (fahrenheit - FREEZING_POINT);
} // end fahrenheitToCelsius
```

Las constantes locales nombradas en este método facilitan la comprensión del código.

Constantes de instancia nombradas: repaso del capítulo 7

En el siguiente nivel superior de escala, algunas veces se desea una constante que sea una propiedad permanente de un objeto y accesible a todos los métodos de instancia asociados con ese objeto. Estas constantes se denominan constantes de instancia nombradas o, simplemente, *constantes de instancia*. A continuación se presenta un ejemplo de declaración de constante de instancia que identifica una propiedad permanente de un objeto `Person`:

```
public final String SOCIAL_SECURITY_NUMBER;
```

Una declaración de constante de instancia difiere de una declaración de constante local nombrada en tres formas: 1) una declaración de constante de instancia debe aparecer en la parte superior de la definición de clase, y no dentro de un método; 2) una declaración de constante de instancia está precedida por un modificador de acceso `public` o `private` y 3) aunque es legal inicializar una constante de instancia en una declaración, es más común inicializarla en un constructor.

Constantes de clase nombradas

En el siguiente nivel superior de escala, algunas veces se desea una constante que sea la misma para todos los objetos en una clase. En otras palabras, se quiere algo que sea como una variable de clase, pero que sea constante. Estas constantes se denominan constantes de clase nombradas o, simplemente, *constantes de clase*. En el capítulo 5 el lector aprendió acerca de dos constantes de clase definidas por la clase `Math` API, `PI` y `E` de Java. Ahora aprenderá cómo escribir sus propias constantes de clase. Para declarar una constante de clase, use esta sintaxis:

<*private-or-public*> `static final` <*type*> <*variable-name*> `=` <*initial-value*>`;`

Una declaración de constante de clase difiere de una declaración de constante de instancia en dos formas: 1) una constante de clase incluye el modificador `static`; 2) una constante de clase debe inicializarse como parte de su declaración.[1] Si más tarde se intenta asignar un valor a una constante de clase, se genera un error de compilación.

Así como ocurre en una constante de instancia, una declaración de constante de clase debe ir precedida por un modificador de acceso `public` o `private`. Si la constante es necesaria sólo dentro de la clase (y no fuera de ésta), debe hacerse `private`. Lo anterior permite modificar la constante sin molestar a nadie que previamente haya elegido usar la constante del lector en uno de sus programas. No obstante, si se desea que la constante esté a disposición de otras clases, resulta conveniente hacerla `public`. Es seguro hacer lo anterior porque el modificador `final` la vuelve inmutable (inmodificable). En la siguiente sección se presentan ejemplos de constantes de clase `public` incrustadas en una clase de utilidad.

La siguiente clase `Human` contiene una constante nombrada `NORMAL_TEMP`. Se hace una constante de clase (con los modificadores `static` y `final`) porque todos los objetos `Human` tienen la misma temperatura normal de 36.4°C. Se hace una constante de clase `private` porque es necesaria sólo dentro de la clase `Human`.

```
public class Human
{
```

[1] Aunque es relativamente raro, es legal declarar una constante de clase como parte de un bloque inicializador estático. Respecto a bloques inicializadores, consulte el sitio http://java.sun.com/docs/books/tutorial/java/javaOO/initial.html

```
  private static final double NORMAL_TEMP = 98.6;
  private double currentTemp;
  ...
  public boolean isHealthy()
  {
    return Math.abs(currentTemp - NORMAL_TEMP) < 1;
  } // end isHealthy

  public void diagnose()
  {
    if ((currentTemp - NORMAL_TEMP) > 5)
    {
      System.out.println("Go to the emergency room now!");
      ...
} // end class Human
```

A continuación se hará un resumen de cuándo es necesario utilizar los tres tipos diferentes de constantes nombradas. Use una constante local nombrada si la constante es necesaria sólo dentro de un método. Use una constante de instancia si la constante describe una propiedad permanente de un objeto. Use una constante de clase si la constante es una propiedad de la colección de todos los objetos en la clase o de la clase en general.

Posiciones de las declaraciones

A continuación se abordan algunas cuestiones sobre estilo de codificación. Se recomienda colocar juntas todas las declaraciones de constantes de clase, arriba de todas las declaraciones de constantes de instancia. Colocar las declaraciones en la parte superior las destaca más, y es idóneo que las constantes de clase destaquen lo más posible, puesto que tienen el alcance más amplio. En forma semejante, se recomienda colocar todas las declaraciones de variables de clase arriba de todas las declaraciones de variables de instancia. A continuación se proporciona la secuencia preferida de declaraciones dentro de una clase dada:

constantes de clase
constantes de instancia
variables de clase
variables de instancia
constructores
métodos

9.5 Escritura de su propia clase Utility

Hasta el momento se han implementado métodos que resuelven problemas para una clase particular. Suponga que se desea implementar métodos que sean más de propósito general, de modo que clases múltiples e imprevistas puedan usarlos. Estos tipos de métodos se denominan *métodos utility*. En el pasado, ya se han usado métodos de la clase `Math`; por ejemplo, `Math.round` y `Math.sqrt`. En esta sección, el lector aprenderá a escribir sus propios métodos de utilidad como parte de una clase utility.

Vea la clase `PrintUtilities` en la figura 9.3. Contiene constantes y métodos de utilidad orientados a impresoras. Las dos constantes, `MAX_COL` y `MAX_ROW` siguen la pista de la columna máxima y del renglón máximo para una hoja de papel de tamaño estándar. Si se cuenta con múltiples clases que imprimen reportes, estas constantes pueden ser de ayuda para asegurar uniformidad en el tamaño del reporte. El método `printCentered` imprime una cadena dada centrada horizontalmente. El método `printUnderlined` imprime una cadena dada con rayas por abajo de ella. Estos métodos se colocan en una clase utility porque ejecutan rutinas de impresión que pueden ser necesarias para muchas otras clases.

En la clase `PrintUtilities` observe que todas las constantes y todos los métodos usan los modificadores `public` y `static`. Esto es normal para miembros de clases de utilidad. Los modificadores `public` y `static` facilitan el acceso de otras clases a miembros de `PrintUtilities`.

```
/*******************************************************************
* PrintUtilities.java
* Dean & Dean
*
* Esta clase contiene constantes y métodos para una impresión sofisticada.
*******************************************************************/

public class PrintUtilities
{
  public static final int MAX_COL = 80; // last allowed column
  public static final int MAX_ROW = 50; // last allowed row

  //*****************************************************************

  // Print given string horizontally centered.

  public static void printCentered(String s)
  {
    int startingCol; // starting point for string
    startingCol = (MAX_COL / 2) - (s.length() / 2);

    for (int i=0; i<startingCol; i++)
    {
      System.out.print(" ");
    }
    System.out.println(s);
  } // end printCentered

  //*****************************************************************

  // Print given string with dashes underneath it.

  public static void printUnderlined(String s)
  {
    System.out.println(s);
    for (int i=0; i<s.length(); i++)
    {
      System.out.print("-");
    }
  } // end printUnderlined
} // end class PrintUtilities
```

Figura 9.3 Ejemplo de clase de utilidad que maneja la impresión con necesidades especiales.

9.6 Utilización de miembros de clase en conjunción con miembros de instancia

Ahora se considerará un problema que requiere una combinación de miembros de instancia y miembros de clase. El objetivo consiste en modelar una colección de alcancías de un centavo. Con cada inserción de una moneda en cualquier alcancía, se quiere que el programa imprima "clink" e incremente la cuenta de monedas para esa alcancía y el conteo total de monedas. Cuando el número total de monedas excede una meta establecida, el programa debe imprimir "¡Hora de gastar!" Luego, el programa debe imprimir el número total de monedas que hay en cada alcancía y el número total de monedas que hay en todas las alcancías.

La clase primaria

Aborde el problema más crítico lo más pronto posible.

La parte más importante de este problema es la utilización de miembros de instancia y de clase. Así, este hecho se abordará de inmediato. Se necesita una clase que describa de manera simultánea alcancías de un centavo y la colección de todas las alcancías para estas monedas. La figura 9.4 muestra el diagrama de una clase UML para una clase

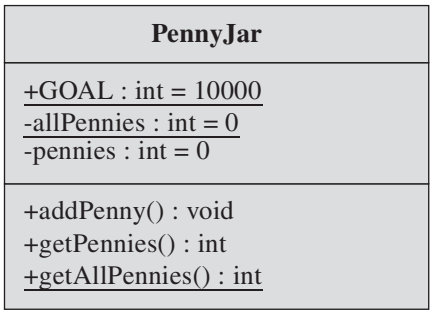


Figura 9.4 Clase que describe las alcancías de manera individual y como grupo.

`PennyJar` que hace lo que se desea. Para manejar las monedas en un objeto individual `PennyJar`, usa miembros de instancia: `pennies`, `addPenny` y `getPennies`. Para manejar las monedas en la colección de todas las alcancías, usa miembros de clase: `GOAL`, `AllPennies` y `getAllPennies`. En el diagrama UML puede decirse que estos tres miembros son miembros de clase porque están subrayados (como quizá recuerde el lector, las normas UML sugieren subrayar todos los miembros de clase). El diagrama UML no incluye ningún método `main`, de modo que esta clase no se ejecuta por sí misma. En efecto, se está empezando con una perspectiva de implementación del problema y desarrollando el programa en forma ascendente, ya que el enfoque actual está en los detalles de los miembros de instancia y de clase.

Observe la figura 9.5. Contiene una implementación de la clase `PennyJar`. Primero se analizarán las declaraciones de constantes y de variables de `PennyJar`:

- `GOAL` constituye el número de monedas a alcanzar que deben ahorrarse entre todas las alcancías combinadas. Como tal, es un miembro de clase y usa el modificador `static`. Puesto que la cantidad a alcanzar es fija, `GOAL` es una constante nombrada y usa el modificador `final` y se escribe sólo con mayúsculas. `GOAL` se inicializa en `10000`, que representa $100.00. Presumiblemente, cuando el usuario alcanza la cantidad `GOAL`, vacía todas las alcancías y gasta todo el dinero a lo loco.
- La variable `AllPennies` almacena el total de monedas que hay en todas las alcancías. Puesto que `AllPennies` es un atributo de todas las alcancías, es un miembro de clase y usa el modificador `static`. Aunque como valor inicial por defecto podría justamente aceptarse el cero, de forma explícita se inicializa en cero para recalcar lo que se desea.
- La variable `pennies` es una variable de instancia normal. De nuevo, aunque como valor inicial por defecto podría justamente aceptarse el cero, de forma explícita se inicializa en cero para recalcar lo que se desea.

A continuación se analizarán las definiciones del método:

- El método `getPennies` es un método de acceso típico, y recupera el valor de la variable de instancia `pennies`. Acceder a la variable de instancia `pennies` significa que `getPennies` debe ser un método de instancia. El hecho de que `getPennies` es un método de instancia puede verse porque en su encabezado no hay ningún modificador `static`.
- El método `addPenny` simula la adición de una moneda a una alcancía. Actualiza la variable de instancia `pennies` para la alcancía a la que se ha introducido la moneda y actualiza la variable de clase `allPennies` para la colección de alcancías. Accede a la variable de instancia `pennies` significa que `addPenny` debe ser un método de instancia y que, como tal, en su encabezado no hay ningún modificador `static`.
- El método `getAllPennies` recupera el valor de la variable de clase `AllPennies`. Puesto que `getAllPennies` trata sólo con información de la clase a escala**,** resulta conveniente hacerlo un método de clase. Puede verse que `getAllPennies` es un método de clase debido al modificador `static` que aparece en su encabezado.

Controlador

La figura 9.6 contiene un controlador para la clase `PennyJar`. Observe cómo `main` llama a los métodos de instancia `addPenny` y `getPennies` al crear primero objetos `PennyJar`. Asigna los objetos re-

```
/*****************************************************************
* PennyJar.java
* Dean & Dean
*
* Esta clase cuenta monedas para alcancías individuales y para todas
* las alcancías combinadas.
*****************************************************************/

public class PennyJar
{
  public static final int GOAL = 10000;
  private static int allPennies = 0;
  private int pennies = 0;

  //***************************************************************

  public int getPennies()
  {
    return this.pennies;
  }

  //***************************************************************

  public void addPenny()
  {
    System.out.println("Clink!");
    this.pennies++;
    PennyJar.allPennies++;

    if (PennyJar.allPennies >= PennyJar.GOAL)
    {
      System.out.println("Time to spend!");
    }
  } // end addPenny

  //***************************************************************

  public static int getAllPennies()
  {
    return PennyJar.allPennies;
  }
} // end class PennyJar
```

variables de clase

variables de instancia

método de instancia

método de clase

Figura 9.5 Clase `PennyJar` que ilustra miembros de instancia y miembros de clase.

cientemente creados a las variables de referencia `pennyJar1` y `pennyJar2`. Luego usa estas variables de referencia para llamar a los métodos de instancia.

Observe cómo `main` usa una técnica diferente para llamar a `getAllPennies`. En vez de prefijar `getAllPennies` con una variable de referencia (`pennyJar1` o `pennyJar2`), `main` llama a `getAllPennies` al prefijarlo con el nombre de la clase `PennyJar`. Lo anterior es porque `getAllPennies` es un método de clase, no un método de instancia. ¿Sería correcto omitir el prefijo del nombre de la clase y simplemente llamar directamente a `getAllPennies`? No. No es posible omitir el prefijo del nombre de la clase porque `getAllPennies` está en otra clase. Si `main` se hubiera fusionado con la clase `PennyJar` para ejecutar el programa desde esa clase, entonces el prefijo `PennyJar` podría omitirse de la llamada al método `getAllPennies`. Sin embargo, no hace daño incluir el prefijo del nombre

```
/*************************************************************
* PennyJarDriver.java
* Dean & Dean
*
* Esta clase controla la clase PennyJar.
*************************************************************/

public class PennyJarDriver
{
  public static void main(String[] args)
  {
    PennyJar pennyJar1 = new PennyJar();
    PennyJar pennyJar2 = new PennyJar();

    pennyJar1.addPenny();
    pennyJar1.addPenny();
    pennyJar2.addPenny();
    System.out.println(pennyJar1.getPennies());
    System.out.println(PennyJar.getAllPennies());
  } // end main
} // end class PennyJarDriver
```

Output:

```
Clink!
Clink!
Clink!
2
3
```

Figura 9.6 Controlador de la clase `PennyJar` en la figura 9.5.

de la clase para una llamada a un método de clase. Facilita copiar y pegar, y es de ayuda para que el código sea más autodocumentado.

9.7 Resolución de problemas con miembros de clase y miembros de instancia en una clase de listas ligadas (opcional)

El programa previo `PennyJar` se considera justo como un programa "juguete", aunque algunas veces los programas juguete pueden ayudar a aprender nuevas técnicas. En esta sección se usará el enfoque de diseño basado en casos descritos en el capítulo 8, y el programa `PennyJar` se vuelve en algo más práctico.

Programa FundRaiser

Adapte un programa previo a un nuevo propósito.

Cada alcancía es un agente en el proceso de colectar dinero. Considere que el agente es un fiscal humano. Entonces la colección de todas las alcancías se transforma en un grupo de personas que trabaja en una actividad organizada para aumentar fondos monetarios. Las alcancías se convierten en solicitudes de donativos, y el GOAL de 10 000 monedas de un centavo se vuelve un GOAL de 10 000 dólares en donativos. Ahora viene lo mejor: sustituya la clase `PennyJar` por una clase `Agent` más general, y sustituya la clase `PennyJarDriver` por la clase `FundRaiser`.

La clase `Agent` debe contar con métodos semejantes a los métodos en la clase `PennyJar`. El método de instancia `getPennies` puede sustituirse por un método de instancia `getValue`; el método de instancia `addPenny`, por un método de instancia `addValue`, y el método de clase `getAllPennies`, por el método de clase `getAllValues`. Una vez hecho lo anterior, el lector debe preguntarse si puede haber una mejor manera de asignar el trabajo realizado por estos métodos.

Puesto que las alcancías pasivas se han transformado en agentes humanos activos, el lector debe preguntarse: "¿Dónde se tomó la decisión de gastar el dinero?" Esta decisión de alto nivel debe realizarse en el controlador. Así, a medida que se pasa del programa `PennyJar` al programa `FundRaiser`, es necesario cambiar la ubicación de las decisiones finales de gastar. Esto significa que la constante `GOAL` debe desplazarse hacia la clase `FundRaiser`.

En forma semejante, el lector debe preguntarse: "¿Dónde se toman las decisiones sobre las contribuciones individuales?" "¿Se toman al nivel de la gerencia en la clase `FundRaiser` o al nivel de los agentes individuales en la clase `Agent`?" Los donantes hablan a los agentes individuales, de modo que la información debe entrar al programa a través de un método `Agent`, quizás el método `addValue`. Puesto que ahora las decisiones sobre las contribuciones individuales están descentralizadas, es necesario coordinar esfuerzos desde arriba. Es necesario planificar la conducción de campañas publicitarias dirigidas centralmente. Para efectuar cada una de estas campañas es necesario definir otro método de clase: `addAllValues`. En un intento por obtener múltiples donativos, el método `addAllValues` se desplaza a través de todos los agentes y hace que cada agente llame a `addValue`. Después de cada campaña se llama al método de clase `getAllValues` para ver si el valor total excede a `GOAL`. En caso afirmativo, imprimir el valor total y salir.

En un incrementador de fondos real, es posible pronosticar cuántos donantes habrá. En forma semejante, en el programa FundRaiser, se desea poder manipular una cantidad desconocida de agentes donantes. El truco consiste en plantear las cosas como un "tesoro de caza": un viaje en el que cada destino intermedio proporciona el siguiente destino. Cada agente indica dónde se encuentra el próximo agente, hasta que se llega al último, quien indica que eso es todo.

Use una lista ligada para acceder a todos los objetos desde un punto.

Esta estrategia se usa para el método `getAllValues` y para el método `addAllValues`. En lugar de simplemente leer un valor previamente acumulado desde una variable de clase como `AllPennies`, el método `getAllValues` acumula valores desde agentes individuales. Así se evita la duplicación de datos y se elimina la necesidad de contar con una variable de clase `allValues`. También libera a cada agente individual de la tarea de sumar la contribución actual a una variable `allValues` además de sumarla a su propio `valor`.

No obstante, la clase `Agent` sigue requiriendo una variable de clase: una variable de referencia que indique a los métodos de clase dónde iniciar sus recorridos. Esta variable de referencia de clase se denomina `listOfAgents`, y siempre contiene una referencia al primer objeto a visitar. Cada objeto contiene una variable de referencia de instancia denominada `nextAgent`, que refiere al siguiente objeto a visitar. En el último objeto de la lista, la variable de referencia `nextAgent` contiene `null`. Esto indica que el recorrido ha terminado. Esta estructura se denomina *lista ligada*, y la variable de referencia a una simple clase, `listOfAgents`, señala al objeto en el *encabezado* de la lista.

Inicialmente, en la lista no hay objetos, y el valor en la variable `listOfAgents` es `null`. El constructor para cada nuevo objeto `Agent` inserta ese objeto al inicio de la lista ligada de la clase `Agent`, usando el siguiente algoritmo:

set `this` agent's `nextAgent` to `listOfAgents`
set `listOfAgents` to `this` agent

Esto significa que el primer objeto visitado en un recorrido a través de todos los objetos es el último objeto que se construye, y que el último objeto que se visita es el primero que se construye. En otras palabras, la secuencia de visitas es opuesta a la secuencia de construcciones.

En la figura 9.7 se muestra el diagrama de clase UML para el programa. Como de costumbre, la clase controladora (`FundRaiser`) cuenta con un método `main`. También tiene una constante de clase, `GOAL`, que establece el criterio de detención. La clase `Agent` tiene un constructor, dos métodos de clase `públicos` (`getAllValues` y `addAllValues`), así como dos métodos de instancia `privados` (`getValue` y `addValue`). Observe que los dos métodos de clase `públicos` proporcionan la única ruta de acceso a todo lo que hay en esta clase. ¿Recuerda el lector la regla de que un método de clase no puede acceder directamente a un miembro de instancia? Si esa regla es válida, y todo lo demás es algún tipo de miembro de instancia, ¿cómo pueden estos dos métodos de clase acceder al resto? Estos métodos no acceden directamente a ningún miembro de instancia. Acceden directamente a la variable de clase `listOfAgents`. Esa variable de clase les proporciona una referencia a un objeto, y cada objeto les pro-

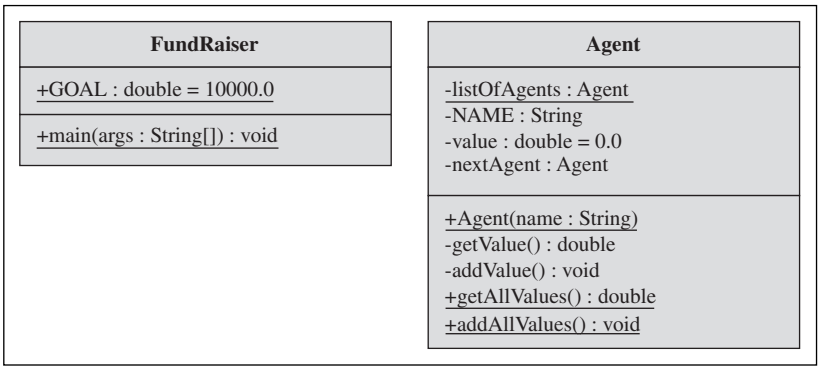


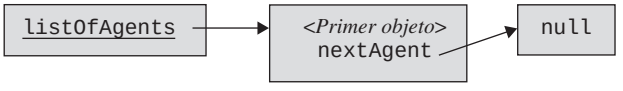
Figura 9.7 Diagrama de clase UML para el controlador del programa FundRaiser.

porciona la referencia para el siguiente objeto. Estas referencias a objetos permiten que los miembros de clase accedan de manera indirecta a todo lo demás.

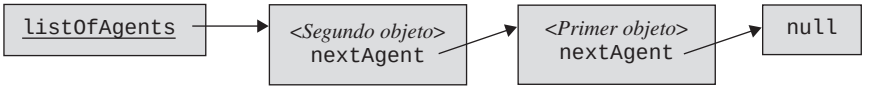
Ya es hora de considerar el código de implementación. En la figura 9.8 se muestra la clase controladora `FundRaiser`. El controlador pide al usuario introducir un número deseado de agentes, y el ciclo `for` avanza a través de un proceso que introduce el nombre de cada agente e instancia un objeto `Agent` con ese nombre. El siguiente ciclo `do` inicia cada campaña para incrementar los fondos al llamar al método `addAllValues`. Luego determina el resultado de esa campaña al llamar al método `getAllValues`. A continuación, imprime el resultado acumulado después de esa campaña. Si el resultado sigue siendo menor que `GOAL`, lanza otra campaña hasta que se alcanza el objetivo. Observe que este controlador no contiene ninguna referencia a objetos `Agent` en particular. Ni siquiera contiene una referencia a la lista ligada de agentes. Todo el acceso a los datos en la clase `Agent` es controlado por los métodos de clase `Agent` y por `addAllValues` y `getAllValues`. Los datos están bien encapsulados, lo cual, en efecto, ¡es algo bastante positivo!

En la figura 9.9*a* se muestra la primera parte de la clase `Agent`. Inicialmente la variable de clase `listOfAgents` contiene `null`, ya que al principio no hay objetos en la lista. Ahora se considerará el constructor. La primera instanciación asigna este valor `null` a la variable de referencia `nextAgent` del primer objeto. Luego usa la referencia `this` para asignar el primer objeto a la variable de clase `listOfAgents`. La segunda instanciación asigna la referencia ahora en la variable de referencia `listOfAgents` (una referencia al primer objeto) a la variable de referencia `nextAgent` del segundo objeto. Luego usa la referencia `this` para asignar el segundo objeto a la variable de clase `listOfAgents`. Así, la lista ligada de objetos se estructura como sigue:

después de la primera instanciación:



después de la segunda instanciación:



El resto del código en la figura 9.9*a* es directo. El método de instancia `getValue` devuelve el valor que está en la variable de instancia `value`. El método de instancia `addValue` asigna una entrada del usuario a la variable de instancia `value`.

En la figura 9.9*b* se muestran los dos métodos de clase en la clase `Agent`. El método `addAllValues` empieza en el objeto referido por la variable de referencia de la clase, `listOfObjects`. Llama al método de instancia `addValue` de ese objeto para recuperar la entrada y agregarla a la variable de instancia `value` de ese objeto. Luego usa la variable de instancia `nextAgent` del objeto actual para encontrar el siguiente objeto en la lista, y repite el proceso hasta que `nextAgent` es `null`. El método `getAllValues` inicializa las variables locales, a `totalValue` y a `agent`. Luego avanza a través de los objetos justamente como lo hizo el método `addAllValues`, sumando a `totalValue` el valor de-

```
/*****************************************************************
* FundRaiser.java
* Dean & Dean
*
* Este programa administra a los agentes que incrementan fondos.
*****************************************************************/

import java.util.Scanner;

public class FundRaiser
{
  public static final double GOAL = 10000.00;

  //*************************************************************

  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int numberOfAgents;
    double totalValue;
    String name;

    System.out.print("Enter total number of agents: ");
    numberOfAgents = stdIn.nextInt();
    stdIn.nextLine();
    for (int i=0; i<numberOfAgents; i++)
    {
      System.out.print("Enter agent name: ");
      name = stdIn.nextLine();
      new Agent(name);
    }
    do
    {
      Agent.addAllValues();
      totalValue = Agent.getAllValues();
      System.out.printf("Total value = $%,.2f\n", totalValue);
    } while (totalValue < GOAL);
    System.out.println("Time to Spend!");
  } // end main
} // end FundRaiser class
```

Observe que este controlador no requiere seguir la pista de ninguna referencia a objetos.

Todos los accesos ulteriores a la clase `Agent` es a través de métodos de clase.

Figura 9.8 Nivel superior del programa FundRaiser.
Esto controla la clase `Agent` en las figuras 9.9*a* y 9.9*b*.

vuelto por el método de instancia `getValue`. Cuando la referencia `agent` se vuelve `null`, se detiene y devuelve el `totalValue` acumulado.

La sesión de muestra a continuación indica lo que hace el programa. Observe que la secuencia empleada cuando se suman nuevos objetos es opuesta a la secuencia empleada cuando se crearon los objetos originales. Esto se debe a que cada objeto nuevo se inserta al inicio de la lista ligada, más que al final, como normalmente sería de esperar. Cada nuevo objeto puede colocarse al final de la lista y hacer que la secuencia sea la misma, aunque esto requeriría una variable de clase adicional y más código en el constructor.

Sesión muestra:

```
Enter total number of agents: 3
Enter agent name: Bavitha
Enter agent name: Alan
Enter agent name: Rebecca
```

```
/*************************************************************
* Agent.java
* Dean & Dean
*
* Clase que describe agentes que reúnen valores cuantitativos.
*************************************************************/

import java.util.Scanner;

public class Agent
{
  private static Agent listOfAgents = null;    // Inicio de la lista.

  private final String NAME;
  private double value = 0.0;
  private Agent nextAgent;                     // Próximo en la lista.

  //*********************************************************

  public Agent(String name)
  {
    this.NAME = name;
    this.nextAgent = listOfAgents;
    listOfAgents = this;
  } // end constructor

  //*********************************************************

  private double getValue()
  {
    return this.value;
  }

  //*********************************************************

  private void addValue()
  {
    Scanner stdIn = new Scanner(System.in);

    System.out.printf("Enter %s's contribution: ", this.NAME );
    this.value += stdIn.nextDouble();
  } // end addValue
```

Esto inserta cada objeto nuevo al inicio de la lista ligada.

Figura 9.9*a* Primera parte de la clase `Agent` del programa FundRaiser. Este código y el código en la figura 9.9*b* están controlados por la clase `FundRaiser` en la figura 9.8.

```
Enter Rebecca's contribution: 6 000
Enter Alan's contribution: 6 000
Enter Bavitha's contribution: 6 000
Total value = $18 000.00
Time to Spend!
```

Clase API `LinkedList`

La biblioteca API de Sun contiene varias clases que manejan colecciones de datos. Estas clases se denominan *marco de colecciones de Java* o *colecciones de API*. En el siguiente capítulo se analizará en deta-

```
  //*********************************************************

  public static double getAllValues()
  {
    double totalValue = 0.0;
    Agent agent = listOfAgents;
    while (agent != null)
    {
      totalValue += agent.getValue();
      agent = agent.nextAgent;
    }
    return totalValue;
  } // end getAllValues

  //*********************************************************

  public static void addAllValues()
  {
    Agent agent = listOfAgents;
    while (agent != null)
    {
      agent.addValue();
      agent = agent.nextAgent;
    }
  } // end addAllValues
} // end class Agent
```

Esto recupera la ubicación del siguiente objeto en la lista.

Figura 9.9*b* Métodos de clase en la clase `Agent` del programa FundRaiser. Este código y el código en la figura 9.9*a* están controlados por la clase `FundRaiser` en la figura 9.8.

lle una de estas colecciones de clases, la clase `ArrayList`. En este capítulo, y específicamente en esta sección, se han estudiado las listas ligadas. A menudo, los programadores implementan listas ligadas a partir de cero, como se muestra en esta sección, aunque como una alternativa, también implementan listas ligadas usando la clase `LinkedList`, otra clase de las colecciones de Java. Para aprender sobre la clase `LinkedList` y todas las demás colecciones de clases, el lector debe consultar la página http://java.sun.com/javase/6/docs/technotes/guides/collections/

Resumen

- Las variables de clase tienen un modificador `static`. Use variables de clase para atributos de la colección de todos los objetos en la clase. Use variables de instancia para atributos de objetos individuales.
- Recuerde que las variables de clase tienen un alcance más amplio que las variables de instancia, y que las variables de instancia tienen un alcance más amplio que las variables locales. Para mejorar el encapsulamiento, debe intentar el uso de variables cuyo alcance sea más estrecho, en lugar de más amplio.
- Un método de instancia puede acceder directamente a miembros de clase, así como a miembros de instancia.
- Un método de clase puede acceder directamente a miembros de clase, pero no puede acceder directamente a miembros de instancia. Para acceder a un miembro de clase desde un método de clase, es necesario usar un prefijo punto del nombre de la clase.
- Utilice métodos de clase para procesos relacionados con el grupo de todos los objetos en una clase, para procesos que deben existir antes que se defina cualquier objeto (como `main`), para métodos de clase de ayuda y para utilidades de propósito general.

- Las constantes de instancia tienen sólo un modificador `final`. Úselas para atributos permanentes de objetos individuales.
- Use constantes de clase para datos permanentes que no estén asociados con ningún objeto particular. Las constantes de clase usan los modificadores `final` y `static`.
- Puede usar una variable de clase para referirse a una lista ligada arbitrariamente larga de objetos de una clase. Esto permite que otra clase acceda a todos los objetos de la clase sólo a través de métodos de clase, y la otra clase no requiere de ninguna referencia a objetos específicos.

Preguntas de revisión

§9.2 Variables de clase

1. Normalmente, debe utilizar el modificador de acceso `private` para variables de clase. (F/V)
2. ¿Cuándo debe declarar una variable como una variable de clase, en oposición a una variable de instancia?
3. ¿Cuáles son los valores por defecto de las variables de clase?

§9.3 Métodos de clase

4. En la clase `Mouse4` en la figura 9.2, suponga que usted cuenta con un método cuyo inicio es `public int getAge()`. Suponga que desea llamar a este método desde otra clase. ¿Cuál es el error en la siguiente sentencia?

```
int age = Mouse4.getAge();
```

5. El acceso a miembros:
   a) Es correcto usar el `this` en un método de clase. (C/V)
   b) Es correcto usar el nombre de la clase como prefijo cuando se llama a un método de clase. (C/V)
   c) Dentro de un método `main` es correcto omitir el prefijo del nombre de la clase antes del nombre de otro método de clase que se esté llamando. (C/V)
6. Es legal acceder a un miembro de clase desde un método de instancia y también desde un constructor. (C/V)
7. Es legal acceder directamente a un miembro de instancia desde un método de clase. (C/V)
8. ¿Cuáles son cuatro razones comunes para convertir un método en un método de clase?

§9.4 Constantes nombradas

9. ¿Qué palabra clave convierte una variable en una constante?
10. Si el lector desea que una constante nombrada usada por un método de instancia tenga el mismo valor sin importar a cuál objeto accede, la declaración debe incluir el modificador `static`. (C/V)
11. Una constate de clase debe inicializarse dentro de un constructor. (C/V)
12. Suponga que se tiene un programa para calificar qué instancia múltiples objetos examen desde una clase `Exam`. Escriba una declaración para una calificación mínima aprobatoria constante. Suponga que la calificación mínima aprobatoria para todos los exámenes es 59.5.

§9.5 Escritura de la propia clase de utilidad

13. Los miembros de una clase de utilidad normalmente deben usar los modificadores `private` y `static`. (C/V)

Ejercicios

1. [Después de §9.2] Se tiene una clase con una variable de clase. Todos los objetos de la clase obtienen una copia por separado de la variable de clase. (C/V)
2. [Después de §9.2] En general, ¿por qué deben preferirse variables locales en lugar de variables de instancia, y variables de instancia en lugar de variables de clase?
3. [Después de §9.2] Dado un programa que sigue la pista de los detalles de un libro con ayuda de una clase `Book`, para cada una de las siguientes variables del programa, especifique si debe ser una variable local, una variable de instancia o una variable de clase.

`bookTitle` (el título de un libro particular)
`averagePrice` (el precio medio de todos los libros)
`price` (el precio de un libro particular)
`i` (una variable de indexación usada para recorrer todos los libros)

4. [Después de §9.3] Si un método accede a una variable de clase y también a una variable de instancia, el método:
   a) debe ser un método local;
   b) debe ser un método de instancia;
   c) debe ser un método de clase;
   d) puede ser un método de clase o un método de instancia: depende de otros factores

5. [Después de §9.3] Si se intenta acceder directamente a un método de instancia desde un método de clase, aparece un mensaje de error como el siguiente:

```
Non-static <nombre del método> no puede referirse desde un contexto
estático
```

Normalmente, ¿cómo es posible arreglar este error?

6. [Después de §9.3] Considere el siguiente programa.

```
public class Test
{
  private int x;
  private static int y;
  public void doIt()
  {
    x = 1;
    y = 2;
  }
  public static void tryIt()
  {
    x = 3;
    y = 4;
  }
  public static void main(String[] args)
  {
    doIt();
    tryIt();
    Test t = new Test();
    t.doIt();
    Test.doIt();
    Test.tryIt();
  }
} // end Test class
```

   a) Marque todas las líneas del código que tienen un error de compilación.
   b) Para cada línea que contiene un error de compilación, explique por qué es incorrecta.
   Observe lo siguiente:
   - En las declaraciones de las variables y en los encabezados de los métodos no hay ningún error, de modo que no marque ninguna de estas líneas como si tuviese un error.
   - Para cada error de compilación, simplemente mencione la razón por la que ocurre el error. En particular, no resuelva el problema modificando el código hasta deshacerse de todos los errores de compilación.

7. [Después de §9.4] ¿Por qué es seguro declarar a las constantes nombradas como `públicas` (`public`)?

8. [Después de §9.4] Escriba declaraciones idóneas para las siguientes constantes. En cada caso, decida si incluir o no la palabra clave `static`, así como incluir una inicialización en la declaración. Asimismo, haga lo más accesible posible a cada constante, de manera consistente con la protección de corrupción fortuita.
   a) El año de nacimiento de cada persona.
   b) La cadena de formato `"%-25s%, 13.2f%, 13.2f%(,15.2f\n"`, para usar en varias sentencias `printf` en un solo método.
   c) La "razón áurea" o razón ancho/largo de un rectángulo de oro. Es igual a (sqrt(5) − 1)/2 = 0.6180339887498949.

9. [Después de §9.5] Escriba una clase de utilidad denominada `RandomDistribution`, que contiene los cuatro siguientes métodos de clase. El lector debe poder implementar todos estos métodos con llamadas a métodos de clase `Math` y/o llamadas a uno de los métodos `uniform` dentro de la clase `RandomDistribution`.

a) Escriba un método denominado `uniform` que genere un número aleatorio `double` a partir de una distribución continua que sea uniforme entre los valores `double min` y `max`.

b) Escriba otro método (sobrecargado) denominado `uniform` que genere un número aleatorio `int` a partir de una distribución discreta que sea uniforme entre los valores `int min` y `max`, inclusive estos dos puntos extremos.

c) Escriba un método denominado `triangular` que genere un número aleatorio `int` a partir de una distribución triangular simétrica que varíe entre los valores `int min` y `max`, inclusive estos dos puntos extremos. (*Sugerencia:* Haga dos llamadas a la versión `int` del método `uniform` anterior.)

d) Escriba un método denominado `exponential` que genere un número aleatorio `double` a partir de una distribución exponencial cuyo tiempo esperado entre la llegada de eventos aleatorios es igual a `averageTimeInterval`. Éste es el algoritmo:

`return` ← `averageTimeInterval` * $\log_e(1.0 - \texttt{Math.random})$

10. [Después de §9.6] En el programa `PennyJar`, el prefijo punto `PennyJar` se usa para acceder a miembros `PennyJar`. Hay cuatro prefijos punto `PennyJar` en la clase `PennyJar` y un prefijo punto en la clase `PennyJarDriver`. Para cada uno de estos prefijos, ¿es válido omitirlo?

11. [Después de §9.11] Programa `PetMouse`:
El programa crea una lista ligada de objetos. La variable de referencia de la clase, `pets`, refiere al primer objeto en la lista, y cada objeto ulterior contiene una variable de referencia de instancia que refiere al siguiente objeto, excepto que la variable de referencia de instancia en el último objeto refiere a `null`. Observe que las mascotas individuales son *objetos anónimos*, en el sentido de que no poseen nombres por separado. La variable de referencia de clase, `pets`, en realidad refiere al primer objeto en la lista, aunque conceptualmente refiere a todos los objetos en la lista. Observe que se usa la misma palabra, `next`, para una variable local en un método de clase y una variable de instancia, ya que ambas variables en realidad hablan de lo mismo, por lo que no sería lógico usar términos diferentes.

```
 1  /*************************************************************
 2  * PetMouseDriver.java
 3  * Dean & Dean
 4  *
 5  * Esto crea y muestra una lista ligada de objetos simples.
 6  *************************************************************/
 7
 8  public class PetMouseDriver
 9  {
10    public static void main(String[] args)
11    {
12      new PetMouse();
13      new PetMouse();
14      new PetMouse();
15      PetMouse.list();
16    } // end main
17  } // end class PetMouseDriver
```

```
 1  /*************************************************************
 2  * PetMouse.java
 3  * Dean & Dean
 4  *
 5  * Esto crea y muestra una lista ligada de objetos simples.
 6  *************************************************************/
 7
 8  import java.util.Scanner;
 9
10  public class PetMouse
11  {
12    private static PetMouse pets;  // apunta a la lista de pets
13
14    private String name;
```

```
15    private PetMouse next;
16
17    //*************************************************
18
19    // Inserta cada objeto nuevo al inicio de una lista existente.
20
21    public PetMouse()
22    {
23      Scanner stdIn = new Scanner(System.in);
24
25      this.next = pets;
26      System.out.print("Enter name: ");
27      this.name = stdIn.nextLine();
28      pets = this;
29    } // end constructor
30
31    //*************************************************
32
33    public static void list()
34    {
35      PetMouse next = pets;
36
37      while (next != null)
38      {
39        System.out.print(next.name + "  ");
40        next = next.next;
41      }
42      System.out.println();
43    } // end list
44  } // end class PetMouse
```

Use lo siguiente para rastrear el programa PetMouse. Observe cómo la variable de clase `pets` está abajo del encabezado `PetMouse`, aunque separada de los tres objetos. Se ha mostrado el valor inicial de `pets`: `null`.

input
cutie
sugar
fluffy

Driver	PetMouse										
		static	list	obj1		obj2		obj3			
line#	***line#***	**pets**	**next**	**name**	**next**	**name**	**next**	**name**	**next**	**output**
		null								

Soluciones a las preguntas de revisión

1. Cierto.
2. Es necesario declarar una variable que sea una variable de clase, en contraposición a una variable de instancia si la variable contiene datos que están asociados con la clase como un todo. Debe usar variables de clase para describir propiedades de los objetos de una clase que deban ser compartidos por todos los objetos.
3. Los valores por defecto para las variables de clase son los mismos que para las variables de instancia del mismo tipo. Éstos son los valores por defecto:

   los tipos enteros obtienen 0

   los tipos de punto flotante obtienen 0.0

   los tipos `booleanos` obtienen `false`

   los tipos de referencia obtienen `null`

4. Debido a que no tiene modificador `static`, `getAge` es un método de instancia. La llamada de `Mouse4.getAge( )` genera un prefijo punto `Mouse4`. Es ilegal usar un nombre de clase (`Mouse4`) como prefijo para una llamada a un método de instancia. Para llamar a un método de instancia, es necesario usar un prefijo punto de variable de referencia.
5. El acceso a miembros:
   a) Falso. El `this` no puede usarse en un método de clase.
   b) Cierto. Siempre es posible usar el nombre de la clase como prefijo de método de clase.
   c) Cierto, si el método `main` se "fusiona" en la misma clase que el otro método.
   Falso, si el otro método está en una clase diferente.
   Incluir el prefijo del nombre de la clase permite desplazar más tarde el método `main` a otra clase.
6. Cierto. Es posible acceder a un miembro de clase desde un método de instancia y también desde un constructor, basta prefijar el miembro de clase con el nombre de la clase.
7. Falso. Un miembro de instancia puede accederse desde un método de clase sólo si el nombre del método se prefija con una referencia a un objeto particular.
8. Es necesario convertir un método en un método de clase:
   a) Si se tiene un método que usa variables de clase y/o llama a métodos de clase, entonces es un buen candidato para volverse un método de clase.
   b) Si puede ser necesario llamar a un método inclusive cuando no hay objetos de la clase del método, entonces es necesario convertirlo en un método de clase.
   c) El método principal debe ser un método de clase. Si un método principal usa métodos de ayuda que no implican miembros de instancia, entonces los métodos de ayuda deben ser métodos de clase.
   d) Si se tiene un método de propósito general autosustentable, hágalo un método de clase.
9. La palabra clave `final` convierte una variable en una constante.
10. Cierto. Use `static` para que una constante sea la misma para todos los objetos.
11. Falso. Normalmente, una constante de clase debe inicializarse como parte de su declaración. Si más tarde se le asigna un valor, inclusive dentro de un constructor, se genera un error de compilación.
12. Declaración para una calificación mínima aprobatoria:

```
private static final double MIN_PASSING_SCORE = 59.5;
```

13. Falso. Los miembros de una clase de utilidad normalmente deben usar los modificadores `public` y `static`.

CAPÍTULO 10

Arreglos y `listas de arreglos`

Objetivos

- Comparar un arreglo con otros objetos.
- Crear e iniciar arreglos.
- Copiar valores de un arreglo a otro.
- Desplazar datos en un arreglo.
- Hacer histogramas.
- Buscar un arreglo para datos particulares.
- Ordenar datos.
- Crear y usar arreglos de dos dimensiones.
- Crear y usar arreglos de objetos.
- Ver cómo la clase `ArrayList` hace más flexibles los arreglos.
- Almacenar primitivos en una `ArrayList`.
- Pasar objetos anónimos hacia y desde métodos.
- Aprender cómo usar ciclos for-each.

Relación de temas

10.1 Introducción

En el pasado se ha visto que los objetos suelen contener más de un dato, y que cada uno de los diferentes datos posee un nombre distinto. Ahora se considerará un tipo especial de objeto que contiene varios

artículos del mismo tipo y usa el mismo nombre para todos ellos. El lenguaje natural cuenta con medios para asignar un solo nombre a una población: "grupo" de lobos, "ganado" vacuno, "manada" de leones, "pandilla" de zarigüeyas, "grupo" de hurones, etc. Java cuenta con una forma de hacer lo mismo.

Cuando se tiene una colección de cosas del mismo tipo y se desea usar el mismo nombre para todas ellas, es posible definirlas a todas juntas en un *arreglo*. Cada miembro del arreglo se denomina más formalmente *elemento* del arreglo. Para distinguir los distintos elementos en el arreglo, se usa el nombre del arreglo más un número que identifica la posición del elemento dentro del arreglo. Por ejemplo, si se almacena una colección de títulos de canciones en un arreglo denominado `songs`, el título de la primera canción se identifica diciendo `songs[0]`, y el título de la segunda canción se distingue diciendo `songs[1]`. Como se muestra en este ejemplo, los elementos del arreglo empiezan en la posición 0. Los números de posición de un arreglo (0, 1, 2, etc.) se denominan más formalmente *índices*. En la siguiente sección se dice más sobre los índices de un arreglo.

Hay una ventaja importante al usar un nombre para todo un grupo de artículos semejantes y distinguirlos sólo por un número. Esto puede llevar a un código más simple. Por ejemplo, si se requiere almacenar el título de 100 canciones, es posible declarar 100 variables por separado. Pero qué lástima sería escribir 100 sentencias de declaración y seguir la pista de 100 nombres de variables diferentes. La solución más simple es utilizar un arreglo y declarar una sola variable, una variable arreglo `songs`.

Los lectores que desean una introducción temprana a los arreglos tienen la opción de leer las secciones 10.1 a 10.6 después de completar el capítulo 4. La conexión natural entre el capítulo 4 y este capítulo es que el capítulo 4 describe ciclos y los arreglos dependen bastante de los ciclos.

Empezando con la sección 10.7, los arreglos se presentan en un contexto orientado a objetos, donde los arreglos son miembros de una clase. Se analizan técnicas para buscar un arreglo y ordenar un arreglo. Se describen diversas estructuras organizacionales para los arreglos, arreglos de dos dimensiones y arreglos de objetos. Luego se presentan las `listas de arreglos`, que son semejantes a los arreglos aunque proporcionan más flexibilidad. Las `ArrayLists` crecen dinámicamente a medida que se agregan elementos, y es fácil insertar o eliminar elementos en medio de las `ArrayLists`. Por último, se describe un tipo especial de ciclo `for` denominado ciclo for-each, que es particularmente útil para el procesamiento de los elementos en una `ArrayList`.

10.2 Fundamentos de arreglos

En esta sección se muestra cómo realizar operaciones sencillas en un arreglo, como cargar un arreglo con datos e imprimir un arreglo. Para ilustrar estas operaciones se hará referencia al arreglo `phoneList` en la figura 10.1. Este arreglo contiene una lista de cinco números telefónicos de marcación rápida para un teléfono celular. El primer número telefónico es 8167412000; el segundo, 2024561111, y así sucesivamente.

Introducción de los elementos de un arreglo

Para trabajar con un arreglo es necesario introducir los elementos de un arreglo. Por ejemplo, para imprimir el contenido de un arreglo, es necesario introducir el primer elemento del arreglo, imprimirlo, intro-

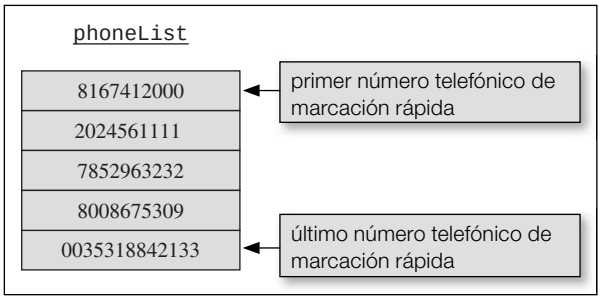


Figura 10.1 Arreglo ejemplo, arreglo con cinco elementos para contener una lista de números telefónicos de marcación rápida.

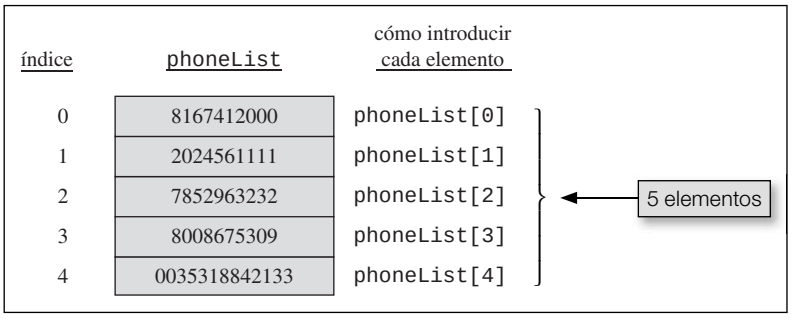


Figura 10.2 Introducción de elementos en un arreglo `phoneList`.

ducir el segundo elemento del arreglo, imprimirlo y así sucesivamente. Para introducir un elemento a un arreglo, se especifica el nombre del arreglo, seguido por el índice del elemento escrito entre corchetes. En la figura 10.2 se muestra cómo introducir los elementos individuales al arreglo `phoneList`. El índice del primer elemento es 0, de modo que el primer elemento se introduce con `phoneList[0]`. ¿Por qué el índice del primer elemento es 0 en lugar de 1? El índice es una medida de cuán lejos se está del inicio del arreglo. Si se está justo al principio, la distancia al principio es 0. Así, el primer elemento usa 0 como valor de su índice.

⚠ Los programadores novatos a menudo creen que el último índice en un arreglo es igual al número de elementos en el arreglo. Por ejemplo, un programador novato podría pensar que el último índice en el arreglo `phoneList` es igual a 5 porque este arreglo tiene 5 elementos. No es así. El primer índice es 0, y el último es 4. Intente recordar esta regla importante: el último índice en un arreglo es igual a uno menos que el número de elementos en el arreglo. Si se intenta introducir un elemento del arreglo con un índice mayor que el último índice o menor que cero, el programa se cae. De modo que si se especifica `phoneList[5]` o `phoneList[-1]`, el programa se cae. Como parte de esa caída, la máquina virtual Java (JVM) imprime un mensaje de error con la palabra "`ArrayIndexOutOfBoundsException`" en el mensaje. `ArrayIndexOutOfBoundsException` es una *excepción*. En el capítulo 15 se aborda el estudio de las excepciones, pero por el momento considere que una excepción es un tipo de error sofisticado que pueden usar los programadores para determinar la fuente de un error.

Ahora que ya se sabe cómo introducir un elemento de un arreglo, se pondrá en práctica. A continuación se muestra cómo cambiar el primer número telefónico a 2013434:

```
phoneList[0] = 2013434;
```

y he aquí cómo imprimir el segundo número:

```
System.out.println(phoneList[1]);
```

Tome en cuenta que algunas personas usan el término "subíndice" en lugar de "índice" porque la identificación con subíndices constituye la forma normal en inglés de representar un elemento dentro de un grupo. En otras palabras, x_0, x_1, x_2, etc., en escritura normal es lo mismo que `x[0]`, `x[1]`, `x[2]` y así sucesivamente en Java.

Programa ejemplo

A continuación se verá cómo los arreglos se usan en el contexto de un programa completo. En la figura 10.3, el programa SpeedDialList solicita al usuario la cantidad de números telefónicos de marcación rápida que serán introducidos, llena el arreglo `phoneList` con los números telefónicos introducidos por el usuario, e imprime la lista creada de números de marcación rápida. Para llenar un arreglo e imprimir los elementos de un arreglo, típicamente es necesario recorrer cada elemento del arreglo con ayuda de una variable índice que se incrementa desde cero hasta el índice del último elemento lleno del arreglo. A menudo, las operaciones de incremento de la variable índice se implementan con ayuda de un ciclo `for`. Por ejemplo, el programa SpeedDialList usa el siguiente encabezado del ciclo `for` para incrementar una variable índice, `i`:

```
for (int i=0; i<sizeOfList; i++)
```

Con cada iteración para el ciclo `for`, `i` corre desde 0 hasta 1 hasta 2 y así sucesivamente, e `i` sirve como un índice para los diferentes elementos en el arreglo `phoneList`. A continuación se muestra cómo el ciclo coloca un número telefónico en cada elemento:

```
phoneList[i] = phoneNum;
```

```
/*************************************************************************
* SpeedDialList.java
* Dean & Dean
*
* Este programa crea una lista de números telefónicos de marcación
* rápida a teléfonos celulares e imprime la lista creada.
*************************************************************************/

import java.util.Scanner;

public class SpeedDialList
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    long[] phoneList;   // lista de números telefónicos
    int sizeOfList;     // cantidad de números telefónicos
    long phoneNum;      // un número telefónico introducido

    System.out.print(
      "¿Cuántos números telefónicos de marcación rápida le gustaría introducir? ");
    sizeOfList = stdIn.nextInt();
    phoneList = new long[sizeOfList];

    for (int i=0; i<sizeOfList; i++)
    {
      System.out.print("Enter phone number: ");
      phoneNum = stdIn.nextLong();
      phoneList[i] = phoneNum;
    } // end for

    System.out.println("\nSpeed Dial List:");
    for (int i=0; i<sizeOfList; i++)
    {
      System.out.println((i + 1) + ". " + phoneList[i]);
    } // end for
  } // end main
} // end class SpeedDialList
```


Sesión muestra:

```
¿Cuántos números telefónicos de marcación rápida le gustaría introducir? 2
Número telefónico introducido: 8167412000
Número telefónico introducido: 2024561111

Speed Dial List:
1. 8167412000
2. 2024561111
```

Figura 10.3 Programa SpeedDialList que muestra cómo crear, llenar e imprimir un arreglo.

10.3 Declaración y creación de arreglos

En la sección previa se mostró cómo realizar operaciones simples en un arreglo. Al hacerlo, el centro de atención fue la introducción de elementos en un arreglo. En esta sección se aborda otro concepto clave: la declaración y creación de arreglos.

Declaración de arreglos

Un arreglo es una variable y, como tal, es necesario declararla antes de poder usarla. Para declarar un arreglo, se usa esta sintaxis:

<*element-type*>`[]` <*array-variable*>`;`

El <*array-variable*> es el nombre del arreglo. Los corchetes vacíos indican que la variable está definida como un arreglo. El <*element-type*> indica el tipo de cada elemento en el arreglo: `int`, `double`, `char`, `String`, y así sucesivamente.

A continuación se muestran algunos ejemplos de declaración de arreglos:

```
double[] salaries;
String[] names;
int[] employeeIds;
```

La variable `salaries` es un arreglo cuyos elementos son de tipo `double`. La variable `names` es un arreglo cuyos elementos son de tipo `String`. Y por último, la variable `employeeIds` es un arreglo cuyos elementos son de tipo `int`.

Java proporciona un formato de declaración alternativo para arreglos, donde los corchetes van después del nombre de la variable. He aquí de lo que se está hablando:

```
double salaries[];
```

Los dos formatos son idénticos en términos de funcionalidad. La mayoría de quienes están en el negocio prefieren el primer formato, que es el que se usa aquí, aunque el lector debe tomar en cuenta el formato alternativo en caso de que lo vea en el código de otra persona.

Creación de arreglos

Un arreglo es un objeto, aunque es un tipo especial de objeto. Así como ocurre con cualquier objeto, un arreglo contiene un grupo de datos. Y como con cualquier objeto, un arreglo puede crearse o instanciarse usando el operador `new`. Ésta es la sintaxis para crear un objeto del arreglo con el operador `new` y asignar el objeto del arreglo a una variable arreglo:

<*array-variable*> `=` `new` <*element-type*>`[`<*array-size*>`];`

El <*element-type*> indica el tipo de cada elemento en el arreglo. El <*array-size*> indica el número de elementos en el arreglo. El siguiente fragmento de código crea un arreglo de 10 elementos de `longs`:

```
long[] phoneList;
phoneList = new long[10];
```

Estas dos líneas realizan tres operaciones: 1) la primera línea declara la variable `phoneList`; 2) el código en el recuadro con línea punteada crea el objeto del arreglo, y 3) el operador de asignación asigna una referencia al objeto del arreglo en la variable `phoneList`.

En una declaración, es legal combinar operaciones de declaración, creación y asignación de un arreglo. El siguiente ejemplo hace exactamente eso. Reduce el código previo de dos líneas a sólo una:

```
long[] phoneList = new long[10];
```

Aquí, para el tamaño del arreglo se usa una constante (10), aunque no se requiere usar una constante. Para indicar el tamaño del arreglo puede usarse cualquier expresión. En la figura 10.3, el programa SpeedDialList pide al usuario el tamaño del arreglo, almacena el tamaño introducido en una variable

`sizeOfList` y usa `sizeOfList` para la creación del arreglo. Éste es el código para la creación del arreglo desde el programa SpeedDialList:

```
phoneList = new long[sizeOfList];
```

Iniciación de los elementos de un arreglo

Por lo general, se quiere declarar y crear un arreglo en un sitio y asignar valores a los elementos del arreglo en otro sitio. Por ejemplo, el siguiente fragmento de código declara y crea un arreglo `temperatures` en una declaración, y asigna valores al arreglo `temperatures` en otra declaración, dentro de un ciclo.

```
double[] temperatures = new double[5];
for (int i=0; i<5; i++)
{
  temperatures[i] = 98.6;
}
```

declara y crea un arreglo

asigna un valor al $i^{ésimo}$ elemento del arreglo

Por otra parte, algunas veces se quiere declarar y crear un arreglo, y asignar valores al arreglo, todo en la misma declaración. Esto se denomina *iniciador del arreglo*, cuya sintaxis es ésta:

<*element-type*>`[]` <*array-variable*> `=` `{`<*value1*>`,` <*value2*>`,` . . .`,` <*valuen*>`};`

El código a la izquierda del operador de asignación declara una variable del arreglo usando una sintaxis que ya se ha visto antes. El código a la derecha del operador de asignación especifica una lista de valores separados por una coma que están asignados a los elementos del arreglo. Observe este ejemplo:

```
double[] temperatures = {98.6, 98.6, 98.6, 98.6, 98.6};
```

Al comparar la declaración anterior con el fragmento de código `temperatures` previo, puede verse que es lo mismo en términos de funcionalidad pero diferente en términos de estructura. Diferencias fundamentales: 1) es de una línea, en lugar de cinco líneas; 2) no hay operador `new`; 3) no hay valor del tamaño del arreglo. Sin este valor, ¿cómo es posible que el compilador conozca el tamaño del arreglo? El tamaño del arreglo está determinado por el número de valores en la lista de valores de los elementos. En el ejemplo anterior, en la lista del iniciador hay cinco valores, de modo que el compilador crea un arreglo con cinco elementos.

Se presentaron dos soluciones para asignar valores a un arreglo temperaturas. ¿Cuál es mejor: el fragmento de código de cinco líneas o el iniciador del arreglo de una línea? Aquí se prefiere la solución del iniciador porque es más simple. Sin embargo, recuerde que la técnica del iniciador del arreglo sólo puede usarse si se conocen los valores asignados cuando el arreglo se declara por primera vez. Para el ejemplo de las temperaturas, los valores asignados se conocen cuando el arreglo se declara por primera vez: cada temperatura se inicia en 98.6, la temperatura normal del cuerpo humano en grados Fahrenheit. El uso de iniciadores de arreglos debe limitarse a situaciones en las que el número de valores asignados es razonablemente pequeño. Para el ejemplo de temperaturas, el número de valores asignados es razonablemente pequeño: cinco. Si es necesario seguir la pista de una centena de temperaturas, sería legal usar la solución del iniciador del arreglo, aunque podría ser algo fastidioso:

```
double[] temperatures =
{
  98.6, 98.6, 98.6, 98.6, 98.6, 98.6, 98.6, 98.6, 98.6, 98.6,
  <repetir ocho veces la línea anterior>
  98.6, 98.6, 98.6, 98.6, 98.6, 98.6, 98.6, 98.6, 98.6, 98.6
}
```

Valores por defecto

Ya se sabe cómo iniciar explícitamente los elementos de un arreglo con un iniciador del arreglo. Pero, en caso de no utilizar un iniciador del arreglo, ¿qué obtienen por defecto los elementos de un arreglo? Un arreglo es un objeto, y los elementos de un arreglo son las variables de instancia de un objeto del arreglo. Como tales, los elementos de un arreglo obtienen valores por defecto una vez que se crea el arreglo; así

como cualesquiera otras variables de instancia obtienen valores por defecto. Éstos son los valores por defecto para los elementos de un arreglo:

Tipo de los elementos del arreglo	Valor por defecto
entero	0
de punto flotante	0.0
`boolean`	`false`
de referencia	`null`

Así, ¿cuáles son los valores por defecto para los elementos en el siguiente arreglo?

```
double[] rainfall = new double[365];
String[] colors = new String[5];
```

El arreglo `rainfall` obtiene 0.0 por cada uno de sus 365 elementos. El arreglo `colors` obtiene `null` para cada uno de sus cinco elementos.

10.4 Propiedad `length` en un arreglo y arreglos parcialmente llenos

Como ya se ilustró, al trabajar con un arreglo es común recorrer cada elemento en el arreglo. Al hacer lo anterior, es necesario conocer el tamaño del arreglo y/o el número de elementos llenos en el arreglo. En esta sección se estudia cómo obtener el tamaño de un arreglo y cómo seguir la pista del número de elementos llenos en un arreglo.

Propiedad `length` de un arreglo

Suponga que se tiene un arreglo `colors` de cinco elementos que ha sido iniciado como sigue:

```
String[] colors = {"blue", "gray", "lime", "teal", "yellow"};
```

He aquí cómo imprimir este arreglo:

```
for (int i=0; i<5; i++)
{
  System.out.println(colors[i]);
}
```

tamaño del arreglo duramente codificado

Esto funciona bien, pero suponga que en su código hay varios ciclos relacionados con el color, cada uno de los cuales usa `i < 5`. Si el programa se modifica para que quepan más colores, y el arreglo de cinco elementos se modifica a un arreglo de 10 elementos, es necesario cambiar todas las apariciones de `i < 5` a `i < 10`. A fin de evitar este trabajo de mantenimiento, ¿no sería más práctico sustituir `i < 5` o `i < 10` por algo genérico, como `i <` tamaño del arreglo? Esto puede hacerse usando la propiedad `length` del arreglo `color`. Todo objeto del arreglo contiene una propiedad `length` que almacena el número de elementos en el arreglo. La propiedad `length` se denomina "propiedad", aunque en realidad es simplemente una variable de instancia con modificadores `public` y `final`. El modificador `public` indica que `length` es directamente accesible sin necesidad de un método de acceso. El modificador `final` hace de `length` una constante nombrada; de modo que es posible actualizarla. He aquí cómo puede usarse la propiedad `length`:

```
for (int i=0; i<colors.length; i++)
{
  System.out.println(colors[i]);
}
```

número de elementos en el arreglo

Propiedad `length` del arreglo contra método `String length`

¿Recuerda el lector dónde más ha visto la palabra `length` en el lenguaje Java? La clase `String` proporciona un método `length` para recuperar el número de caracteres en una cadena. Recuerde que `String length` es un método, de modo que al llamarlo es necesario usar paréntesis de rastreo. Por otra parte, la longitud de un arreglo es una constante, de modo que al acceder a él no se usa paréntesis de rastreo. En la figura 10.4, el programa SpeedDialList2 ilustra estos conceptos. Observe que `phoneNum.length( )` usa paréntesis al comprobar la longitud de la cadena `phoneNum` como parte de la validación de entrada.

```
/*****************************************************************
* SpeedDialList2.java
* Dean & Dean
*
* Este programa crea una lista de números telefónicos de marcación
* rápida e imprime la lista creada. Usa un arreglo parcialmente lleno.
*****************************************************************/

import java.util.Scanner;

public class SpeedDialList2
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String[] phoneList = new String[100]; // números telefónicos
    int filledElements = 0;     // cantidad de números telefónicos
    String phoneNum;            // un número telefónico introducido

    System.out.print("Introducir el número telefónico (o q para salir): ");
    phoneNum = stdIn.nextLine();
    while (!phoneNum.equalsIgnoreCase("q") &&
           filledElements < phoneList.length)
    {
      if (phoneNum.length() < 1 || phoneNum.length() > 16)
      {
        System.out.println("Entrada inválida." +
          " Debe introducir entre 1 y 16 caracteres.");
      }
      else
      {
        phoneList[filledElements] = phoneNum;
        filledElements++;
      }
      System.out.print("Introducir el número telefónico (o q para salir): ");
      phoneNum = stdIn.nextLine();
    } // end while

    System.out.println("\nSpeed Dial List:");
    for (int i=0; i<filledElements; i++)
    {
      System.out.println((i + 1) + ". " + phoneList[i]);
    } // end for
  } // end main
} // end class SpeedDialList2
```

La propiedad `length` del arreglo no usa paréntesis.

El método `String length` usa paréntesis.

Actualiza el número de elementos llenos.

Usa `filledElements` para imprimir el arreglo.

Figura 10.4 Programa SpeedDialList2 que procesa un arreglo parcialmente lleno, usando la propiedad `length` del arreglo y el método `String length`.

Asimismo, observe que `phoneList.length` no usa paréntesis al comprobar el número de elementos en el arreglo `phoneList` para asegurarse que hay espacio para otro número telefónico.

Si el lector es como nosotros, puede tener bastantes dificultades para recordar cuándo usar paréntesis y cuándo no. Intente usar el acrónimo mnemónico ANCS, que significa Arreglos No, Cadenas Sí. "Arreglos No" significa que los arreglos no usan paréntesis cuando especifican longitud. "Cadenas Sí" significa que las cadenas sí usan paréntesis cuando especifican longitud. Si al lector no le agradan los APCL,[1] puede intentar un método más analítico para recordar la regla del paréntesis. Los arreglos son objetos de un caso especial que no tienen métodos; en consecuencia, la longitud de un arreglo debe ser una constante, no un método. Y las constantes no usan paréntesis.

Arreglos parcialmente llenos

En la figura 10.4 observe cómo el programa SpeedDialList2 declara que el arreglo `phoneList` tiene 100 elementos. El programa solicita repetidamente al usuario que introduzca un número telefónico o presione q para salir. Típicamente, el usuario introduce menos números que el máximo de 100. Eso resulta en que el arreglo `phoneList` esté parcialmente lleno. Si se tiene un arreglo parcialmente lleno, en oposición a uno completamente lleno, es necesario seguir la pista del número de elementos llenos en el arreglo, de modo que sea posible procesar los elementos llenos en forma distinta a como se procesan los elementos que no están llenos. Observe cómo el programa SpeedDialList2 usa la variable `filledElements` para seguir la pista de la cantidad de números telefónicos en el arreglo. `filledElements` empieza en cero y se incrementa cada vez que el programa almacena un número telefónico en el arreglo. Para imprimir el arreglo, el programa usa `filledElements` en el siguiente encabezado del ciclo `for`.

```
for (int i=0; i<filledElements; i++)
```

Es bastante común que los programadores introduzcan accidentalmente elementos no llenos en un arreglo parcialmente lleno. Por ejemplo, suponga que el ciclo `for` de `SpeedDialList2` es algo así como:

```
for (int i=0; i<phoneList.length; i++)
{
  System.out.println((i + 1) + ". " + phoneList[i]);
} // end for
```

El uso de `phoneList.length` en el encabezado del ciclo `for` funciona bastante bien para imprimir un arreglo totalmente lleno, pero no para imprimir un arreglo parcialmente lleno. En el programa SpeedDialList2, los elementos no llenos llevan `null` (el valor por defecto para una cadena), de modo que el ciclo `for` anterior imprimiría `null` para cada uno de los elementos no llenos. Y esto produce usuarios confundidos y molestos. ☹

10.5 Copia de un arreglo

En las secciones previas, la atención se centró en detalles de la sintaxis de los arreglos. En las secciones siguientes, el tema será menos la sintaxis y más el lado relacionado con la aplicación de las cosas. En esta sección se analiza un problema de objetivo general: cómo copiar de un arreglo a otro.

Uso de arreglos para almacenar precios de una tienda departamental

Suponga que se usan arreglos para almacenar los precios de una tienda departamental: un arreglo para los precios de cada mes. A continuación se presenta el arreglo para los precios de enero:

```
double[] pricesJanuary = {1.29, 9.99, 22.50, 4.55, 7.35, 6.49};
```

La intención es usar el arreglo de enero como punto de partida para los arreglos de los otros meses. Específicamente, se quiere copiar los precios de enero en los arreglos de los otros meses y modificar los pre-

[1] APCL = Acrónimos prácticos de cuatro letras.

cios de los otros meses cuando sea necesario. La siguiente declaración crea el arreglo para los precios de febrero. Observe cómo `pricesJanury.length` asegura que el arreglo de febrero tiene la longitud que el arreglo de enero.

```
double[] pricesFebruary = new double[pricesJanuary.length];
```

Suponga que se desea que los valores en el arreglo de febrero sean los mismos que en el arreglo de enero, excepto por la segunda entrada, que se quiere modificar de 9.99 a 10.99. En otras palabras, se quiere algo como esto:

Salida:

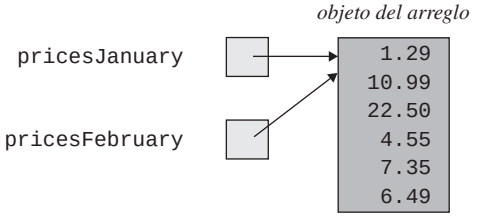
```
   Ene      Feb
 1.29     1.29
 9.99    10.99
22.50    22.50
 4.55     4.55
 7.35     7.35
 6.49     6.49
```

Para minimizar el esfuerzo y evitar errores al introducir reentradas, sería perfecto hacer que la computadora copie los valores del primer arreglo en el segundo arreglo y luego simplemente modifique un elemento del segundo arreglo que es necesario cambiar. El siguiente fragmento de código, ¿funcionaría?

```
pricesFebruary = pricesJanuary;
pricesFebruary[1] = 10.99;
```

No es una buena idea.

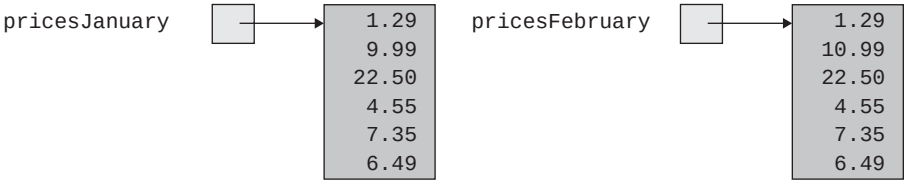
El nombre de un arreglo es sólo una referencia. Contiene la dirección de un sitio en la memoria donde empiezan los datos del arreglo. Así, `pricesFebruary = pricesJanuary;` se obtiene la dirección de los datos de `pricesJanuary` y la dirección se copia en `pricesFebruary`. Así, `pricesFebruary` y `pricesJanuary` se refieren a los mismos datos físicos. Esta cuestión se ilustra con la siguiente figura:



El problema con el hecho de que `pricesFebruary` y `pricesJanuary` se refieran a los mismos datos físicos es que si se modifican los datos en uno de los arreglos, entonces automáticamente se cambian los datos en el otro arreglo. Por ejemplo, la declaración anterior `pricesFebruary[1] = 10.99`; actualiza no sólo el segundo elemento de `pricesFebruary`, sino también el segundo elemento de `pricesJanuary`. Y no es esto lo que se quiere.

Usualmente, cuando se hace una copia de un arreglo, se quiere que el original y la copia apunten hacia objetos distintos del arreglo. Para hacer lo anterior, es necesario asignar uno por uno los elementos del arreglo. Vea el programa ArrayCopy en la figura 10.5. Usa un ciclo `for` para asignar uno por uno elementos de `pricesJanuary` a elementos de `pricesFebruary`.

Esto es lo que produce el código en la figura 10.5:



```
/*************************************************************
* ArrayCopy.java
* Dean & Dean
*
* Esto copia un arreglo y modifica la copia.
*************************************************************/

public class ArrayCopy
{
  public static void main(String[] args)
  {
    double[] pricesJanuary =
      {1.29, 9.99, 22.50, 4.55, 7.35, 6.49};
    double[] pricesFebruary = new double[pricesJanuary.length];

    for (int i=0; i<pricesJanuary.length; i++)
    {
      pricesFebruary[i] = pricesJanuary[i];
    }
    pricesFebruary[1] = 10.99;

    System.out.printf("%7s%7s\n", "Ene", "Feb");
    for (int i=0; i<pricesJanuary.length; i++)
    {
      System.out.printf("%7.2f%7.2f\n",
        pricesJanuary[i], pricesFebruary[i]);
    }
  } // end main
} // end class ArrayCopy
```

Figura 10.5 Programa ArrayCopy que copia un arreglo y luego modifica la copia.

`System.arraycopy`

El copiado de datos de un arreglo en otro es una operación bastante común, de modo que los diseñadores de Java cuentan con un método especial, `System.arraycopy`, sólo para ese fin. Permite copiar cualquier número de elementos desde cualquier sitio en un arreglo hacia cualquier sitio en otro arreglo. He aquí cómo puede usarse para copiar el arreglo `pricesJanuary` en la figura 10.5 en el arreglo `pricesFebruary`.

```
System.arraycopy(pricesJanuary, 0, pricesFebruary, 0, 6);
pricesFebruary[1] = 10.99;
```

El primer argumento es el nombre del arreglo fuente; es decir, el nombre del arreglo del cual se está copiando. El segundo argumento es el índice del primer elemento del arreglo fuente a copiar. El tercer argumento es el nombre del arreglo destino; es decir, el nombre del arreglo al que se está copiando. El cuarto argumento es el índice del primer elemento a sustituir en el arreglo de destino. El argumento final es el número total de elementos a copiar.

10.6 Resolución de problemas mediante casos con arreglos

Aprenda mediante ejemplos.

En esta sección se presentan dos estudios de caso basados en arreglos. Para cada estudio de caso se presenta un problema y luego se analiza su solución. Lo importante de estos estudios de caso no es que el lector memorice los detalles. Más bien, se trata de adquirir la sensibilidad de cómo resolver problemas orientados a arreglos. Así, cuando el lector es un programador en el mundo real, tiene un "costal de trucos" a su disposición. Probablemente deba modificar las soluciones del estudio de caso para ajustarlas a sus problemas específicos del mundo real, aunque está bien. Después de todo, el lector debe aprender a ganarse su sustento.

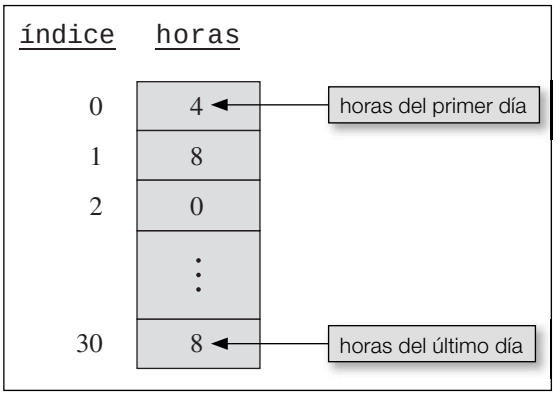


Figura 10.6 Arreglo que contiene horas laborales calendarizadas para los 31 días siguientes.

Desplazamiento de valores de elementos del arreglo

Considere el arreglo `hours` en la figura 10.6. Este arreglo contiene las horas laborales calendarizadas para una persona durante un periodo de 31 días. El primer elemento (`hours[0]`) contiene las horas laborales calendarizadas para la persona durante el día actual. El último elemento (`hours[30]`) contiene las horas laborales calendarizadas para la persona durante el día que será el trigésimo día. Al principio de cada día nuevo es necesario desplazar las horas laborales a posiciones con menor índice. Por ejemplo, el valor `hours[1]` debe desplazarse al elemento `hours[0]`. Esto debe tener sentido cuando al darse cuenta que se está pasando a un nuevo día, es necesario hacer que lo que eran las horas laborales calendarizadas para el día siguiente, `hours[1]`, se conviertan en las horas laborales calendarizadas del día actual, `hours[0]`.

Ahora se considerará el código Java que realiza esta operación de desplazamiento. Se quiere desplazar el valor de cada elemento `hours` a su elemento adyacente de menor índice. En otras palabras, se desea copiar el valor del segundo elemento en el primer elemento, copiar el valor del tercer elemento en el segundo elemento y así sucesivamente. Luego, se quiere asignar un valor introducido por el usuario al último elemento. Éste es el código:

```
for (int d=0; d<hours.length-1; d++)
{
  hours[d] = hours[d+1];
}
```

A fin de desplazar valores a posiciones con menor índice, es necesario empezar en el extremo del índice bajo y trabajar hacia el otro extremo.

```
System.out.print("Enter last day's scheduled hours: ");
hours[hours.length-1] = stdIn.nextInt();
```

Hay varias cosas que observar respecto a este fragmento de código. Es correcto usar una expresión entre los corchetes; se usa `hours [d + 1]` para introducir el elemento después del elemento `hours[d]`. Observe cómo primero se desplazan elementos en el extremo del índice bajo. ¿Qué ocurriría si el desplazamiento se inicia en el extremo del índice alto? Se rescribiría el siguiente elemento que se quisiera mover y todo el arreglo terminaría llenándose con el valor que originalmente estaba en el elemento más alto. Esto no está bien.

Cálculo de un promedio variable

Tome prestado código y modifíquelo.

Ahora se tomará prestado código del ejemplo anterior y se aplicará a otro problema. Suponga que es necesario presentar un promedio móvil durante cuatro días del Índice Industrial Dow Jones (IIDJ) al final de las actividades cotidianas de la Bolsa de Valores. Suponga que ya cuenta con un arreglo de cuatro elementos que contiene los valores del IIDJ al final del día de cada uno de los últimos cuatro días, con el valor de hace cuatro días en el índice 0, el valor de hace tres días en el índice 1, el valor de hace dos días en el índice 2 y el valor de ayer en el índice 3. Para el promedio móvil de cuatro días de hoy, se quiere la suma de los valores de los tres últimos días más el valor para hoy. Esto significa que es necesario desplazar todo en el arreglo a po-

siciones de menor índice e insertar el valor para hoy en el extremo del índice alto. Luego es necesario sumar todos en el arreglo y dividir entre la longitud del arreglo. Presumiblemente, usted guardará el arreglo desplazado en algún sitio y luego hará lo mismo nuevamente al final de cada día futuro. Es posible efectuar el desplazamiento y sumar en ciclos separados, aunque es más fácil hacer ambas cosas en el mismo ciclo, como se muestra en la figura 10.7.

Para contar con diferentes longitudes temporales, es mejor no codificar duramente la longitud del arreglo. En lugar de eso, siempre debe usarse <*nombre del arreglo*>.`length`. Considere cuidadosamente cada límite. Observe que el índice `[d + 1]` en el miembro derecho de la primera declaración en el ciclo interno `for` es uno más que el valor de la variable de conteo `d`. Recuerde que el valor más alto de índice en un arreglo siempre es uno menos que la longitud del arreglo. En consecuencia, el valor más alto de la variable de conteo debe ser igual a la longitud del arreglo menos dos. Ésta es la razón por la cual la condición de continuación del ciclo es `d<days.length – 1`. Observe también que el nuevo valor final para el arreglo se inserta después que termina el ciclo, y luego este valor final se incluye en la suma antes de calcular el promedio. He aquí un ejemplo de lo que hace el programa:

```
/**************************************************************************
* MovingAverage.java
* Dean & Dean
*
* Este programa contiene una operación que desplaza cada elemento
* del arreglo al siguiente elemento más bajo y carga una nueva
* entrada en el elemento final.
**************************************************************************/

import java.util.Scanner;

public class MovingAverage
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int[] days = {9400, 9500, 9600, 9700}; // rising market
    double sum;
    int samples;

    System.out.print("Introduzca el número de días a evaluar: ");
    samples = stdIn.nextInt();
    for (int j=0; j<samples; j++)
    {
      // shift down and sum
      sum = 0.0;
      for (int d=0; d<days.length-1; d++)
      {
        days[d] = days[d+1];
        sum += days[d];
      }
      System.out.print("Enter next day's value: ");
      days[days.length-1] = stdIn.nextInt();
      sum += days[days.length-1];
      System.out.printf(
        "Moving average = %5.0f\n", sum / days.length);
    }
  } // end main
} // end class MovingAverage
```

Esto desplaza a posiciones de índices menores.

Esto acumula los valores ya desplazados.

Esto desplaza el último valor.

Figura 10.7 Cálculo de un promedio variable.

Sesión muestra:

```
Enter number of days to evaluate: 4
Enter next day's value: 9800
Moving average =  9650
Enter next day's value: 9800
Moving average =  9725
Enter next day's value: 9700
Moving average =  9750
Enter next day's value: 9600
Moving average =  9725
```

Un promedio variable es más continuo que una gráfica instantánea, pero observe que sus valores se retrasan.

Hay una forma más simple para efectuar desplazamientos: ¿Recuerda el lector el método API `arraycopy` mencionado en la sección previa? Es posible usarlo para implementar desplazamientos a posiciones de menor índice con este fragmento de código:

```
System.arraycopy(days, 1, days, 0, days.length-1);
System.out.print("Enter next day's value: ");
days[days.length-1] = stdIn.nextInt();
```

Conceptualmente, el método `arraycopy` copia todo desde el elemento 1 hasta el último elemento en un arreglo temporal, y luego lo copia desde este arreglo temporal de vuelta en el arreglo original empezando en el elemento 0. Así se elimina el ciclo interno `for` en la figura 10.7. Desafortunadamente, el ciclo interno `for` también se usó para calcular la suma necesaria para el promedio. Sin embargo, hay un truco que puede usarse para hacer más eficiente un programa como éste cuando el arreglo es muy grande. Si se sigue la pista de la suma de todos los elementos en el arreglo, cada vez que se desplazan los valores del elemento en el arreglo, sólo es posible corregir la suma, en lugar de volver a calcularla por completo. Para corregir la suma, se resta el valor desplazado de salida y se suma el valor desplazado de entrada, como se muestra a continuación:

```
sum -= days[0];
System.arraycopy(days, 1, days, 0, days.length-1);
System.out.print("Enter next day's value: ");
days[days.length-1] = stdIn.nextInt();
sum += days[days.length-1];
```

Histogramas

En esta subsección se usa un arreglo como parte de un programa para elaborar histogramas. Pero antes de presentar el programa, es conveniente hacer una revisión de los histogramas. Un *histograma* es una gráfica que presenta cantidades para un conjunto de categorías. Típicamente, indica las cantidades de las categorías con barras: barras más cortas se igualan con cantidades pequeñas y barras largas se igualan con cantidades más grandes. Por ejemplo, el histograma en la figura 10.8 muestra cantidades de postres congelados producidos en Estados Unidos en 2003.[2] Los histogramas constituyen una forma conocida para presentar datos estadísticos porque proporcionan una representación rápida y clara de la distribución de los datos.

Suponga que se tienen tres monedas. Cuando las tres se lanzan al mismo tiempo, se tiene interés en la posibilidad de no obtener ninguna cara, de obtener una cara, de obtener dos caras y de obtener tres caras. En otras palabras, se tiene interés en la distribución de frecuencias del número de caras.

Aproxime una solución matemática mediante simulación.

La distribución de frecuencias puede calcularse matemáticamente (con la fórmula de la distribución binomial) pero, en lugar de ello, se decide escribir un programa para simular el lanzamiento de las monedas. Si se simulan suficientes lanzamientos de las monedas, entonces los resultados se aproximarán al resultado calculado matemáticamente.

[2] National Agricultural Statistics Service, *Frozen Dessert Production Histogram*, en Internet en http://www.usda.gov/nass/nasskids/glossary_1.html

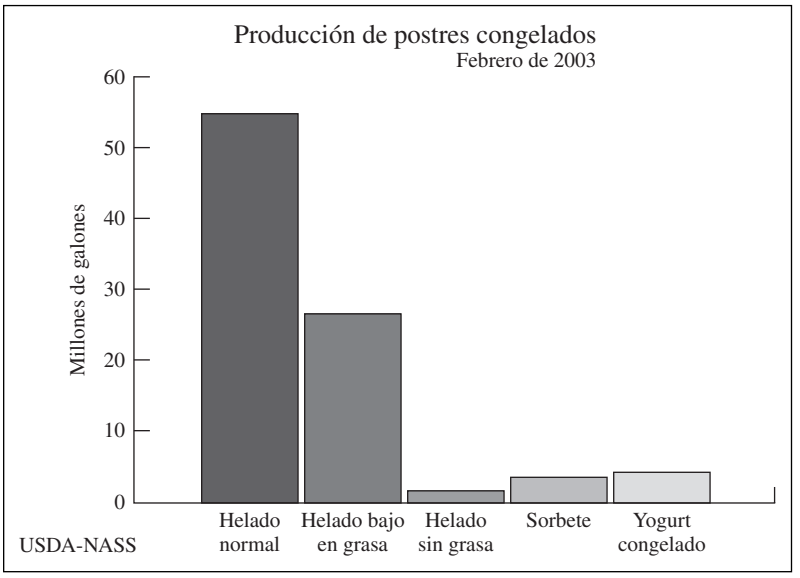


Figura 10.8 Ejemplo de histograma.

En el programa es necesario simular que las monedas se lanzan un millón de veces. Debe imprimir los resultados de la simulación en forma de histograma. Para cada uno de los cuatro casos (cero caras, una cara, dos caras y tres caras) imprima una serie de asteriscos, donde el número de éstos es proporcional al número de veces que ocurrió el caso. Cada serie de asteriscos representa una barra de un histograma. Esto debe tener más sentido si se observa la siguiente muestra de salida:

```
Número de veces que ocurre el conteo de una cara:
 0  124960 **************
 1  375127 ****************************************
 2  375261 ****************************************
 3  124652 **************
```

Observe el primer renglón de asteriscos. Se trata de una "barra" horizontal que describe gráficamente el número de veces que ocurre el caso en que no se obtiene ninguna cara. El cero a la izquierda es la etiqueta para el caso en que no se obtiene ninguna cara. El 124960 es el número específico de veces que ocurre el caso de cero caras. O dicho de otra forma, 124960 es la *frecuencia* del caso en que no se obtiene ninguna cara. Observe que las frecuencias de cero caras y tres caras (124960 y 124652, respectivamente) son casi iguales, y que las frecuencias de los casos una cara y dos caras (375127 y 375261, respectivamente) también son casi iguales. Asimismo, observe que las frecuencias de los casos en que se obtienen dos caras y tres caras son aproximadamente la tercera parte de las frecuencias de los casos en que se obtienen una cara y dos caras. Siempre resulta una buena idea usar algún tipo de cálculo independiente para pronosticar cómo sería una respuesta de la computadora. Para este problema simple, es relativamente fácil calcular una respuesta exacta. Si "T" significa "cruz" y "H" significa "cara", éstos son todos los resultados posibles de los lanzamientos:

Compare los resultados del programa con los resultados pronosticados.

TTT (0 caras)
TTH (1 cara)
THT (1 cara)
THH (2 caras)
HTT (1 cara)
HTH (2 caras)
HHT (2 caras)
HHH (3 caras)

Observe que sólo hay una forma de obtener cero caras y una sola forma de obtener tres caras, aunque hay tres formas de obtener una cara y tres formas de obtener dos caras. Así, las frecuencias de cero caras y de tres caras deben ser, cada una, la tercera parte de la frecuencia de una cara o dos caras. Si observa los números y las longitudes de las barras en la muestra de salida anterior, verá que el resultado de la computadora se ajusta, en efecto, a esta expectativa.

Vea al programa CoinFlips en la figura 10.9. Hace lo que se desea que haga. Simula el lanzamiento simultáneo de tres monedas un millón de veces, e imprime los resultados de la simulación en forma de

```
/**************************************************************
* CoinFlips.java
* Dean & Dean
*
* Esto genera un histograma de lanzamientos de monedas.
**************************************************************/

public class CoinFlips
{
  public static void main(String[] args)
  {
    final int NUM_OF_COINS = 3;       // número de repeticiones
    final int NUM_OF_REPS = 1000000;  // de monedas

    // El arreglo frequency contiene el número de veces que ha
    // ocurrido un número particular de caras.
    int[] frequency = new int[NUM_OF_COINS + 1];
    int heads;             // caras en el grupo actual de monedas
    double fractionOfReps; // conteo de caras / repeticiones
    int numOfAsterisks;    // asteriscos en una barra del histograma

    for (int rep=0; rep<NUM_OF_REPS; rep++)
    {
      // perform a group of flips
      heads = 0;
      for (int i=0; i<NUM_OF_COINS; i++)
      {
        heads += (int) (Math.random() * 2);
      }
      frequency[heads]++;   // actualiza la papelera idónea
    } // end for
    System.out.println(
      "Número de veces que ha ocurrido un conteo de cada cara:");
    for (heads=0; heads<=NUM_OF_COINS; heads++)
    {
      System.out.print(
        " " + heads + "  " + frequency[heads] + " ");
      fractionOfReps = (float) frequency[heads] / NUM_OF_REPS;
      numOfAsterisks = (int) Math.round(fractionOfReps * 100);

      for (int i=0; i<numOfAsterisks; i++)
      {
        System.out.print("*");
      }
      System.out.println();
    } // end for
  } // end main
} // end class CoinFlips
```

Este ciclo llena las papeleras de frequency. Cada iteración simula un grupo de lanzamientos simultáneos de tres monedas.

Este ciclo imprime el histograma. Cada iteración imprime una barra del histograma.

Figura 10.9 Programa CoinFlips que genera un histograma para simulación de lanzamientos de monedas.

histograma. Utiliza un arreglo `frequency` de cuatro elementos para seguir la pista del número de veces que ocurre el valor de un conteo de una cara. Cada elemento en el arreglo `frequency` se denomina *papelera*. En general, una papelera contiene el número de ocurrencias de un elemento. Para el programa CoinFlips, el elemento `frequency[0]` es la primera papelera, y contiene el número de veces que ninguno de los tres lanzamientos de las monedas resulta en cara. El elemento `frequency[1]` es la segunda papelera, y contiene el número de veces que una de las tres monedas resulta en cara. Al cabo de cada iteración de la simulación del lanzamiento de tres monedas, el programa suma uno a la papelera idónea. Por ejemplo, si una iteración particular genera una cara, el programa incrementa la papelera `frequency[1]`. Y si una iteración particular genera dos caras, el programa incrementa la papelera `frequency[2]`.

A continuación se analizará cómo el programa CoinFlips imprime las barras de asteriscos del histograma. Como se especifica en la segunda llamada en la figura 10.9, el segundo ciclo `for` grande imprime el histograma. Cada iteración del ciclo `for` imprime la etiqueta de la papelera (0, 1, 2 o 3) y luego la frecuencia para esa papelera. Luego calcula el número de asteriscos a imprimir al dividir la frecuencia en la papelera actual entre el número total de repeticiones y multiplicar por 100. A continuación, use un ciclo interno `for` para mostrar el número calculado de asteriscos.

10.7 Búsqueda de un arreglo

Para usar un arreglo, es necesario introducir sus elementos individuales. Si se conoce la ubicación del elemento en que se tiene interés, entonces el elemento se introduce simplemente al escribir entre corchetes el índice del elemento. Pero si se ignora la ubicación del elemento, entonces es necesario buscarlo. Por ejemplo, suponga que se está escribiendo un programa que sigue la pista de los alumnos que se inscriben para tomar los cursos que ofrece una escuela. Se supone que el programa es capaz de agregar un estudiante, quitar un estudiante, consultar los datos de un estudiante, y así sucesivamente. Todas estas operaciones requieren que primero se busque el estudiante en un arreglo de estudiantes (inclusive la operación de agregar un estudiante requiere una búsqueda para asegurar que el estudiante no está ya en el arreglo). En esta sección se presentan dos técnicas para buscar en un arreglo.

Búsqueda secuencial

Si el arreglo es corto (si tiene menos de aproximadamente 20 datos), la mejor forma de buscarlo es la más simple: el arreglo se recorre secuencialmente y el valor de cada elemento del arreglo se compara con el valor que se busca. Una vez que se encuentra una coincidencia, se hace algo y se regresa. A continuación se presenta una descripción en pseudocódigo del algoritmo de búsqueda secuencial:

```
i ← 0
while i < number of filled elements
   if list[i] equals the searched-for value
      <hacer algo y detener el ciclo>
   increment i
```

Adapte algoritmos genéricos a situaciones específicas.

Típicamente, los algoritmos son más genéricos que las implementaciones de Java. Parte de la resolución del problema es el proceso de adaptar algoritmos genéricos a situaciones específicas. En este caso, el código "hacer algo" es diferente para casos distintos. El método `findStudent` en la figura 10.10 ilustra una implementación del algoritmo de búsqueda secuencial. Este método particular podría formar parte de una clase `Course` que implementa un curso académico. La clase `Course` almacena el nombre de un curso, un arreglo de identificaciones de los estudiantes inscritos en el curso y el número de estudiantes en el curso. El método `findStudent` busca la identificación de un estudiante dado en el arreglo de identificaciones de los estudiantes. Si se encuentra el estudiante, el método regresa el índice de la identificación encontrada. En caso contrario, devuelve -1. Observe cómo el código de `findStudent` coincide con la lógica del algoritmo de búsqueda secuencial. En particular, observe cómo `findStudent` implementa *<hacer algo y detener el ciclo>* con una declaración `return i`. Esta declaración implementa "hacer algo" al regresar el índice de la identificación encontrada del estudiante. Implementa "detener el ciclo" al regresar del método y terminar el ciclo a la vez.

```
/*****************************************************************
* Course.java
* Dean & Dean
*
* Esta clase representa un curso particular en una escuela.
*****************************************************************/

public class Course
{
  private String courseName;   // nombre del curso
  private int[] ids;           // identificaciones de los estudiantes en el curso
  private int filledElements;  // número de elementos ocupados

  //***************************************************************

  public Course(String courseName, int[] ids, int filledElements)
  {
    this.courseName = courseName;
    this.ids = ids;
    this.filledElements = filledElements;
  } // end constructor

  //***************************************************************

  // Este método devuelve el índice de la identificación encontrada o -1 si no
  // se encontró la identificación.

  public int findStudent(int id)
  {
    for (int i=0; i<filledElements; i++)
    {
      if (ids[i] == id)
      {
        return i;
      }
    } // end for

    return -1;
  } // end findStudent
} // end class Course
```

Figura 10.10 Clase con método de búsqueda secuencial (`findStudent`).

Al analizar el método `findStudent`, el lector podría preguntarse: "¿Cuál es el interés práctico del índice devuelto?" Para hacer algo con una identificación (id) en el arreglo `ids`, es necesario conocer el índice de la identificación. Si el índice de la identificación se desconoce de antemano, el método `findStudent` encuentra este índice para el lector. Más tarde en este capítulo se verá cómo llamar a un método de búsqueda y usar el índice devuelto al ordenar un arreglo y cuándo sumar un nuevo valor a un arreglo. El lector sigue preguntándose ¿"cuál es el interés práctico del -1 devuelto cuando no se encuentra la identificación?" El -1 puede ser usado por el módulo que llama para verificar el caso de una identificación inválida de un estudiante.

La figura 10.11 contiene una clase `CourseDriver` que controla la clase `Course` en la figura 10.10. La clase `CourseDriver` es bastante directa. Crea un arreglo de identificaciones de estudiantes, almacena el arreglo en un objeto `Course`, solicita al usuario la identificación de un estudiante particular y luego llama a `findStudent` para ver si ese estudiante particular está tomando el curso. Para no complicar las cosas, se usa un iniciador a fin de crear el arreglo `ids`. Para un controlador de propósito más

general, quizá sea conveniente sustituir el iniciador con un ciclo que de manera repetida solicite al usuario la introducción de la identificación de un estudiante o q para salir. Si se elige esa opción, entonces es necesario almacenar el número de elementos ocupados en una variable `filledElements` y pasar la variable `filledElements` como el tercer argumento en la llamada al constructor `Course`. La llamada al constructor es algo así como:

```
Course course = new Course("CS101", ids, filledElements);
```

Búsqueda binaria

Si se tiene un arreglo con un gran número de elementos, por ejemplo 100 000, una búsqueda secuencial típicamente requiere bastante tiempo. Si ese arreglo debe buscarse muchas veces, a menudo vale la pena usar una búsqueda binaria. El nombre búsqueda binaria se debe al hecho de que separa en dos una lista de valores y reduce la búsqueda justamente a la mitad de la lista separada en dos.

Para que una búsqueda binaria funcione en un arreglo, éste debe ordenarse de modo que todo esté escrito en algún tipo de orden alfabético o numérico. En la siguiente sección se describe uno de los mu-

```
/*********************************************************************
* CourseDriver.java
* Dean & Dean
*
* Esta clase crea un objeto curso y busca la identificación
* de un estudiante dentro del objeto Course recientemente creado.
*********************************************************************/

import java.util.Scanner;

public class CourseDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int[] ids = {4142, 3001, 6020};
    Course course = new Course("CS101", ids, ids.length);
    int id;       // se está buscando la identificación del estudiante
    int index;    // Índice de la identificación buscada o -1 en caso de no encontrarla

    System.out.print("Enter 4-digit ID: ");
    id = stdIn.nextInt();
    index = course.findStudent(id);
    if (index >= 0)
    {
      System.out.println("búsqueda del índice de la identificación " + index);
    }
    else
    {
      System.out.println("no se encontró");
    }
  } // end main
} // end class CourseDriver
```

Sesión muestra:

```
Enter 4-digit ID: 3001
found at index 1
```

Figura 10.11 Controlador para el programa que ilustra una búsqueda secuencia.

chos métodos de ordenamiento que hay. Este ordenamiento inicial requiere más tiempo que una simple búsqueda secuencial, pero sólo es necesario hacerlo una vez.

Ya que se ha ordenado el arreglo, es posible efectuar una búsqueda binaria para encontrar rápidamente valores en el arreglo; inclusive cuando el arreglo es extremadamente largo. Una búsqueda secuencial requiere una cantidad de tiempo proporcional a la longitud del arreglo. Una búsqueda binaria requiere una cantidad de tiempo proporcional al logaritmo de la longitud del arreglo. Cuando un arreglo es muy largo, la diferencia entre lineal y logarítmico es enorme. Por ejemplo, suponga que la longitud es 100 000. Resulta que $\log_2(100\,000) \approx 17$. Puesto que 17 es aproximadamente 6 000 veces menor que 100 000, la búsqueda binaria es alrededor de 6 000 veces más rápida que una búsqueda secuencial para un arreglo de 100 000 elementos.

Vea el método `binarySearch` en la figura 10.12 y, en particular, observe el modificador `static`. Para implementar la búsqueda puede usarse un método de instancia o un método de clase. En la subsección previa, la búsqueda se implementó con un método de instancia. Esta vez se hará con un método de clase, que es idóneo si se desea que un método se utilice genéricamente. A fin de hacerlo genérico (es decir, para que sea utilizable por programas diferentes), el método debe colocarse en una clase por separado y hacer del método uno de clase. Puesto que se trata de un método de clase, programas diferentes pueden llamar fácilmente al método `binarySearch`, usando el nombre de clase `binarySearch`, en lugar de usar un objeto que llama. Por ejemplo, si el método `binarySearch` se coloca en la clase `Utilities`, el método `binarySearch` puede llamarse como se muestra a continuación:

```
Utilities.binarySearch(
  <nombre del arreglo>, <número de elementos ocupados>, <valor que se busca>);
```

```
public static int binarySearch(
  int[] array, int filledElements, int value)
{
  int mid;                        // índice del elemento medio
  int midValue;                   // valor del elemento medio
  int low = 0;                    // índice del elemento más bajo
  int high = filledElements - 1; // índice del elemento más alto

  while (low <= high)
  {
    mid = (low + high) / 2;       // siguiente punto medio
    midValue = array[mid];        // y el valor ahí
    if (value == midValue)
    {
      return mid;                 // ¡encontrarlo!
    }
    else if (value < midValue)
    {
      high = mid - 1;             // la próxima vez, usar la mitad inferior
    }
    else
    {
      low = mid + 1;              // la próxima vez, usar la mitad superior
    }
  } // end while

  return -1;
} // end binarySearch
```

Figura 10.12 Método que efectúa una búsqueda binaria de un arreglo ya escrito en orden ascendente.

En la llamada al método `binarySearch`, observe el argumento del arreglo. Siendo un método de clase, `binarySearch` no puede acceder a variables de instancia. Más específicamente, no puede acceder al arreglo buscado como una variable de instancia. Así, el arreglo buscado debe pasarse como un argumento. Esto permite que el método sea usado desde fuera de su clase.

Separe el problema en problemas más pequeños.

Antes de abordar los detalles del código en el método `binarySearch`, se analizará la estrategia básica: *dividir y vencer*. Primero se identifica el elemento medio en el arreglo ordenado. Luego se conjetura si el elemento buscado está antes o después del elemento medio. Si está antes del elemento medio, el intervalo de la búsqueda se reduce a la mitad inferior del arreglo (la mitad con los elementos de menor índice). Si, por otra parte, el valor buscado está después del elemento medio, entonces el intervalo de la búsqueda se reduce a la mitad superior del arreglo. Luego se repite el proceso. En otras palabras, dentro de la mitad inferior reducida del arreglo se identifica el elemento medio, se conjetura si el valor buscado está antes o después del elemento medio y el intervalo de la búsqueda se reduce en consecuencia. Cada vez que se hace lo anterior, el problema se reduce a la mitad, lo cual permite volver a empezar rápidamente la búsqueda del valor buscado, en caso de haberlo. Separar el arreglo a la mitad es la parte "dividir" de la expresión "dividir y vencer". La determinación del valor buscado dentro de una de las mitades es la parte "vencer".

A continuación se verá cómo el método `binarySearch` implementa el algoritmo dividir y vencer. El método declara las variables `mid`, `low` y `high` que siguen la pista de los índices del elemento medio y los dos elementos en los extremos del intervalo de búsqueda del arreglo. Como ejemplo, observe el dibujo izquierdo en la figura 10.13. Al usar un ciclo `while`, el método calcula repetidamente `mid` (el índice del elemento medio) y comprueba si el valor del elemento `mid` es el valor buscado. Si el valor del elemento `mid` es el valor buscado, entonces el método devuelve el índice `mid`. En caso contrario, el método reduce el intervalo de la búsqueda a la mitad inferior o a la mitad superior del arreglo. Como ejemplo del proceso de reducción del intervalo de la búsqueda, vea la figura 10.13. El método repite el ciclo hasta que se encuentra el valor buscado o el intervalo de la búsqueda se reduce hasta el punto en que el índice `low` es mayor que el índice `high`.

10.8 Ordenamiento de un arreglo

Las computadoras son particularmente buenas para almacenar grandes cantidades de datos y acceder rápidamente a esos datos. Como se aprendió en la sección previa, la búsqueda binaria constituye una téc-

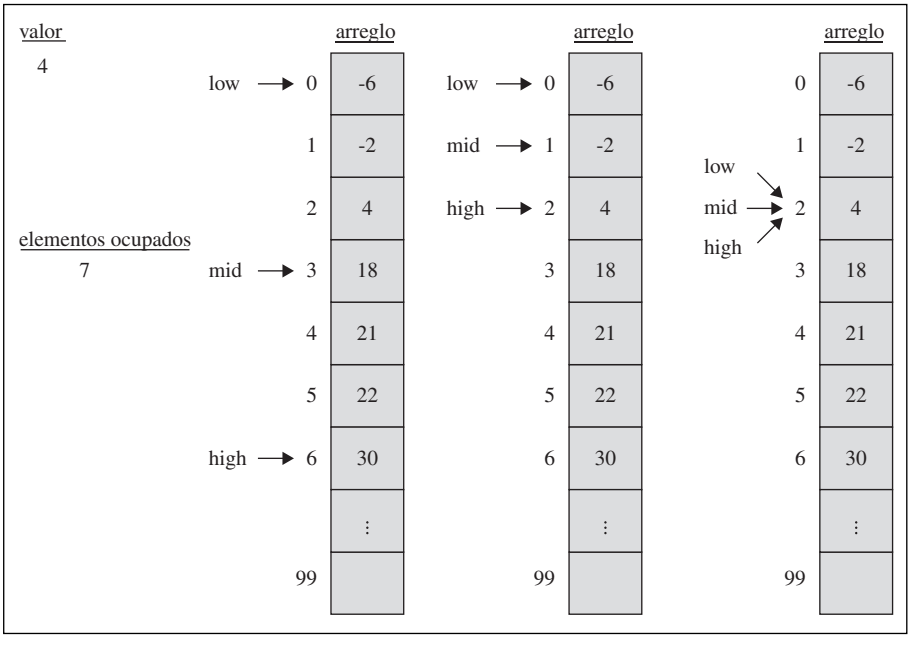


Figura 10.13 Ejemplo de ejecución del método `binarySearch` en la figura 10.12.

nica efectiva para encontrar y acceder rápidamente a datos. A fin de preparar los datos para una búsqueda binaria, es necesario que los datos estén ordenados. El ordenamiento de los datos se hace no sólo para efectos de búsqueda binaria. Las computadoras también ordenan datos de modo que sea más fácil exhibirlos de manera amigable para el usuario. Si el lector consulta sus correos electrónicos, seguramente están ordenados por fecha, donde el más reciente es el primero. La mayor parte de organizadores de correo electrónico también permiten ordenar los correos siguiendo otros criterios, como usando el remitente o el tamaño del correo. En esta sección se describen los fundamentos de cómo se realiza el ordenamiento. Primero se presenta un algoritmo de ordenamiento y luego, su implementación en forma de un programa que ordena los valores en un arreglo.

Ordenamiento por selección

Hay muchos algoritmos de ordenamiento con grados de complejidad y eficiencia variables. A menudo, la mejor manera de resolver un problema en una computadora es la forma en que una persona resolvería naturalmente el problema en forma manual. Para ilustrar esta idea, se mostrará cómo convertir uno de los algoritmos humanos comunes de ordenamiento de cartas en un programa de ordenamiento en Java.

Si en un juego de naipes una persona ordena las cartas, quizás use el algoritmo de *ordenamiento por selección*. Suponga que primero se ordenan las cartas más bajas. Se busca y selecciona la carta más baja y se mueve hacia el lado de cartas bajas del grupo de naipes en la mano. El lado de las cartas bajas del grupo de naipes es donde se tienen las cartas que ya han sido ordenadas. Luego, se busca la siguiente carta más baja, pero al hacerlo, sólo se buscan las cartas que están en la porción no ordenada del grupo de naipes. La carta encontrada se mueve a la segunda posición del lado de cartas bajas del grupo de naipes. Este proceso de búsqueda y movimiento se repite hasta que ya no quedan cartas en la porción no ordenada del grupo de naipes que se tiene en la mano.

Como primer paso en la implementación de la lógica del ordenamiento por selección, se analizará una solución en pseudocódigo. Antes se dijo "Este proceso de búsqueda y movimiento se repite". Siempre que hay una repetición, debe pensarse en la utilización de un ciclo. El siguiente algoritmo usa un ciclo para repetir el proceso de búsqueda y movimiento. Observe cómo `i` sigue la pista del sitio en que empieza la búsqueda. La primera vez que se ejecuta el ciclo, la búsqueda empieza en el primer elemento (en el índice 0). La próxima vez, la búsqueda empieza en la segunda posición. Cada vez que se ejecuta el ciclo, se encuentra el valor más pequeño y se mueve a la porción ordenada de la lista (la i indica el sitio en la lista en que se desea colocar el valor más pequeño).

```
for (i ← 0; i < list's length; i++)
    find the smallest value in the list from list[i] to the end of the list
    swap the found value with list[i]
```

Una imagen dice más que mil palabras, de modo que se proporciona una figura (10.14) que muestra en acción al algoritmo de ordenamiento por selección. Las cinco imágenes muestran las diversas etapas de una lista que está siendo ordenada usando el algoritmo de ordenamiento por selección. Las porciones en blanco de la lista están desordenadas. Toda la lista original a la izquierda es blanca, lo cual indica que

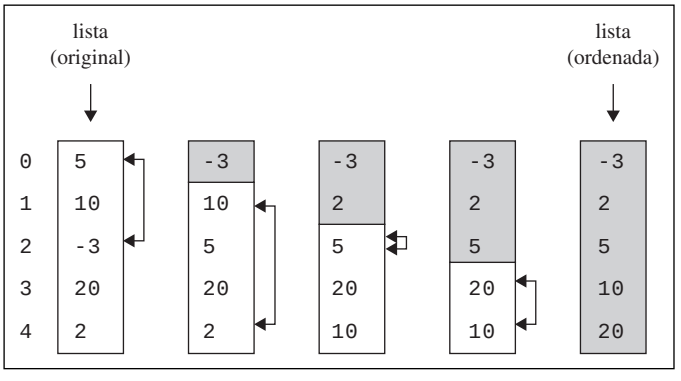


Figura 10.14 Ejemplo de ejecución del algoritmo Selection Sort.

está completamente desordenada. Las porciones sombreadas de la lista están ordenadas. Toda la lista a la derecha está sombreada, lo cual indica que está completamente ordenada. Las flechas bidireccionales muestran lo que ocurre después que se encuentra un valor más pequeño. El valor más pequeño (en la parte inferior de la flecha bidireccional) se desplaza hasta la parte superior de la porción desordenada de la lista. Por ejemplo, al ir de la primera a la segunda imagen, el valor más pequeño, −3, se ha desplazado hasta la posición del 5 en la parte superior de la porción desordenada de la lista.

A continuación se implementará una versión en Java del algoritmo de ordenamiento por selección. Puede usarse un método de instancia o un método de clase. En la sección previa, la búsqueda binaria se implementó con un método de clase. Para práctica adicional, aquí se hará lo mismo para el ordenamiento por selección. Al implementar este tipo de ordenamiento con un método de clase, resulta fácil llamarlo desde cualquier programa que requiera ordenar una lista de números: simplemente se prefija la llamada al método con el nombre de clase punto.

Vea la clase `Sort` en la figura 10.15. Observe cómo el método `sort` semeja bastante al pseudocódigo porque el método `sort` usa diseño arriba-abajo. En lugar de incluir el código de búsqueda del valor más pequeño dentro del método `sort`, éste llama al método de ayuda `indexOfNextSmallest`. En lugar de incluir el código de desplazamiento del elemento dentro del método `sort`, el método `sort` llama al método de ayuda `swap`. La única diferencia sustancial entre el método `sort` y el algoritmo de ordenamiento es que el ciclo `for` del método `sort` detiene la iteración un elemento antes de llegar a la parte inferior del arreglo. Esto es porque no es necesario llevar a cabo una búsqueda cuando ya se ha llegado al último elemento (ya se sabe que el último elemento es el valor mínimo para el resto de la lista). No fue necesario preocuparse sobre estos detalles de eficiencia con el algoritmo porque los algoritmos tratan más sobre lógica fundamental que con los detalles hechos por uno.

Paso de arreglos como argumentos

La figura 10.16 contiene un controlador para la clase `Sort` de la figura 10.15. La mayor parte del código es directa, pero por favor tome nota de argumento `studentIds` en la llamada al método `Sort.sort`. Éste es un ejemplo de paso de un arreglo a un método. Un arreglo es un objeto, y como tal, `StudentIds` es una referencia a un objeto de un arreglo. Como quizás el lector recuerde de la sección "Paso de referencias como argumentos" en el capítulo 7, un argumento de referencia (en la llamada a un método) y su parámetro de referencia correspondiente (en el encabezado de un método) señalan al mismo objeto. Así, en caso de que se actualice el objeto del parámetro de referencia desde el método, simultáneamente se actualiza el argumento del objeto de referencia en el módulo que llama. Al aplicar este razonamiento al programa `Sort`, cuando la referencia `StudentIds` se pasa al método `sort` y se ordena el arreglo ahí, no es necesario devolver el arreglo actualizado (ordenado) con una declaración `return`. Esto se debe a que la referencia `StudentIds` apunta al mismo objeto del arreglo que está ordenado dentro del método `sort`. Así, en el método `sort` no se incluye una declaración `return`, y el método funciona a la perfección.

Ordenamiento con un método API de Java

Verifique la eficiencia de los métodos API.

Cuando un arreglo tiene más de aproximadamente 20 elementos, es mejor usar un algoritmo más eficiente que el relativamente simple algoritmo Selection Sort recientemente descrito. Y con toda seguridad, el API de Java tiene un método de ordenamiento que usa un algoritmo de ordenamiento más eficiente. Se trata del método `sort` en la clase `Arrays`.

A continuación se presenta la estructura del método `sort` de la clase `Arrays`:

```
import java.util.Arrays;
...
  int[] studentIds = {. . .};
  ...
  Arrays.sort(studentIds);
```

Se recomienda el uso de este método API para ordenamientos difíciles. Se trata de un método sobrecargado, de modo que también funciona para arreglos de otros tipos de variables primitivas.

```
/*****************************************************************
* Sort.java
* Dean & Dean
*
* Esta clase usa un ordenamiento por selección para ordenar un simple arreglo.
*****************************************************************/

public class Sort
{
  public static void sort(int[] list)
  {
    int j;                       // índice del valor más pequeño

    for (int i=0; i<list.length-1; i++)
    {
      j = indexOfNextSmallest(list, i);
      swap(list, i, j);
    }
  } // end sort

  //***************************************************************

  private static int indexOfNextSmallest(
    int[] list, int startIndex)
  {
    int minIndex = startIndex;  // índice del valor más pequeño

    for (int i=startIndex+1; i<list.length; i++)
    {
      if (list[i] < list[minIndex])
      {
        minIndex = i;
      }
    } // end for
    return minIndex;
  } // end indexOfNextSmallest

  //***************************************************************

  private static void swap(int[] list, int i, int j)
  {
    int temp;                    // almacenamiento temporal del número

    temp = list[i];
    list[i] = list[j];
    list[j] = temp;
  } // end swap
} // end Sort
```

Figura 10.15 Clase `Sort` que contiene un método que ordena un arreglo de enteros en forma creciente.

10.9 Arreglos de dos dimensiones

Los arreglos son buenos para agrupar datos relacionados. Hasta el momento, los datos se han agrupado usando arreglos unidimensionales estándar. Si los datos relacionados están organizados en forma de tabla, considere el uso de un arreglo de dos dimensiones. En esta sección se describen los arreglos de dos dimensiones.

```
/*************************************************************
* SortDriver.java
* Dean & Dean
*
* Esto ejercita el ordenamiento por selección en la clase Sort.
*************************************************************/

public class SortDriver
{
  public static void main(String[] args)
  {
    int[] studentIds = {3333, 1234, 2222, 1000};

    Sort.sort(studentIds);
    for (int i=0; i<studentIds.length; i++)
    {
      System.out.print(studentIds[i] + " ");
    }
  } // end main
} // end SortDriver
```

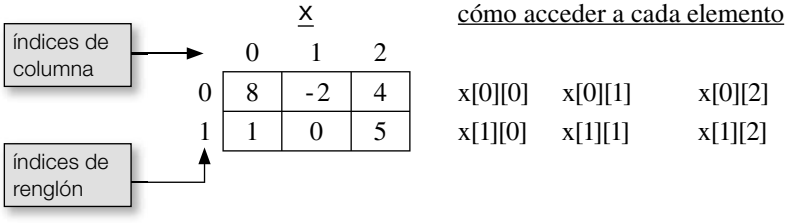
llamada al método `sort`

Figura 10.16 Controlador que ejercita el método `sort` en la figura 10.15.

Sintaxis de los arreglos de dos dimensiones

Los arreglos de dos dimensiones usan la misma sintaxis básica que los arreglos unidimensionales, excepto por un segundo par de corchetes (`[]`). Cada par de corchetes contiene un índice. Según la práctica de programación estándar, el primer índice identifica el renglón y el segundo identifica la posición de la columna dentro de un renglón.

Por ejemplo, a continuación se presenta un arreglo de dos renglones por tres columnas denominado `x`:



Los datos a la derecha, bajo el encabezado de la columna "cómo acceder", muestran cómo acceder a cada uno de los seis elementos en el arreglo. Así, para acceder al valor 5, en el índice de renglón 1 e índice de columna 2, se especifica `x[1][2]`.

Así como ocurre con arreglos unidimensionales, hay dos formas de asignar valores a los elementos de un arreglo de dos dimensiones. Puede usarse un iniciador del arreglo, donde la asignación de elementos es parte de la declaración del arreglo. O bien, pueden usarse declaraciones de asignación estándar, donde las declaraciones de asignación están separadas de la declaración y creación del arreglo. Primero se describirá la técnica del iniciador del arreglo. A continuación se presenta cómo es posible declarar el arreglo x de dos dimensiones anterior y asignar valores a sus elementos, usando un iniciador del arreglo:

iniciador para un arreglo de 2 renglones por 3 columnas

```
int[][] x = {{8, -2, 4}, {1, 0, 5}};
```

Observe que el iniciador del arreglo contiene dos grupos internos, donde cada grupo representa un renglón. `{8, -2, 4}` representa el primer renglón. `{1, 0, 5}` representa el segundo renglón. Observe que los

```
int[ ][ ] x = new int[2][3];
for (int i=0; i<x.length; i++)
{
  for (int j=0; j<x[0].length; j++)
  {
    System.out.print("Enter value for row " + i + ", col " + j + ": ");
    x[i][j] = stdIn.nextInt();
  } // end for j
} // end for i
```

Declara y crea un arreglo de 2 renglones por 3 columnas.

Asigna un valor al elemento en el renglón i, columna j.

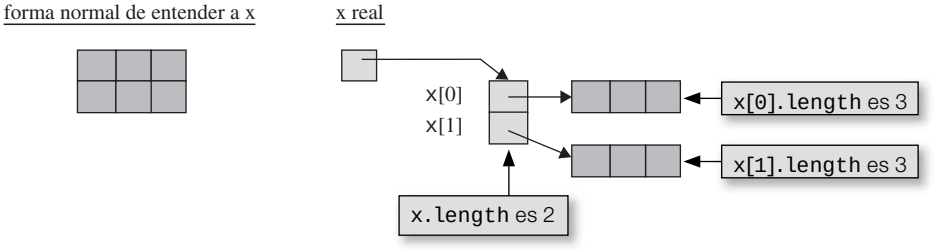
Figura 10.17 Asignación de valores en un arreglo de dos dimensiones usando ciclos `for` anidados y la propiedad `length`.

elementos y los grupos están separados por comas, y que cada grupo interno y todo el conjunto de grupos internos están escritos entre paréntesis de llaves.

La técnica del iniciador del arreglo puede usarse sólo si se conocen los valores asignados cuando el arreglo se declara por primera vez. En caso contrario, es necesario proporcionar declaraciones de asignación de los elementos del arreglo que estén separadas de la declaración y creación del arreglo. Por ejemplo, el fragmento de código en la figura 10.17 declara y crea el arreglo `x` en una declaración, y asigna valores a elementos `x` en una declaración por separado, dentro de los ciclos `for` anidados.

Cuando se trabaja con arreglos de dos dimensiones, suele ser muy común usar ciclos `for` anidados. En la figura 10.17 observe el ciclo `for` externo con variable de índice `i`, así como el ciclo `for` interno con variable de índice `j`. El ciclo `for` externo itera a través de cada renglón, en tanto el ciclo `for` interno itera a través de cada elemento en un renglón particular.

La primera línea de la figura 10.17 declara que `x` es un arreglo de 2 renglones por 3 columnas con 6 elementos en total. Así, sería de esperar que la propiedad `x.length` del primer ciclo `for` contenga un 6. No es así. Inclusive si resulta normal (y útil) pensar que `x` es una caja rectangular que contiene 6 elementos `int`, en realidad `x` es una referencia a un arreglo de 2 elementos y cada uno de los dos elementos es una referencia a su propio arreglo de 3 elementos de `int`s. La siguiente figura ilustra lo que se está hablando:



Puesto que en realidad `x` es una referencia a un arreglo de 2 elementos, `x.length` contiene el valor 2. O bien, si `x` se considera, como en el caso "normal" (figura superior izquierda), `x.length` contiene el número de renglones en `x`. Como puede verse antes, `x[0]` es una referencia a un arreglo de 3 elementos. Por tanto, `x[0].length` contiene el valor 3. O bien, si `x` se considera, como en el caso "normal" (figura superior izquierda), `x[0].length` contiene el número de columnas en `x`. Lo importante de todo esto es que la propiedad `length` puede usarse para iterar a través de los elementos en un arreglo de dos dimensiones. En la figura 10.17 observe cómo el primer ciclo usa `x.length` para iterar a través de cada renglón en `x`, y observe cómo el segundo ciclo usa `x[0].length` para iterar a través de cada columna en `x`.

Ejemplo

Estos conceptos de un arreglo de dos dimensiones se pondrán en práctica mediante el uso del arreglo de dos dimensiones en el contexto de un programa completo. El programa, elaborado para una compañía de aviación de Kansas y Missouri, indica a los clientes cuándo se espera la llegada de los vuelos a varios

aeropuertos de Kansas y Missouri. Usa un arreglo de dos dimensiones para almacenar los tiempos de vuelo entre ciudades, y la salida se exhibe como se muestra a continuación:

```
      Wch   Top   KC   Col   StL
Wch     0    22   30    42    55
Top    23     0   14    25    37
 KC    31     9    0    11    28
Col    44    27   12     0    12
StL    59    41   30    14     0
```

El vuelo de Topeka a Columbia se realiza en 25 minutos.

Renglones distintos corresponden a ciudades de origen diferente. Columnas diferentes corresponden a ciudades destino distintas. Las etiquetas son abreviaturas de los nombres de las ciudades: "Wch" representa Wichita, Kansas; "Top", Topeka, Kansas; "KC", Kansas City, Missouri; "Col", Columbia, Missouri; "StL", St. Louis, Missouri. Así, por ejemplo, el vuelo de Topeka a Columbia se realiza en 25 minutos. ¿En cuánto tiempo se hace el vuelo de regreso, de Columbia a Topeka? En 27 minutos. De Columbia a Topeka se hace más tiempo porque el recorrido es en dirección este-oeste, y los aviones tienen que enfrentar los vientos contrarios de la corriente estadounidense oeste-este.

A continuación se analizará el programa empezando con la clase `FlightTimesDriver` de la figura 10.18. Observe cómo el método `main` declara y crea una tabla `flightTimes` con un iniciador del arreglo de dos dimensiones. También, observe cómo el iniciador escribe por sí mismo cada renglón de la tabla en una línea. Esto no es un requerimiento del compilador, sino que se hace para obtener un código elegante y autodocumentado. Es autodocumentado porque los lectores pueden identificar fácilmente cada renglón de la tabla de datos al observar un solo renglón del código. Después de iniciar la tabla `flightTimes`, `main` inicia un arreglo unidimensional de nombres de ciudades y luego llama al constructor `flightTimes`, al método `displayFlightTimesTable` y al método `promptForlightTime`. A continuación se analizan el constructor y estos dos métodos.

Las figuras 10.19*a* y 10.19*b* contienen el meollo del programa: la clase `FlightTimes`. En la figura 10.19*a*, el constructor inicia los arreglos de las variables de instancia `flightTimes` y `cities` con los datos pasados a él por la llamada al constructor del controlador. Observe que asigna las referencias pasadas del arreglo `ft` y `c` a las variables de instancia usando el operador =. Previamente, se aprendió cómo usar un ciclo `for`, no el operador =, para hacer una copia de un arreglo. ¿Por qué es aceptable aquí el operador =? Porque no hay necesidad de hacer una segunda copia de estos arreglos. Después de la primera operación de asignación del constructor, la variable de instancia `flightTimes` de referencia al arreglo y el parámetro `ft` de referencia al arreglo apuntan al mismo objeto en el arreglo, lo cual es correcto. En forma semejante, después de la segunda operación de asignación del constructor, la variable de instancia `cities` de referencia al arreglo y el parámetro `c` de referencia al arreglo apuntan al mismo objeto en el arreglo.

El método `promptForFlightTime` de la figura 10.19*a* pide al usuario el nombre de una ciudad de partida y una ciudad de destino e imprime el tiempo de vuelo para ese vuelo. Más específicamente, imprime una leyenda de números y los nombres de sus ciudades asociadas (1 = Wichita, 2 = Topeka, etc.), pide al usuario que introduzca números para las ciudades de partida y de destino, e imprime el tiempo de vuelo entre las ciudades especificadas. Observe cómo los números de las ciudades introducidas por el usuario empiezan en 1, en lugar de 0 (1 = Wichita). Eso hace que el programa sea más amigable para el usuario porque normalmente la gente prefiere iniciar el conteo en uno y no en cero. Internamente, el programa almacena nombres de ciudades en un arreglo. Puesto que el arreglo empieza con un índice 0, el programa debe traducir entre números introducidos por el usuario (que empiezan en 1) e índices de ciudades en el arreglo (que empiezan en 0). Observe cómo se hace esto con `+1` y `-1` en el método `promptForFlightTime`.

El método `displayFlightTimesTable` en la figura 10.19*b* muestra la tabla con los tiempos de vuelo. Al hacerlo, emplea una técnica de formateo interesante. Primero considera las dos constantes locales nombradas, que son cadenas de formato definidas por separado. Desde hace algún tiempo, en los argumentos de las llamadas al método `printf` se han estado usando cadenas de formato literal insertadas en cadenas de texto. Pero en lugar de insertar cadenas de formato literal, algunas veces resulta más fácil comprender si se declaran como constantes nombradas por separado. Si uno se regresa y cuenta los espacios en la tabla de seis columnas de los tiempos de vuelo, se observa que el ancho de cada columna es exactamente de cinco espacios. Así, las etiquetas en la parte superior de las columnas y los números en las columnas deben formatearse de modo que se utilicen exactamente cinco espacios. Por tanto, la ca-

dena de formato para las etiquetas (`CITY_FMT_STR`) debe ser `"%5s"`, y la cadena de formato para las entradas enteras (`TIME_FMT_STR`) debe ser `"%5d"`. El uso de constantes nombradas para cadenas de formato permite que cada cadena de formato se use en muchos sitios, lo cual hace más fácil y seguro modificarlas en cualquier instante ulterior: simplemente se cambian los valores asignados a las constantes nombradas al inicio del método.

En el método `displayFlightTimesTable` observe los encabezados de los tres ciclos `for`. En todos se usa la propiedad `length` para sus condiciones de terminación. Puesto que `length` contiene a 5, el

```
/**************************************************************************
* FlightTimesDriver.java
* Dean & Dean
*
* Esto administra una tabla de tiempos de vuelo entre ciudades.
**************************************************************************/

public class FlightTimesDriver
{
  public static void main(String[] args)
  {
    int[][] flightTimes =
    {
      {0, 22, 30, 42, 55},
      {23, 0, 14, 25, 37},
      {31, 9, 0, 11, 28},
      {44, 27, 12, 0, 12},
      {59, 41, 30, 14, 0}
    };
    String[] cities = {"Wch", "Top", "KC", "Col", "StL"};
    FlightTimes ft = new FlightTimes(flightTimes, cities);

    System.out.println("\ntiempos de vuelo para Aerolíneas KansMo:\n");
    ft.displayFlightTimesTable();
    System.out.println();
    ft.promptForFlightTime();
  } // end main
} // end class FlightTimesDriver
```

Sesión muestra:

```
tiempos de vuelo para Aerolíneas KansMo:

        Wch  Top    KC  Col  StL
  Wch     0   22    30   42   55
  Top    23    0    14   25   37
  KC     31    9     0   11   28
  Col    44   27    12    0   12
  StL    59   41    30   14    0

1 = Wch
2 = Top
3 = KC
4 = Col
5 = StL
Enter departure city's number: 5
Enter destination city's number: 1
Flight time = 59 minutes.
```

Figura 10.18 Controlador de la clase `FlightTimes` en las figuras 10.19*a* y 10.19*b*.

programa debe ejecutarse correctamente si las condiciones de terminación de la longitud se sustituyen con 5 duramente codificados. Pero no lo haga. El uso de la propiedad `length` hace más *escalable* la implementación. Escalable significa que es fácil cambiar la cantidad de datos que usa el programa. Por ejemplo, en el programa `FlightTimes`, el uso de una condición de terminación de ciclo `cities.length` significa que si se modifica el número de ciudades en el programa, éste sigue funcionando bien.

Arreglos multidimensionales

Los arreglos pueden tener más de dos dimensiones. Los arreglos con tres o más dimensiones usan la misma sintaxis básica, excepto que tienen corchetes adicionales. El primer par de corchetes corresponde

```
/**********************************************************************
* FlightTimes.java
* Dean & Dean
*
* Esto administra una tabla de tiempos de vuelo entre ciudades.
**********************************************************************/

import java.util.Scanner;

public class FlightTimes
{
  private int[][] flightTimes; // tabla de tiempos de vuelo
  private String[] cities;     // ciudades en la tabla flightTimes

  //*******************************************************************

  public FlightTimes(int[][] ft, String[] c)
  {
    flightTimes = ft;
    cities = c;
  }

  //*******************************************************************

  // Desplegado para el usuario solicitando ciudades y la impresión
  // del tiempo de vuelo asociado.

  public void promptForFlightTime()
  {
    Scanner stdIn = new Scanner(System.in);
    int departure;   // índice de la ciudad de origen
    int destination; // índice de la ciudad de destino

    for (int i=0; i<cities.length; i++)
    {
      System.out.println(i+1 + " = " + cities[i]);
    }
    System.out.print("Introducir el número de la ciudad de salida: ");
    departure = stdIn.nextInt() - 1;
    System.out.print("Introducir el número de la ciudad de destino: ");
    destination = stdIn.nextInt() - 1;
    System.out.println("Flight time = " +
      flightTimes[departure][destination] + " minutos.");
  } // end promptForFlightTime
```

Imprime la leyenda del número de ciudad.

Figura 10.19*a* Clase `FlightTimes` que exhibe los tiempos de vuelo entre ciudades, parte A.

```
  //*****************************************************************

  // Este método imprime una tabla con todos los tiempos de vuelo.

  public void displayFlightTimesTable()
  {
    final String CITY_FMT_STR = "%5s";
    final String TIME_FMT_STR = "%5d";

    System.out.printf(CITY_FMT_STR, ""); // vacía la esquina superior
    izquierda para (int col=0; col<cities.length; col++)
    {
      System.out.printf(CITY_FMT_STR, cities[col]);
    }
    System.out.println();

    for (int row=0; row<flightTimes.length; row++)
    {
      System.out.printf(CITY_FMT_STR, cities[row]);
      for (int col=0; col<flightTimes[0].length; col++)
      {
        System.out.printf(TIME_FMT_STR, flightTimes[row][col]);
      }
      System.out.println();
    } // end for
  } // end displayFlightTimesTable
} // end class FlightTimes
```

cadenas de formato

Figura 10.19*b* Clase `flightTimes` que muestra los tiempos de vuelo entre ciudades, parte B.

a la escala más grande, y cada par subsiguiente de nidos de corchetes dentro del par previo, corresponde a niveles progresivamente menores de la escala. Por ejemplo, suponga que la línea aérea Missouri-Kansas decide "dar luz verde" y expande su flota con nuevos aviones de energía solar y con nuevos aviones de energía eólica que consumen hidrógeno. Los tiempos de vuelo de los nuevos aviones son diferentes a los tiempos de vuelo de los aviones originales que funcionan con combustible normal. Así, requieren sus propias tablas de tiempos de vuelo. La solución consiste en crear un arreglo tridimensional donde la primera dimensión especifique el tipo de avión: 0 para los aviones a combustible normal, 1 para los aviones de energía solar y 2 para los aviones a energía eólica. A continuación se muestra cómo declarar la nueva variable de instancia del arreglo tridimensional `FlightTimes`:

```
private int[][][] flightTimes;
```

10.10 Arreglos de objetos

En la sección previa se aprendió que un arreglo de dos dimensiones es realmente un arreglo de referencias donde cada referencia apunta a un objeto de un arreglo. A continuación se considerará un escenario relacionado. Se considerará un arreglo de referencias donde cada referencia apunta a un objeto definido por el programador. Por ejemplo, suponga que se desea almacenar el total de ventas para cada empleado de ventas en una tienda departamental. Si la empleada de ventas Amanda vende dos artículos por $55.45 y $22.01, entonces se quiere almacenar 77.46 como valor total de sus ventas. Los datos de venta del empleado pueden almacenarse en un arreglo, `clerks`, donde cada elemento contiene una referencia a un objeto `SalesClerk`. Cada objeto `SalesClerk` contiene el nombre de un empleado y las ventas totales de ese empleado. La figura 10.20 ilustra lo que se está hablando.

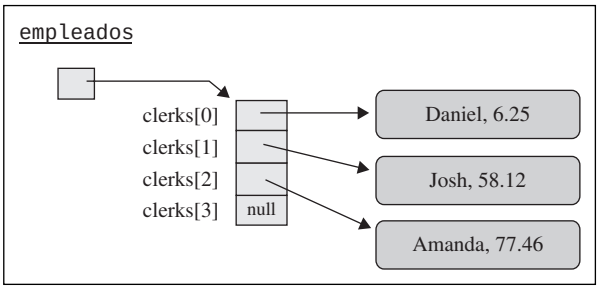


Figura 10.20 Arreglo de objetos que almacena datos de los empleados de ventas.

El arreglo `clerks` es un arreglo de referencias. Sin embargo, la mayoría de quienes están en el negocio se refieren a él como un arreglo de objetos, que es lo que también se hace aquí. Un arreglo de objetos no es tan distinto de un arreglo de primitivos. En ambos casos, a cada elemento del arreglo se accede con corchetes (por ejemplo, `clerks[0]`, `clerks[1]`). Sin embargo, hay algunas diferencias que deben tomarse en cuenta, y estas diferencias constituyen el centro de interés de esta sección.

Necesidad de instanciar arreglos de objetos y los objetos en ese arreglo

Con un arreglo de primitivos se efectúa una instanciación: se instancia el objeto del arreglo y ya está. Pero con un arreglo de objetos, es necesario instanciar el objeto del arreglo y también es necesario instanciar cada objeto elemento almacenado en el arreglo. Es fácil olvidar el segundo paso, la instanciación de objetos elemento individuales. Si se olvida, entonces los elementos contienen valores por defecto de `null`, como se ilustra con `clerk[3]` en la figura 10.20. Para la parte vacía de un arreglo parcialmente lleno, `null` está bien, pero para la parte de un arreglo que se supone está lleno, es necesario recubrir `null` con una referencia a un objeto. A continuación se presenta un ejemplo de cómo crear un arreglo de objetos: más específicamente, de cómo crear el arreglo de objetos `clerks` que se muestra en la figura 10.20. Observe las instanciaciones por separado, con el operador `new`, para el arreglo `clerks` y para cada objeto `SalesClerk`.

```
SalesClerk[] clerks = new SalesClerk[4];
clerks[0] = new SalesClerk("Daniel", 6.25);
clerks[1] = new SalesClerk("Josh", 58.12);
clerks[2] = new SalesClerk("Amanda", 77.46);
```

Imposibilidad de acceder directamente a datos de un arreglo

Con un arreglo de primitivos es posible acceder directamente a los datos del arreglo: los primitivos. Por ejemplo, el siguiente fragmento de código muestra cómo es posible asignar e imprimir el primer valor de lluvia en un arreglo `rainfall`. Observe cómo se accede directamente al valor con `rainfall[0]`.

```
double[] rainfall = new double[365];
rainfall[0] = .8;
System.out.println(rainfall[0]);
```

Por el contrario, con un arreglo de objetos, en condiciones normales no es posible acceder directamente a los datos del arreglo: las variables dentro de los objetos. Puesto que las variables dentro de los objetos suelen ser `private`, normalmente es necesario llamar a un constructor o un método para acceder a ellas. Por ejemplo, el siguiente fragmento de código muestra cómo usar un constructor para asignar Daniel y 6.25 al primer objeto en el arreglo `clerks`. También muestra cómo es posible usar métodos de acceso para imprimir el nombre del objeto y los datos de venta.

```
SalesClerk[] clerks = new SalesClerk[4];
clerks[0] = new SalesClerk("Daniel", 6.25);
System.out.println(
  clerks[0].getName() + ", " + clerks[0].getSales());
```

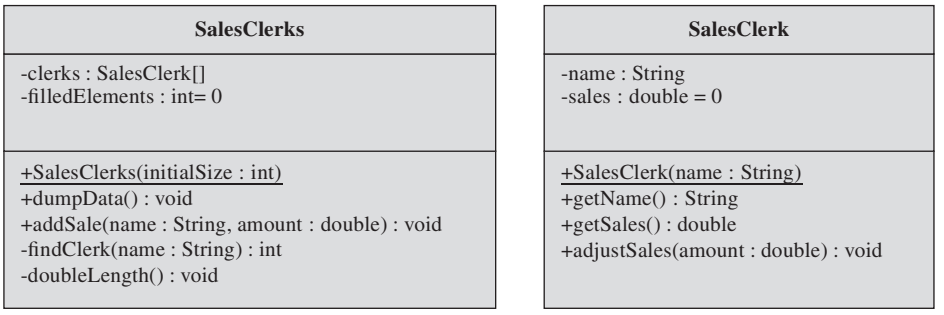


Figura 10.21 Diagrama de clase UML para el programa SalesClerks.

Programa SalesClerks

Empiece con un diagrama de clase UML para obtener una comprensión en perspectiva.

A continuación se implementará un programa completo que agrega ventas e imprime ventas para un grupo de empleados de ventas en una tienda departamental. Como de costumbre, primero se adquiere una comprensión en perspectiva de la cuestión al presentar un diagrama de clase UML. El diagrama de clase en la figura 10.21 muestra dos clases. La clase `SalesClerks` representa datos de ventas para toda la tienda departamental, y la clase `SalesClerk` representa las ventas totales para un empleado de ventas particular.

La clase `SalesClerk` contiene dos variables de instancia: `clerks` y `filledElements`; `clerks` es un arreglo de objetos `SalesClerk`; `filledElements` almacena el número de elementos que se han colocado hasta el momento en el arreglo `clerks`. Para un ejemplo de `filledElements`, consulte la figura 10.20, donde `filledElements` debe ser 3. El constructor `SalesClerks` instancia el arreglo `clerks`, usando el parámetro `initialSize` del constructor para el tamaño del arreglo.

La clase `SalesClerk` contiene cuatro métodos: `dumpData`, `addSale`, `findClerk` y `doubleLength`. El método `dumpData` es el más directo de los cuatro. Imprime todos los datos en el arreglo `clerks`. El término *dump* (vertedero o vaciado) es un término de computación que hace referencia a un simple despliegue (sin formatear) de los datos de un programa. Vea el método `dumpData` en la figura 10.22*b* y compruebe que imprime los datos en el arreglo `clerks`.

El método `addSale` procesa una venta para un empleado de ventas particular. Más específicamente, el método `addSale` encuentra el empleado de ventas especificado por su parámetro `name` y actualiza el total de ventas de ese empleado con el valor especificado por su parámetro `amount`. Para encontrar al empleado de ventas, al método `addSale` llama al método de ayuda `findClerk`. Este método realiza una búsqueda secuencial a través del arreglo `clerks` y devuelve el índice del empleado encontrado, o -1 si no se encuentra al empleado. Si no se encuentra al empleado, `addSale` agrega un nuevo objeto `SalesClerk` al arreglo `clerks` a fin de almacenar la nueva transacción de ventas. Al agregar un nuevo objeto `SalesClerk` al arreglo `clerks`, `addSale` comprueba para asegurarse de que en el arreglo `clerks` haya espacio disponible para el nuevo objeto `SalesClerk`. Si el arreglo `clerks` está completamente lleno (es decir, que `filledElements` es igual a `clerks.length`), entonces `addSale` debe hacer algo para proporcionar más elementos. Ahí es donde el método de ayuda `doubleLength` llega al rescate.

El método `doubleLength`, como sugiere su nombre, duplica el tamaño del arreglo `clerks`. Para hacer lo anterior, instancia un nuevo arreglo, `clerks2`, cuya longitud es el doble de la longitud del arreglo `clerks` original. Luego, copia todos los datos del arreglo `clerks` en los elementos de menor numeración en el arreglo `clerks2`. Finalmente, asigna el arreglo `clerks2` al arreglo `clerks`, de modo que el arreglo `clerks` apunte a un nuevo arreglo más largo. Vea los métodos `addSale`, `findClerk` y `doubleLength` en las figuras 10.22*a* y 10.22*b*, y compruebe que hacen lo que se supone que deben hacer.

La clase `SalesClerk`, que se muestra en el lado derecho de la figura 10.21, es bastante directo. Contiene dos variables de instancia, `name` y `sales`, para el nombre del agente de ventas y las ventas totales del empleado. Contiene dos métodos de acceso: `getName` y `getSales`. Contiene un método

```
/*****************************************************************
* SalesClerks.java
* Dean & Dean
*
* Esta clase almacena nombres y ventas de empleados de ventas.
*****************************************************************/

class SalesClerks
{
  private SalesClerk[] clerks;     // contiene nombres y ventas
  private int filledElements = 0;  // número de elementos llenos

  //***************************************************************

  public SalesClerks(int initialSize)
  {
    clerks = new SalesClerk[initialSize];
  } // end SalesClerks constructor

  //***************************************************************

  // Procesa una venta para el empleado cuyo nombre ha sido pasado.
  // Si el nombre no está ya en el arreglo clerks, crea un nuevo objeto e
  // inserta una referencia a éste en el siguiente elemento del arreglo,
  // duplicando la longitud del arreglo en caso de ser necesario.

  public void addSale(String name, double amount)
  {
    int clerkIndex = findClerk(name);

    if (clerkIndex == -1)          // agrega un nuevo empleado
    {
      if (filledElements == clerks.length)
      {
        doubleLength();
      }
      clerkIndex = filledElements;
      clerks[clerkIndex] = new SalesClerk(name);
      filledElements++;
    } // end if

    clerks[clerkIndex].adjustSales(amount);
  } // end addSale
```

Figura 10.22*a* Clase `SalesClerks`, parte A.

`adjustSales` que actualiza las ventas totales del empleado al sumar la `amount` pasada a la variable de instancia `sales`. Observe la clase `SalesClerk` en la figura 10.23 y compruebe que hace lo que se supone que debe hacer.

Luego considere el método `main` en la clase `SalesClerkDriver` en la figura 10.24. En una declaración, instancia un objeto `SalesClerks`, pasando al constructor `SalesClerks` un valor de longitud inicial del arreglo igual a 2. Luego, repetidamente solicita al usuario la introducción del nombre del empleado y el valor de las ventas y llama al método `addSale` para insertar los datos de entrada en el objeto `SalesClerks`. El ciclo termina cuando el usuario introduce una q para el nombre siguiente. Luego, el método `main` llama a `dumpData` para exhibir los datos acumulados de las ventas.

```
  //*************************************************************

  // Imprime todos los datos: nombre del empleado y ventas.

  public void dumpData()
  {
    for (int i=0; i<filledElements; i++)
    {
      System.out.printf("%s: %6.2f\n",
        clerks[i].getName(), clerks[i].getSales());
    }
  } // end dumpData

  //*************************************************************

  // Busca el nombre dado. Si lo encuentra, devuelve el índice.
  // En caso contrario, regresa -1.

  private int findClerk(String name)
  {
    for (int i=0; i<filledElements; i++)
    {
      if (clerks[i].getName().equals(name))
      {
        return i;
      }
    } // end for
    return -1;
  } // end findClerk

  //*************************************************************

  // Duplica la longitud del arreglo.

  private void doubleLength()
  {
    SalesClerk[] clerks2 = new SalesClerk[2 * clerks.length];
    System.arraycopy(clerks, 0, clerks2, 0, clerks.length);
    clerks = clerks2;
  } // end doubleLength
} // end class SalesClerks
```

Figura 10.22*b* Clase `SalesClerks`, parte B.

10.11 La clase `ArrayList`

Como ha aprendido el lector a lo largo de todo este capítulo, los arreglos permiten trabajar con una lista ordenada de datos relacionados. Los arreglos funcionan a la perfección para muchas listas; no obstante, si se tiene una lista donde es difícil pronosticar el número de elementos, no funcionan tan bien. Si se desconoce el número de elementos, es necesario 1) empezar con un tamaño de arreglo suficientemente grande para permitir la posibilidad de un número muy grande de elementos, o 2) crear un nuevo arreglo más grande cada que el arreglo esté lleno y se requiera más espacio para más elementos. La primera solución representa un desperdicio de memoria de la computadora, ya que requiere asignar espacio para un gran arreglo en el cual la mayor parte de los elementos no se usan. La segunda solución es lo que se hizo en el programa `SalesClerks` del método `doubleLength`. Funciona bien en términos de ahorro de memoria, pero requiere que el programador haga trabajo extra (al escribir el código que crea un arreglo más grande).

```
/*****************************************************************
* SalesClerk.java
* Dean & Dean
*
* Esta clase almacena y recupera los datos de un empleado de ventas.
*****************************************************************/

public class SalesClerk
{
  private String name;          // nombre del empleado
  private double sales = 0.0;  // ventas totales del empleado

  //*************************************************************

  public SalesClerk(String name)
  {
    this.name = name;
  }

  //*************************************************************

  public String getName()
  {
    return name;
  }

  public double getSales()
  {
    return sales;
  }

  //*************************************************************

  // Ajusta las ventas totales del empleado al sumar la venta que se ha pasado.

  public void adjustSales(double amount)
  {
    sales += amount;
  }
} // end class SalesClerk
```

Figura 10.23 Clase `SalesClerk`.

Para ayudar con listas donde es difícil pronosticar el número de elementos, el personal de Sun presentó la clase `ArrayList`. Esta clase se construye usando un arreglo, pero el arreglo está oculto en el fondo, de modo que no es posible acceder directamente a él. Con un arreglo en el fondo, la clase `ArrayList` es capaz de proporcionar la funcionalidad fundamental que conlleva un arreglo estándar. Con sus métodos, la clase `ArrayList` es capaz de proporcionar funcionalidad adicional que ayuda cuando se desconoce el número de elementos. En esta sección se analiza cómo crear una `ArrayList` y cómo usar sus métodos.

Cómo crear una `ArrayList`

La clase `ArrayList` está definida en el paquete API `java.util` de Java, de modo que para usar la clase, es necesario proporcionar una declaración `import` como ésta:

```
import java.util.ArrayList;
```

```
/*****************************************************************
* SalesClerksDriver.java
* Dean & Dean
*
* Esto controla la clase SalesClerks.
*****************************************************************/

import java.util.Scanner;

public class SalesClerksDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    SalesClerks clerks = new SalesClerks(2);
    String name;

    System.out.print("Introducir el nombre del empleado (q para salir): ");
    name = stdIn.nextLine();
    while (!name.equals("q"))
    {
      System.out.print("Introducir el nombre del empleado (q para salir): ");
      clerks.addSale(name, stdIn.nextDouble());
      stdIn.nextLine();          // flush newline
      System.out.print("Enter clerk's name (q to quit): ");
      name = stdIn.nextLine();
    } // end while
    clerks.dumpData();
  } // end main
} // end SalesClerksDriver
```

Sesión muestra:

```
Introducir el nombre del empleado (q para salir): Daniel
Introducir la cantidad de la venta: 6.25
Introducir el nombre del empleado (q para salir): Josh
Introducir la cantidad de la venta: 58.12
Introducir el nombre del empleado (q para salir): Amanda
Introducir la cantidad de la venta: 40
Introducir el nombre del empleado (q para salir): Daniel
Introducir la cantidad de la venta: -6.25
Introducir el nombre del empleado (q para salir): Josh
Introducir la cantidad de la venta: 12.88
Introducir el nombre del empleado (q para salir): q
Daniel:   0.00
Josh:  71.00
Amanda:  40.00
```

Figura 10.24 Controlador para el programa `SalesClerks` en las figuras 10.22*a*, 10.22*b* y 10.23.

Para iniciar una variable de referencia `ArrayList`, se usa esta sintaxis:

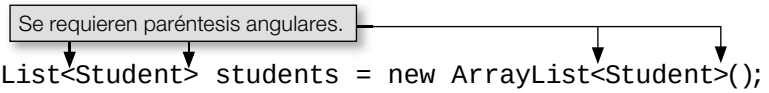
`ArrayList<`*tipo de elemento*`>` *variable de referencia* `= new ArrayList<`*tipo de elemento*`>();`[3]

[3] Es legal omitir <*tipo de elemento*> cuando se usan `ArrayLists`. Si se omite, entonces la `ArrayList` puede almacenar diferentes tipos de elementos. Esto puede parecer emocionante, pero no se requiere con tanta frecuencia. Y hay desventajas si se omite <*tipo de elemento*>:

a) Obliga al programador a usar el operador tipo cast al asignar un elemento extraído a una variable.

b) Elimina la comprobación de tipo para asignar valores a la `ArrayList` (puesto que entonces es legal asignar cualquier tipo).

Observe los paréntesis angulares en *tipo de elemento* y en *variable de referencia*. Los paréntesis angulares forman parte de la sintaxis requerida. Como se indica con las cursivas, *tipo de elemento* y *variable de referencia* son descripciones. Normalmente, los paréntesis angulares se usan para tales descripciones, aunque se intentará no hacerlo al describir la sintaxis de `ArrayList` porque los paréntesis angulares usados para descripciones podrían confundirse con los paréntesis angulares requeridos por `ArrayList`. Es necesario sustituir *tipo de elemento* con el tipo para los elementos de `ArrayList`. Es necesario sustituir *variable de referencia* con una variable de referencia real. Por ejemplo, suponga que se ha definido una clase `Student`, y que se quiere una `ArrayList` de objetos `Student`. He aquí cómo crear tal `ArrayList`, llamada `students`:



Además de los paréntesis angulares, hay dos cuestiones adicionales que destacan en el ejemplo anterior. Primero, no hay especificación de tamaño. Esto se debe a que los objetos `ArrayList` empiezan sin elementos y automáticamente crecen para dar cabida a cuantos elementos se les agreguen. Segundo, el tipo de elemento, `Student`, es un nombre de clase. En `ArrayList`, para el tipo de elemento es necesario especificar un nombre de clase, no un tipo primitivo. Especificar un nombre de clase significa que `ArrayList` sólo pueden contener referencias a objetos. No pueden contener primitivos, como `int` o `double`. Esto es técnicamente cierto, aunque hay una forma fácil de imitar el almacenamiento de primitivos en una `ArrayList`. En la siguiente sección se analizará este tema.

Agregación de elementos a una `ArrayList`

Para convertir una `ArrayList` vacía instanciada en algo útil es necesario agregarle elementos. Para agregar un elemento al final de una `ArrayList` se usa la sintaxis:

ArrayList-reference-variable`.add(`*item*`);`

El *item* que se agrega debe ser del mismo tipo que el tipo de elemento especificado en la declaración de la `ArrayList`. Quizás el tipo más simple de elemento objeto es una cadena, de modo que se empezará con una `ArrayList` de cadenas. Suponga que se quiere escribir un fragmento de código que crea este objeto `ArrayList`:

	colores
0	“rojo”
1	“verde”
2	“azul”

Intente escribir el código antes de continuar. Cuando haya terminado, compare su respuesta con esto:

```
import java.util.ArrayList;
. . .
ArrayList<String> colores = new ArrayList<String>();
colores.add("rojo");
colores.add("verde");
colores.add("azul");
```

El orden en que se agregan los elementos determina la posición de éstos. Puesto que primero se agregó “rojo”, está en la posición con índice 0. Como después se agregó “verde”, está en la posición con índice 1. En forma semejante, “azul” está en la posición con índice 2.

Encabezados API

Al describir la clase `ArrayList` se usarán *encabezados API* para presentar los métodos de la clase `ArrayList`. Como puede recordar del capítulo 5, API significa Interfaz de Programación de Aplicacio-

nes (*Application Programming Interface*), y los encabezados API son los encabezados de los códigos fuente para los métodos y constructores en la biblioteca Sun de las clases Java preconstruidas. Los encabezados API indican cómo usar los métodos y los constructores al mostrar sus parámetros y tipos de retorno. Por ejemplo, éste es el encabezado API para el método `pow` de la clase `Math`:

```
public static double pow(double num, double power)
```

La línea anterior indica todo lo necesario para usar el método `pow`. Para llamar a este método, deben pasarse dos argumentos `double`: un argumento para la base y otro para la potencia. El modificador `static` indica que es necesario anteponer (prefijar) la llamada con el nombre de la clase y luego escribir un punto. El valor de retorno `double` indica insertar la llamada al método en un sitio que pueda usar un valor `double`. He aquí un ejemplo que calcula el volumen de una esfera:

```
double volume = 1.333333333 * Math.PI * Math.pow(radius, 3)
```

Cómo introducir elementos en una `ArrayList`

Con arreglos estándar, para leer y actualizar un elemento se usan corchetes. Pero con una `ArrayList` no se usan corchetes. En lugar de ello, para leer el valor de un elemento se usa un método read y para actualizarlo, un método `set`.

He aquí el encabezado API para el método `get` de `ArrayList`:

```
public E get(int index)
```

El parámetro `index` especifica la posición del elemento deseado dentro del objeto que llama en `ArrayList`. Por ejemplo, la siguiente llamada a un método recupera el segundo elemento en una `ArrayList` `colors`:

```
colors.get(1);
```

Si el parámetro `index` se refiere a un elemento inexistente, entonces ocurre un error de tiempo de ejecución. Por ejemplo, si `colors` contiene tres elementos, entonces genera el error de tiempo de ejecución:

```
colors.get(3);
```

En el encabezado API del método `get`, observe el error `E` de tipo de retorno:

```
public E get(int index)
```

La `E` significa "elemento" y representa el tipo de datos de los elementos de la `ArrayList`, sin importar qué tipos de datos son. Entonces, si se declara que una `ArrayList` contiene elementos cadena, entonces el método `get` devuelve un valor cadena, y si se declara que una `ArrayList` tiene elementos `Student`, entonces el método `get` devuelve un valor `Student`. La `E` en el encabezado del método `get` es un nombre genérico para un tipo de elemento. El uso de un nombre genérico para un tipo es un concepto importante que se presenta de nuevo con otros métodos. Es suficientemente importante para justificar una analogía pedagógica.

El uso de un tipo de retorno genérico es como decir que una persona entra en un supermercado para comprar "alimentos". Es mejor usar un término genérico como alimentos que un término específico como brócoli. ¿Por qué? Porque podría terminarse adquiriendo *Goma de Mascar Princesa* en lugar de brócoli. Al especificar los alimentos genéricos como el "tipo de retorno", ya no hay la posibilidad de adquirir Goma de Mascar Princesa en lugar de brócoli, como parece que entiende su prebecario.[4]

El uso de un nombre genérico para un tipo es posible con `ArrayList` porque la clase `ArrayList` se define como una *clase genérica*, al usar `<E>` en su encabezado de clase:

```
public class ArrayList<E>
```

Para usar `ArrayList` no es necesario comprender los detalles de la clase genérica, pero si el lector desea conocerlos, debe visitar el sitio http://java.sun.com/docs/books/tutorial/java/generics/index.html

[4] Esta analogía está tomada de las aventuras de la vida real del prebecario Jordan Dean.

Cómo actualizar un elemento de una ArrayList

Ahora, como compañero del método get se tiene el método set. El método set permite asignar un valor a un elemento de una ArrayList. He aquí el encabezado API para el método set de ArrayList:

```
public E set(int index, E elem)
```

En el encabezado API del método set, el parámetro index especifica la posición del elemento motivo de interés. Si index se refiere a un elemento inexistente, entonces ocurre un error de tiempo de ejecución. Si index es válido, entonces set asigna el parámetro elem al elemento especificado. Observe que elem se declara con E para su tipo. Así como ocurre con el método set, la E representa el tipo de datos de los elementos de ArrayList. Así, elem es del mismo tipo que el tipo de los elementos de ArrayList. Este ejemplo ilustra lo que se está hablando:

```
String mixedColor;
ArrayList<String> colors = new ArrayList<String>();

colors.add("rojo");
colors.add("verde");
colors.add("azul");
mixedColor = colors.get(0) + colors.get(1);
colors.set(2, mixedColor);
```

Observe que mixedColor se declara como una cadena y que colors se declara como una ArrayList de cadenas. Así, en la última declaración cuando mixedColor se usa como el segundo argumento en la llamada al método set, el argumento es, en efecto, del mismo tipo que los elementos de color.

¿Puede el lector determinar a qué se parece la ArrayList colors después de la ejecución del fragmento de código? Haga un dibujo de la ArrayList colors antes de continuar. Una vez que termine, compare su respuesta con esto:

colores

0	"rojo"
1	"verde"
2	"rojoverde"

En el encabezado API del método set, observe el tipo de retorno, E. La mayor parte de los métodos reguladores (*mutators*)/set simplemente asignan un valor y eso es todo. Además de asignar un valor, el método set de ArrayList también devuelve un valor: el valor del elemento especificado previo al elemento que se está actualizando. Usualmente, no es necesario hacer nada con el valor original, de modo que simplemente se llama a set y el valor devuelto desaparece. Eso es lo que ocurre en el fragmento de código anterior. Pero si se desea hacer algo con el valor original, es fácil porque set lo devuelve.

Métodos ArrayList adicionales

Ahora ya se han explicado los métodos más importantes para la clase ArrayList. Hay algunos métodos más, y en la figura 10.25 se proporcionan encabezados API y descripciones breves para cinco de ellos. A medida que se lee la figura, es de esperar que el lector considere que la mayor parte de los métodos son directos. Sin embargo, puede ser necesario clarificar algunas cuestiones. Al buscar en una ArrayList la primera ocurrencia de un parámetro elem ya pasado, el método indexOf declara que el tipo de elem es Object. El tipo Object significa que el parámetro puede ser cualquier tipo de objeto. Por supuesto, si el tipo real del parámetro es diferente del tipo de elementos en la ArrayList, entonces la búsqueda de indexOf termina vacía y devuelve −1 para indicar que no se encontró elem. Por cierto, hay mucho más que decir sobre el tipo Object (en realidad es una clase Object) en el capítulo 13. Previamente se cubrió un método add de un parámetro que agrega un elemento al final de la ArrayList. El método sobrecargado add de dos parámetros en la figura 10.25 agrega un elemento en una posición especificada dentro de la ArrayList.

`public void add(int index, E elem)`
Empezando con la posición `index` especificada, el método `add` desplaza los elementos originales en y por arriba de la posición del índice a posiciones próximas con índices superiores. Luego inserta el parámetro `elem` en la posición `index` especificada.

`public int indexOf(Object elem)`
Busca la primera ocurrencia del parámetro `elem` dentro de la lista y devuelve la posición del índice del elemento encontrado. Si el elemento no se encuentra, el método `indexOf` devuelve −1.

`public boolean isEmpty()`
Devuelve `true` si la `ArrayList` no contiene elementos.

`public int lastIndexOf(Object elem)`
Busca la última ocurrencia del parámetro `elem` dentro de la lista y devuelve la posición del índice del elemento encontrado. Si el elemento no se encuentra, el método `indexOf` devuelve −1.

`public E remove(int index)`
Elimina y devuelve el elemento en la posición `index` especificada. Para manipular la ausencia del elemento eliminado, el método `remove` desplaza una posición todos los elementos con mayor índice hacia posiciones con índices menores.

`public int size()`
Devuelve el número de elementos que actualmente están en la `ArrayList`.

Figura 10.25 Encabezados y descripciones API para algunos métodos adicionales `ArrayList`.

Ejemplo survivor

A fin de reforzar lo que se ha aprendido hasta el momento, se considerará cómo una clase `ArrayList` se utiliza en un programa de trabajo completo. Observe el programa Survivor[5] en la figura 10.26. Crea una lista de miembros sobrevivientes de una tribu al instanciar un objeto `ArrayList` y llamando a `add` para agregar miembros de la tribu a la lista. Luego, al azar escoge uno de los miembros de la tribu y lo elimina de la lista. Imprime un mensaje de disculpas por el miembro eliminado de la tribu y un mensaje adicional para el resto de los miembros de la tribu.

Observe el formato de los miembros de la tribu en la línea de salida inferior en la figura 10.26: una lista cuyos elementos están separados por comas está escrita entre corchetes. ¿Puede el lector encontrar el código `Survivor` que imprime esta lista? Si busca corchetes y un ciclo, olvídelo; ahí no están. Entonces, ¿cómo se logra imprimir la lista entre corchetes? En la declaración final `println` que aparece en la parte inferior del programa, la `ArrayList tribe` está concatenada a una cadena. Esto provoca que la JVM haga algo de trabajo tras bambalinas. Si se intenta concatenar una `ArrayList` a una cadena o imprimir una `ArrayList`, la `ArrayList` devuelve una lista escrita entre corchetes (`[]`) de elementos de `ArrayList` separados por comas. Y esto es exactamente lo que ocurre cuando se ejecuta la última declaración en la figura 10.26.

10.12 Almacenamiento de primitivos en una `lista de arreglos`

Como ya se mencionó, las `ArrayList` almacenan referencias. Por ejemplo, en el programa Survivor, `tribe` es una `ArrayList` de cadenas, y las cadenas son referencias. Si es necesario almacenar primitivos en una `ArrayList`, puede hacerse directamente, pero si los primitivos están envueltos en clases envoltorio,[6] los objetos envueltos resultantes pueden almacenarse en una `ArrayList`. En esta sección se muestra cómo hacer lo anterior.

[5] Survivor es una marca registrada de CBS Broadcasting Inc.

[6] Si el lector quiere recordar su conocimiento sobre las clases envoltorio, debe consultar el capítulo 5.

```
/*********************************************************************
* Survivor.java
* Dean & Dean
*
* Esta clase crea una ArrayList de sobrevivientes.
* Escoge al azar un miembro de la tribu y lo elimina.
*********************************************************************/

import java.util.ArrayList;

public class Survivor
{
  public static void main(String[] args)
  {
    int loserIndex;
    String loser;
    ArrayList<String> tribe = new ArrayList<String>();

    tribe.add("Richard");
    tribe.add("Jerri");
    tribe.add("Colby");
    tribe.add("Amber");
    tribe.add("Rupert");
    loserIndex = (int) (Math.random() * tribe.size());
    loser = tribe.remove(loserIndex);
    System.out.println("Lo siento, " + perdedor +
      ". La tribu ha hablado. Debes irte de inmediato.");
    System.out.println("Remaining: " + tribe);
  } // end main
} // end Survivor
```

Salida típica:

```
Lo siento, Colby. La tribu ha hablado. Debes irte de inmediato.
Restantes: [Richard, Jerri, Amber, Rupert]
```

Figura 10.26 Programa Survivor.

Ejemplo Stock Average

El programa StockAverage en la figura 10.27 lee valores ponderados de la Bolsa y los almacena en una `ArrayList`. En términos simplificados, un valor ponderado de la Bolsa es el precio de mercado de una acción multiplicado por un número que escala ese precio hacia arriba o hacia abajo para reflejar la importancia de la compañía de acciones en el mercado global. Después que el programa StockAverage almacena los valores ponderados de la Bolsa en una `ArrayList`, el programa calcula el promedio de todos los valores ponderados de la Bolsa introducidos. ¿Por qué es idónea una `ArrayList` para calcular un promedio de la Bolsa? El tamaño de una `ArrayList` crece según sea necesario. Esto funciona bien para promedios de la Bolsa porque hay muchos promedios de la Bolsa (también denominados índices de la Bolsa), y para su cálculo se usan diferentes números de acciones. Por ejemplo, el Índice Industrial Dow Jones usa valores de la Bolsa de 30 compañías, mientras el índice Russell 3000 usa valores de la Bolsa de 3 000 compañías. Debido a que el programa StockAverage usa una `ArrayList`, funciona bien para ambas situaciones.

El programa StockAverage almacena valores de la Bolsa en una `ArrayList` denominada `stocks`. Los valores de la Bolsa se originan a través del usuario en forma de `doubles`, como 25.6, 36.0, etc. Como sabe el lector, las `ArrayList` no pueden almacenar primitivos; sólo pueden almacenar referencias. Así, el programa StockAverage envuelve los `doubles` en objetos envoltorio `Double` justo antes de almacenarlos en la `ArrayList stocks`. Como puede imaginarse, un *objeto envoltorio* es una instan-

cia de una clase envoltorio, y cada objeto envoltorio almacena un valor primitivo "envuelto". El lector no debe preocuparse mucho por los objetos envoltorio para `ArrayList`. Casi siempre es posible pretender que las `ArrayList` pueden contener primitivos. Ejemplo: la siguiente línea del programa Stock Average parece agregar un primitivo (`stock`) a la lista `ArrayList` `stocks`:

```
stocks.add(stock);
```

Lo que ocurre realmente tras bambalinas es que el primitivo `stock` se convierte automáticamente en un objeto envoltorio antes que se agregue a la `ArrayList` `stocks`. En realidad, sólo hay una cosa de la cual preocuparse cuando se trabaja con primitivos en una `ArrayList`. Cuando se crea un objeto `ArrayList` para contener valores primitivos, el tipo que se especifica en los corchetes debe ser la versión envuelta de tipo primitivo; es decir, `Double` en lugar de `double`, `Integer` en vez de `int` y así sucesivamente. Esta línea del programa StockAverage ilustra lo que se está hablando:

```
ArrayList<Double> stocks = new ArrayList<Double>();
```

Autoboxing y unboxing

En casi todos los sitios, es legal usar valores primitivos y objetos envoltorio como sinónimos. La forma en que funciona esto es que la JVM automáticamente envuelve valores primitivos y desenvuelve objetos envoltorio cuando es apropiado hacerlo. Por ejemplo, si la JVM ve un valor `int` a la derecha de una declaración de asignación y una variable `Integer` a la izquierda, piensa, hmmm, para que esto funcione, necesito convertir el valor `int` en un objeto envoltorio `Integer`. Luego saca su hule espuma para empacar cacahuates y cinta en rollo y envuelve el valor `int` en un objeto envoltorio `Integer`. Este proceso se denomina *autoboxing*. Por otra parte, si la JVM ve un objeto envoltorio `Integer` a la derecha de una declaración de asignación y una variable `int` a la izquierda, piensa, hmmm, para que esto funcione, tengo que extraer el valor `int` del objeto envoltorio `Integer`. Luego procede a rasgar la cubierta del objeto envoltorio `Integer` y obtiene el valor `int` que está en el interior. Este proceso se denomina *unboxing*.

De manera más formal, autoboxing es el proceso de envolver automáticamente un valor primitivo en una clase envoltorio idónea siempre que hay un intento por usar un valor primitivo en un sitio que espera una referencia. Consulte la declaración `stocks.add(stock);` en la figura 10.27. Esta declaración hace que ocurra autoboxing. La llamada al método `stocks.add` espera un argumento de referencia.

Específicamente, espera que el argumento sea una referencia a un objeto envoltorio `Double` (puesto que `stocks` se declara como una `ArrayList` de referencias `Double`). Cuando la JVM ve un argumento de valor primitivo (`stock`), automáticamente envuelve el argumento en una clase envoltorio `Double`.

De manera más formal, unboxing es el proceso de extracción automática de un valor primitivo de un objeto envoltorio siempre que hay un intento por usar un objeto envoltorio en un sitio que espera un primitivo. Consulte la declaración `stock = stocks.get(i);` en la figura 10.27. Esta declaración hace que ocurra unboxing. Puesto que `stock` es una variable primitiva, la JVM espera que le sea asignado un valor primitivo. Cuando la JVM ve un objeto envoltorio a la derecha de la declaración de asignación (`stocks` contiene objetos envoltorio `Double`) y `get(i)` recupera el `i`-ésimo de estos objetos envoltorios, automáticamente extrae el valor primitivo del objeto envoltorio.

Autoboxing y unboxing se realizan en forma automática tras bambalinas. Eso facilita el trabajo del programador. ¡Sí!

10.13 Ejemplo de `lista de arreglo` utilizando objetos anónimos y el ciclo for-each

Los *objetos anónimos* y los *ciclos for-each* son constructos de programación particularmente útiles cuando se usan junto con `ArrayList`. En esta sección se presentan detalles del ciclo for-each y de objetos anónimos al mostrar cómo se usan en el contexto de un programa `ArrayList`. Pero antes de abordar el programa se proporcionan introducciones breves para los dos nuevos constructos.

Usualmente, cuando se crea un objeto, la referencia al objeto se almacena inmediatamente en una variable de referencia. Así, es posible referirse al objeto más tarde mediante el uso del nombre de la va-

```
/*************************************************************
* StockAverage.java
* Dean & Dean
*
* Este programa usa una ArrayList para almacenar valores stock
* introducidos por el usuario. Imprime el valor promedio de la Bolsa.
*************************************************************/

import java.util.Scanner;
import java.util.ArrayList;

public class StockAverage
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    ArrayList<Double> stocks = new ArrayList<Double>();
    double stock;                          // un valor stock
    double stockSum = 0;                   // suma de los valores stock

    System.out.print("Introduzca un valor stock (-1 para salir): ");
    stock = stdIn.nextDouble();

    while (stock >= 0)
    {
      stocks.add(stock);
      System.out.print("Introduzca un valor stock (-1 para salir): ");
      stock = stdIn.nextDouble();
    } // end while

    for (int i=0; i<stocks.size(); i++)
    {
      stock = stocks.get(i);
      stockSum += stock;
    }

    if (stocks.size() != 0)
    {
      System.out.printf("\nAverage stock value = $%.2f\n",
        stockSum / stocks.size());
    }
  } // end main
} // end class StockAverage
```

Ésta debe ser una clase envoltorio, ¡no un tipo primitivo!

El autoboxing se lleva a cabo aquí.

El unboxing se lleva a cabo aquí.

Figura 10.27 Programa StockAverage que ilustra una `ArrayList` de objetos `Double`.

riable de referencia. Si se crea un objeto y la referencia al objeto no se asigna de inmediato a una variable de referencia, se ha creado un objeto anónimo. Se denomina anónimo porque carece de nombre.

Un ciclo for-each es una versión modificada del ciclo `for` tradicional. Puede usarse siempre que es necesario iterar a través de todos los elementos en una colección de datos. Una `ArrayList` es una colección de datos y, como tal, los ciclos for-each pueden usarse para iterar a través de todos los elementos en una `ArrayList`.

Ejemplo de una tienda que vende osos de juguete

Suponga que se desea modelar una tienda donde se venden osos de juguete a la medida. Para representar cada oso se requiere una clase `Bear`; para representar la tienda, una clase `BearStore`, y para "contro-

```
/*************************************************************
* Bear.java
* Dean & Dean
*
* Esta clase modela un oso de juguete.
*************************************************************/

public class Bear
{
  private final String MAKER; // fabricante del oso
  private final String TYPE;  // tipo de oso

  //*********************************************************

  public Bear(String maker, String type)
  {
    MAKER = maker;
    TYPE = type;
  }

  //*********************************************************

  public void display()
  {
    System.out.println(MAKER + " " + TYPE);
  }
} // end Bear class
```

Figura 10.28 Clase que representa un oso de juguete.

lar" el programa, una clase `BearStoreDriver`. Se empezará por analizar la clase `Bear` en la figura 10.28. La clase `Bear` define dos constantes de instancia nombradas que representan dos propiedades permanentes de un oso particular: 1) `MAKER`, el fabricante del oso, como Gund, y 2) `TYPE`, el tipo de oso, como "pooh" u "oso de camping enojado". Un constructor inicia estas dos constantes de instancia y un método `display` las muestra.

Ahora se analizará la primera parte de la clase `BearStore` que se muestra en la figura 10.29*a*. La clase `BearStore` tiene una variable de instancia, `bears`, que se declara como una `ArrayList` de referencias `Bear`. Contiene la colección de osos de juguete de la tienda. El método `addStdBear`s de la clase `BearStore` llena la `ArrayList` `bears` con un número específico de osos teddy normales. Ésta es la declaración que agrega un oso teddy normal a la `ArrayList`:

```
bears.add(new Bear("Acme", "teddy café"));
```

La declaración instancia un objeto `Bear` y pasa la referencia del objeto `Bear` a la llamada al método `bears.add`.

La declaración no asigna la referencia del objeto `Bear` a una variable de referencia `Bear`. Puesto que no hay asignación a una variable de referencia `Bear`, éste es un ejemplo de objeto anónimo. Como alternativa, la declaración hubiera podido escribirse con una variable de referencia `Bear` como ésta:

```
Bear stdBear = new Bear("Acme", "teddy café");
bears.add(stdBear);
```

Sin embargo, ¿por qué molestarse usando dos declaraciones en lugar de una? La nueva referencia del oso se almacena en la `ArrayList` `bears`, y es ahí donde se procesa. No es necesario almacenarla en un segundo sitio (por ejemplo, en la variable de referencia `stdBear`), de modo que en aras de la compactación del código, no lo haga.

```
/*****************************************************************
* BearStore.java
* Dean & Dean
*
* Esta clase implementa una tienda que vende osos de juguete.
*****************************************************************/

import java.util.Scanner;
import java.util.ArrayList;

public class BearStore
{
  ArrayList<Bear> bears = new ArrayList<Bear>();

  //***************************************************************

  // Llena la tienda con el número especificado de osos de juguete normales.

  public void addStdBears(int num)
  {
    for (int i=0; i<num; i++)
    {
      bears.add(new Bear("Acme", "teddy café"));
    }
  } // end addStdBears

  //***************************************************************

  // Llena la tienda con el número especificado de osos de juguete a la medida.

  public void addUserSpecifiedBears(int num)
  {
    for (int i=0; i<num; i++)
    {
      bears.add(getUserSpecifiedBear());
    }
  } // end addUserSpecifiedBears
```

objeto anónimo como argumento

El objeto anónimo devuelto se convierte en el argumento en esta llamada al método.

Figura 10.29*a* Clase que implementa una tienda que vende osos de juguete, parte A.

Ahora se analizará la parte inferior de la clase `BearStore` que se muestra en la figura 10.29*a*. El método `getUserSpecifiedBear` de la clase `BearStore` solicita al usuario el nombre y el tipo de un fabricante de un oso a la medida, y devuelve el oso recientemente creado. Ésta es la declaración `return`:

```
return new Bear(maker, type);
```

Observe que para el nuevo oso no hay variable de referencia. Así, el nuevo oso se considera como un objeto anónimo. La declaración `return` devuelve el nuevo oso al método `addUserSpecifiedBears`, donde se agrega a la `ArrayList bears`.

Cuándo usar un objeto anónimo

El programa Bearstore contiene varios ejemplos específicos del uso de objetos anónimos. En general, se observa que los objetos anónimos se usan en dos circunstancias:

1. Cuando un objeto de reciente creación se pasa a un método o constructor. Por ejemplo:

   ```
   bears.add(new Bear("Gund", "Teddy"));
   ```

```
  //*******************************************************

  // Solicita al usuario el nombre y el tipo del fabricante y devuelve el oso.

  private Bear getUserSpecifiedBear()
  {
    Scanner stdIn = new Scanner(System.in);
    String maker, type;

    System.out.print("Enter bear's maker: ");
    maker = stdIn.nextLine();
    System.out.print("Enter bear's type: ");
    type = stdIn.nextLine();
    return new Bear(maker, type);
  } // end getUserSpecifiedBear

  //*******************************************************

  // Imprime todos los osos en la tienda.

  public void displayInventory()
  {
    for (Bear bear : bears)
    {
      bear.display();
    }
  } // end displayInventory

  //*******************************************************

  public static void main(String[] args)
  {
    BearStore store = new BearStore();
    store.addStdBears(3);
    store.addUserSpecifiedBears(2);
    store.displayInventory();
  } // end main
} // end BearStore class
```

objeto anónimo como valor de retorno

ciclo for-each

Figura 10.29*b* Clase que implementa una tienda que vende osos de juguete, parte B.

2. Cuando un objeto de reciente creación se devuelve desde un método. Por ejemplo:

```
return new Bear(maker, type);
```

Controlador insertado

En la parte inferior de la clase `BearStore` se ha insertado el controlador del programa, `main`. `main` instancia un objeto `BearStore`, agrega tres osos estándar a la tienda, agrega a la tienda dos osos especificados por el usuario y luego muestra el inventario de osos de la tienda al llamar a `displayInventory`. Al exhibir el inventario de la tienda, el método `displayInventory` introduce cada oso en la `bears` `ArrayList` con ayuda de un ciclo for-each. En la siguiente subsección se aprenderán detalles del ciclo for-each.

Ciclo for-each

Como ya se mencionó, un ciclo for-each puede usarse siempre que sea necesario iterar a través de todos los elementos en una colección de datos. Ésta es la sintaxis de un ciclo for-each para una `ArrayList`:

```
for (<tipo de elemento> <nombre del elemento> : <variable de referencia de ArrayList>)
{
  ...
}
```

Y he aquí un ejemplo para el ciclo for-each del método `displayInventory` de la figura 10.29*b*:

```
for (Bear bear : bears)
{
  bear.display();
}
```

Observe cómo el encabezado del ciclo for-each coincide con la sintaxis anterior: `bears` es una variable de referencia de `ArrayList`, `bear` es el nombre de un elemento en la `ArrayList` `bears`, y `Bear` es el tipo de cada elemento. Es legal escoger cualquier nombre para el elemento pero, como siempre, debe elegirse un nombre descriptivo, como `bear` en este ejemplo. Con cada iteración del ciclo for-each, el nombre del elemento se usa para referirse al elemento actual. Por ejemplo, `bear.display()` llama al método `display` para el elemento `bear` actual.

Acaso el lector se pregunta: ¿por qué el ciclo for-each se denomina así aun cuando no hay "each" en la sintaxis? Porque la mayoría de las personas dice "para cada" al leer el encabezado de un ciclo for-each. Por ejemplo, al leer el ciclo for-each de `displayInventory`, la mayoría de las personas diría "Para cada oso en la colección, hacer lo siguiente".

Observe que, como alternativa, es posible implementar el método `displayInventory` usando un ciclo `for` tradicional en lugar de un ciclo for-each. Ésta es la implementación con un ciclo tradicional `for`:

```
for (int i=0; i<bears.size(); i++)
{
  bears.get(i).display();
}
```

Es preferible la implementación con el ciclo for-each, ya que es más simple. No es necesario declarar una variable índice, y tampoco calcular y especificar el valor de los índices primero y último de la `ArrayList`.

Tome en cuenta que el ciclo for-each puede usarse para más que simplemente `ArrayList`. Puede usarse para iterar a través de cualquier colección de objetos. Más específicamente, es posible usarlo para arreglos y para cualquiera de las *clases de colección* de Java. `ArrayList` es una clase de colección, y para aprender sobre las otras clases de colección, consulte http://java.sun.com/javase/6/docs/technotes/guides/collections/

El siguiente fragmento de código ilustra cómo usar un ciclo for-each con un arreglo estándar. Imprime los números en un arreglo `primes`.

```
int[] primes = {1, 2, 3, 5, 7, 11};
for (int p : primes)
{
  System.out.println(p);
}
```

El ciclo for-each es fantástico, aunque es necesario tener en cuenta varias cuestiones al utilizarlo. 1) Fue introducido en Java 5.0, de modo que no funciona con compiladores anteriores. 2) El ciclo for-each no usa una variable índice para recorrer sus elementos. Esto puede ser benéfico en cuanto a que lleva a un código menos revuelto. Pero es una desventaja si dentro del ciclo se requiere un índice. Por ejemplo, suponga que se tiene el arreglo `primes`, como antes, y que se desea imprimir lo siguiente:

```
primes[0] = 1
primes[1] = 2
...
primes[5] = 11
```

Los números entre corchetes son valores de índice. Entonces, si se ha implementado una solución con un ciclo for-each, es necesario agregar una variable índice al código e incrementarla cada vez que se ejecuta el ciclo. Por otra parte, si se ha implementado una solución con un ciclo for tradicional, ya se cuenta con una variable índice integrada.

10.14 Lista de arreglos *versus* arreglos estándar

Hay mucho traslape en la funcionalidad de una ArrayList y un arreglo estándar. Así, ¿cómo puede decidirse cuál usar? La respuesta es distinta para situaciones diferentes. Cuando decida sobre una implementación, considere esta tabla:

Ventajas de una ArrayList sobre un arreglo estándar	Ventajas de un arreglo estándar sobre una ArrayList
1. Es fácil aumentar el tamaño de una ArrayList: simplemente se llama a add.	1. Un arreglo estándar usa corchetes para acceder a los elementos del arreglo (lo cual es más fácil que usar los métodos get y set).
2. Es fácil para el programador insertar un elemento o eliminar un elemento en o desde el interior de una ArrayList: simplemente se llama a add o a remove y se especifica la posición del índice del elemento.	2. Un arreglo estándar es más eficiente cuando se almacenan valores primitivos.

Al observar la primera ventaja de ArrayList en la tabla, que es fácil aumentar el tamaño de una ArrayList, piense en la cantidad de trabajo necesario para aumentar el tamaño de un arreglo estándar. Para un arreglo estándar, el programador necesita instanciar un arreglo más grande y luego copiar el contenido del arreglo anterior en el nuevo arreglo más grande. Por otra parte, para una ArrayList, el programador simplemente debe llamar al método add. Observe que tras bambalinas, la JVM debe invertir algo de esfuerzo para implementar el método add, aunque el esfuerzo se mantiene en un mínimo. Las ArrayList se implementan con ayuda de un arreglo estándar subyacente. Usualmente, el arreglo subyacente cuenta con un mayor número de elementos que la ArrayList, de modo que es fácil agregar otro elemento a la ArrayList: la JVM simplemente toma prestado un elemento no utilizado del arreglo subyacente. Como programador, no es necesario preocuparse por estos detalles o codificarlos: el "préstamo" se lleva a cabo automáticamente.

La segunda ventaja de la ArrayList en la tabla, que es fácil para el programador insertar un elemento o eliminar un elemento en o desde el interior de una ArrayList, es cierta, aunque sólo porque es fácil para los programadores no significa que es fácil para la JVM. En realidad, la JVM debe realizar bastante trabajo cuando agrega o elimina desde el interior de una ArrayList. Para insertar un elemento, la JVM debe ajustar su arreglo subyacente al desplazar elementos con índices superiores a fin de dar cabida al nuevo elemento. Y para eliminar un elemento, la JVM debe ajustar su arreglo subyacente al desplazar elementos con índices superiores para recubrir el elemento eliminado.

Puesto que las ArrayList y los arreglos estándar son ineficientes cuando se trata de insertar un elemento en una lista o eliminar un elemento de una lista, si se están efectuando bastantes inserciones y eliminaciones, es necesario considerar una estructura diferente: una *lista ligada*. Una lista ligada es una secuencia de elementos, donde cada elemento contiene un dato más una referencia (una "liga") que apunta al siguiente elemento. Una lista ligada puede crearse usando exactamente los mismos procedimientos que se usan para crear una ArrayList. Simplemente se sustituye la clase ArrayList por la clase LinkedList. Si el lector está interesado en conocer los detalles de LinkedList, debe consultar la clase de colección LinkedList en el sitio API de Java Sun en la red.

La tabla anterior indica que un arreglo estándar es más eficiente que una ArrayList cuando se trata de almacenar valores primitivos. ¿Por qué? Recuerde que antes que una ArrayList almacene un valor primitivo, debe envolverlo en un objeto envoltorio. El proceso de envoltura requiere tiempo, lo cual explica la causa de la ineficiencia.

Resumen

- Los arreglos facilitan la representación y manipulación de colecciones de datos semejantes. Para acceder a los elementos de los arreglos se usa <*array-name*> [index], donde index es un entero no negativo que empieza en cero.
- Un arreglo puede crearse e iniciarse por completo con una declaración como ésta:
  <*tipo de elemento*>[] <*nombre del arreglo*> = {*elemento0*, *elemento1*, ...};
- No obstante, por lo general, resulta más fácil posponer la iniciación y usar new para crear un arreglo de objetos no iniciados, como se muestra a continuación.
  <*tipo de elemento*>[] <*nombre del arreglo*> = new <*tipo de elemento*>[*tamaño del arreglo*];
- Es posible leer o escribir directamente en un elemento de un arreglo al insertar un valor de índice idóneo escrito entre corchetes después del nombre del arreglo en cualquier instante después que se ha creado el arreglo.
- Todo arreglo incluye automáticamente una propiedad public denominada length a la cual puede accederse directamente con el nombre del arreglo. El valor de índice más alto es <*array-name*>. length – 1.
- Para copiar un arreglo, cada uno de sus elementos se copia individualmente, o puede usarse el método System.arraycopy para copiar cualquier subconjunto de elementos en un arreglo hacia cualquier ubicación en otro arreglo.
- Un histograma es un arreglo de elementos donde el valor de cada elemento es el número de ocurrencias de algún evento.
- Una búsqueda secuencial constituye una forma aceptable para una comparación en un arreglo cuya longitud sea menor que aproximadamente 20, aunque para arreglos largos, primero es necesario ordenar el arreglo con el método Arrays.sort y luego efectuar una búsqueda binaria.
- Un arreglo de dos dimensiones es un arreglo de arreglos, declarado con dos conjuntos de corchetes después de la identificación del tipo de elemento. Es posible instanciarlo con un iniciador o con new seguido del tipo de elemento y dos especificaciones del tamaño del arreglo entre corchetes.
- Para crear un arreglo de objetos se requieren múltiples instanciaciones. Después de instanciar el arreglo, también es necesario instanciar los objetos elemento individuales dentro del arreglo.
- Si se requiere insertar en un arreglo o eliminar elementos de un arreglo en forma repetida, debe considerarse la utilización de una ArrayList en lugar de un arreglo estándar. Cuando una ArrayList se declara o instancia para un grupo de elementos Car, es necesario incluir entre paréntesis angulares el tipo de elemento que contiene, como se muestra a continuación:
  ```
  ArrayList<Car> car = new ArrayList<Car>();
  ```
- Una ArrayList sólo almacena objetos. Java automáticamente hace necesario realizar conversiones entre primitivos y primitivos envueltos, de modo que no es necesario preocuparse sobre esto, aunque si se desea una ArrayList de primitivos como int, es necesario declararla con el tipo envuelto, como sigue:
  ```
  ArrayList<Integer> num = new ArrayList<Integer>();
  ```
- Es posible pasar en forma anónima objetos hacia métodos y desde métodos.
- Debe usarse un ciclo for-each para iterar a través de una colección de datos.

Preguntas de revisión

§10.2 Fundamentos de arreglos

1. Es legal almacenar int y también double en un simple arreglo estándar. (F/C)
2. Dado un arreglo denominado myArray, al primer elemento se accede usando myArray[0]. (F/C)

§10.3 Declaración y creación de arreglos

3. Proporcione una declaración para un arreglo de cadenas denominado names.
4. Considere el encabezado de cualquier método main:

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

   ¿Qué tipo de ente es args?

5. Suponga que se crea un arreglo con la declaración:

```
char[] choices = new char[4];
```

¿Cuál es el valor por defecto en un elemento típico de este arreglo? ¿Es basura o algo en particular?

§10.4 Propiedad `length` en un arreglo y arreglos parcialmente llenos

6. El valor de la longitud de un arreglo es igual al valor del máximo índice aceptable del arreglo. (F/C)

§10.5 Copia de un arreglo

7. Dado

```
String letters = "abcdefghijklmnopqrstuvwxyz";
char alphabet[] = new char[26];
```

escriba un ciclo `for` que inicie `alphabet` con los caracteres que hay en `letters`.

8. Escriba una declaración simple que copie todos los elementos en

```
char arr1[] = {'x', 'y', 'z'};
```

en los tres últimos elementos de

```
char arr2[] = new char[26];
```

§10.6 Resolución de problemas mediante casos con arreglos

9. En el programa MovingAverage en la figura 10.7, suponga que se desea desplazarse en la otra dirección. ¿Cómo escribiría el lector el encabezado de un ciclo interno `for` y cómo escribiría la declaración de asignación del arreglo en el ciclo `for` interno?

10. ¿Qué tipo de valores contiene un histograma típico "papelera"?

§10.7 Búsqueda en un arreglo

11. Es posible buscar `ids` en arreglos para un elemento igual a `id` con sólo esto:

```
int i;
for (i=0; i<ids.length && id != ids[i]; i++)
{ }
if (<expresión booleana>)
{
  return i;
}
```

¿Cuál es la *<expresión booleana>* que indica que se ha encontrado `i`?

§10.8 Ordenamiento de un arreglo

12. Se decidió usar métodos de clase para implementar un algoritmo de ordenamiento. ¿Cuál es la ventaja?

13. ¿A qué clase pertenece el método `sort` API de Java?

§10.9 Arreglos de dos dimensiones

14. Se mencionó que un arreglo de dos dimensiones es un arreglo de arreglos. Considere la siguiente declaración:

```
double[][] myArray = new double[5][8];
```

En el contexto de la expresión arreglo de arreglos, ¿qué significa `myArray[3]`?

§10.10 Arreglos de objetos

15. Al crear un arreglo de objetos es necesario instanciar el objeto del arreglo, así como cada objeto elemento que esté almacenado en el arreglo. (F/C)

§10.11 La clase `ArrayList`

16. ¿En qué forma es más polivalente una `ArrayList` que un arreglo?

17. Para evitar errores de tiempo de ejecución, siempre es necesario especificar el tamaño de una `ArrayList` cuando se declara. (F/C)

18. ¿Cuál es el tipo de retorno del método `get` de la clase `ArrayList`?

19. Si se llama al método `add (i, x)` de `ArrayList`, ¿qué ocurre al elemento que originalmente está en la posición `i`?

§10.12 Almacenamiento de primitivos en una `lista de arreglos`

20. Específicamente, ¿en qué circunstancias se lleva a cabo autoboxing?

21. Escriba una declaración que agregue el valor `double`, 56.85, al final de una `ArrayList` existente denominada `prices`.

22. Escriba una declaración que muestre todos los valores en una `ArrayList` de `Doubles` denominada `prices`. Escriba toda la lista entre corchetes y use una coma y un espacio para separar valores diferentes en la lista.

§10.13 Ejemplo de `lista de arreglo` utilizando objetos anónimos y el ciclo for-each

23. ¿Qué es un objeto anónimo?

24. Es necesario usar un ciclo for-each, y no un ciclo tradicional `for`, siempre que sea necesario iterar a través de una colección de elementos. (F/C)

§10.14 `Lista de arreglos` *versus* arreglos estándar

25. Dado lo siguiente:

- Se cuenta con una clase `WeatherDay` que almacena la información climatológica de un día.
- Se desea almacenar en `WeatherDay` objetos para todo un año.
- La tarea fundamental del programa es ordenar objetos `WeatherDay` (por ejemplo, ordenarlos por temperatura, según la velocidad del viento, etcétera).

¿Dónde almacenaría objetos `WeatherDay`: en una `ArrayList` o en un arreglo estándar? Explique su respuesta.

Ejercicios

1. [Después de §10.2] El número de índice del último elemento en un arreglo de longitud 100 es ____.

2. [Después de §10.3] Declare un arreglo denominado `scores` que contenga valores `double`.

3. [Después de §10.3] Proporcione una simple declaración que inicie `myList` en todo unos. `myList` es un arreglo de 5 elementos de `int`.

4. [Después de §10.4] Programa Zoo Animals:

Como parte de su periodo de aprendizaje en el nuevo zoológico de Parkville, al lector se le pide escribir un programa que siga la pista de los animales que hay en el zoológico. El lector desea que el programa sea de propósito general, de modo que una vez terminado, pueda venderlo a zoológicos de todo el mundo y enriquecerse. Así, el lector decide crear una clase genérica `Zoo`.

Escriba una clase `Zoo`. Su clase no tiene que hacer mucho: simplemente manipula la creación e impresión de objetos `Zoo`. Para tener una mejor idea de la funcionalidad de la clase `Zoo` se proporciona un método `main`:

```
public static void main(String[] args)
{
  Zoo zoo1 = new Zoo();
  String[] animals = {"cerdo", "zarigüeya", "ardilla", "Chihuahua"};
  Zoo zoo2 = new Zoo(animals, "Parkville");
  animals[0] = "tigre blanco";
  Zoo zoo3 = new Zoo(animals, "San Diego");
  zoo1.display();
  zoo2.display();
  zoo3.display();
}
```

Cuando se ejecuta el método `main`, debe imprimir esto:

```
El zoo está vacío.
Parkville zoo: cerdo, zarigüeya, ardilla, Chihuahua
San Diego zoo: tigre blanco, zarigüeya, ardilla, Chihuahua
```

Aunque no es necesario, se alienta al lector a que escriba un programa completo a fin de probar su clase `Zoo`.

5. [Después de §10.5] Suponga que el siguiente fragmento de código compila y ejecuta. ¿Cuál es la salida? Sea preciso cuando muestre su salida.

```
char[] a = new char[3];
char[] b;
for (int i=0; i<a.length; i++)
{
  a[i] = 'a';
}
b = a;
b[2] = 'b';
System.out.println("a[1]=" + a[1] + ", a[2]=" + a[2]);
System.out.println("b[1]=" + b[1] + ", b[2]=" + b[2]);
```

6. [Después de §10.5] ¿Qué es necesario agregar al siguiente fragmento de código de modo que todos los valores excepto los dos primeros (100000.0 y 110000.0) sean copiados de `allSalaries` a `workerSalaries`?

```
double[] allSalaries = {100000.0, 110000.0, 25000.0, 18000.0,
  30000.0, 9000.0, 12000.0};
double[] workerSalaries;
```

7. [Después de §10.5] Se supone que el siguiente programa invierte el orden de los elementos en el arreglo `simpsons`. Compila y ejecuta, pero no funciona correctamente.

```
public class Reverse
{
  public static void main(String[] args)
  {
    String[] simpsons = {"Homer", "Flanders", "Apu"};
    reverse(simpsons);
    System.out.println(
      simpsons[0] + " " + simpsons[1] + " " + simpsons[2]);
  } // end main

  public static void reverse(String[] list)
  {
    String[] temp = new String[list.length];
    for (int i=0; i<list.length; i++)
    {
      temp[i] = list[list.length-i-1];
    }
    list = temp;
  } // end reverse
} // end class Reverse
```

a) ¿Qué imprime el programa?

b) Arregle el programa escribiendo una o más líneas de código alternativo para la línea `list = temp;`. No se permite modificar ningún otro código; simplemente proporcione un código alternativo para esa línea.

8. [Después de §10.6] Escriba un programa que implemente el ejemplo descrito al inicio de la sección 10.6. Su programa debe desplazar los elementos del arreglo de la posición x a la posición x – 1 según se describió en esa sección. (Mueva el valor en la posición 1 a la posición 0; mueva el valor en la posición 2 a la posición 1 y así sucesivamente.)

Empiece creando dos arreglos, `double [] initialHours` y `double[] hours`. En su declaración, inicie `initialHours` con los valores `{8, 8, 6, 4, 7, 0, 0, 5}`, pero no inicie `hours` cuando lo declare e instancie con sus 31 elementos. En lugar de ello, inicie `hours` después de su creación usando `System.arraycopy` para copiar todos los valores en `initialHours` en los primeros elementos en `hours`. Luego, efectúe una operación de desplazamiento hacia abajo y cargue cero en elemento más alto (el nuevo).

9. [Después de §10.7] Escriba un método de clase denominado `allPositive` que reciba un arreglo denominado `arr` de valores `double` y devuelva `true` si todos los valores de los elementos son positivos y devuelva `false` en caso contrario. Use modificadores de acceso idóneos. Haga que el método sea accesible desde fuera de su clase.

10. [Después de §10.7] Suponga que ya ha escrito exitosamente una clase denominada `Students` que maneja los registros de estudiantes para la oficina de inscripciones. Suponga que la clase `Students`:

- Contiene una variable de instancia `studentIds`: un arreglo de `ints` que contiene los números de identificación del estudiante.

- Contiene un constructor de un parámetro que inicia la variable de instancia `studentIds`.
- Contiene el método `main`:

```
public static void main(String[] args)
{
  Students s1 = new Students(new int[] {123, 456, 789});
  Students s2 = new Students(new int[] {123, 456, 789, 555});
  Students s3 = new Students(new int[] {123, 456, 789});
  if (s1.equals(s2))
  {
    System.out.println("s1 == s2");
  }
  if (s1.equals(s3))
  {
    System.out.println("s1 == s3");
  }
} // end main
```

Escriba un método `public` denominado `equals` para la clase `Students` que prueba si dos objetos `Students` son iguales. El método `equals` debe escribirse de modo que el método `main` anterior produzca esta salida:

```
s1 == s3
```

Sólo proporcione el código para el método solicitado `equals`; no proporcione el código para toda la clase `Students`.

11. [Después de §10.8] Dada la siguiente lista de arreglos, use el algoritmo de ordenamiento por selección para ordenar el arreglo. Muestre cada paso en el proceso de ordenamiento por selección. No proporcione el código, simplemente muestre imágenes del arreglo `list` después del desplazamiento de cada elemento.

`list` (original)

0	12
1	2
2	−4
3	0
4	9

`list` (sorted)

0	−4
1	0
2	2
3	9
4	12

12. [Después de §10.8] El algoritmo de ordenamiento por inserción constituye una alternativa para el algoritmo de ordenamiento por selección para ordenar cantidades pequeñas de objetos (del orden de 20 o menos). No es tan eficiente como el ordenamiento por selección para arreglos, aunque es ligeramente más eficiente para otros tipos de colecciones de datos. El siguiente código implementa el algoritmo de ordenamiento por inserción:

```
10  public static void insertionSort(int[] list)
11  {
12    int temp;
13    int j;
14
15    for (int i=1; i<list.length; i++)
16    {
17      temp = list[i];
18      for (j=i; j>0 && temp<list[j-1]; j--)
19      {
20        list[j] = list[j-1];
21      }
22      list[j] = temp;
23    } // end for
24  } // end insertionSort
```

Observe que el alcance de la variable de conteo `j` rebasa el alcance del ciclo `for` en que se utiliza. Suponga que se ha instanciado un arreglo de `int` y que se ha llamado al método `insertionSort` con una referencia

a este arreglo pasado como un parámetro. Rastree la ejecución de este método, usando el siguiente encabezado y entradas iniciales:

Ordenamiento					<arreglos>				
	Ordenamiento por inserción				arr1				
línea#	**(lista)**	**i**	**j**	**temp**	**longitud**	**0**	**1**	**2**	**3**
					4	3333	1234	2222	1000
10	arr1								

13. [Después de §10.8] Rastree el siguiente código y muestre la salida exacta.

```
 1  public class ModifyArray
 2  {
 3    public static void main(String[] args)
 4    {
 5      int sum = 0;
 6      int[] list = new int[3];
 7
 8      for (int i=0; i<3; i++)
 9      {
10        list[i] = i + 100;
11      }
12      modify(list, sum);
13      for (int i=0; i<3; i++)
14      {
15        System.out.print(list[i] + " ");
16      }
17      System.out.println("\nsum = " + sum);
18    }
19
20    public static void modify(int[] list, int sum)
21    {
22      int temp = list[0];
23
24      list[0] = list[list.length - 1];
25      list[list.length - 1] = temp;
26      for (int i=0; i<3; i++)
27      {
28        sum += list[i];
29      }
30    }
31  } // end ModifyArray
```

Use el siguiente encabezado de rastreo:

ModifyArray								<arreglos>				
	main			modify				arr1				
línea#	**i**	**suma**	**lista**	**(lista)**	**(suma)**	**temp**	**i**	**longitud**	**0**	**1**	**2**	**salida**

14. [Después de §10.9] Especifique una declaración simple que inicie todo en unos un arreglo de `int` denominado `myTable`. El arreglo debe ser de dos dimensiones con 2 renglones y 3 columnas.

15. [Después de §10.9] Escriba un método denominado `getMask` que reciba un simple parámetro denominado `table` que sea un arreglo de dos dimensiones de `int`. El método `getMask` debe crear y devolver un arreglo *mask* para el arreglo de tabla pasado. El término de programación *mask* se refiere a un arreglo que se construye a partir de otro arreglo y que sólo contiene ceros y unos. Para cada elemento en el arreglo `mask`, si el elemento correspondiente del arreglo contiene un número positivo, el elemento del arreglo `mask` debe contener un 1. Y si el elemento correspondiente del arreglo original contiene un cero o un número negativo, entonces el elemento del arreglo `mask` debe contener un 0. Observe este ejemplo:

parámetro `table`

5	−2	3	1
0	14	0	6
3	6	−1	4

arreglo devuelto

1	0	1	1
0	1	0	1
1	1	0	1

Observe:

- Su método no debe modificar el contenido del arreglo de tabla pasado.
- Su método debe funcionar con cualquier tabla de tamaño arbitrario, no sólo con la tabla de 3 renglones y 4 columnas mostrada en el ejemplo.
- Use modificadores de acceso idóneos. Suponga que el método debe ser accesible desde fuera de su clase. Al decidir si el método debe ser uno de clase o uno de instancia, observe que el método no introduce ninguna variable de instancia (sólo introduce un parámetro).

16. [Después de §10.10] Suponga que se tiene la siguiente clase `City`:

```
public class City
{
  private String name;
  private double north;          // latitud norte en grados
  private double west;           // longitud oeste en grados

  //************************************************************

  public City(String name, double latitude, double longitude)
  {
    this.name = name;
    this.north = latitude;
    this.west = longitude;
  } // end constructor

  //************************************************************

  public void display()
  {
    System.out.printf("%12s%6.1f%6.1f\n", name, north, west);
  }
} // end class City
```

Escriba un fragmento de código que produzca un arreglo de objetos `City` que contenga el nombre, la latitud y la longitud de las cuatro ciudades siguientes y exhiba el contenido de estos arreglos como se muestra a continuación:

```
Nueva York  41.0  74.0
     Miami  26.0  80.0
   Chicago  42.0  88.0
   Houston  30.0  96.0
```

17. [Después de §10.11] ¿Qué hace el método `remove` de `ArrayList`?

18. [Después de §10.12] Proporcione una simple declaración (una declaración de iniciación) que declare una `ArrayList` denominada `evenNumbers` y le asigne una nueva `ArrayList` recientemente instanciada. La `ArrayList` instanciada debe ser capaz de almacenar enteros.

19. [Después de §10.12] Use la `ArrayList` `evenNumbers` creada en el ejercicio previo para proporcionar un fragmento de código que almacene los 10 primeros números pares en la `ArrayList` `evenNumbers`. En otras palabras, coloque 0 en el primer elemento de `evenNumbers`, 2 en el segundo elemento de `evenNumbers`, . . . , 18 en el décimo elemento de `evenNumbers`. Debe usar un ciclo `for` estándar para su fragmento de código.

20. [Después de §10.13] Proporcione una versión más elegante (aunque funcionalmente equivalente) de este fragmento de código:

```
ArrayList<Car> cars = new ArrayList<Car>();
Car car1 = new Car("Mustang", 2006, "tigre con rayas");
cars.add(car1);
Car car2 = new Car("MiniCooper", 2006, "verde lima");
cars.add(car2);
```

21. [Después de §10.13] Suponga que se cuenta con una `ArrayList` de direcciones de calles que se ha iniciado y llenado como sigue:

```
ArrayList<String> addressList = new ArrayList<String>();
addressList.add("1600 Pennsylvania Avenue");
addressList.add("221B Baker Street");
...
addressList.add("8700 N.W. River Park Drive");
```

Proporcione un ciclo for-each (no un ciclo estándar `for`) que imprima las direcciones de `addressList`, una dirección por línea.

22. [Después de §10.14] Suponga que se desea mantener una lista de ciudades descritas por la clase `City` definida en el ejercicio 16, y suponga que se desea poder insertar o eliminar ciudades en cualquier sitio de la lista a fin de mantener cierto orden a medida que se agregan o eliminan elementos. ¿Debe usar un arreglo o una `ArrayList`, y por qué?

Soluciones a las preguntas de revisión

1. Falso. Los tipos de elementos de datos en un arreglo particular deben ser los mismos.
2. Cierto.
3. Declaración para un arreglo de cadenas denominado `names`:

```
String[] names;
```

4. El parámetro `args` en `main` es un arreglo de cadenas.
5. Los elementos de un arreglo son como las variables de instancia en un objeto. Los valores por defecto de los elementos del arreglo no son basura. El valor por defecto de un elemento `char[]` es un carácter especial cuyo valor numérico subyacente es `0`.
6. Falso. El valor del máximo índice aceptable es uno menos que la longitud del arreglo.
7. Este fragmento de código inicia el arreglo de caracteres alphabet:

```
for (int i=0; i<26; i++)
{
  alphabet[i] = letters.charAt(i);
}
```

8. Es posible copiar:

```
arr1[] = {'x', 'y', 'z'}
```

al final de:

```
arr2[] = new char[26]
```

con la siguiente declaración:

```
System.arraycopy(arr1, 0, arr2, 23, 3);
```

9. En el programa MovingAverage, para desplazarse en la otra dirección, el encabezado del ciclo interno `for` es:

```
for (int d=days.length-1; d>0; d--)
```

La declaración de asignación del elemento en el arreglo en este ciclo es:

```
days[d] = days[d-1];
```

10. Un histograma "papelera" contiene el número de ocurrencias de un evento.
11. La expresión booleana que indica que se ha encontrado `i` es:

```
(ids.length != 0 && i != ids.length)
```

12. La ventaja de usar métodos de clase es que el método de ordenamiento puede usarse con cualquier arreglo pasado, no sólo sobre un arreglo específico de variables de instancia.
13. El método `sort` API de Java está en la clase `Arrays`.
14. `myArray[3]` se refiere al cuarto renglón, que es un arreglo de ocho valores `double`.
15. Cierto.

16. Con una `ArrayList` es posible insertar y eliminar elementos en cualquier sitio de la secuencia, y la longitud de la lista aumenta y disminuye dinámicamente.
17. Falso. Normalmente, cuando se declara una `ArrayList` no se especifica ningún tamaño para ésta.
18. El tipo de retorno del método `get` es `E`, que se refiere al tipo de cada elemento en la `ArrayList`.
19. El elemento que originalmente está en la posición `i` se desplaza a la posición del siguiente índice superior.
20. El autoboxing se lleva a cabo cuando un primitivo se está usando en un sitio que espera una referencia.
21. `prices.add(56.85);`
22. `System.out.println(prices);`
23. Un objeto anónimo es un objeto que está instanciado pero que no se almacena en una variable.
24. Falso. Para iterar a través de una colección de elementos es posible usar un ciclo `for` tradicional (o un ciclo for-each).
25. Los objetos `WeatherDay` deben almacenarse en un arreglo estándar.
    Explicación:
    - No es necesario que el arreglo crezca o disminuya, puesto que el tamaño se ha fijado en 365 (y los arreglos estándar tienen un tamaño fijo).
    - Con el ordenamiento, es necesario acceder a los objetos bastante a menudo (lo cual es más fácil con arreglos estándar).

CAPÍTULO 11

Detalles de tipo y mecanismos de codificación alternativa

Objetivos

- Comprender mejor las relaciones y diferencias entre tipos de datos primitivos y su apreciación en cuanto a sus limitaciones individuales.
- Comprender cómo los códigos numéricos identifican caracteres.
- Aprender las reglas para conversiones automáticas de tipo y los riesgos en la conversión explícita.
- Comprender los modos prefijo/sufijo insertados para operadores de incremento/decremento.
- Comprender expresiones de asignación insertadas.
- Aprender dónde y cómo las expresiones del operador condicional pueden abreviar el código.
- Ver cómo la evaluación de cortocircuito ayuda a evitar operaciones problemáticas.
- Ver cómo funciona la declaración vacía.
- Aprender a usar declaraciones `break` en ciclos.
- Opcionalmente, usar caracteres Unicode en aplicaciones GUI.

Relación de temas

11.1 Introducción

En los capítulos 3 y 4 se aprendieron los fundamentos del lenguaje Java. Entre otras cosas se aprendió sobre tipos de datos, conversiones de tipo y declaraciones de control. En este capítulo se describen algunos tipos de datos adicionales y conversiones de tipo adicionales. También se describen algunos mecanismos alternativos para codificar declaraciones de control.

En el capítulo 3 se presentaron algunos tipos de números enteros y de punto flotante de Java, y en el capítulo 5 se mostró cómo encontrar los límites de sus intervalos. En este capítulo se verán dos tipos de enteros más, y para todos los tipos numéricos se aprenderá la cantidad de almacenamiento necesaria, la precisión proporcionada y cómo usar los límites de los intervalos. En el capítulo 3 se presentó el uso del tipo carácter, `char`. En este capítulo se verá que cada carácter posee un valor numérico subyacente y se aprenderá cómo usar estos valores. En el capítulo 3 se presentó el tipo de conversión con el operador tipo cast. En este capítulo se aprenderá más sobre conversiones de tipo. En el capítulo 3 se presentaron los operadores de incremento y decremento. En este capítulo se descubrirá que es posible mover las posiciones de estos operadores (antes o después de la variable) para controlar el momento en que actúan. En el capítulo 3 se introdujeron los operadores de asignación. En este capítulo se verá cómo es posible insertar asignaciones en expresiones a fin de reducir más el código.

En el capítulo 4 se presentaron varios tipos de evaluaciones condicionales. En este capítulo se aprenderá sobre el operador condicional que puede asumir uno de dos valores posibles, dependiendo de una condición `boolean`. También se aprenderá sobre la evaluación de cortocircuito que puede prevenir errores al detener una evaluación condicional "peligrosa" en ciertas situaciones. Asimismo, se aprenderá más sobre ciclos. Específicamente, se abordarán ciclos de cuerpo vacío y ciclos que terminan desde el interior del cuerpo del ciclo. Y se verán técnicas de codificación alternativas para encabezados del ciclo `for`.

El material en este capítulo mejorará su comprensión de varios detalles y sutilezas de Java. Esto ayudará a evitar problemas en primer lugar, y le ayudará a crear códigos más eficientes y sencillos de mantener. También le ayudará a depurar un código que tenga problemas. Puede ser su código o el de alguien más. Como programador del mundo real, el lector debe trabajar con códigos de otras personas, por lo que es necesario comprender lo que hace ese código.

Mucho del material de este capítulo hubiera podido insertarse desde antes en varios sitios en el texto. Sin embargo, no era necesario, por eso se pospuso hasta ahora a fin de no entorpecer presentaciones anteriores. La reunión de estos detalles en un capítulo en este punto del libro constituye una excelente oportunidad para efectuar un repaso. A medida que avance en este capítulo, integre este nuevo material a lo que ha aprendido antes y vea cómo se enriquece su comprensión de estos temas.

11.2 Tipos entero y de punto flotante

Esta sección complementa el material sobre tipos de datos numéricos que se abordó en la sección 3.13 del capítulo 3.

Tipos entero

Los tipos entero comprenden a los números enteros (números sin punto decimal). En la figura 11.1 se muestran los cuatro tipos entero. Los tipos están ordenados en términos de requerimiento creciente de espacio para almacenamiento en la memoria. Las variables tipo `byte` requieren sólo ocho bits, de modo que toman la menor cantidad de almacenamiento. Si se tiene un programa que ocupa demasiado espacio en la memoria, es posible usar tipos menores para variables que contengan valores pequeños. El uso de tipos menores significa que se requiere menos espacio de almacenamiento en la memoria. Ahora que la memoria se ha vuelo relativamente barata, los tipos `byte` y `short` no se usan muy a menudo.

Para introducir los valores mínimo y máximo de un entero, se usan las variables nombradas `MIN_VALUE` y `MAX_VALUE` que vienen con la clase envoltorio del entero. Como se aprendió en el capítulo 5, `Integer` y `Long` son las clases envoltorio para los tipos de datos `int` y `long`. Y como es de esperar, `Byte` y `Short` son las clases envoltorio para los tipos de datos `byte` y `short`. Entonces, he aquí cómo imprimir el valor `byte` máximo:

```
System.out.println("Largest byte = " + Byte.MAX_VALUE);
```

El tipo por defecto para una constante entera es `int`. Sin embargo, podría ser necesaria una constante entera que sea demasiado grande para un `int`. En ese caso, explícitamente es posible forzar una constante entera a ser `long` al agregar un sufijo `l` o `L` a la constante entera. Por ejemplo, suponga que se está escribiendo un programa sistema solar y que se desea almacenar la antigüedad de la Tierra en una varia-

Tipo	Almacenamiento	MIN_VALUE de la clase envoltorio	MAX_VALUE de la clase envoltorio
byte	8 bits	−128	127
short	1 6 bits	−32 768	32 767
int	32 bits	−2 147 483 648	2 147 483 647
long	64 bits	$\approx -9*10^{18}$	$\approx 9*10^{18}$

Figura 11.1 Propiedades de los tipos de datos enteros de Java.

ble nombrada `ageOfPlanet`. La antigüedad de la Tierra es de 4.54 miles de millones de años, cifra más grande que el `MAX_VALUE` de `Integer`: 2 147 483 647. Esto genera un error de compilación:

```
long ageOfPlanet = 4540000000;
```

Pero esto, con el sufijo `L`, funciona bien:

```
long ageOfPlanet = 4540000000L;
```

⚠ Cuando se declara una variable numérica, es necesario tener la certeza de que el tipo elegido sea suficientemente grande para manejar el mayor valor que el programa es capaz de asignarle. Si un valor no cabe en el espacio de la memoria proporcionado, esto se denomina *desbordamiento*. Los errores de desbordamiento son espectaculares, como se ilustra con el programa ByteOverflowDemo en la figura 11.2.

El desbordamiento de enteros invierte el signo, de modo que el programa ByteOverflowDemo imprime 128 negativo en lugar del resultado correcto, 128 positivo. En este ejemplo, la magnitud del error

```
/*************************************************************
* ByteOverflowDemo.java
* Dean & Dean
*
* This demonstrates integer overflow.
*************************************************************/

public class ByteOverflowDemo
{
  public static void main(String[] args)
  {
    byte value = 64;

    System.out.println("Initial byte value = " + value);
    System.out.println("Byte maximum = " + Byte.MAX_VALUE);
    value += value;
    System.out.println("Twice initial byte value = " + value);
  } // end main
} // end ByteOverflowDemo class
```

Salida:

```
Initial byte value = 64
Byte maximum = 127
Twice initial byte value = -128
```

¡Un error muy grande!

Figura 11.2 Programa ByteOverflowDemo que ilustra el problema de desbordamiento.

Tipo	Almacenamiento	Precisión	MIN_VALUE de la clase envoltorio	MAX_VALUE de la clase envoltorio
`float`	32 bits	6 dígitos	$\approx 1.2 * 10^{-38}$	$\approx 3.4*10^{38}$
`double`	64 bits	15 dígitos	$\approx 2.2 * 10^{-308}$	$\approx 1.8*10^{308}$

Figura 11.3 Propiedades de los tipos de datos de punto flotante de Java.

es ¡aproximadamente dos veces la magnitud del máximo valor permisible! El desbordamiento también origina inversión de signo para los tipos `short`, `int` y `long`. En tales casos, el compilador no detecta el problema, y la Máquina Virtual de Java (JVM) tampoco. Java ejecuta el programa sin ningún problema y felizmente genera un error masivo. Al final, depende del programador. Siempre que haya duda, ¡es necesario usar un tipo más grande!

Tipos de punto flotante

Como se sabe, los números de punto flotante son números reales: números que permiten dígitos distintos de cero a la derecha de un punto decimal. Esto significa que es posible usar números de punto flotante para abarcar valores fraccionarios: valores que son menores que uno. En la figura 11.3 se muestran los dos tipos de punto flotante: `float` y `double`.

⚠ Observe la columna Precisión en la figura 11.3. Precisión se refiere al número aproximado de dígitos que el tipo puede representar con exactitud. Por ejemplo, puesto que los tipos `float` tienen seis dígitos de precisión, si se intenta almacenar 1.2345678 en una variable `float`, en realidad se almacena una versión redondeada: un número como 1.234568. Los primeros seis dígitos (1.23456) son precisos, pero el resto del número es impreciso. Los valores `double` tienen 15 dígitos de precisión: bastante mejor que los valores `float` con sus seis dígitos de precisión. La precisión relativamente baja de un `float` puede conducir a errores por redondeo significativos cuando se restan dos números próximos en valor. Si los números son suficientemente próximos, entonces la diferencia es un número muy pequeño donde los dígitos que están más a la derecha son meras aproximaciones.

Este error por redondeo surge cuando se tienen cálculos repetitivos. Puesto que la memoria ahora es relativamente poco costosa, se considera que `float` es un tipo de dato arcaico, por lo que es necesario evitar su uso. Una excepción es cuando se especifica un color. Varios métodos en la clase `Color` API de Java emplean parámetros y/o valores de retorno tipo `float`.

Tome en cuenta que los números de punto flotante funcionan peor que los números enteros cuando se trata de tener precisión. Por ejemplo, al comparar el tipo `float` de 32 bites y el tipo `int` de 32 bites, el de punto flotante es menos preciso. Los números `float` tienen seis dígitos de precisión, mientras los números `int` tienen nueve dígitos de precisión. En forma semejante, cuando se comparan el tipo `double` de 64 bits y el tipo `long` de 64 bits, el tipo de punto flotante tiene menos precisión. Los números `double` tienen 15 dígitos de precisión, mientras los números `long` tienen 19. ¿Por qué los números de punto flotante pierden precisión? Algunos de los bits en los números de punto flotante se usan para especificar el exponente que permite que estos números cubran intervalos mucho más grandes en magnitud que el intervalo que pueden cubrir los números enteros. Esto reduce los bits disponibles para suministrar precisión.

Como se aprendió en el capítulo 5, `Float` y `Double` son las clases envoltorio para los tipos de datos `float` y `double`. Para acceder a los valores mínimo y máximo de tipos de datos de punto flotante, se usan las constantes de clase nombradas `MIN_NORMAL` y `MAX_VALUE` de las clases `Float` y `Double`. `MAX_VALUE` es el máximo valor positivo del tipo de datos de punto flotante, y `MIN_NORMAL` es el menor valor positivo de mayor precisión del tipo de datos de punto flotante. Un `MIN_NORMAL` de punto flotante es cualitativamente distinto de un `MIN_VALUE` entero. En lugar de ser un gran valor negativo, un `MIN_NORMAL` de punto flotante es una pequeña fracción positiva. Entonces, ¿cuáles son los límites de los números negativos de punto flotante? El número negativo de mayor magnitud que puede contener una variable de punto flotante es `-MAX_VALUE`. El número negativo de menor magnitud que una variable de punto flotante puede contener sin ningún riesgo es `-MIN_NORMAL`, que es una pequeña fracción negativa.

En realidad, es posible que una variable de punto flotante contenga un número cuya magnitud es menor que `MIN_NORMAL`. Puede contener un valor tan pequeño como un `MIN_VALUE` de punto flotante, que es aproximadamente $1.4 * 10^{-45}$ para `float` y aproximadamente $4.9 * 10^{-324}$ para `double`. Pero el `MIN_VALUE` de un número de punto flotante sólo tiene un bit de precisión, lo cual puede originar un error significativo en el cálculo de un resultado: sin ninguna indicación explícita de que está presente un error. Éste es un ejemplo del peor tipo de error, porque puede permanecer sin identificar durante bastante tiempo. En consecuencia, con números de punto flotante, siempre debe usarse `MIN_NORMAL` en lugar de `MIN_VALUE`.

La constante por defecto del tipo punto flotante es `double`. Si una variable se declara como `float`, debe agregarse un sufijo `f` o `F` a todas las constantes de punto flotante que van ahí, como se muestra a continuación:

```
float gpa1 = 3.22f;
float gpa2 = 2.75F;
float gpa3 = 4.0;
```

error de compilación, porque 4.0 es un `double`

Debido a que la `f` y la `F` son sufijos, 3.22f y 2.75F son valores `float` de 32 bits, de modo que es legal asignarlos a las variables `float gpa1` y `gpa2` de 32 bits. Pero 4.0 es un valor `double` de 64 bits, e intentar asignarlo a la variable `float gpa3` de 32 bits genera un error de compilación.

Para escribir un número de punto flotante en notación científica, es necesario escribir `e` o `E` antes del valor del exponente base 10. Si el exponente es negativo, debe insertarse un signo menos entre la `e` o la `E` y el valor del exponente. Si el exponente es positivo, puede usarse un signo más después de la `e` o la `E`, aunque ésta no es una práctica estándar. En cualquier caso, jamás debe haber ningún espacio en blanco dentro de la especificación del número. Por ejemplo:

```
double x = -3.4e4;
double y = 5.6E-4;
```

equivalente a −34000.0

equivalente a 0.00056

11.3 Tipo `char` y el conjunto de caracteres ASCII

Esta sección complementa el material sobre el tipo `char` que se estudió en la sección 3.20 del capítulo 3.

Valores numéricos subyacentes

Para la mayor parte de lenguajes de programación, incluyendo Java, cada carácter posee un valor numérico subyacente. Por ejemplo, el carácter 'A' posee el valor subyacente 65 y el carácter 'B' posee el valor subyacente 66. Para la mayor parte de los lenguajes de programación, incluyendo Java, los valores numéricos de los caracteres se obtienen del conjunto de caracteres *American Standard Code for Information Interchange* (ASCII, que se pronuncia "aski"). Vea el conjunto de caracteres ASCII en la tabla ASCII en la figura 11.4 y confirme que el valor subyacente del carácter 'A' es 65.

Entonces, ¿de qué se trata el hecho de tener valores numéricos subyacentes para los caracteres? Con los valores numéricos subyacentes, para la JVM es más fácil determinar el orden de los caracteres. Por ejemplo, puesto que el valor del carácter 'A' es 65 y el valor del carácter 'B' es 66, la JVM puede determinar fácilmente que 'A' está antes que 'B'. Y conocer el orden de los caracteres es necesario para realizar operaciones de ordenamiento de cadenas. Por ejemplo, suponga que a las cadenas "peach", "pineapple" y "apple" se les aplicará un método de ordenamiento que compara los primeros caracteres 'p', 'p' y 'a' de las palabras y al hacerlo, la JVM busca los caracteres en el código ASCII. Como el valor de 'p' es 112 y el de 'a' es 97, "apple" va primero. Luego, el método de ordenamiento compara los segundos caracteres en "peach" y "pineapple". Puesto que el valor de 'e' es 10 y el valor de 'i' es 105, "peach" va antes que "pineapple".

La mayor parte de los caracteres en el conjunto de caracteres ASCII representan símbolos que pueden imprimirse. Por ejemplo, el carácter 'f' representa la letra imprimible *f*. Pero los 32 primeros caracteres y el último carácter en el conjunto de caracteres ASCII son diferentes: son *caracteres de control*.

Estos caracteres ejecutan operaciones que no son de impresión. Por ejemplo, el carácter de inicio de encabezado (valor numérico ASCII igual a 1) ayuda con los datos enviados desde un dispositivo de una computadora a otro. Más específicamente, indica el inicio de los datos transmitidos. Cuando se imprime un carácter de control, podría ser sorprendente lo que aparece en la pantalla. El carácter campana (valor numérico ASCII de 7) normalmente genera un sonido y no exhibe nada, lo cual tiene sentido, pero el carácter de inicio de encabezado muestra algo menos intuitivo. Cuando se imprime el carácter de inicio de encabezado, se obtienen resultados diferentes en entornos diferentes. Por ejemplo, en la ventana de una consola[1] en un entorno Windows se muestra una carita sonriente. En otros entornos aparece un cuadrado en blanco. Observe el siguiente fragmento de código, con salida relacionada desde una ventana de una consola en un entorno Windows:

```
char ch;
for (int code=1; code<=6; code++)
{
  ch = (char) code;
  System.out.print(ch + " ");
}
```

Salida:

☺ ☻ ♥ ♦ ♣ ♠

En el fragmento de código anterior, el operador tipo cast (`char`) usa la tabla ASCII para devolver el carácter asociado con el valor numérico de `code`. Por tanto, si el valor de `code` es 1, entonces el `code` (`char`) devuelve el carácter de inicio del encabezado.

El conjunto de caracteres ASCII fue útil durante los primeros años de la programación de computadoras, aunque ya no es suficiente. Algunas veces se requieren símbolos y caracteres que no están en el conjunto de caracteres ASCII. Por ejemplo, suponga que se desea mostrar una marca de verificación (√) o el símbolo pi (π). Estos dos caracteres no aparecen en la figura 11.4; forman parte de un nuevo esquema de codificación denominado *Unicode*, que es un superconjunto de ASCII. Para conocer más sobre Unicode, puede consultar la sección opcional al final de este capítulo (sección 11.13). En esa sección se muestra cómo acceder a la marca de verificación y al símbolo pi, así como a los muchos otros caracteres enumerados en el estándar Unicode.

Uso del operador + con chars

¿Recuerda el lector cómo puede usar el operador + para concatenar dos cadenas? Este operador también puede usarse para concatenar un `char` con una cadena. Observe este ejemplo:

```
char first = 'J';
char last = 'D';
System.out.println("Hello, " + first + last + '!');
```

Salida:

```
¡HolaJD!
```

Cuando la JVM ve una cadena próxima a un signo +, ejecuta la concatenación al convertir primero en una cadena el operando en el otro lado del signo +. Así, en el ejemplo anterior, la JVM convierte la variable `first` en una cadena y luego concatena la "J" resultante con el fin de "Hello", para formar "Hello, J". La JVM hace lo mismo con cada uno de los dos siguientes caracteres que ve, el carácter almacenado `last` y '!'. Convierte cada uno en una cadena y concatena cada uno con la cadena a su izquierda.

Tome en cuenta que si aplica el operador + a dos caracteres, el operador + no realiza ninguna concatenación; ejecuta adición matemática usando los valores subyacentes ASCII de los caracteres. Observe este ejemplo:

[1] Vea la sección 1.8, Primer programa: Hola mundo del capítulo, para una descripción de cómo ejecutar un programa en la ventana de una consola.

valor numérico	carácter	valor numérico	carácter	valor numérico	carácter	valor numérico	carácter
0	null	32	space	64	@	96	`
1	inicio del encabezado	33	!	65	A	97	a
2	inicio del texto	34	"	66	B	98	b
3	fin del texto	35	#	67	C	99	c
4	fin de la transmisión	36	$	68	D	100	d
5	pregunta	37	%	69	E	101	e
6	reconocimiento	38	&	70	F	102	f
7	campana audible	39	'	71	G	103	g
8	retroceso	40	(	72	H	104	h
9	tabulador horizontal	41	)	73	I	105	i
10	avance de línea	42	*	74	J	106	j
11	tabulador vertical	43	+	75	K	107	k
12	avance de forma	44	,	76	L	108	l
13	regreso de carro	45	-	77	M	109	m
14	desplazamiento hacia fuera	46	.	78	N	110	n
15	desplazamiento hacia dentro	47	/	79	O	111	o
16	escape de liga de datos	48	0	80	P	112	p
17	dispositivo de control 1	49	1	81	Q	113	q
18	dispositivo de control 2	50	2	82	R	114	r
19	dispositivo de control 3	51	3	83	S	115	s
20	dispositivo de control 4	52	4	84	T	116	t
21	reconocimiento negativo	53	5	85	U	117	u
22	tiempo muerto síncrono	54	6	86	V	118	v
23	bloque de fin de transmisión	55	7	87	W	119	w
24	cancelar	56	8	88	X	120	x
25	fin del medio	57	9	89	Y	121	y
26	sustituto	58	:	90	Z	122	z
27	escape	59	;	91	[	123	{
28	separador de archivos	60	<	92	\	124	|
29	separador de grupos	61	=	93	]	125	}
30	separador de registros	62	>	94	^	126	~
31	separador de unidades	63	?	95	_	127	borrar

Figura 11.4 Tabla ASCII
Estos caracteres y sus valores de código son los mismos que los 128 primeros caracteres en Unicode, que se analiza en la sección 11.13.

```
char first = 'J';
char last = 'D';
System.out.println(first + last + ", ¿Qué hay?");
```

Salida:

```
142, ¿Qué hay?
```

La salida que se pretende es: `JD, ¿Qué hay?` ¿Por qué el fragmento de código imprime 142 en lugar de JD? La JVM evalúa los operadores + (así como la mayor parte de los otros operadores) de izquierda a derecha, de modo que al evaluar el argumento de `println`, primero evalúa `first + last`. Puesto que `first` y `last` son variables `char`, la JVM realiza adición matemática usando los valores subyacentes

ASCII de los caracteres. `first` contiene a 'J' y el valor de J es 74. `last` contiene a 'D' y el valor de J es 68. Así, `first + last` se evalúa en 142.

Hay dos formas para arreglar el código anterior. Es posible cambiar las dos primeras líneas a inicios de cadenas como sigue:

```
String first = "J";
String last = "D";
```

O es posible insertar una cadena vacía a la izquierda del argumento de `println` como sigue:

```
System.out.println("" + first + last + ", What's up?");
```

11.4 Conversiones de tipo

Esta sección complementa el material sobre conversión de tipo que se estudió en la sección 3.19 del capítulo 3.

Java es un lenguaje de programación *enérgicamente escrito*, de modo que cada variable y cada valor dentro del programa se definen como poseedores de un tipo de datos particular. Así como ocurre con todos los lenguajes de programación enérgicamente escritos, es necesario tener cuidado cuando se trabaja con más de un tipo de datos. En esta sección se aprenderá cómo algunos, pero no todos, los tipos de datos se convierten en otros tipos de datos. Java efectúa en forma automática algunas conversiones de tipo, y permite obligar algunas otras conversiones de tipo. En cualquier caso, debe tenerse cuidado. Las conversiones inadecuadas de tipo pueden causar problemas.

Para imaginarse lo que es permitido en términos de conversiones de tipo, aprenda el esquema de orden en la figura 11.5. En términos crudos, esta imagen muestra cuáles tipos pueden "quedarle" a otros tipos. Por ejemplo, un valor `byte` con ocho bits puede quedarle a una variable `short` que contiene 16 bits porque un ente de ocho bits es más "estrecho" que uno de 16. En este texto se usan los términos "más estrecho" y "más ancho" para describir los tamaños del tipo, aunque debe tomarse en cuenta que éstos no son términos formales; otras personas no los usan. Observe que el tipo `boolean` no aparece en esta figura. No es posible convertir entre los tipos numérico y `boolean`.

Promoción

Hay dos clases de conversión de tipo: la *promoción* (conversión de tipo automática) y el *type casting* (conversión de tipo forzada). Aunque esta última conversión ya se ha estudiado, se hará un breve repaso, pero primero se analizará la promoción.

Una promoción es una conversión implícita. Es cuando un tipo de operando se convierte automáticamente sin necesidad de usar un operador tipo `cast`. Ocurre cuando hay un intento por usar un tipo más estrecho en un sitio que espera un tipo más ancho; es decir, ocurre cuando se sigue el flujo indicado por las flechas en la figura 11.5. La promoción a menudo ocurre en declaraciones de asignación. Si la expresión a la derecha de una declaración de asignación se evalúa como un tipo más estrecho que el tipo de la variable a la izquierda de la declaración de asignación, entonces durante la asignación el tipo más estrecho a la derecha es promovido al tipo más ancho a la izquierda. Observe estos ejemplos de promoción:

```
long x = 44;
float y = x;
```

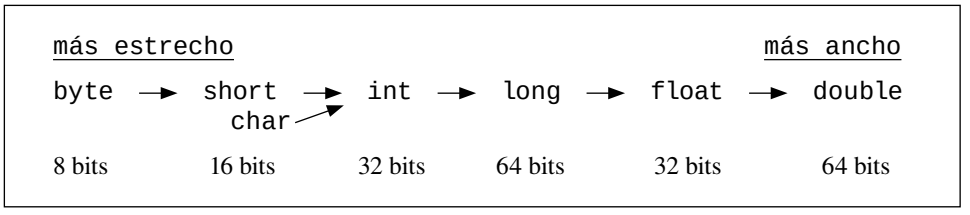


Figura 11.5 Esquema de orden en la conversión de tipo.

En la primera declaración, 44 es un `int`. El `int` 44 es más estrecho que el `long` `x`, de modo que la JVM promueve 44 a un `long` y luego realiza la asignación. En la segunda declaración de asignación, `x` es un `long`. El `long` `x` es más estrecho que el `float` de modo que la JVM promueve `x` a un `float` y luego realiza la asignación.

Observe estos ejemplos adicionales de promoción:

```
double z = 3 + 4.5;
int num = 'f' + 5;
```

expresiones mixtas

Las expresiones de la derecha son *expresiones mixtas*. Una expresión mixta contiene operandos de diferentes tipos de datos. Dentro de una expresión mixta, el operando más estrecho promueve automáticamente al tipo de operando más ancho.

En la primera declaración anterior, el `int` 3 es más estrecho que el `double` 4.5, de modo que la JVM promueve 3 a `double`, antes de sumarlo a 4.5. En la segunda declaración anterior, ¿se sabe qué operando, 'f' o 5, es promovido para coincidir con el otro? 'f' es un `char` y 5 es un `int`, y en la figura 11.5 se muestra que `char` es más estrecho que `int`. Por tanto, la JVM promueve 'f' a un `int`. Más específicamente, puesto que el valor numérico subyacente de f es 102 (véase la figura 11.4), la JVM promueve 'f' a 102. Luego, la JVM suma 102 a 5 y el 107 resultante lo asigna a `num`.

La promoción suele ocurrir como parte de declaraciones de asignación, expresiones mixtas y llamadas a métodos. Ya se han visto ejemplos con declaraciones de asignación y expresiones mixtas; a continuación se abordarán promociones con llamadas a métodos. Como ya se mencionó, las conversiones se llevan a cabo siempre que hay un intento por usar un tipo más estrecho en un sitio que espera un tipo más ancho. Entonces, si un argumento se pasa a un método y el parámetro del método está definido como un tipo más ancho que el tipo del argumento, entonces el tipo del argumento se promociona a fin de coincidir con el tipo del parámetro. El programa de la figura 11.6 muestra un ejemplo de este comportamiento. ¿Puede determinar el lector qué promoción se lleva a cabo dentro del programa? El argumento `x` es un `float` y promueve a un `double`. El argumento 3 es un `int` y también se promueve a un `double`.

```
/************************************************
* MethodPromotion.java
* Dean & Dean
*
* Promote type in method call
************************************************/

public class MethodPromotion
{
  public static void main(String[] args)
  {
    float x = 4.5f;

    printSquare(x);
    printSquare(3);
  }

  private static void printSquare(double num)
  {
    System.out.println(num * num);
  }
} // end class MethodPromotion
```

promoción automática

Salida:

```
20.25
9.0
```

Figura 11.6 Programa que demuestra la promoción de tipo en la llamada a un método.

Type casting

El type casting es uno de conversión de tipo explícita. Ocurre cuando se usa un operador tipo cast para convertir el tipo de una expresión. He aquí la sintaxis para usar un operador tipo cast:

(*type*) *expression*

Es legal usar un operador tipo cast para convertir cualquier tipo numérico en cualquier otro tipo numérico; es decir, la conversión puede ir en cualquier dirección del diagrama del esquema de orden en la figura 11.5. Por ejemplo, el siguiente fragmento de código convierte la `x double` en la `y int`.

```
double x = 12345.6;
int y = (int) x;
System.out.println("x = " + x + "\ny = " + y);
```

¿Qué ocurre si se omite el operador tipo cast (`int`)? Se obtiene un error de compilación porque se estaría asignando directamente un `double` a un `int` y eso está prohibido (en el diagrama del esquema de orden en la figura 11.5 no hay ninguna flecha del tipo `double` al tipo `int`). ¿Por qué es ilegal asignar directamente un número de punto flotante a un `int`? Porque los números de punto flotante pueden tener fracciones y los `int` no pueden manejar fracciones.

¿Sabe el lector lo que imprime el fragmento de código anterior? `x` permanece sin cambio [aun cuando se le aplicó (`int`)] y `y` obtiene la porción entera de `x` con la fracción de `x` truncada, no redondeada. De modo que ésta es la salida:

```
x  = 12345.6
y  = 12345
```

El programa en la figura 11.7 ilustra aún más el uso de los operadores tipo cast. Solicita al usuario que introduzca un valor ASCII (un valor entero entre 0 y 127). Luego imprime el carácter asociado con ese valor ASCII y también el siguiente carácter en la tabla ASCII. En el programa, ¿qué hacen los dos operadores tipo cast? El primero devuelve la versión `char` de `asciiValue`, una variable `int`. El segundo devuelve la versión `char` de `asciiValue + 1`. Las operaciones de conversión de tipo son necesarias para imprimir `ch` y `nextCh` como caracteres, en lugar de hacerlo como enteros. ¿Qué ocurriría si se omitieran los operadores tipo cast? Se obtendrían errores de compilación, porque se estaría asignando directamente un `int` a un `char`, lo cual está prohibido según el esquema de orden en la figura 11.5.

¿Por qué es ilegal asignar directamente un número a `char`? Podría pensarse que es seguro asignar un número entero pequeño, como un `byte` con ocho bits, a un `char` con 16 bits. Es ilegal asignar directamente un número a un `char` porque los números pueden ser negativos y un `char` no es capaz de manejar este tipo de números (el valor subyacente de un `char` es un número positivo entre 0 y 65535).

11.5 Modos prefijo/sufijo para operadores de incremento/decremento

Esta sección complementa el material que se estudió en la primera parte del capítulo 3, en la sección 3.17 (Operadores incremento/decremento), y usa técnicas ya estudiadas en el capítulo 3, en la sección 3.18 (Rastreo).

El operador incremento tiene dos modos distintos: el *modo prefijo* y el *modo sufijo*. El modo prefijo es cuando se escribe `++` antes de la variable que ha de incrementarse. El uso del modo prefijo provoca que la variable se incremente antes que se use el valor de la variable. Por ejemplo:

```
y = ++x        es equivalente a      x = x + 1;
                                     y = x;
```

El modo sufijo ocurre cuando se escribe `++` después de la variable que ha de incrementarse. El uso del modo sufijo provoca que la variable se incremente después que se usa el valor de la variable. Por ejemplo:

```
y = x++        es equivalente a      y = x;
                                     x = x + 1;
```

```
/*************************************************************
* PrintCharFromAscii.java
* Dean & Dean
*
* This illustrates manipulation of ASCII code values.
*************************************************************/

import java.util.*;

public class PrintCharFromAscii
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int asciiValue;  // user entered ASCII value
    char ch;         // the asciiValue's associated character
    char nextCh;     // the character after ch in the ASCII table

    System.out.print("Enter an integer between 0 and 127: ");
    asciiValue = stdIn.nextInt();
    ch = (char) asciiValue;
    nextCh = (char) (asciiValue + 1);
    System.out.println("Entered number: " + asciiValue);
    System.out.println("Associated character: " + ch);
    System.out.println("Next character: " + nextCh);
  } // end main
} // end class PrintCharFromAscii
```

Observe los operadores tipo cast (`char`).

Sesión muestra:

```
Enter an integer between 0 and 127: 67
Entered number: 67
Associated character: C
Next character: D
```

Figura 11.7 Programa que ilustra el uso de cast para convertir códigos de caracteres en caracteres.

Para tener una mejor impresión de cómo funciona, rastree este fragmento de código:

```
1  int x, y;
2
3  x = 4;
4  y = ++x;
5  System.out.println(x + " " + y);
6  x = 4;
7  y = x++;
8  System.out.println(x + " " + y);
```

He aquí el rastreo:

línea#	x	y	salida
1	?	?	
3	4		
4	5		
4		5	
5			5 5
6	4		
7		4	
7	5		
8			5 4

Preste atención a las comillas.

He aquí una pregunta de repaso para ayudarlo a desarrollar sus habilidades para depurar un programa. ¿Cuáles hubieran sido los resultados si los argumentos `println` hubieran sido (`x + ' ' + y`)? En lugar de especificar la versión de cadena de un espacio, esto hubiera especificado la versión de carácter de un espacio, lo cual hubiera hecho que la computadora considerase el argumento como una expresión matemática en lugar de una concatenación de cadenas. Puesto que `x` y `y` son enteros, hubiera promovido el carácter espacio a su valor numérico subyacente, que es 32 (vea la figura 11.4). La primera declaración de impresión hubiera sumado (5 + 32 + 5) e impreso 42. La segunda declaración hubiera sumado (5 + 32 + 4) e impreso 41.

Los modos prefijo y sufijo del operador decremento funcionan igual que el operador incremento, aunque restan uno en vez de sumar uno. Para tener una idea de cómo funcionan, rastree este fragmento de código:

```
1  int a, b, c;
2
3  a = 8;
4  b = --a;
5  c = b-- + --a;
6  System.out.println(a + " " + b + " " + c);
```

línea#	a	b	c	salida
1	?	?	?	
3	8			
4	7			
4		7		
5	6			
5			13	
5		6		
6				6 6 13

A continuación se analizará la línea 5 con más detalle:

```
c = b-- + --a;
```

Como puede adivinarse, al ejecutar esta declaración, la JVM primero disminuye el valor de `a`. Esto debe tener sentido cuando se consulta la tabla de prioridad del operador en el apéndice 2 y se confirma que el operador decremento tiene una prioridad muy alta. La JVM también ejecuta el operador decremento pronto, aunque su ejecución consiste en usar el valor original de `b` e incrementar a `b` después. La tabla de precedencia de los operadores muestra que el operador `+` tiene prioridad más alta que el operador `=`, de modo que la JVM suma a continuación el valor original de `b` al valor disminuido de `a`. Finalmente, la JVM asigna la suma a `c`.

Para muchas personas, la línea 5 resulta particularmente confusa. Este ejemplo se presentó porque quizás el lector podría observar este tipo de situación en el código de otra persona, pero si desea que su código sea comprensible, le recomendamos no hacer esto. Es decir, no inserte expresiones `++` o `--` en otras expresiones. En lugar de intentar hacer en una declaración todo lo que hace la línea 5, sería más comprensible dividir la línea 5 en tres declaraciones por separado, como se muestra a continuación:

```
5a  a--;
5b  c = b + a;
5c  b--;
```

De todas formas, la JVM efectúa la evaluación en pasos por separado, de modo que al escribirla no incurre en ninguna penalización de desempeño. Requiere más espacio en la página, pero la mayoría de las personas estará de acuerdo en que es más fácil de leer.

Cuando se escribe un código, ¿cómo se decide qué modo usar: prefijo o sufijo? Depende del resto del código. Casi siempre, a fin de evitar que el código sea confuso, las operaciones incremento y decremento se colocan en líneas distintas. Así, no importa cuál modo se use, aunque el más común es el sufijo.

11.6 Asignaciones insertadas

Esta sección complementa el material que se estudió en los capítulos 3 y 4. Específicamente, complementa la sección 3.11 del capítulo 3, sobre declaraciones de asignación y el material del ciclo `while` del capítulo 4, sección 4.8.

Inserción de una asignación en otra asignación

Algunas veces, las asignaciones se insertan como expresiones en declaraciones más grandes. Cuando esto ocurre, recuerde que 1) una expresión de asignación se evalúa en el valor asignado y 2) los operadores de asignación presentan asociatividad de derecha a izquierda. Para ver estos conceptos en acción considere este fragmento de código:

```
1  int a, b = 8, c = 5;
2
3  a = b = c;
4  System.out.println(a + " " + b + " " + c);
```

lo mismo que: `a = (b = c);`

La línea 3 muestra una expresión de asignación insertada dentro de una declaración de asignación más grande. ¿Cuál de los dos operadores de asignación ejecuta primero la JVM? Puesto que los operadores de asignación presentan asociatividad de derecha a izquierda, la JVM ejecuta primero la operación de asignación de la derecha. ¿En qué se evalúa la expresión `b = c`? Se evalúa en 5 porque el valor asignado, `c`, es 5. Al evaluar la línea 3, la parte `b = c` de la declaración se sustituye con 5 a fin de reducir la declaración a:

```
a = 5.
```

He aquí cómo se vería el rastreo del fragmento de código:

línea#	a	b	c	salida
1	?	8	5	
3		5		
3	5			
4				5 5 5

Inserción de una asignación en una condición de ciclo

Salvo por una asignación pura múltiple como `a = b = c`, resulta mejor evitar insertar asignaciones múltiples como expresiones en otras declaraciones, ya que eso dificulta la comprensión del código. No obstante, suele ser práctica común insertar una sola asignación como una expresión en una condición de ciclo. Por ejemplo, la figura 11.8 contiene un programa que promedia un conjunto de puntajes de entrada. Observe la asignación (`score = stdIn.bextDouble()`) dentro de la condición `while`. Si, por ejemplo, el usuario responde al desplegado introduciendo 80, entonces `score` adquiere el valor 80, la expresión de asignación dentro del paréntesis se evalúa en 80 y el encabezado del ciclo `while` se convierte en:

```
while (80 != -1)
```

Puesto que la condición es verdadera, la JVM ejecuta el cuerpo del código. Si la expresión de asignación no estuviese insertada en la condición del ciclo `while`, aparecería dos veces: una arriba del encabezado del ciclo y de nuevo en la parte inferior del ciclo. El hecho de insertar la asignación en la condición ayuda a mejorar la estructura del ciclo.

Algunas veces también se observan asignaciones insertadas en argumentos de métodos e índices de arreglos. Esto reduce aún más el código. A veces la compacidad es benéfica en cuanto a que puede condu-

```
/*****************************************************************
* AverageScore.java
* Dean & Dean
*
* This program averages input scores.
*****************************************************************/

import java.util.Scanner;

public class AverageScore
{
  public static void main(String[] args)
  {
    double score;
    double count = 0;
    double totalScore = 0;
    Scanner stdIn = new Scanner(System.in);

    System.out.print("Enter a score (or -1 to quit): ");
    while ((score = stdIn.nextDouble()) != -1)
    {
      count++;
      totalScore += score;
      System.out.print("Enter a score (or -1 to quit): ");
    }
    if (count > 0)
    {
      System.out.println("Average score = " + totalScore / count);
    }
  } // end main
} // end AverageScore class
```

asignación insertada

Figura 11.8 Programa AverageScore que ilustra el uso de asignaciones incrustadas.

cir a un código menos revuelto y así más fácil de comprender. Pero no debe irse demasiado lejos en tratar de reducir el código porque la compacidad algunas veces puede llevar a un código difícil de comprender (es decir, puede producir un código más *críptico*). Algunos programadores han sido despedidos por tratar de hacer programas "inteligentes" lo más resumidos posible. Si éste es el caso del lector, se recomienda redirigir los esfuerzos a intentar escribir programas lo más comprensibles que sea posible. Sigue siendo posible utilizar códigos compactos, pero a manera que ayude, y no que entorpezca, la comprensibilidad.

11.7 Expresiones del operador condicional

Esta sección complementa la sección 4.3 (Declaraciones `if`) del capítulo 4.

Sintaxis y semántica

Cuando se quiere usar una condición lógica para determinar cuál de dos valores alternativos es válido, en lugar de utilizar la forma "si, entonces" de la declaración `if`, puede usarse una expresión de operador condicional. El operador condicional es el único operador *ternario* de Java. Ternario significa tres. El condicional relaciona tres operandos con los dos símbolos, `?` y `:`. El `?` va entre los operandos primero y segundo, y el `:` va entre los operandos segundo y tercero.

Ésta es la sintaxis:

<*condición*> `?` <*expresión1*> `:` <*expresión2*>

Si la condición es `true`, la expresión de operador condicional se evalúa en el valor de *expresión1* e ignora *expresión2*. Si la condición es `false`, la expresión de operador condicional se evalúa en el valor de *expresión2* e ignora *expresión1*. Considere que *expresión1* es la parte verdadera de una declaración "if, else". Piense que *expresión2* es la parte falsa de una declaración "if, else".

Por ejemplo, considere la expresión

```
(x>y) ? x : y
```

El paréntesis alrededor de la condición no es necesario porque `>` tiene mayor prioridad que el par `?:`, aunque se recomienda usarlo porque mejora la legibilidad. ¿Qué hace la JVM cuando ve esta expresión?

- Compara `x` y `y`.
- Si `x` es mayor, asigna a la expresión el valor de `x`.
- Si `x` no es mayor, asigna a la expresión el valor de `y`.

¿Sabe el lector cuál es la funcionalidad general que implementa la expresión? Encuentra el máximo de dos números. Esto puede demostrarse introduciendo números de muestra. Suponga x = 2 y y = 5. He aquí cómo la expresión evalúa el máximo, 5:

```
(2>5) ? 2 : 5  ⇒
(false) ? 2 : 5  ⇒
5
```

Uso del operador condicional

Una expresión de operador condicional no puede aparecer por sí misma en una línea porque no es una declaración completa. Simplemente es parte de una declaración: una expresión. El siguiente fragmento de código incluye dos ejemplos de expresiones de operador condicional insertadas:

```
int score = 58;
boolean extraCredit = true;

score += (extraCredit ? 2 : 0);
System.out.println(
  "grade = " + ((score>=60) ? "pass" : "fail"));
```

¿Cómo funciona? Puesto que `extraCredit` es `true`, el valor elegido por el primer operador condicional es 2. Luego, `score` aumenta en dos unidades, desde su valor inicial de 58 a 60. Puesto que (`score>=60`) se evalúa en `true`, el valor elegido por el segundo operador condicional es "pass". Luego, la declaración `println` imprime:

```
grade = pass
```

En el fragmento de código anterior es aconsejable que el paréntesis esté como se muestra, aunque en aras de mejorar las habilidades de lector para depurar el programa se analizará lo que ocurre si se omiten los pares entre paréntesis. Como se muestra en la tabla de precedencia de operadores en el apéndice 2, el operador condicional tiene mayor prioridad que el operador `+=`. En consecuencia, sería legal omitir el paréntesis en la declaración de asignación `+=`. En la declaración `println`, el operador condicional tiene menor prioridad que el operador `+`, de modo que es necesario mantener el paréntesis que rodea la expresión del operador condicional. Puesto que el operador `>=` tiene mayor prioridad que el operador condicional, sería legal omitir el paréntesis que rodea la condición `score>=60`. Observe cómo se omiten los espacios en la condición `score>=60` pero se incluyen espacios alrededor del `?` y del `:` que separan los tres componentes de la expresión del operador condicional. Este estilo mejora la legibilidad.

El operador condicional puede usarse para evitar declaraciones `if`. El código del operador condicional podría parecer más eficiente que el código de la declaración `if` porque el código fuente es más corto, aunque el bytecode generado suele ser más largo. Éste es otro ejemplo de algo que el lector podría ver en el código de otra persona, pero debido a que es relativamente difícil de comprender, se recomienda usarlo con moderación en su propio código. Por ejemplo, la declaración `score += (extraCredit ? 2 : 0);` en el fragmento de código anterior es más bien críptica. Un mejor estilo sería incrementar la variable `score` como se muestra:

```
if (extraCredit)
{
  score += 2;
}
```

11.8 Revisión de evaluación de expresiones

Los cálculos manuales ayudan a comprender.

Hasta ahora en este capítulo se han aprendido algunos detalles sobre tipo y sobre operadores. Este aprendizaje será de ayuda para depurar códigos con problemas, así como para evitar problemas en primer lugar. Para tener la certeza de que el lector ha comprendido los detalles, a continuación se resolverán problemas de práctica de evaluación de expresiones.

Práctica de evaluación de expresiones con concatenación de caracteres y cadenas

Observe las tres siguientes expresiones. Intente evaluarlas antes de ver las respuestas ulteriores. Mientras efectúa las evaluaciones, recuerde que si tiene dos o más operadores con la misma prioridad, debe usar asociatividad de izquierda a derecha (es decir, efectuar primero la operación a la izquierda). De modo que en la primera expresión es necesario realizar la operación + en `'1' + '2'` antes de intentar efectuar la segunda operación +.

1. `'1' + '2' + "3" + '4' + '5'`
2. `1 + 2 + "3" + 4 + 5`
3. `1 + '2'`

Éstas son las respuestas:

1.

```
'1' + '2' + "3" + '4' + '5' ⇒
49  + 50 + "3" + '4' + '5' ⇒
99 + "3" + '4' + '5' ⇒
"993" + '4' + '5' ⇒
"9934" + '5' ⇒
"99345"
```

Al sumar dos `chars`, use sus valores numéricos subyacentes ASCII.

Cuando la JVM ve una cadena cerca de un signo +, concatena al convertir primero en una cadena el operando que está al otro lado del signo +.

2.

```
1 + 2 + "3" + 4 + 5 ⇒
3 + "3" + 4 + 5 ⇒
"33" + 4 + 5 ⇒
"334" + 5 ⇒
"3345"
```

La asociatividad izquierda-derecha indica sumar los dos números a la izquierda.

3.

```
1 + '2' ⇒
1 + 50 ⇒
51
```

Expresión mixta: el `char` es promovido a un `int`, usando el valor numérico subyacente ASCII para '2'.

Práctica de evaluación de expresiones con conversión de tipo y varios operadores

Suponga lo siguiente:

```
int a = 5, b = 2;
double c = 3.0;
```

Intente evaluar las siguientes expresiones antes de ver las respuestas ulteriores.

1. `(c + a / b) / 10 * 5`
2. `a + b++`
3. `4 + --c`
4. `c = b = a % 2`

Éstas son las respuestas:

1.

```
(c + a / b) / 10 * 5 ⇒
(3.0 + 5 / 2) / 10 * 5 ⇒
(3.0 + 2) / 10 * 5 ⇒
5.0 / 10 * 5 ⇒
0.5 * 5 ⇒
2.5
```

Expresión mixta: el `int` es promovido a un `double`.

`/` y `*` tienen la misma prioridad. Primero realice la operación a la izquierda.

2.

```
a + b++ ⇒
5 + 2 ⇒
7
```

Use el valor original de `b`, 2, en la expresión. Después, el valor de `b` se incrementa a 3.

3.

```
4 + --c ⇒
4 + 2.0 ⇒
6.0
```

El valor original de `c` disminuye a 2.0 antes de usarlo en esta expresión.

4.

```
c = b = a % 2 ⇒
c = b = 5 % 2 ⇒
c = b = 1 ⇒
c = 1 ⇒
1.0
```

No introduzca valores para variables que no están a la izquierda de las asignaciones.

La asignación `b = 1` se evalúa en `1`.

`c` es un `double`, de modo que el resultado es un `double`.

Más práctica de evaluación de expresiones

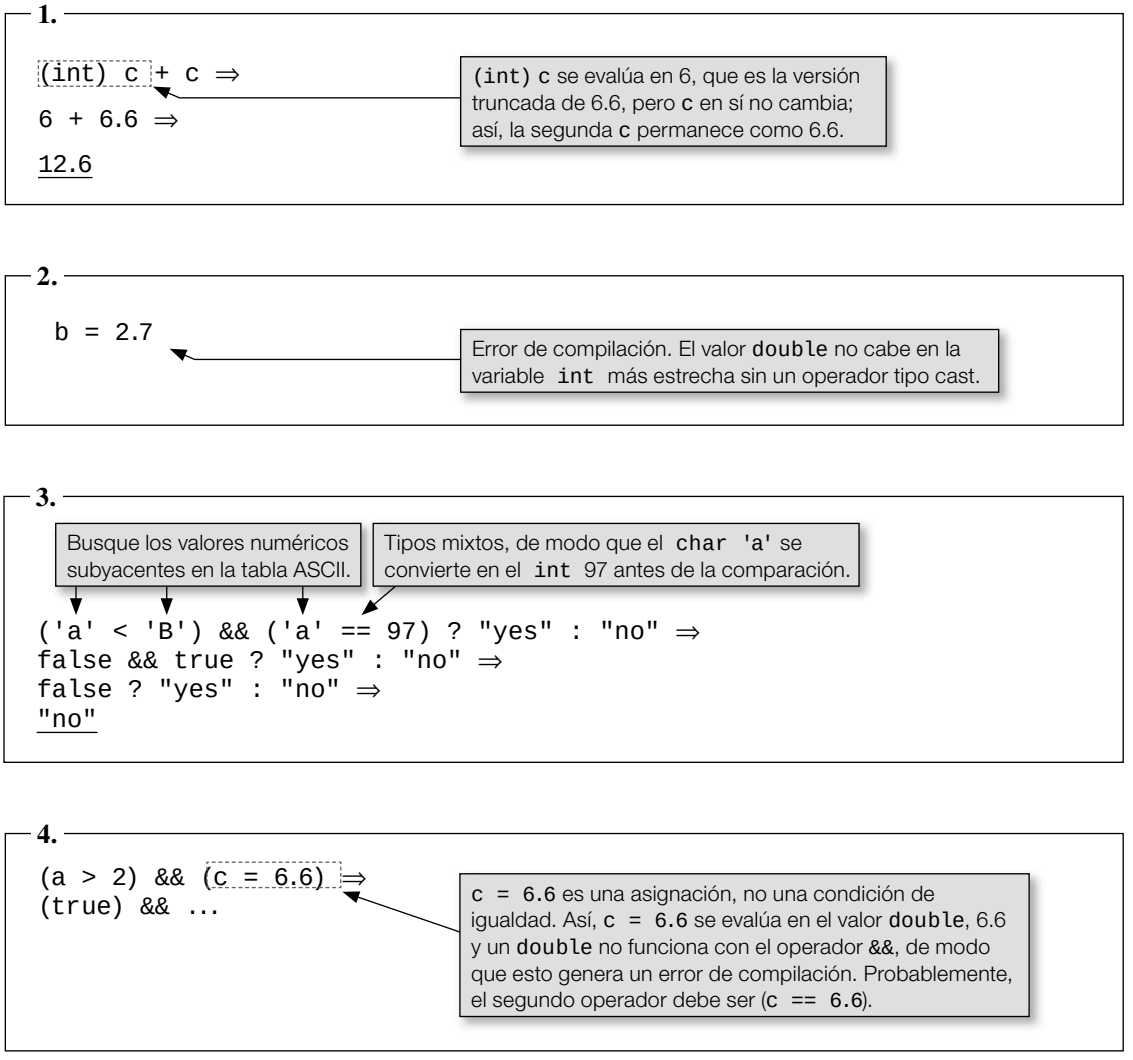
Suponga lo siguiente:

```
int a = 5, b = 2;
double c = 6.6;
```

Intente evaluar las siguientes expresiones antes de ver las respuestas ulteriores.

1. `(int) c + c`
2. `b = 2.7`
3. `('a' < 'B') && ('a' == 97) ? "yes" : "no"`
4. `(a >2) && (c = 6.6)`

Éstas son las respuestas:



11.9 Evaluación de cortocircuito

Esta sección complementa el material sobre el operador lógico && que se estudió en la sección 4.4 del capítulo 4 y el material sobre el operador lógico || que se estudió en la sección 4.5 del capítulo 4.

Considere el programa en la figura 11.9. Calcula el porcentaje de disparos de un jugador de baloncesto e imprime el mensaje asociado. Observe el encabezado de la declaración `if`, que se repite aquí para mayor claridad. En particular, observe la operación de división con `attempted` en el denominador.

```
if ((attempted > 0) && ((double) made / attempted) >= .5)
```

```
/*****************************************************************
* ShootingPercentage.java
* Dean & Dean
*
* This program processes a basketball player's shooting percentage.
*****************************************************************/

import java.util.Scanner;

public class ShootingPercentage
{
  public static void main(String[] args)
  {
    int attempted; // number of shots attempted
    int made;       // number of shots made
    Scanner stdIn = new Scanner(System.in);
    System.out.print("Number of shots attempted: ");
    attempted = stdIn.nextInt();
    System.out.print("Number of shots made: ");
    made = stdIn.nextInt();

    if ((attempted > 0) && ((double) made / attempted) >= .5)
    {
      System.out.printf("Excellent shooting percentage - %.1f%%\n",
        100.0 * made / attempted);
    }
    else
    {
      System.out.println("Practice your shot more.");
    }
  } // end main
} // end class ShootingPercentage
```

Si `attempted` es cero, la división entre cero no ocurre.

Use `%%` para imprimir un signo de porcentaje.

Sesión muestra:

```
Number of shots attempted: 0
Number of shots made: 0
Practice your shot more.
```

Segunda sesión muestra:

```
Number of shots attempted: 12
Number of shots made: 7
Excellent shooting percentage - 58.3%
```

Figura 11.9 Programa que ilustra la evaluación de cortocircuito.

Mediante la división, siempre es necesario pensar en, y no tratar de evitar, la división entre cero. Si `attempted` es igual a cero, ¿la JVM intentará dividir entre cero? ¡No! La evaluación de cortocircuito salva la situación.

Evaluación de cortocircuito significa que la JVM deja de evaluar una expresión siempre que el resultado de la expresión sea seguro. Más específicamente, si el lado izquierdo de una expresión `&&` se evalúa como `false`, entonces el resultado de la expresión es seguro (`false &&` cualquier cosa se evalúa como `false`) y el lado derecho se omite. En forma semejante, si el lado izquierdo de una expresión `||` se evalúa como `true`, entonces el resultado de la expresión es seguro (`true ||` cualquier cosa se evalúa como `true`) y el lado derecho se omite. Así, en la declaración `if` en la figura 11.9, si `attempted` es igual a cero, entonces el lado izquierdo del operador `&&` se evalúa como `false`, y el lado derecho se omite, evitándose así la división entre cero.

Entonces, ¿cuál es el beneficio de la evaluación de cortocircuito?

Utilice el comportamiento preintegrado.

1. Se evitan los errores: puede ayudar a prevenir problemas al permitir evitar una operación ilegal en el lado derecho de una expresión.

2. Desempeño: puesto que ya se conoce el resultado, la computadora no tiene que desperdiciar tiempo en el cálculo del resto de la expresión.

Al margen, observe el %% en la declaración `printf` en la figura 11.9. Se trata de un especificador de conversión para el método `printf`. A diferencia de otros especificadores de conversión, es un ente independiente; no posee ningún argumento que se conecte a él. Simplemente imprime el carácter de porcentaje. Observe el % impreso al final de la segunda sesión muestra en la figura 11.9.

11.10 Declaración vacía

Esta sección complementa el material sobre ciclos que se estudió en el capítulo 4.

Algunas veces es posible colocar toda la funcionalidad de un ciclo dentro de su encabezado. Por ejemplo:

```
for (int i=0; i<1000000000; i++)
{ }
```

El compilador de Java requiere la inclusión de una declaración para el cuerpo del ciclo `for`, inclusive si la declaración no hace nada. Las llaves vacías anteriores ({}) forman una declaración compuesta[2] y satisfacen ese requerimiento. En esta sección se aprenderá una forma alternativa para satisfacer ese requerimiento. Se aprenderá sobre la declaración vacía.

Uso de la declaración vacía

La *declaración vacía* consta de un punto y coma en sí. Use esta declaración en sitios en que el compilador requiera una declaración, aunque no sea necesario hacer nada. Por ejemplo, el siguiente ciclo `for` puede usarse como una forma "rápida y sucia" para agregar un retraso a su programa:

```
monster.display();
for (int i=0; i<1000000000; i++)
  ;
monster.erase();
```

Convención del código: Coloque la declaración vacía en una línea e introduzca sangría.

Observe cómo la declaración vacía es idónea aquí porque todo el trabajo se realiza en el encabezado del ciclo `for`, donde `i` corre hasta mil millones. Para contar todo esto se necesita tiempo. En función de la velocidad de la computadora, podría requerirse desde una fracción de segundo hasta cinco segundos.

Entonces, ¿por qué se desea agregar un retraso al programa? Suponga que se está escribiendo un programa de juegos que requiere la aparición de un monstruo sólo durante cierto intervalo. Para implementar esta función es necesario imprimir el monstruo, ejecutar el ciclo de retraso y luego borrar al monstruo.

Quizá sea aconsejable usar el fragmento de código anterior como parte de un intento de primer corte en la implementación del retraso, pero no lo use para su implementación final. ¿Por qué? Porque introduce retraso que depende de la velocidad de la computadora que ejecuta el programa. Con retrasos variados, las computadoras lentas podrían tener monstruos que se rezagan demasiado y las computadoras rápidas podrían tener monstruos que desaparecen demasiado pronto. En una implementación final, debe tratar de usar el método `sleep` de la clase `Thread` para implementar el retraso. El método `sleep` permite especificar precisamente la cantidad del retraso en milisegundos. Para usar el método `sleep` es necesario comprender el manejo de excepciones, que se analizará en el capítulo 14.[3]

En el fragmento del código anterior, observe la llamada a la convención de codificación. ¿Puede el lector pensar por qué es una buena idea colocar la declaración vacía en una línea por sí misma? Si la declaración vacía se coloca en una línea por sí misma y luego se introduce una sangría, los lectores lo percibirán. Por otra parte, si la declaración vacía se coloca al final de la línea de la declaración previa, quizá

[2] La declaración compuesta, definida en el capítulo 4, es un grupo de ninguna o más declaraciones escritas entre llaves.

[3] Esto agrega un retraso de 1 000 milisegundos (lo cual es igual a 1 segundo):

```
try {Thread.sleep(1000);}
catch (InterruptedException e) { }
```

los lectores no la noten. Ver el código es una parte importante para hacerlo comprensible. Y hacer códigos comprensibles hace más fácil su mantenimiento.

Evitar el uso indebido y accidental de la declaración vacía

Una situación bastante común para los programadores es la creación accidental de declaraciones vacías. Debido a que en la mayor parte de las líneas del código Java se escribe un punto y coma, resulta fácil adquirir la costumbre de escribir el punto y coma al final de cada línea de código que se escribe. Si esto se hace al final del encabezado de un ciclo, se genera una declaración vacía. Puede ser que el código compile y se ejecute sin que se reporte ningún error, aunque los resultados podrían ser misteriosos. He aquí un ejemplo:

```
System.out.print("¿Quieres jugar un juego (s/n)? ");
while (stdIn.next().equals("y"));
{
  <The code to play the game goes here.>
  System.out.print("¿Jugar otro juego (s/n)? ");
}
```

Este punto y coma crea una declaración vacía.

El punto y coma al final del encabezado del ciclo `while`, ¿genera un error de compilación? No: el punto y coma actúa como la única declaración (una declaración vacía) que está dentro del ciclo `while`. Los paréntesis de llave ulteriores forman una declaración compuesta. La declaración compuesta no es parte del ciclo `while`; se ejecuta después de la terminación del ciclo `while`.

Entonces, ¿qué hace el código? Primero, suponga que el usuario introduce `n`. En el encabezado del ciclo `for`, la JVM compara el valor introducido n con 'y'. La condición del ciclo es `false`, de modo que la JVM omite el cuerpo del ciclo `while`, la declaración vacía. Luego, la JVM ejecuta la declaración compuesta e intenta jugar un juego. Éste es un error lógico: la JVM intenta jugar un juego aun cuando el usuario ha introducido `n`.

Ahora suponga que el usuario introduce y. En el encabezado del ciclo `while`, la JVM compara el valor introducido y con 'y'. La condición del ciclo es `true`, de modo que la JVM calcula el cuerpo del ciclo `while`, la declaración vacía. Luego, la JVM regresa al encabezado del ciclo `for` y ejecuta nuevamente la llamada al método `stdIn.next()`. La JVM espera que el usuario introduzca otro valor. Pero el usuario no sabe que se supone que debe introducir algo porque no hay mensaje. Éste es un error lógico especialmente desagradable porque el programa no produce ninguna salida errónea y tampoco ningún mensaje de error. Esto significa que no hay ninguna ayuda en cuanto a qué hacer.

Es posible producir estos mismos tipos de errores lógicos al escribir un punto y coma después de los encabezados "if", "else if" o "else". Estos punto y coma efectivamente crean declaraciones vacías, que a menudo son introducidas accidentalmente durante el desarrollo o la depuración del programa. Esté alerta a las declaraciones vacías, y siempre que vea una, ¡sospeche y verifíquela! Mejor aún, minimice la confusión al final maximizando las precauciones al principio.

La prisa produce errores.

11.11 Declaración `break` dentro de un ciclo

Esta sección complementa el material sobre ciclos que se estudió en el capítulo 4.

En el capítulo 4 se introdujo el uso de la declaración `break` dentro de una declaración `switch`. Termina la declaración `switch` y transfiere el control a la siguiente declaración después de la declaración `switch`. Además, la declaración `break` puede usarse dentro de un ciclo `while`, `do` o `for`. Hace lo mismo que cuando está dentro de una declaración `switch`. El `break` termina el ciclo que está inmediatamente encerrado y transfiere el control a la siguiente declaración después de la parte inferior del ciclo. Se dice "inmediatamente encerrado" porque es posible tener un `break` que esté anidado dentro de múltiplos ciclos. El `break` está asociado con el ciclo que lo rodea inmediatamente.

El programa DayTrader en la figura 11.10 ilustra lo que se denomina "comercio de días". Es una forma de jugar donde la gente compra y vende acciones en la Bolsa cada día con la esperanza de ganar

más dinero mediante el movimiento a corto plazo de la Bolsa. Este programa sigue la pista del balance de un corredor durante un periodo de tres meses (para los días = 1 a 90). El balance original es $1 000. En este modelo simple, al inicio de cada día, el corredor retiene la mitad del balance inicial en ahorros e invierte la otra mitad en la Bolsa de valores. El rendimiento de las inversiones al final del día es igual a la inversión multiplicada por un número aleatorio entre 0 y 2. Así, el rendimiento de las inversiones varía desde cualquier punto a partir de cero hasta el doble de la inversión original. Cada día, el corredor suma al balance en ahorros el rendimiento de las inversiones. Si el balance disminuye por abajo de $1 o crece por arriba de $5 000, el corredor renuncia.

Antes de analizar la declaración `break` en la figura 11.10, observe el argumento (`day - 1`) en la declaración final `printf`. Se encuentra después del ciclo `for`, de modo que el alcance de `day` necesita ser más grande que el alcance del ciclo `for`. Ésta es la razón de que se haya declarado antes del ciclo `for` con las otras variables locales. Pero, ¿por qué se restó 1 en la declaración `printf`? Debido a que la operación `day++` en el tercer compartimiento del encabezado del ciclo `for` incrementa `day` una vez más, después de la transacción que lleva el balance a un valor de terminación. Si se ha olvidado restar 1 en la declaración `printf`, éste sería un error cometido por uno.

Ahora observe la declaración `break` del programa DayTrader. Si alguna vez el balance se sale del intervalo de $1 a $5 000, el control del programa salta inmediatamente a la siguiente declaración abajo del ciclo `for`. Si el programa se ejecuta varias veces se verá que algunas veces esto origina que el ciclo termine antes que `day` llegue a 90. Cada que se ejecuta el programa se obtiene un resultado diferente porque este programa usa `Math.random` para generar un número aleatorio en el intervalo entre 0.0 y 1.0.

```
/*************************************************************
* DayTrader.java
* Dean & Dean
*
* This simulates stock market day trading.
*************************************************************/

public class DayTrader
{
  public static void main(String[] args)
  {
    double balance = 1000.00; // money that's retained
    double moneyInvested;     // money that's invested
    double moneyReturned;     // money that's earned at end of day
    int day;                  // current day, ranges from 1 to 90

    for (day=1; day<=90; day++)
    {
      if (balance < 1.0 || balance > 5000.0)
      {
        break;
      }
      balance = moneyInvested = balance / 2.0;
      moneyReturned = moneyInvested * (Math.random() * 2.0);
      balance += moneyReturned;
    } // end for

    System.out.printf("final balance on day %d: $%4.2f\n",
      (day - 1), balance);
  } // end main
} // end DayTrader
```

Figura 11.10 Programa DayTrader que ilustra el uso de la declaración `break`.

Tome en cuenta que en realidad nunca es necesario usar una declaración `break` para implementar esta capacidad de terminación prematura del ciclo. Por ejemplo, las declaraciones `if` y `break` del programa DayTrader pueden eliminarse si el encabezado del ciclo `for` se cambia a:

```
for (day=1; day<=90 && !(balance < 1.0 || balance > 5000.0); day ++)
```

No caiga en la trampa de usar la declaración `break` muy a menudo. Usualmente, alguien que lea su programa sólo verá el encabezado del ciclo para imaginar cómo termina el ciclo. Al usar la declaración `break`, se obliga al lector a buscar dentro del ciclo las condiciones de terminación de éste. Y esto hace más difícil de comprender su programa. A pesar de ello, en ciertas situaciones, la declaración `break` mejora la legibilidad, en lugar de entorpecerla. La declaración `break` del programa DayTrader es un ejemplo de dónde esta declaración mejora la legibilidad.

11.12 Detalles para el encabezado del ciclo `for`

Esta sección complementa el material del ciclo `for` que se estudió en la sección 4.10 del capítulo 4.

Omisión de uno o más componentes del encabezado del ciclo `for`

Se permite, aunque no es tan común, omitir los componentes primero y/o tercero en el encabezado del ciclo `for`. Por ejemplo, para imprimir una cuenta regresiva a partir de un número introducido por el usuario, podría hacerse lo siguiente:

```
System.out.print("Introduzca el número que inicia la cuenta regresiva: ");
count = stdIn.nextInt();
for (; count>0; count--)
{
  System.out.print(count + " ");
}
System.out.println("Liftoff!");
```

no hay componente de iniciación

En realidad, es legal omitir cualquiera de los componentes del encabezado del ciclo `for`, en la medida en que los dos punto y coma sigan apareciendo entre paréntesis. Por ejemplo, es posible inclusive escribir un encabezado del ciclo `for` como éste:

```
for (;;)
```

Cuando se omite un componente (el segundo) de condición del encabezado de un ciclo `for`, la condición se considera verdadera para todas las iteraciones del ciclo. Con una condición permanentemente verdadera, a menudo un ciclo así es infinito y un error lógico. Pero éste no siempre es el caso. Puede terminarse usando una declaración `break` como:

```
for (;;)
{
  ...
  if (<condición>)
  {
    break;
  }
}
```

El lector debe comprender el ejemplo anterior en caso de que vea un código semejante en el programa de otra persona. Pero es más bien críptico y, como tal, debe evitar escribir su propio código de esa forma.

Iniciación múltiple y componentes de actualización

Para casi todos los ciclos `for`, una variable de índice es todo lo que se necesita. Pero a partir de ahora, se requieren dos o más variables de índice. Para cumplir esta necesidad, es posible incluir una lista de ini-

ciaciones separadas por comas en el encabezado de un ciclo `for`. La salvedad para las iniciaciones es que sus variables de índice deben ser del mismo tipo. Al trabajar en concierto con las iniciaciones separadas por comas también es posible incluir una lista de actualizaciones separadas por comas en el encabezado de un ciclo `for`. Los siguientes fragmentos de código y salida asociada muestran de qué se está hablando. En el encabezado del ciclo `for` observe las dos variables de índice, `up` y `down`, así como sus iniciaciones y componentes de actualización separadas por comas.

```
System.out.printf("%3s%5s\n", "Up", "Down");
for (int up=1,down=5; up<=5; up++,down--)
{
  System.out.printf("%3d%5d\n", up, down);
}
```

Salida:

```
Up  Down
 1     5
 2     4
 3     3
 4     2
 5     1
```

Como es el caso con muchas otras técnicas presentadas en este capítulo, el uso de iniciación múltiple y componentes de actualización en un ciclo `for` es una especie de arte. Conduce a un código más reducido, lo cual puede ser bueno o malo. Si el código reducido es legible, úselas. Si el código se vuelve más críptico, no lo haga.

11.13 Apartado GUI: Unicode (opcional)

Antes se aprendió que los caracteres obtienen sus valores numéricos subyacentes del conjunto de caracteres ASCII. Esto es cierto para los 128 caracteres que se muestran en la figura 11.4, pero tome en cuenta que en el mundo hay más de 128 caracteres. El conjunto de caracteres ASCII contiene los caracteres del alfabeto latino —de la A a la Z—, pero no contiene los caracteres de otros alfabetos. Por ejemplo, no contiene los caracteres de los alfabetos griego, cirílico y hebreo. Los diseñadores del lenguaje Java querían que Java fuese de propósito general, de modo que deseaban poder producir salida de texto para muchos idiomas distintos usando muchos alfabetos diferentes. Para manejar los caracteres adicionales, los diseñadores de Java tuvieron que usar un conjunto de caracteres más grande que el conjunto de caracteres ASCII. Así, adoptaron el estándar *Unicode*. Este estándar define valores numéricos subyacentes para un conjunto enorme de 65 536 caracteres.

¿Por qué en el estándar Unicode hay 65 536 caracteres? Porque quienes diseñaron el estándar Unicode (el Consorcio Unicode) decidieron que 16 bits eran suficientes para representar todos los caracteres necesarios en un programa de computadora.[4] Y 16 bits pueden representar 65 536 caracteres. He aquí las representaciones binarias de los cuatro primeros caracteres y del último carácter:

```
0000 0000 0000 0000
0000 0000 0000 0001
0000 0000 0000 0010
0000 0000 0000 0011
. . .
1111 1111 1111 1111
```

[4] El centro de atención es el estándar Unicode original, que es un subconjunto del estándar Unicode actual. El estándar Unicode original es suficiente para casi toda la programación en Java. El estándar Unicode original usa 16 bits para todos los caracteres. El estándar Unicode actual utiliza bits adicionales para caracteres adicionales que no caben en el conjunto de 65 536 caracteres del estándar Unicode original. Para más detalles, consulte la página http://www.unicode.org/

Observe que cada renglón es una permutación diferente de ceros y unos. Si se escribieran todas estas permutaciones, se verían 63 536 renglones. Entonces, con 16 bits, es posible representar 65 536 caracteres. La fórmula para determinar el número de permutaciones (y en consecuencia el número de renglones y el número de caracteres) es 2 elevado a la potencia del número de bits. En otras palabras, $2^{16} = 65\,536$.

El mismo razonamiento puede aplicarse para determinar por qué hay 128 caracteres en el conjunto de caracteres ASCII. En 1963 (cuando los dinosaurios vagaban por la Tierra), quienes diseñaron el conjunto de caracteres ASCII decidieron que siete bits bastarían para representar todos los caracteres necesarios en un programa de computadora. $2^7 = 128$, de modo que siete bits pueden representar 128 valores únicos.

Puesto que la tabla ASCII fue y es un estándar tan conocido con muchos lenguajes de programación, los diseñadores de Unicode decidieron usar el conjunto de caracteres ASCII como un subconjunto del conjunto de caracteres Unicode. Insertaron los caracteres del conjunto de caracteres ASCII en las 128 primeras ranuras del conjunto de caracteres Unicode. Esto significa que los programadores pueden encontrar los valores numéricos de estos caracteres consultando una simple tabla ASCII; no tienen que leer el enorme conjunto de caracteres Unicode.

Números hexadecimales

Los números normales se expresan como potencias de 10, pero puesto que las computadoras son binarias y 16 es una simple potencia de 2 ($16 = 2^4$), es una práctica común expresar cantidades de computadora en base 16 (usando potencias de 16), en lugar de hacerlo en base 10 (usando potencias de 10). Los números base 10 se denominan números decimales. Los números base 16 se denominan números *hexadecimales*. Los lugares en los números decimales se denominan dígitos. Los lugares en los números hexadecimales se denominan *hexits*, aunque más a menudo se denominan simplemente *dígitos hexadecimales*. Los números base 10 usan 10 símbolos: 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. Los números base 16 usan 16 símbolos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e y f (las mayúsculas de la A a la F se consideran equivalentes a las minúsculas de la a a la f). Así, los números hexadecimales a menudo incluyen uno o más de los seis caracteres alfabéticos, así como uno o más de los caracteres numéricos normales.

En Java, cualquier entero puede escribirse en forma decimal o hexadecimal. Si se desea que un número se interprete como hexadecimal, es necesario prefijarlo con el carácter `0x`. Entonces, si se ve algo como `0x263A`, por ejemplo, puede reconocerse como un número hexadecimal. Para la mayor parte de las personas, los números hexadecimales no son muy intuitivos. Sin embargo, resulta bastante fácil efectuar conversiones. Simplemente se utiliza el método `toString` de dos parámetros de `Integer`:

```
Integer.toString(<starting-number>, <desired-base>)
```

Por ejemplo, si se desea conocer el equivalente decimal de `0x263A`, se escribe lo siguiente:

```
System.out.println(Integer.toString(0x263A, 10));
```

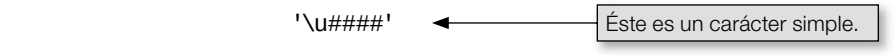
Esto genera un resultado de 9786. A la inversa, si se desea conocer el equivalente hexadecimal de 9786, escriba esto:

```
System.out.println(Integer.toString(9786, 16));
```

Esto genera un resultado de `263a`. Observe que el resultado de este método no incluye el prefijo `0x`, y que usa minúsculas para los dígitos hexadecimales alfabéticos.

Secuencia de escape Unicode

Siempre que se escribe un entero, es posible hacerlo en forma decimal o hexadecimal. De manera semejante, es posible especificar un carácter al escribir su valor numérico en forma decimal o hexadecimal y luego convirtiéndolo con el operador tipo cast (`char`). Java también proporciona otra forma para especificar un carácter. Puede usarse la *secuencia de escape Unicode*. La secuencia de escape Unicode es `\u` seguido inmediatamente por los dígitos hexadecimales de un número hexadecimal. He aquí de qué se está hablando:



Cada # significa un dígito hexadecimal. Aquí se decidió mostrar esto con comillas simples, no normales, para recalcar que la secuencia de escape de seis elementos es sólo un carácter simple, no una cadena. Sin

embargo, es justo como cualquier otra secuencia de escape, de modo que es posible insertar el `\u####` en cualquier sitio de una cadena. La `u` debe ser minúscula, y debe haber exactamente cuatro dígitos hexadecimales.[5]

Uso del Unicode en programas de Java

Si se desea imprimir caracteres usando secuencias de escape Unicode, puede usarse `System.out.println` en un entorno basado en texto para los 128 primeros caracteres, pero para los otros caracteres, `System.out.println` en un entorno basado en texto no funciona de manera consistente. Esto se debe a que los entornos basados en texto reconocen justo la porción ASCII de la tabla Unicode; es decir, los 128 primeros caracteres. Para imprimir todos los caracteres en la tabla Unicode, es necesario usar los comandos de interfaz gráfica de usuarios (GUI) en un entorno GUI.

El programa en la figura 11.11 proporciona una ventana GUI y la usa para ilustrar una pequeña muestra de los muchos caracteres disponibles. El arreglo `codes` contiene valores de código `int` para las secuencias de escape Unicode para los primeros caracteres en bloques de caracteres que se decidió mostrar. Estas secuencias de escape Unicode promueven en forma automática de tipo `char` a tipo `int` en la asignación de iniciación. El arreglo denominado `descriptions` contiene una simple descripción `String` para cada bloque de caracteres.

Para la ventana, se utiliza una instancia de la clase `JFrame` API de Java, que está en el paquete `javax.swing`. El tamaño de la ventana se establece en 600 pixeles de ancho y 285 pixeles de alto. En la ventana se incluye un simple objeto `JTextArea` denominado `area`, y se activa su capacidad de envolver en línea. El método `append` de `JTextArea` se usa para agregar cada nueva cadena o carácter a cualquier cosa que ya se encuentre ahí.

Antes de caer en un ciclo se presenta algo de información sobre las fuentes (*fonts*). El ciclo externo `for` exhibe el valor del primer número del código en uno de los bloques escogidos de caracteres y luego la descripción de ese bloque. El ciclo interno `for` muestra los 73 primeros caracteres en ese bloque. En el argumento del método `append`, observe cómo se agrega el conteo del ciclo, `j`, al valor inicial Unicode a fin de obtener cada valor Unicode individual como un `int`. Luego ese `int` se cambia a un `char`. A continuación, las `""` concatenadas convierten ese `char` en un `String`, que coincide con el tipo de parámetro del método `append`.

En la figura 11.12 se muestra la salida GUI que genera este programa. Los caracteres en el arreglo `codes` en la figura 11.11 son las secuencias de escape Unicode para el primer carácter en cada bloque de caracteres mostrado en la figura 11.12. Los cuadrados huecos indican números de código que no tienen símbolos asociados o simplemente que no están presentes en la biblioteca del software de la computadora. Observe que los bloques cirílico y griego incluyen caracteres tanto superior como inferior, y que incluyen algunos caracteres adicionales más allá de los valores finales normales de Ω (ω) y Я (я), respectivamente. Estos (y otros) caracteres adicionales son necesarios para algunos de los lenguajes individuales en las familias de lenguajes que usan estos alfabetos. Por supuesto, los caracteres mostrados en la figura 11.12 son apenas una pequeña muestra de todos los caracteres en Unicode.

Observe que en general los diferentes caracteres mostrados en la figura 11.12 tienen anchos distintos. Para obtener caracteres con ancho constante, es necesario cambiar el tipo de fuente a algo como Courier New. Esto puede hacerse, así como cambiar el estilo a negritas y el tamaño a 10 puntos, insertando una declaración como ésta:

```
area.setFont(new Font("Courier New", Font.BOLD, 10));
```

Suponga que se desea conocer el valor Unicode para ≈. Éste es el último operador matemático mostrado en la figura 11.12. Como se indica por el tercer valor `codes` en el programa UnicodeDisplay, el primer operador matemático tiene un valor de código hexadecimal de 0x2200.

---

[5] Los caracteres complementarios Unicode tienen valores numéricos que requieren más de cuatro dígitos hexadecimales. Para especificar uno de estos caracteres complementarios, use una representación `int` decimal o hexadecimal del carácter, o prefije la representación `\u` de los cuatro dígitos hexadecimales menos significativos con una representación u idónea en el intervalo que va desde `\uD800` hasta `\uDFFF`. El prefijo, denominado *subrogado*, no tiene asociación de carácter independiente. (Consulte la documentación en la clase `Character` de Java y en http://www.unicode.org/Public/UNIDATA/BLocks.txt.) También hay otro esquema subrogado que representa caracteres con un valor base de ocho bits y múltiples subrogados de ocho bits. Este último esquema se utiliza en comunicaciones.

```
/*********************************************************
* UnicodeDisplay.java
* Dean & Dean
*
* This prints unicode characters.
*********************************************************/

import javax.swing.*;
import java.awt.Font;

public class UnicodeDisplay
{
  public static void main(String[] args)
  {
    int[] codes = {'\u0391',
                   '\u0410',
                   '\u2200',
                   '\u2500',
                   '\u2700'};
    String[] descriptions = {"Greek",
                             "Cyrillic (Russian)",
                             "mathematical operators",
                             "box drawing",
                             "dingbats"};
    JFrame window = new JFrame("Some Unicode Characters");
    JTextArea area = new JTextArea();
    Font font = area.getFont();

    window.setSize(600,285);     // pixel width, height
    window.setDefaultCloseOperation(JFrame.EXIT _ ON _ CLOSE);
    window.add(area);
    area.setLineWrap(true);
    area.append("Font type, style, and size: " +
      font.getFontName() + ", " + font.getSize() + "\n");
    for (int i=0; i<codes.length; i++)
    {
      area.append("0x" + Integer.toString(codes[i], 16) +
        " " + descriptions[i] + ":\n");
      for (int j=0; j<=72; j++)
      {
        area.append((char) (codes[i] + j) + "  ");
      }
      area.append("\n");
    }
    window.setVisible(true);
  } // end main
} // end UnicodeDisplay
```

Figura 11.11 Programa que usa GUI para exhibir una muestra de caracteres Unicode.

El valor máximo del ciclo interno `for` en la figura 11.11 es 72. El valor hexadecimal de 72 es 4 × 16 + 8 = 0x0048. Así, el valor hexadecimal Unicode del último operador matemático mostrado en la figura 11.12 es 0x2200 + 0x0048 = 0x2248. Algunas veces es posible usar un procesador de texto como ayuda para encontrar el valor Unicode del símbolo especial que busca. Por ejemplo, en Microsoft Word, seleccione Insert/Symbol/Mathematical Operators y luego seleccione ≈. A continuación, lea el valor hexadecimal Unicode del símbolo seleccionado en el campo "Carácter code" cerca de la parte inferior de la

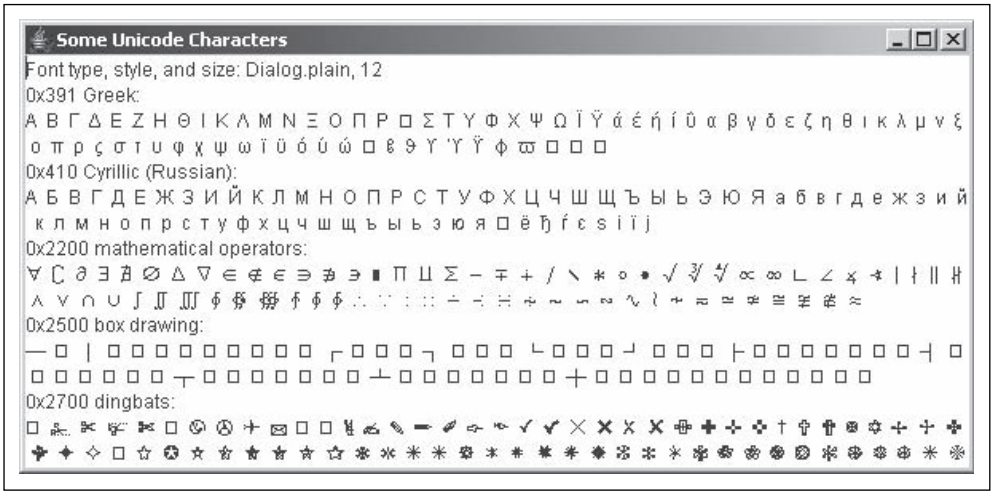


Figura 11.12 Salida producida por el programa en la figura 11.11.

ventana Symbol. Encontrará que esto también indica que el valor hexadecimal Unicode para el carácter ≈ es 0x2248.

Búsquelo. Todo lo que Unicode tiene que ofrecer puede encontrarse en el sitio http://www.unicode.org Si el lector consulta ese sitio, busque una liga Code Charts y haga clic ahí. Así debe acceder a una página que le permite explorar en las diversas subtablas dentro de la enorme tabla Unicode. Intente encontrar la liga Latin Basic. Eso lo lleva a la subtabla Basic Latin, que es equivalente a la tabla ASCII. La subtabla se denomina Latin porque contiene el alfabeto latino: a, b, c, etc. Visite algunas de las otras subtablas para tener una idea de lo que está a disposición. En cada subtabla verá un conjunto de caracteres, y para cada carácter, verá el código Unicode equivalente.

También hay otros diversos estándares para asignar números a los caracteres. Las aplicaciones de las computadoras algunas veces incluyen tablas de traducción para efectuar conversiones entre sus propios esquemas de codificación de caracteres y Unicode. De todos modos, esté alerta. Las traducciones no siempre funcionan como le agradaría al lector, y caracteres especiales pueden cambiar de maneras sorprendentes cuando se transfiere texto con caracteres especiales de una aplicación a otra.

Resumen

- El desbordamiento numérico provoca errores graves. Siempre que haya alguna duda acerca de la habilidad particular de un tipo para contener un valor que podría serle asignado, debe cambiarse a un tipo más grande.
- Los números de punto flotante tienen un intervalo más grande que los enteros, pero para una cantidad de memoria dada, proporcionan menos precisión.
- El conjunto de caracteres ASCII proporciona valores numéricos para los símbolos en un tablero estándar.
- Puesto que los caracteres se representan como números, `ch1 + ch2` se evalúa como la suma de los valores ASCII para las variables `char`, `ch1` y `ch2`.
- La conversión de tipo permite colocar un valor numérico en una variable numérica de un tipo diferente, pero debe tenerse cuidado en no producir desbordamiento o truncamiento indeseado al hacerla.
- A usar un prefijo, un operador incremento (++) o un operador decremento (--) cambia el valor de la variable antes que esa variable participe en otras operaciones de la expresión. Cuando se usa como sufijo, un operador incremento o un operador decremento cambia el valor de la variable después que ésta participa en otras operaciones de la expresión.
- Si una declaración contiene operadores de asignación múltiple, primero se evalúa la asignación que está más a la derecha.

- Algunas veces resulta útil insertar una asignación en una condición, aunque es necesario evitar el uso excesivo de operaciones insertadas de incremento, decremento y asignación.
- Una expresión de operador condicional proporciona una evaluación condicional compacta. Si lo que está antes de ? es `true`, se usa lo que está después de `?`. En caso contrario, debe usarse lo que está después de `:`.
- La evaluación de cortocircuito significa que la JVM deja de evaluar una expresión siempre que la salida de la expresión se vuelve cierta. Use esta característica para evitar operaciones ilegales.
- Use una declaración `break` con moderación para terminar prematuramente los ciclos.
- En su forma extendida, Unicode proporciona códigos numéricos hasta para un millón de caracteres diferentes, que es posible especificar como enteros decimales o hexadecimales o con una secuencia de escape Unicode. A fin de ver los caracteres Unicode para códigos superiores a 127 es necesario desplegarlos en una ventana GUI.

Preguntas de revisión

§11.2 Tipos entero y punto flotante

1. Para cada tipo de datos entero, ¿cuántos bits de almacenamiento se usan?
2. ¿Cómo se escribe la constante decimal 1.602×10^{-19} como `double`?
3. ¿Cuál es la precisión aproximada (número de dígitos decimales exactos) de cada uno de los tipos de punto flotante?

§11.3 Tipo char y el conjunto de caracteres ASCII

4. ¿Cuántos caracteres distintos describe el conjunto de caracteres ASCII?
5. ¿Qué número puede sumarse a una variable `char` escrita con mayúsculas a fin de convertirla en minúsculas?

§11.4 Conversiones de tipo

6. Suponga la declaración:

```
public final double C = 3.0E10;  // velocidad de la luz en cm/s
```

Escriba una declaración de impresión en Java que use un operador tipo cast para mostrar el valor de `C` en este formato:

```
30000000000
```

7. La siguiente declaración, ¿es correcta o genera un error de tiempo de compilación? (correcta/error)

```
float price = 66;
```

8. La siguiente declaración, ¿es correcta o genera un error de tiempo de compilación? (correcta/error)

```
boolean done = (boolean) 0;
```

9. La siguiente declaración, ¿es correcta o genera un error de tiempo de compilación? (correcta/error)

```
float price = 98.1;
```

§11.5 Modos prefijo/sufijo para operadores de incremento/decremento

10. ¿Cuál es el valor de `z` después que se ejecutan las siguientes declaraciones?

```
int z, x = 3;
z = --x;
z += x--;
```

§11.6 Asignaciones insertadas

11. Escriba una declaración en Java que iguale `w`, `x` y `y` al valor actual de `z`.

§11.7 Expresiones del operador condicional

12. Suponga que `x` es igual a 0.43. Dado el siguiente encabezado de la declaración `switch`, ¿en qué se evalúa la expresión de control del encabezado de `switch`?

```
switch (x>0.67 ? 'H' : (x>0.33 ? 'M' : 'L'))
```

§11.8 Revisión de evaluación de expresiones

13. Suponga lo siguiente:

```
int a = 2;
int b = 6;
float x = 8.0f;
```

Evalúe cada una de las siguientes expresiones, usando estas directrices:

- Como se mostró en la sección 11.8, escriba cada paso de la evaluación en una línea por separado y use el símbolo ⇒ entre los pasos.
- Evalúe cada expresión independientemente de las otras expresiones; en otras palabras, use los valores supuestos anteriores para la evaluación de cada expresión.
- Los problemas de evaluación de expresiones pueden ser engañosos. Alentamos al lector a que compruebe su trabajo mediante la ejecución de un código de prueba en una computadora.
- En caso de haber un error de compilación, especifique "error de compilación".

a) `a + 25 / (x + 2)`
b) `7 + a * --b / 2`
c) `a * --b / 6`
d) `a + b++`
e) `a - (b = 4) % 7`
f) `b = x = 23`

§11.9 Evaluación de cortocircuito

14. Suponga que `expr1` y `expr2` son expresiones que se evalúan en valores `boolean`. Suponga que `expr1` se evalúa en `true`. Cuando la computadora evalúa cada una de las siguientes expresiones, ¿evalúa `expr2`? En caso afirmativo, simplemente diga "sí". En caso contrario, explique por qué no y use el término "evaluación de cortocircuito" en su explicación.

a) `expr1 || expr2`
b) `expr1 && expr2`

15. Suponga lo siguiente:

```
int a = 2;
boolean flag = true;
```

Evalúe la siguiente expresión:

```
a < 3 || flag && !flag
```

§11.10 Declaración vacía

16. Suponga que el siguiente fragmento de código está dentro de un programa que compila exitosamente. ¿Qué imprime el fragmento de código? *Sugerencia:* Ésta es una pregunta engañosa. Estudie cuidadosamente el código.

```
int x = 1;
while (x < 4);
{
  System.out.println(x);
  x++;
}
```

§11.11 Declaración `break` dentro de un ciclo

17. Por lo general, es necesario evitar el uso de `break`, excepto en declaraciones `switch` porque el uso de declaraciones `switch` obliga a los lectores a buscar condiciones de terminación dentro de los cuerpos de los ciclos. (F/C)

§11.12 Detalles para el encabezado del ciclo `for`

18. Suponga que el siguiente fragmento de código está dentro de un programa que compila exitosamente. ¿Qué imprime el fragmento de código?

```
for (int i=0,j=0; ; i++,j++)
{
  System.out.print(i + j + " ");
}
```

§11.13 Apartado GUI: Unicode (opcional)

19. ¿Cuál es el símbolo hexadecimal para el número decimal 13?

20. Los valores Unicode para los caracteres son los mismos que los valores ASCII en el intervalo de `0x00` a `0xFF`. (F/C)

Ejercicios

1. [Después de §11.2] Si un entero se desborda, ¿qué tipo de error se produce: de tiempo de compilación, de ejecución o lógico?

2. [Después de §11.4] ¿Cuántos bits se usan para almacenar un valor `char`?

3. [Después de §11.4] ¿Qué imprime esto?: `System.out.println('A' + 2);`

4. [Después de §11.6] Suponga que `a` y `b` son variables `boolean`. ¿Cuáles son sus valores después que se ejecuta esta declaración?

```
a =!((b=4<=5) && (a=4>=5));
```

Sugerencia: Primero vuelva a escribir la declaración para hacerla más legible.

5. [Después de §11.7] Suponga lo siguiente:

```
int a = 2;
float x = 8.0f;
boolean flag = true;
```

Evalúe:

```
(flag) ? (a = (int) (x + .6)) : a
```

6. [Después de §11.8] Suponga lo siguiente:

```
int a = 10;
int b = 2;
double x = 6.0;
```

Evalúe cada una de las siguientes expresiones. Siga estas directrices:

- Como se mostró en la sección 11.8, escriba cada paso de la evaluación en una línea por separado y use el símbolo ⇒ entre los pasos.
- Evalúe cada expresión independientemente de las otras expresiones; en otras palabras, use los valores supuestos anteriores para la evaluación de cada expresión.
- Los problemas de evaluación de expresiones pueden ser engañosos. Alentamos al lector a que compruebe su trabajo mediante la ejecución de un código de prueba en una computadora.
- En caso de haber un error de compilación, especifique "error de compilación".

a) `a - 7 / (x - 4)`
b) `8 + a * ++b / 20`
c) `a + b--`
d) `a + (b = 5) % 9`
e) `a = x = -12`

7. [Después de §11.8] Suponga lo siguiente:

```
String s = "hi";
int num = 3;
char ch = 'm';
```

Evalúe cada una de las siguientes expresiones. Siga estas directrices:

- Como se mostró en la sección 11.8, escriba cada paso de la evaluación en una línea por separado y use el símbolo ⇒ entre los pasos.
- Evalúe cada expresión independientemente de las otras expresiones; en otras palabras, use los valores supuestos anteriores para la evaluación de cada expresión.
- Los problemas de evaluación de expresiones pueden ser engañosos. Alentamos al lector a que compruebe su trabajo mediante la ejecución de un código de prueba en una computadora.
- En caso de haber un error de compilación, especifique "error de compilación".

a) `s + (num + 4)`
b) `s + num + 4`
c) `s + '!' + "\""`
d) `num + ch`
e) `'8' + 9`

8. [Después de §11.9] Considere el siguiente fragmento de código. Los números de línea están a la izquierda.

```
1    int a = 2;
2    boolean b = false;
3    boolean c;
4    c = b && ++a == 2;
5    b = a++ == 2;
6    b = !b;
7    System.out.println(a + " " + b + " " + c);
```

Rastree el código usando lo siguiente:

línea#	a	b	c	salida

9. [Después de §11.9] Suponga:

```
boolean a = false;
boolean b;
double c = 2.5;
```

Determine la salida del siguiente fragmento de código:

```
b = a && (++c == 3.5);
a = true || (++c == 3.5);
System.out.println(a + " " + b + " " + c);
```

10. [Después de §11.10] En la sucesión de Fibonacci, cada elemento sucesivo es la suma de los dos elementos previos. Si se empieza con 0 y 1, el siguiente elemento es 0 + 1 = 1. El siguiente elemento es 1 + 1 = 2; el siguiente es 1 + 2 = 3; el siguiente es 2 + 3 = 5, y así sucesivamente. Dada esta declaración:

```
int p, q;
```

Escriba un ciclo `for` que imprima esta parte de la sucesión de Fibonacci:

```
1 2 3 5 8
```

Su solución debe constar sólo de un encabezado de ciclo `for` y luego una declaración vacía: nada más. Por cierto, se recomienda evitar un código como éste para sus programas reales. Este ejercicio es sólo por diversión (diversión para un hacker, en todo caso ☺).

11. [Después de §11.10] Un error común consiste en agregar accidentalmente un punto y coma al final del encabezado de un ciclo. Ejecute el siguiente método `main` en una computadora. ¿Cuál es la salida?

```
public static void main(String[] args)
{
  int i;
  int factorial = 1;

  for (i=2; i<=4; i++);
  {
    factorial *= i;
  }
  System.out.println("i = " + i + ", factorial = " + factorial);
} // end main
```

12. [Después de §11.12] Observe el siguiente programa. Escriba un ciclo `for` que sea funcionalmente equivalente al ciclo `do` dado.

```
import java.util.Scanner;

public class Test
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String entry;
```

```
      do
      {
        System.out.println("Introduzca 'q' para salir: ");
        entry = stdIn.nextLine();
      } while (!entry.equals("q"));
    } // end main
  } // end class Test
```

13. [Después de §11.13] ¿Cuál es el valor hexadecimal del código Unicode para el símbolo ∞ (infinito)? Muestre o explique cómo obtuvo su respuesta.

Soluciones a las preguntas de revisión

1. `byte` = 8 bits, `short` = 16 bits, `int` = 32 bits, `long` = 64 bits.
2. `1.602E-19` o `1.602e-19`.
3. Precisión `float` ≈ 6 dígitos; precisión `double` ≈ 15 dígitos.
4. El conjunto básico de caracteres ASCII describe 128 caracteres distintos.
5. Para convertir mayúsculas en minúsculas, sumar 32. Para hacer lo contrario, restar 32.
6. `System.out.println((long) C);  // ¡no es suficientemente grande!`
7. Esta declaración es correcta:

```
float price = 66;
```

8. Esta declaración genera un error de tiempo de compilación porque es ilegal convertir entre valores numéricos y valores `boolean`:

```
boolean done = (boolean) 0;
```

9. Esta declaración genera un error de tiempo de compilación porque las constantes de punto flotante son `double` por defecto:

```
float price = 98.1;
```

10. El valor `z` es 14. El primer decremento usa el modo prefijo, así que `x` primero se disminuye a 2 y luego 2 se asigna a `z`. El segundo decremento usa modo sufijo, por lo que `x` se disminuye después que su valor se suma a `z`.
11. `w = x = y = z;` o cualquier otra secuencia que tenga `z` a la derecha.
12. La expresión de control del encabezado de `switch` se evalúa en `'M'`.
13. Práctica de evaluación de expresiones:

a) `a + 25 / (x + 2)` ⇒
`( 2 + 25 / (8.0 + 2)` ⇒
`2 + 25 / 10.0` ⇒
`2 + 2.5` ⇒
`4.5`

b) `7 + a * --b / 2` ⇒
`7 + 2 * --6 / 2` ⇒
`7 + 2 * 5 / 2` ⇒
`7 + 10 / 2` ⇒
`7 + 5` ⇒
`12`

c) `a * --b / 6` ⇒
`2 * --6 / 6` ⇒
`2 * 5 / 6` ⇒
`10 / 6` ⇒
`1`

d) `a + b++` ⇒
`2 + 6` (b se actualiza a 7 después que se ha introducido su valor) ⇒
`8`

e) `a - (b = 4) % 7` ⇒
`2 - 4 % 7` ⇒
`2 - 4` ⇒
`-2`

f) `b = x = 23` ⇒
`b = 23.0` ⇒
error de compilación (porque el `float` 23.0 no puede asignarse al `int b` sin un operador tipo cast)

14. ¿Evalúa `expr2`?
a) No. Puesto que el lado izquierdo del operador `||` es `true`, la evaluación de cortocircuito provoca que se ignore el lado derecho del operador `||` (`expr2`) (puesto que el resultado de toda la expresión se evalúa en `true` sin importar el valor de `expr2`).
b) Sí.

15. Suponiendo:

```
int a = 2;
boolean flag = true;
```

```
a < 3 || flag && !flag ⇒
2 < 3 || true && !true ⇒
2 < 3 || true && false ⇒
true || true && false ⇒
true
```
(la evaluación de cortocircuito indica que "true o cualquier cosa" se evalúa en `true`)

16. No imprime nada porque, debido a la declaración vacía, el encabezado del ciclo `while` se ejecuta repetidamente en un ciclo infinito.

17. Cierto. Normalmente, es necesario evitar el uso de `break`, salvo en declaraciones `switch`.

18. El fragmento de código genera un ciclo infinito porque la segunda componente faltante del encabezado del ciclo `for` es verdadera por defecto. La salida es:

```
0 2 4 6 ...
```

19. El símbolo hexadecimal para el número decimal 13 es `d` o `D`.

20. Falso. Son los mismos sólo en el intervalo de `0x00` a `0x7F`.

CAPÍTULO 12

Agregación, composición y herencia

Objetivos

- Comprender cómo en agregaciones y composiciones las cosas están organizadas de forma natural.
- Implementar relaciones de agregación y composición dentro de un programa.
- Comprender cómo puede usarse la herencia para refinar una clase existente.
- Implementar una jerarquía de la herencia dentro de un programa.
- Aprender a escribir constructores para clases derivadas.
- Aprender a sobreponer un método heredado.
- Aprender a evitar la sobreposición.
- Aprender a usar una clase para representar una asociación.

Relación de temas

12.1 Introducción

Antes de este capítulo, los programas que ha creado han sido relativamente simples en términos de la orientación de sus objetos, de modo que ha sido capaz de describir todos los objetos en un programa con una sola clase. Pero para programas más complicados, es necesario considerar la implementación de múltiples clases, una para cada tipo diferente de objeto dentro de un programa. En este capítulo se hace precisamente eso, y el lector se centrará en las diversas formas que hay para organizar clases en un programa con múltiples clases. Primero, se aprenderá a organizar clases que forman parte de una clase contenedora más grande. Cuando las clases están relacionadas así, donde una clase es el todo y las otras clases son partes del todo, las clases forman una *agregación*. Se aprenderá a organizar clases donde una clase, la *clase base*, define características comunes para un grupo de objetos y las otras clases definen características especializadas para cada uno de los diferentes tipos de objetos en el grupo. Cuando las clases están relacionadas de esta manera, las clases forman una jerarquía de la *herencia*. Se denomina así porque las clases especializadas heredan características de la clase base.

Al describir la herencia se presentan varias técnicas para trabajar con clases de una jerarquía de la herencia. Específicamente, se presenta la *sobreposición de métodos*, que permite redefinir un método en una clase especializada que ya estaba definido en la clase base. También se presenta el modificador `final`, que permite evitar que una clase especializada sobreponga un método definido en la clase base.

Como continuación de la presentación inicial de los conceptos de agregación y herencia, se describe cómo las dos estrategias de diseño pueden trabajar de manera conjunta. Algunas veces resulta difícil decidir cuál es la mejor estrategia a utilizar. Para adquirir práctica en la toma de tales decisiones, se guiará parte del camino al lector a través de una actividad de diseño de un programa y desarrollo de la estructura de lo que puede ser un programa sofisticado de juego de cartas. En una sección final opcional se mostrará cómo mejorar la organización al crear una clase de asociación que define un conjunto de características pertenecientes a una relación particular entre clases.

Al mostrar cómo organizar múltiples clases, este capítulo aporta al lector importantes herramientas necesarias para abordar problemas del mundo real. Después de todo, la mayor parte de los proyectos de programación del mundo real son grandes e implican múltiples tipos de objetos. Cuando los objetos se organizan correctamente, los programas son más fáciles de comprender y mantener, ¡lo que es bueno para todos!

12.2 Composición y agregación

Hay dos formas primarias de agregación. Como ya se describió, la agregación normal ocurre cuando una clase es el todo y otras clases forman parte del todo. La otra forma de agregación también define una clase como el todo y a otras clases como partes del todo. Pero tiene una restricción adicional que indica que la clase todo es la propietaria exclusiva de las clases parte. "Propiedad exclusiva" significa que las clases parte no pueden pertenecer a ninguna otra clase mientras sean propiedad de la clase todo. Esta forma de propiedad exclusiva de agregación se denomina *composición*. Con la composición, la clase todo se denomina *compuesto*, las clases parte se denominan *componentes* y el compuesto contiene a los componentes. Se considera que la composición es una forma fuerte de la agregación, puesto que las conexiones compuesto-componentes son fuertes (debido a que cada componente sólo tiene un propietario: el compuesto).

Composición y agregación en el mundo real

El concepto de composición no fue creado para la programación de computadoras; suele usarse a menudo para objetos complejos en el mundo real. Toda criatura viviente y la mayor parte de los productos manufacturados están constituidos por partes. A menudo, cada parte es un subsistema que está integrado por su propio conjunto de subpartes. Junto, todo el sistema forma una jerarquía de composición.

En la figura 12.1 se muestra la jerarquía de composición del cuerpo humano. En la cima de esta jerarquía de composición particular está un cuerpo completo. Un cuerpo humano está compuesto por varios órganos: cerebro, corazón, estómago, huesos, músculos, etc. A su vez, cada uno de estos órganos está compuesto por muchas células, y cada una de estas células está compuesta por muchos orgánulos, como el núcleo (el "cerebro" de una célula) y las mitocondrias (los "músculos" de una célula). Cada orgánulo está compuesto por muchas moléculas. Y finalmente, cada molécula orgánica está compuesta por muchos átomos.

En una jerarquía de composición (así como en una jerarquía de agregación), la relación entre una clase contenedora y una de sus clases parte se denomina relación *tiene-un*. Por ejemplo, cada cuerpo humano tiene un cerebro y tiene un corazón. Recuerde que con una relación de composición, una parte componente está limitada a justo un propietario a la vez. Por ejemplo, un corazón sólo puede estar en un cuerpo a la vez. Aunque la propiedad es exclusiva, es posible que cambie el propietario. Con un trasplante de corazón, un corazón puede pasar a un nuevo propietario, aunque sigue teniendo un propietario a la vez.

Observe los diamantes en la figura 12.1. En el Lenguaje Unificado de Modelado (UML: *Universal Modeling Language*), los diamantes llenos denotan una relación de composición. Indican que un todo tiene la propiedad exclusiva de una parte.

A continuación se considerará un ejemplo de agregación en el que las partes no son propiedad exclusiva del todo. Es posible implementar una escuela como una agregación al crear una clase todo para la

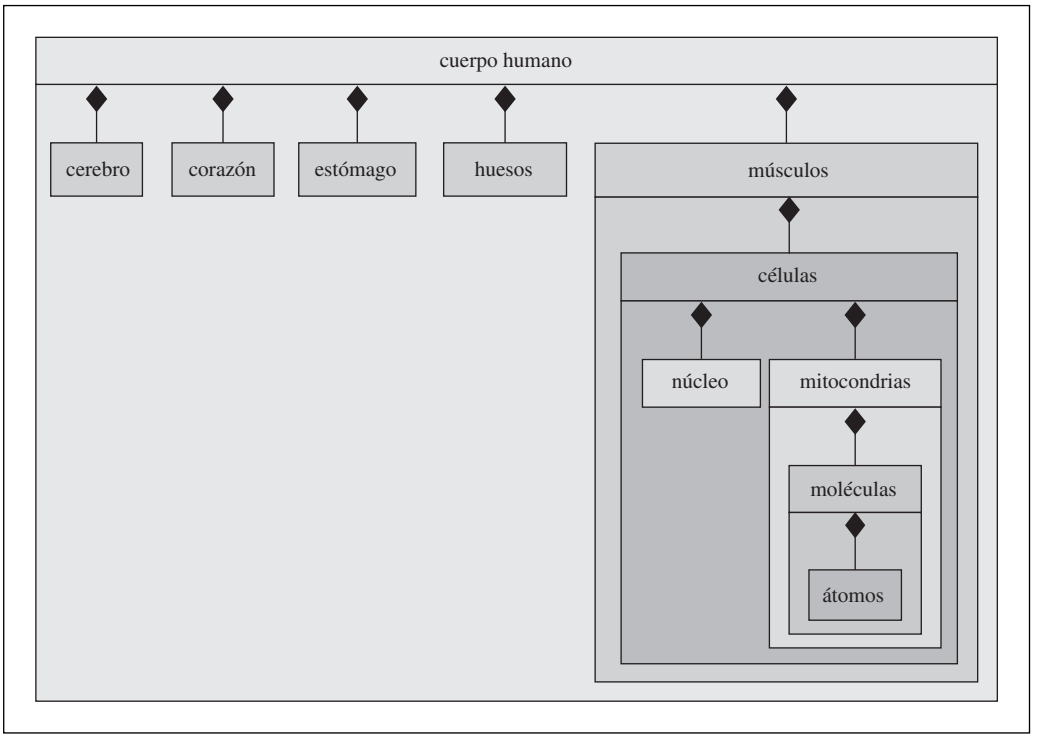


Figura 12.1 Representación parcial de una jerarquía de composición del cuerpo humano.

escuela y clases parte para los diferentes tipos de personas que trabajan y estudian en esa escuela. Las personas no son propiedad exclusiva de la escuela porque una persona puede formar parte de más de una agregación. Por ejemplo, una persona puede asistir a clases en dos escuelas diferentes y ser parte de dos agregaciones escolares. Inclusive, la misma persona podría quizá formar parte de una tercera agregación, de un tipo diferente, como una agregación de inquilinos.

Composición y agregación en un programa de Java

Se considerará un ejemplo que usa ambas relaciones de clase: composición (donde se requiere propiedad exclusiva) y agregación normal (donde no se requiere propiedad exclusiva). Suponga que se está intentando modelar una franquicia de automóviles con un programa de computadora. Puesto que la franquicia está integrada por varias partes no triviales, es un buen elemento para implementarse como agregación. El "todo" (la parte superior de la jerarquía de la agregación) es el concesionario. Por lo general, un negocio consta de dos "partes": las personas y la propiedad. Para no complicar las cosas, suponga que los únicos tipos de personas en la franquicia son el gerente y los vendedores, y suponga que el único tipo de propiedad son los automóviles. El control que el concesionario tiene sobre las personas es limitado. Las personas también pueden tener otros trabajos, y tal vez tienen obligaciones familiares. El concesionario no posee en exclusiva a sus empleados. En consecuencia, la relación entre el concesionario y sus empleados es simplemente de agregación. Pero el concesionario posee en exclusiva sus automóviles. Así, ésta es una relación de composición. Observe que el concesionario puede transferir la propiedad de sus automóviles a sus clientes. Esto está bien porque la composición permite la transferencia de la propiedad. Al usar una metodología de diseño ascendente (*bottom-up design*) para los tres tipos de objetos componentes deben definirse tres clases: `Car`, `Manager` y `SalesPerson`. Luego, debe definirse una clase `Dealership` para el objeto contenedor.

Antes de ver el código del programa Dealership, los conceptos más importantes se considerarán usando un diagrama de clase UML. El diagrama de clase UML en la figura 12.2 muestra las cuatro clases del programa Dealership, así como las relaciones entre ellas. Puesto que ahora el centro de atención lo constituyen simplemente las relaciones entre clases, en cada representación de una clase sólo se incluye el nombre de la clase y se omiten las variables y los métodos. Eso está bien: UML es muy flexible,

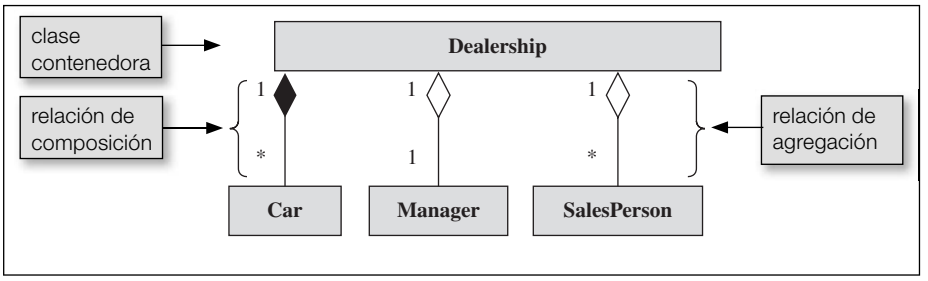


Figura 12.2 Diagrama de clase para el programa Dealership.

y tales omisiones son permitidas por los estándares de UML. UML indica las relaciones de clase con líneas de conexión que unen una clase con otra. Formalmente, cada línea de conexión se denomina *línea de asociación*.

En la figura 12.2 observe los diamantes sobre las líneas de asociación. Los diamantes llenos (como el que está en la línea `Dealership-Car`) indican relaciones de composición, y los diamantes huecos (como los que están en las líneas `Dealership-Manager` y `Dealership-SalesPerson`) indican relaciones de agregación. Los diamantes siempre van cerca de la clase contenedora, de modo que el diagrama de clase en la figura 12.2 indica que la clase contenedora es `Dealership`.

Observe los números y asteriscos escritos al lado de las líneas de asociación. Se trata de *valores de multiplicidad* que UML usa para especificar el número de objetos que participan en asociaciones. Los dos unos en la línea entre `Dealership` y `Manager` indican una asociación uno a uno. Esto significa que hay un gerente para cada concesionario. Si para la clase manager hubiera un valor de multiplicidad 2, esto indicaría dos gerentes para cada concesionario. La combinación de 1 y * en las otras dos líneas de asociación indican asociaciones uno a muchos, donde "muchos" implica un número indefinido. Esto significa que es posible tener muchos automóviles (o ninguno) y muchos vendedores (o ninguno) para un concesionario.

Ahora es tiempo de ir de la fase conceptual, recalcando el diagrama de clase UML del concesionario, a la fase de implementación, con énfasis en el código del programa `Dealership`. Observe la clase `Dealership` en la figura 12.3 y, en particular, observe las variables de instancia `manager`, `people` y `cars` declaradas dentro de la clase `Dealership`. Estas declaraciones de variables de instancia implementan el concepto de la clase `Dealership` que contiene a las otras tres clases. La regla general es que siempre que se tiene una clase que contiene a otra clase, es necesario declarar una variable de instancia dentro de la clase contenedora, de modo que la variable de instancia tenga una referencia a uno o más de los objetos de la clase contenida.

También en la clase `Dealership`, observe el uso de la `ArrayList` para las variables de instancia `people` y `cars`. Típicamente, si se tiene una clase en un diagrama de clase UML con un valor de multiplicidad *, debe usarse una `ArrayList` para implementar la referencia a la clase que está entre asteriscos. Las `ArrayList` son buenas para implementar valores de multiplicidad * porque pueden agrandarse a fin de dar cabida a cualquier número de elementos.

Vea con detenimiento las clases `Car`, `Manager` y `SalesPerson` en las figuras 12.4, 12.5 y 12.6. Simplemente almacenan y recuperan datos. Observe la variable de instancia `sales` de `SalesPerson`: sigue la pista de las ventas totales de un vendedor para el año en curso. No hay métodos para acceder o actualizar la variable de instancia `sales`. Estos métodos se han omitido a fin de evitar un código revuelto y para mantener el centro de atención en el tema a la mano: agregación y composición. En el programa del concesionario, no es necesario contar con estos métodos.

Observe la clase controladora del programa del concesionario de automóviles en la figura 12.7. La mayor parte del código es directa. El método `main` instancia un objeto `Manager`, dos objetos `SalesPerson` y un objeto `Dealership`. Luego, `main` suma los objetos `SalesPerson` y `Car` al objeto `Dealership`. La parte de `main` que merece atención adicional es el uso de las variables locales para los objetos `Manager` y `SalesPerson`, y el uso de objetos anónimos para los objetos `Car`. ¿Por qué la discrepancia? Porque `Manager` y `SalesPerson` se relacionan con la clase `Dealership` con agregación, y `Car` se relaciona con la clase `Dealership` con composición.

```
/*************************************************************
* Dealership.java
* Dean & Dean
*
* Eso representa una organización de venta de automóviles al menudeo.
*************************************************************/

import java.util.ArrayList;

public class Dealership
{
  private String company;
  private Manager manager;
  private ArrayList<SalesPerson> people =
    new ArrayList<SalesPerson>();
  private ArrayList<Car> cars = new ArrayList<Car>();

  //*********************************************************

  public Dealership(String company, Manager manager)
  {
    this.company = company;
    this.manager = manager;
  }

  //*********************************************************

  public void addCar(Car car)
  {
    cars.add(car);
  }

  public void addPerson(SalesPerson person)
  {
    people.add(person);
  }

  //*********************************************************

  public void printStatus()
  {
    System.out.println(company + "\t" + manager.getName());
    for (SalesPerson person : people)
      System.out.println(person.getName());
    for (Car car : cars)
      System.out.println(car.getMake());
  } // end printStatus
} // end Dealership class
```

Contención implementada aquí.

Figura 12.3 Clase `Dealership` para el programa Dealership.

He aquí la regla general para implementar relaciones de agregación. Siempre que dos clases tienen una relación de agregación, es necesario guardar el objeto de la clase contenida en una variable de referencia en la clase contenedora, y también debe guardarlo en otra variable de referencia fuera de la clase contenedora. De esa forma, el objeto puede añadirse a otra agregación y tener dos "propietarios" distintos (el hecho de tener dos propietarios distintos es permitido por la agregación). Cuando esto se ubica en el contexto del programa Dealership, `DealershipDriver` usa variables locales cuando instancia obje-

```
/*******************************
* Car.java
* Dean & Dean
*
* Esta clase implementa un automóvil.
*******************************/

public class Car
{
  private String make;

  //**************************

  public Car(String make)
  {
    this.make = make;
  }

  //**************************

  public String getMake()
  {
    return make;
  }
} // end Car class
```

Figura 12.4 Clase `Car` para el programa Dealership.

```
/*********************************************************
* Manager.java
* Dean & Dean
*
* Esta clase implementa un gerente de ventas de un concesionario de automóviles.
*********************************************************/

public class Manager
{
  private String name;

  //*******************************************************

  public Manager(String name)
  {
    this.name = name;
  }

  //*******************************************************

  public String getName()
  {
    return name;
  }
} // end Manager class
```

Figura 12.5 Clase `Manager` para el programa Dealership.

```
/*************************************************
* SalesPerson.java
* Dean & Dean
*
* Esta clase implementa un vendedor de automóviles.
*************************************************/

public class SalesPerson
{
  private String name;
  private double sales = 0.0; // sales to date

  //*****************************************

  public SalesPerson(String name)
  {
    this.name = name;
  }

  //*****************************************

  public String getName()
  {
    return name;
  }
} // end SalesPerson class
```

Figura 12.6 Clase `SalesPerson` para el programa Dealership.

tos `Manager` y `SalesPerson`. Esto permite que los objetos `Manager` y `SalesPerson` existan de manera independiente del concesionario, lo cual refleja al mundo real.

A continuación se considerará la regla general para implementar relaciones de composición. Siempre que dos clases tienen una relación de composición, el objeto en la clase contenida debe guardarse en una variable de referencia en la clase contenedora, y no debe guardarse en ninguna otra parte. De esa forma, el objeto sólo puede tener un "propietario" (lo cual es requerido por la composición). Cuando esto se ubica en el contexto del programa Dealership, `DealershipDriver` crea objetos anónimos cuando instancia automóviles. Eso otorga al concesionario propiedad exclusiva y control completo sobre los automóviles, lo cual refleja al mundo real.

12.3 Revisión de herencia

Hasta el momento en este capítulo, la atención se ha centrado en las jerarquías de agregación y composición, donde una clase es el todo y las otras clases son parte del todo. Ahora se volverá a las jerarquías de la herencia, que son cualitativamente distintas de las jerarquías de composición. Mientras una jerarquía de composición describe un anidamiento de cosas, una jerarquía de la herencia describe una elaboración de conceptos. El concepto en la parte superior es el más general/genérico, y los conceptos en la parte inferior son los más específicos.

Jerarquías de la herencia en el mundo real

Antes de considerar el código de la jerarquía de la herencia, se pensará en un ejemplo de jerarquía de la herencia en el mundo real. La figura 12.8 describe algunas de las muchas características posibles de los organismos vivientes actualmente sobre la Tierra, donde las características más generales están en la parte superior del diagrama y las características más específicas están en la parte inferior. Aunque esta figura incluye sólo características de organismos vivientes actuales, sirve para reconocer que en el desarrollo de estas características hubo una secuencia temporal natural. Las características en la parte superior se desa-

```
/*******************************************************
* DealershipDriver.java
* Dean & Dean
*
* Esta clase demuestra la composición de un concesionario de automóviles.
*******************************************************/

public class DealershipDriver
{
  public static void main(String[] args)
  {
    Manager ryne = new Manager("Ryne Mendez");
    SalesPerson nicole = new SalesPerson("Nicole Betz");
    SalesPerson vince = new SalesPerson("Vince Sola");
    Dealership dealership =
      new Dealership("Automóviles usados que están bien.", ryne);

    dealership.addPerson(nicole);
    dealership.addPerson(vince);
    dealership.addCar(new Car("GMC"));
    dealership.addCar(new Car("Yugo"));
    dealership.addCar(new Car("Dodge"));
    dealership.printStatus();
  } // end main
} // end DealershipDriver class
```

Para agregaciones, pasar copias de referencias.

Para composiciones, crear objetos anónimos.

Output:

```
OK Used Cars      Ryne Mendez
Nicole Betz
Vince Sola
GMC
Yugo
Dodge
```

Figura 12.7 Controlador para el programa Dealership.

rrollaron primero, y las características en la parte inferior se desarrollaron después. Los primeros tipos de vida, las bacterias, aparecieron en la Tierra hace casi 4 mil millones de años como organismos unicelulares sin particiones internas. Hace aproximadamente 2 300 millones de años, dentro de las células aparecieron un núcleo y otros componentes, dando origen a organismos unicelulares más complejos denominados eucariotes (organismos verdaderamente celulares). Aproximadamente hace 1 300 millones de años aparecieron los primeros animales. Tenían más de una célula y eran vasculares (tenían contenedores y conductos como arteras y venas). Hace alrededor de 510 millones de años, algunos de los animales (vertebrados) desarrollaron espina dorsal y branquias. Aproximadamente hace 325 millones de años aparecieron los primeros reptiles. Luego, hace alrededor de 245 millones de años, aparecieron los primeros mamíferos.

Reconocer la secuencia temporal natural en la jerarquía de la herencia biológica que describe la vida actual es útil por dos razones: 1) Aquí se está hablando sobre "herencia". Lo que está en la parte inferior de la figura, "hereda" de lo que se encuentra arriba, y en la vida real los descendientes heredan de sus ancestros. 2) La evolución natural de la vida de lo simple (en la parte superior de la figura) a lo complejo (en la parte inferior de la figura) constituye un excelente modelo del desarrollo ingenieril de un programa de computadora orientado a objetos. Una buena práctica de diseño consiste en empezar con una implementación relativamente simple y genérica y agregar especializaciones y complejidad en ciclos de diseño ulteriores. Más tarde se verán algunos ejemplos de esto.

Empiece con lo genérico.

Con la composición, ciertas clases contienen a otras clases, pero con la herencia no hay esta relación de contención. Por ejemplo, en la figura 12.8, Animales está por arriba de Mamíferos, pero ningún ani-

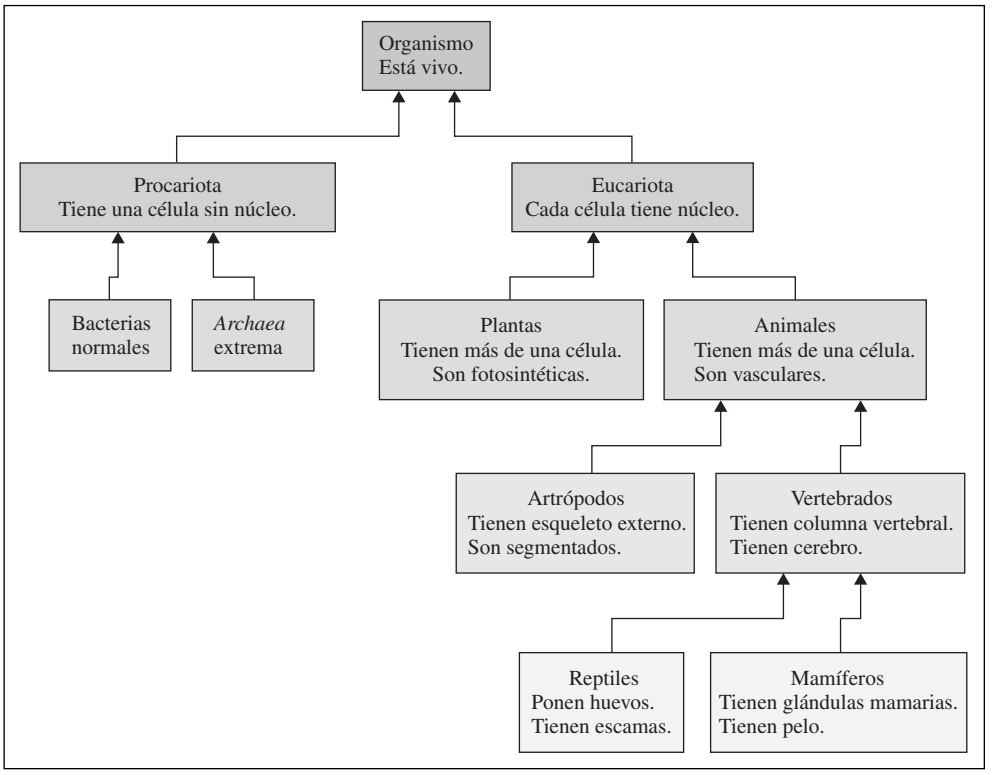


Figura 12.8 Ejemplo biológico de una jerarquía de la herencia.

mal contiene a ningún mamífero. En lugar de eso, un animal es un tipo genérico, y un mamífero es una versión especializada de un animal.

Cada tipo de organismo descendiente hereda algunas características de sus ancestros y añade algunas nuevas características de su parte. En una jerarquía de la herencia, se supone que las características asociadas con un tipo alto en la jerarquía no son todas las características inherentes a un organismo individual viviente. Idealmente, las características asociadas con cada tipo alto en la jerarquía deben ser justamente las características que se "conservan": en realidad, heredadas por todos los tipos descendientes de ese tipo. Así, idealmente, cualquier tipo en la parte inferior de la jerarquía hereda todas las características asociadas con todos los tipos por arriba de él. Por ejemplo, los mamíferos tienen glándulas mamarias y pelo. Y, puesto que los mamíferos son vertebrados, heredan las características vertebradas de poseer una columna vertebral y un cerebro. Y, puesto que los mamíferos también son animales, también heredan las características animales de tener más de una célula y ser vasculares. Y, puesto que los mamíferos también son eucariotes, también heredan la característica eucariota de tener un núcleo en cada célula.

Los tipos en la parte inferior de una jerarquía de la herencia biológica en la vida real no aparecen en la figura 12.8, porque la jerarquía completa es demasiado grande para presentarla en una figura. Lo que está realmente en el fondo son especies, como el *Homo Sapiens* (seres humanos). En la naturaleza, la reproducción sólo es posible entre miembros de la misma especie. En forma semejante, en un programa de computadora OOP ideal, los únicos tipos *realizables* (instanciables) son los tipos que están en la base de las jerarquías de la herencia. El hecho de organizar una jerarquía de la herencia de modo que todos los tipos realizables (instanciables) aparezcan sólo en el nivel más bajo (las *hojas* de un árbol jerárquico) minimiza la duplicación, lo cual minimiza el trabajo de mantenimiento y mejora.

Intente instanciar sólo hojas.

Diagramas de clase UML para jerarquías de la herencia

En la figura 12.9 se muestra un diagrama de clase UML para una jerarquía de la herencia que sigue la pista de las personas asociadas con una tienda departamental. La clase superior, `Persona`, es genérica.

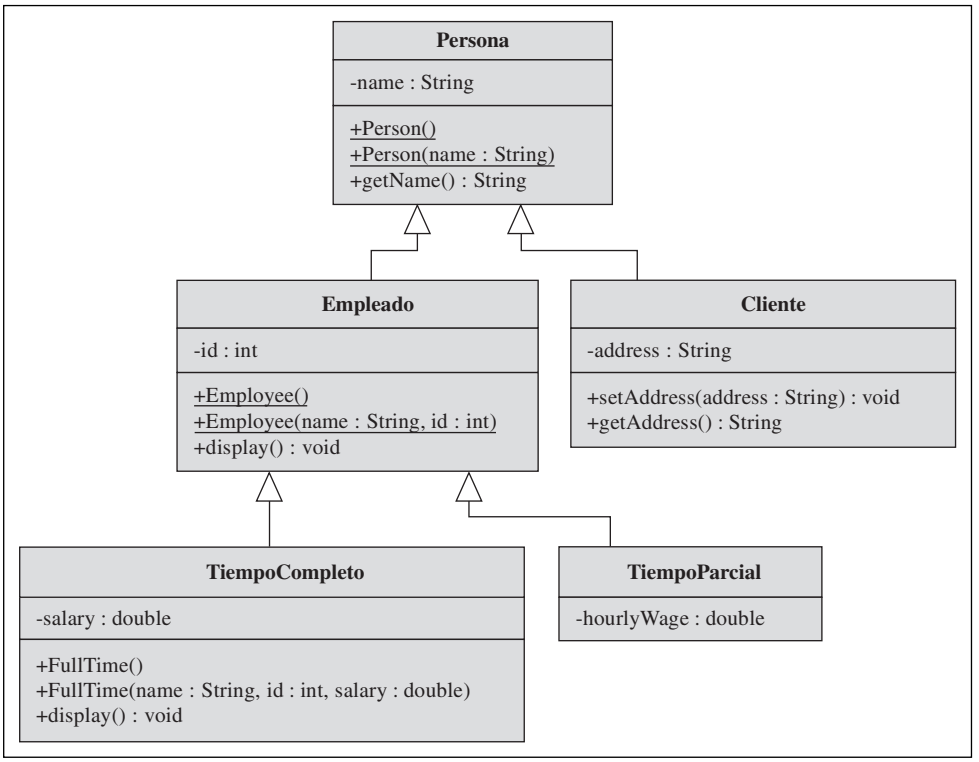


Figura 12.9 Diagrama de clase UML para una jerarquía de la herencia de la clase `Persona`.

Contiene datos y métodos que son comunes a todas las clases en la jerarquía. Las clases abajo de la clase superior son más específicas. Por ejemplo, las clases `Cliente` y `Empleado` describen tipos específicos de personas en la tienda departamental. Puesto que en la tienda hay dos tipos distintos de empleados, la clase `Empleado` tiene dos clases subordinadas para los dos tipos: la clase `TiempoCompleto` para los empleados de tiempo completo y la clase `TiempoParcial` para los empleados de tiempo parcial.

Dentro de una jerarquía de la herencia, las clases inferiores heredan miembros de las clases superiores. Así, las clases `Empleado` y `Cliente` heredan `name` de la clase `Persona`. En forma semejante, las clases `TiempoCompleto` y `TiempoParcial` heredan `id` de la clase `Empleado`. La herencia recorre todo el camino en el árbol de la jerarquía de la herencia, de modo que además de heredar `id` de la clase `Empleado`, las clases `TiempoCompleto` y `TiempoParcial` también heredan `name` de la clase `Persona`.

Dentro de una jerarquía de la herencia, las clases están ligadas por pares. ¿Puede el lector identificar los pares ligados en la figura 12.9? Los cuatro pares de clases ligadas son `Persona-Cliente`, `Persona-Empleado`, `Empleado-TiempoCompleto` y `Empleado-TiempoParcial`. Para cada par de clases ligadas, la clase más general se considera como la *superclase* y la más específica, como la *subclase*.

El hecho de heredar variables y métodos de una superclase permite que una subclase sea un clon de su superclase. Pero hacer una subclase que sólo sea un clon sería tonto, porque en lugar del clon podría usarse la superclase. Siempre es aconsejable que una subclase sea una versión más específica de su superclase. Esto se logra al establecer variables y/o métodos adicionales dentro de la definición de la subclase. Por ejemplo, en la figura 12.9, la clase `Cliente` define una variable de instancia de dirección. Esto significa que los objetos `Cliente` tienen un nombre (heredado de la clase Persona) más una dirección. Las direcciones de los clientes son importantes porque así las tiendas departamentales pueden enviar a sus clientes boletines mensuales de publicidad como "¡Todo debe irse: Gran venta de liquidación!"

Los diagramas de clase UML suelen mostrar las superclases arriba de las subclases. Sin embargo, no siempre es así. Con proyectos grandes se tienen muchas clases y varios tipos diferentes de relaciones entre las clases. Con todo lo anterior, algunas veces resulta imposible dibujar una imagen de jerarquía de clases "limpia" y preservar la disposición tradicional de las superclases arriba de las subclases. Así, algu-

nas veces las subclases aparecen a la izquierda, a la derecha e inclusive arriba de sus superclases. Entonces, ¿cómo puede decidirse cuál es la subclase y cuál es la superclase?

En los diagramas de clase UML se usan una línea continua y una flecha hueca para las relaciones de herencia, donde la flecha apunta a la superclase. En la figura 12.9 observe cómo las flechas apuntan, en efecto, a la superclase.

Terminología de la herencia

Desafortunadamente, los términos superclase y subclase pueden ser engañosos. El "super" en superclase parece implicar que las superclases tienen más capacidad, y el "sub" en subclase parece implicar que las subclases tienen menor capacidad. En realidad, ocurre justamente lo contrario: las subclases tienen más capacidad. Las subclases pueden hacer todo lo que pueden hacer las superclases, y aún más.

Aquí casi siempre se usarán los términos superclase y subclase, puesto que éstos son los términos formales que se usan en Sun, aunque debe tomarse en cuenta que hay una terminología alterna. Los programadores a menudo usan los términos *clase padre* o *clase base* cuando se refieren a una superclase. Y a menudo utilizan los términos *clase hijo* o *clase derivada* para referirse a una subclase. La relación padre-hijo entre clases es importante porque determina la herencia. En una relación padre-hijo humana, el hijo normalmente hereda dinero del padre.[1] La relación padre-hijo de clase imita la relación-padre hijo humana. Pero en una relación padre-hijo de clase, el hijo no hereda dinero; en lugar de ello, hereda las variables y los métodos definidos en la superclase.

Hay otros dos términos relacionados con la herencia que debe conocer el lector. Una clase *ancestro* se refiere a cualquiera de las clases por arriba de una clase particular en una jerarquía de la herencia. Por ejemplo, en la jerarquía de la herencia en la figura 12.9, `Empleado` y `Persona` son ancestros de `TiempoCompleto`. Una clase *descendiente* se refiere a cualquiera de las clases por abajo de una clase particular en una jerarquía de la herencia. Por ejemplo, en la jerarquía de la herencia en la figura 12.9, `Empleado`, `Cliente`, `TiempoCompleto` y `TiempoParcial` son descendientes de `Persona`.

Beneficios de la herencia

Mucho antes de leer ese capítulo, el lector ya estaba convencido sobre el beneficio de modelar sus programas con clases, ¿no es cierto? (En caso de ser necesario recordemos al lector que es porque permiten encapsular cosas.) Así, el lector debe poder ver el beneficio de contar con la clase `Cliente` y también con la clase `Empleado` para un programa de una tienda departamental. Bueno: contar con clases `Cliente` y `Empleado` por separado está bien, pero ¿por qué buscar problemas y proporcionarles una superclase? Si no hay superclase para las clases `Cliente` y `Empleado`, entonces las cosas comunes a los clientes y a los empleados deben definirse en ambas clases. Por ejemplo, en ambas clases se requieren una variable de instancia `name` y un método `getName`. Pero un código redundante siempre es una mala idea. ¿Por qué? Con un código redundante, las tareas de depuración y actualización se vuelven más tediosas. Después de arreglar o mejorar el código en un sitio, el programador debe recordar arreglarlo o mejorarlo también en otro sitio.

En la figura 12.9 observe que las clases a diferentes niveles en la jerarquía contienen diferentes variables de instancia, y que tienen métodos distintos (aunque ambas clases, `Empleado` y `TiempoCompleto`, cuentan con un método `display`, los métodos son diferentes; es decir, se comportan distinto). No hay duplicación funcional y hay máxima *reutilización del código*. La reutilización del código es cuando se tiene un código que proporciona funcionalidad para más de una parte de un programa. Colocar en una superclase un código común proveniente de dos clases, es un ejemplo de reutilización del código. La reutilización del código también puede tener lugar cuando se desea agregar un trozo importante de funcionalidad a una clase existente. La implementación de la funcionalidad podría intentarse agregando código directamente a la clase existente. Pero suponga que la clase funciona perfectamente y que no se quiere ni tocarla por temor de hacerla confusa. O quizá su compañero sabelotodo escribió la clase y el lector no desea arriesgarse a que su compañero se enoje por las modificaciones que usted pudiera hacer al código. No hay problema. Extienda la clase (es decir, elabore una subclase) e implemente la nueva funcionalidad en la clase extendida.

No intente enseñar trucos nuevos a un perro viejo.

[1] El autor John espera que el autor/padre Ray comparta el parecer de esta oración.

Ya se ha visto que la herencia provoca la reutilización del código, y ahora el lector debe estar perfectamente convencido sobre los beneficios de la reutilización del código. Otro beneficio de la herencia es que origina módulos más pequeños (porque las clases se separan en superclases y en subclases). En general, tener módulos más pequeños es bueno porque hay menos códigos que recorrer cuando se buscan errores o se hacen actualizaciones.

12.4 Implementación de jerarquía Persona/Empleado/TiempoCompleto

Para explicar cómo implementar la herencia se implementará la jerarquía Persona/Empleado/TiempoCompleto mostrada en la figura 12.9. En esta sección se implementarán las clases Persona y Empleado, y en la sección 12.6 se implementará la clase TiempoCompleto.

La clase Persona

La figura 12.10 contiene una implementación de la clase Persona. Será una superclase, pero en la clase Persona no hay ningún código que indique que será una superclase. El código especial viene después, cuando se definan las subclases Persona. Es ahí que se indica que Persona es una superclase para estas subclases.

La clase Persona no hace mucho. Simplemente almacena un nombre y permite que el nombre sea recuperado con un método de acceso getName. La clase Persona contiene algo que merece la pena examinar: el constructor de parámetro cero. Normalmente, cuando un controlador instancia una clase Persona, el controlador asigna el nombre de la persona al pasar el argumento de un nombre al constructor de un parámetro. Pero suponga que se desea probar el programa con un objeto Persona, y no se

```
/*******************************************************
* Person.java
* Dean & Dean
*
* Ésta es una clase base para una jerarquía de la herencia.
*******************************************************/

public class Person
{
  private String name = "";

  //*****************************************************

  public Person()
  { }

  public Person(String name)
  {
    this.name = name;
  }

  //*****************************************************

  public String getName()
  {
    return this.name;
  }
} // end Person class
```

Recuerde: una vez que escriba su propio constructor, el constructor automático por defecto de parámetro cero desaparece, y si desea contar con uno, debe escribirlo explícitamente.

Figura 12.10 Clase Persona, superclase de la clase Empleado.

desea liar almacenando un nombre en el objeto `Persona`. El constructor de parámetro cero permite hacer lo anterior. ¿Sabe el lector cuál nombre le será dado a un objeto `Persona` creado por el constructor de parámetro cero? `name` es una variable de instancia en cadena, y el valor por defecto de una variable de instancia en cadena es `null`. Para evitar el espantoso `null` por defecto, observe cómo `null` se inicia en la cadena vacía.

Prueba rápida: ¿es posible lograr la misma funcionalidad al omitir el constructor de parámetro cero puesto que el compilador proporciona automáticamente por defecto un constructor de parámetro cero? No: recuerde que una vez que se escribe cualquier constructor, el compilador ya no proporciona automáticamente por defecto un constructor de parámetro cero.

La clase `Empleado`

La figura 12.11 contiene una implementación de la clase `Empleado` derivada, que proporciona una `id`. Observe la cláusula `extends` en el encabezado de la clase `Empleado`. Para permitir herencia, `extends`, <*superclase*> debe aparecer a la derecha del encabezado de la subclase. Así, `extends Persona` aparece a la derecha del encabezado de la clase `Empleado`.

Observe que la clase `Empleado` define justo una variable de instancia: `id`. ¿Significa esto que un objeto `Empleado` carece de nombre? No. Los objetos `Empleado` tienen nombres porque la clase `Empleado` hereda la variable de instancia `name` de la superclase `Persona`.

El método `display` de la clase `Empleado` es responsable de imprimir la información de un empleado: `name` e `id`. La impresión de `id` es fácil porque `id` se declara dentro de la clase `Empleado`. La impresión de `name` requiere un poco más de tiempo. Puesto que `name` es una variable de instancia `private` en la superclase `Persona`, la clase `Empleado` no puede acceder al nombre directamente (se trata

```
/*****************************************
* Employee.java
* Dean & Dean
*
* Esto describe a un empleado.
*****************************************/

public class Employee extends Person
{
  private int id = 0;

  //***************************************

  public Employee()
  { }

  public Employee(String name, int id)
  {
    super(name);
    this.id = id;
  }

  //***************************************

  public void display()
  {
    System.out.println("name: " + getName());
    System.out.println("id: " + id);
  }
} // end Employee class
```

Esto significa que la clase `Empleado` se deriva de la superclase `Persona`.

Esto llama al constructor `Persona` de un parámetro.

Puesto que `name` es una clase diferente y es `private`, es necesario usar un método de acceso para acceder a ella. Puesto que `getName` es heredada, para ella no se requiere un prefijo de referencia.

Figura 12.11 Clase `Empleado`, derivada de la clase `Persona`.

de la misma interpretación de `private` que siempre se ha tenido). Pero la clase `Empleado` puede acceder al nombre al llamar al método de acceso `getName`. He aquí el código relevante del método `display`:

```
System.out.println("name: " + getName());
```

Como puede recordar el lector, en un método de instancia, si se llama a un método que se encuentra en la misma clase que la clase en que se está actualmente, el prefijo punto de la variable de referencia es innecesario. En forma semejante, en un método de instancia, si se llama a un método que se encuentra en la superclase de la clase en la que se está actualmente, el prefijo punto de la variable de referencia resulta innecesario. Así, en la llamada anterior a `getName` no hay prefijo punto de la variable de referencia.

12.5 Constructores en una subclase

A continuación se analizará el constructor `Empleado` de dos parámetros en la figura 12.11. El objetivo consiste en asignar los valores de `name` e `id` pasados a las variables de instancia asociadas en el objeto `Empleado` instanciado. La asignación a la variable de instancia `id` es fácil porque `id` se declara dentro de la clase `Empleado`. Pero la asignación a la variable de instancia `name` es más difícil porque `name` es una variable de instancia `private` en la superclase `Persona`. En `Persona` no hay método regulador (mutator) `setName`, de modo que ¿cómo se establece `name`? Continúe leyendo. . .

Uso de super para llamar a un constructor de una superclase

Los objetos `Empleado` heredan la variable de instancia `name` de `Persona`. Se concluye que los objetos `Empleado` deben usar el constructor `Persona` para iniciar sus variables de instancia `name` heredadas. Pero, ¿cómo puede un objeto `Empleado` llamar a un constructor `Persona`? Es fácil, una vez que se sabe cómo. Para llamar a un constructor de una superclase, se usa la palabra reservada `super` seguida de paréntesis y una lista de argumentos separados por comas que se desea pasar al constructor. Por ejemplo, vea cómo el constructor `Empleado` en la figura 12.11 llama al constructor `Persona` de un parámetro:

```
super(name);
```

Las llamadas a `super` están permitidas sólo en un sitio particular. Sólo están permitidas desde el interior de un constructor, y deben ser la primera línea dentro de un constructor. Esto debe sonar conocido. En el capítulo 7 se aprendió otro uso de la palabra clave `this`, uso que es distinto al empleo del `this` punto para especificar un miembro de instancia. La sintaxis para este otro uso del `this` es:

```
this(<arguments>);
```

Este tipo de uso del `this` llama a otro constructor (sobrecargado) desde el interior de un constructor en la misma clase. Y recuerde que esta llamada debe hacerla en la primera línea de su constructor.

Por cierto, ¿sería legal tener una llamada al constructor `this` y una llamada al constructor `super` dentro del mismo constructor? No, porque con ambas llamadas al constructor en el mismo constructor, eso significa que sólo una llamada al constructor puede estar en la primea línea. La otra violaría la regla de que las llamadas al constructor deben estar en la primera línea.

Llamadas por defecto al constructor superclase

A los diseñadores de Java en Sun les gusta llamar a constructores superclase porque así se promueve el reúso del software. Si el lector escribe un constructor superclase y no incluye una llamada a otro constructor (con `this` o con `super`), el compilador de Java entra a hurtadillas e inserta por defecto una llamada a un constructor superclase de parámetro cero. Así, aunque el constructor de parámetro cero de `Empleado` de la figura 12.11 tiene un cuerpo vacío, el compilador de Java inserta automáticamente `super();` ahí. Así, estos dos constructores son funcionalmente equivalentes:

```
public Employee()
{ }

public Employee()
```

```
{
  super();
}
```

La llamada explícita `super();` aclara qué está ocurriendo. Siéntase con confianza para introducirla si así lo desea, a fin de hacer más autodocumentado su código.

Siempre que se llama a un constructor, la JVM ejecuta en forma automática el árbol de jerarquías hasta el constructor abuelo más grande y ejecuta primero el constructor abuelo más grande. Luego ejecuta el código en el constructor que está abajo de él y así sucesivamente, de modo que finalmente ejecuta el resto del código en el constructor que se llamó originalmente.[2]

12.6 Sobreposición de métodos

En el capítulo 7 se estudió la sobrecarga de métodos, que es cuando una simple clase contiene dos o más métodos con el mismo nombre pero diferente secuencia de tipos de parámetros. A continuación se abordará un concepto semejante: la *sobreposición de métodos*. Esto ocurre cuando una subclase tiene un método con el mismo nombre, la misma secuencia de tipos de parámetros y el mismo tipo de retorno que un método en una superclase. El término "sobreposición" debe tener sentido cuando se percibe que un método se sobrepone/sustituye a su método de clase asociado. Esto significa, por defecto, que un objeto de la subclase utiliza el método que se sobrepone de la subclase y no el método sobrepuesto de la superclase.

El concepto de objeto de una subclase usando el método de la subclase en lugar del método de la superclase se encuentra en línea con este principio general de la programación: lo local tiene prioridad sobre lo global. ¿Puede pensar el lector en dónde es válida también esta regla? Si una variable local y una variable de instancia tienen el mismo nombre, la variable local asume precedencia cuando se está dentro del método de la variable local. El mismo razonamiento es válido para parámetros que asumen precedencia sobre variables de instancia cuando se está dentro del método del parámetro.

Ejemplo de sobreposición de métodos

Para explicar la sobreposición de métodos se continuará con la implementación del programa Persona/Empleado/TiempoCompleto. Las clases `Persona` y `Empleado` se implementaron en la sección 12.4. La clase `TiempoCompleto` se implementa en la figura 12.12. Observe el método `display` de `TiempoCompleto`. Tiene la misma secuencia de tipos de parámetros que el método `display` y la clase `Empleado` de la figura 12.11. Puesto que la clase `TiempoCompleto` `extends` la clase `Empleado`, el método `display` de `TiempoCompleto` se sobrepone al método `display` de `Empleado`.

Uso de `super` para llamar a un método sobrepuesto

Algunas veces, un objeto de la subclase puede requerir llamar al método sobrepuesto de la superclase. Para realizar esta llamada, la llamada al método debe prefijarse con `super` y luego un punto. Por ejemplo, en la subclase `TiempoCompleto` en la figura 12.12 observe cómo el método `display` llama al método `display` de la superclase con `super.display();`.

Ahora vuelva a mirar la llamada `super.display()` al método en la clase `TiempoCompleto` en la figura 12.12. ¿Qué supone el lector que ocurriría si se olvida prefijar con `super` punto esa llamada al método? Sin el prefijo `display();` se llamaría al método `display` en la clase actual, `TiempoCompleto`, no al método `display` en la superclase. Al ejecutar el método `display` de la clase `TiempoCompleto`, la JVM llamaría de nuevo al método `display` en la clase `TiempoCompleto`. Este proceso se repetiría en un ciclo infinito.

Por cierto, puede tenerse una serie de métodos de sobreposición; es decir, es posible sobreponer un método sobrepuesto, aunque es ilegal tener una serie de prefijos `super` punto encadenados entre sí. En otras palabras, en la jerarquía de la herencia `Persona/Empleado/TiempoCompleto`, suponga que la clase `Persona` contiene un método `display` que es sobrepuesto por las clases `Empleado` y `Tiem-`

[2] Esta secuencia es la misma que ocurre en forma natural en el desarrollo embrionario de una criatura viviente. Las características que se desarrollan primero son las más antiguas.

```
/*************************************************************
* FullTime.java
* Dean & Dean
*
* Esto describe un empleado de tiempo completo.
*************************************************************/

public class FullTime extends Employee
{
  private double salary = 0.0;

  //*********************************************************

  public FullTime()
  { }

  public FullTime(String name, int id, double salary)
  {
    super(name, id);
    this.salary = salary;
  }

  //*********************************************************

  public void display()
  {
    super.display();
    System.out.printf(
      "salary: $%,.0f\n", salary);
  }
} // end FullTime class
```

Esto llama al constructor `Empleado` de dos parámetros.

Este método se *sobrepone* al método `display` definido en la clase `Empleado`.

Esto llama al método `display` definido en la clase `Empleado`.

Figura 12.12 Clase `TiempoCompleto`, que ilustra la sobreposición de métodos.

`poCompleto`. En la clase `TiempoCompleto` sería ilegal llamar al método `display` de la clase `Persona` como sigue:

```
super.super.display();
```

error de compilación

Para llamar al método `display` de la clase `Persona` desde la clase `TiempoCompleto`, es necesario llamar al método `display` de la clase `Empleado`, y depender del método `display` de la clase `Empleado` para llamar al método `display` de la clase `Persona`.

¿Ya observó que `super` tiene dos propósitos distintos? Puede usarse `super` punto para llamar a un método de sobreposición, y también puede usarse `super` con paréntesis (por ejemplo, `super (name);`) para llamar al constructor de una superclase.

Los tipos de retorno deben ser los mismos

Un método que se sobrepondrá debe tener el mismo tipo de retorno que el método al que se va a sobreponer. Si el tipo de retorno es distinto, el compilador genera un error. En otras palabras, si una subclase y una superclase tienen métodos con el mismo nombre, la misma secuencia de tipos de parámetros y tipos de retorno diferentes, entonces el compilador genera un error.

Este error no ocurre muy a menudo porque si se tienen métodos con los mismos nombres y secuencias de tipos de parámetros, por lo general también se desea tener los mismos tipos de retorno. Pero se verá que el error surge a partir de entonces y luego cuando se efectúa una depuración, por lo que conviene estar alerta. Por cierto, si una subclase y una superclase tienen métodos con el mismo nombre y diferentes se-

cuencias de tipos de parámetros, no importa si los tipos de retorno son los mismos. ¿Por qué? Porque tales métodos no están en una relación de sobreposición. Son métodos distintos por completo.

12.7 Utilización de jerarquía `Persona/Empleado/TiempoCompleto`

A continuación se reforzará lo aprendido sobre herencia al considerar lo que ocurre cuando se instancia un objeto del tipo derivado del nivel más bajo y ese objeto se usa para llamar a métodos sobrepuestos y métodos heredados. La figura 12.13 contiene un controlador para la clase `TiempoCompleto`, y la salida ulterior muestra lo que hace. Este controlador instancia un objeto `fullTimer` de la clase `TiempoCompleto`. Luego, el objeto `fullTimer` llama a su método `display`. Como se muestra en la figura 12.12, este método `display` usa `super` para llamar al método `display` de la clase `Empleado`, que imprime el `name` de `fullTimer` y el `name` de `id`. Luego, el método `display` de `fullTimer` imprime el `salary` de `fullTimer`.

En la declaración final en la figura 12.13, el objeto `fullTimer` llama a su método `getName` e imprime el `name` de `fullTimer`. ¡Pero espere un minuto! La clase `TiempoCompleto` no cuenta con un método `getName` y su superclase, `Empleado`, tampoco. Parece que el código está llamando a un método inexistente. ¿Qué está ocurriendo? Lo que ocurre es herencia: producida por estas pequeñas y maravillosas cláusulas `extends`. Debido a que no hay ningún método `getName` definido de manera explícita en su propia clase `TiempoCompleto`, el objeto `fullTimer` recorre su jerarquía de la herencia hasta que encuentra el método `getName`, y luego usa ese método. En este caso, el primer método `getName` encontrado está en la clase `Persona`, de modo que éste es el método que hereda y usa el objeto `fullTimer`. No es necesario usar un `super` punto para introducir el método `getName` (aunque usar `super` punto funciona, en caso de tener curiosidad). Si un método no está en la clase actual, la JVM automáticamente recorre su jerarquía de la herencia y usa la primera definición de ese método que encuentra.

Observe que el controlador no instancia ningún objeto `Empleado` o `Persona`. Simplemente instancia un objeto de una clase que está en el fondo de la jerarquía de la herencia. Ésta es la forma en que debe usarse una buena jerarquía de la herencia. Idealmente, sólo deben instanciarse objetos de clases en el fondo de la jerarquía de la herencia. Idealmente, todas las clases por arriba de las clases en el fondo

```
/*************************************************************
* FullTimeDriver.java
* Dean & Dean
*
* Esto describe un empleado de tiempo completo.
*************************************************************/

public class FullTimeDriver
{
  public static void main(String[] args)
  {
    FullTime fullTimer = new FullTime("Shreya", 5733, 80000);

    fullTimer.display();
    System.out.println(fullTimer.getName());
  }
} // end FullTimeDriver class
```

Salida:

```
name:    Shreya
id:      5733
salary:  $80,000
Shreya
```

Figura 12.13 Controlador de constructores y métodos en una jerarquía de herencia.

están ahí para hacer simples a las clases en el fondo. En la vida real, a menudo se usan las clases que están arriba de las clases en el fondo, aunque la situación ideal es utilizar sólo estas clases.

12.8 El modificador de acceso `final`

El modificador de acceso `final` ya se ha usado desde hace algún tiempo para convertir una variable en una constante nombrada. En esta sección se aprenderá cómo usar `final` para modificar un método y una clase.

Si el modificador `final` se usa en el encabezado de un método, se evita que el método sea sobrepuesto con una nueva definición en una subclase. Esto sería aconsejable si se considera que el método es perfecto y no se quiere que su significado original "quede a la deriva". También podría querer considerar el uso de `final` para que las cosas se acelerasen un poco más. Los métodos que usan el modificador `final` deben ejecutarse más rápido porque el compilador puede generar un código más eficiente para ellos. La eficiencia en el código proviene del hecho de que el compilador no tiene que preparar la posibilidad de herencia. Sin embargo, la mejora en velocidad es minúscula para añadir `final` a un solo método, y quizá no se observe esta mejora, a menos que se tenga un proyecto de programación grande con muchas subclases y `final` se use bastante.

Si el modificador de acceso `final` se usa en el encabezado de una clase, se impide que la clase tenga subclases. Esto sería aconsejable si se tiene una clase que es buena y confiable y se desea preservar su calidad y protegerla del "crecimiento innecesario" (*feature creep*) futuro. Por cierto, si una clase se declara como clase `final`, no tiene caso que en ninguno de sus métodos se especifique `final`. Una clase `final` no puede extenderse, de modo que no existe sobreposición de métodos.

Aun cuando puede ser difícil ver beneficios palpables por el uso del modificador `final`, continúe y úselo cuando sea idóneo hacerlo. E inclusive si el lector no lo utiliza para sus propios programas, es necesario que lo comprenda porque lo verá muy a menudo en las clases de la biblioteca API de Java. Por ejemplo, la clase `Math` se define con el modificador de acceso `final`, de modo que es ilegal extender la clase `Math` y sobreponer cualquiera de sus métodos.

12.9 Utilización de herencia con agregación y composición

Se han descrito varias formas en que las clases pueden estar relacionadas: con agregación, composición y herencia. A continuación se considerará el uso de estas tres relaciones juntas.

Comparación de agregación, composición y herencia

Ambas, la agregación y la composición implementan una relación tiene-un. La agregación y la composición se denominan relaciones tiene-un porque una clase, la clase contenedora, tiene una clase componente en su interior. Por ejemplo, en el programa Dealership de la sección 12.2, un concesionario tiene un gerente de ventas, sin derechos de propiedad exclusivos, razón por la cual el programa Dealership implementa la relación `Dealership-SalesManager` con agregación. Asimismo, un concesionario cuenta con un inventario de automóviles, con derechos de propiedad exclusivos, razón por la cual el programa `Dealership` implementa la relación `Dealership-Car` con composición.

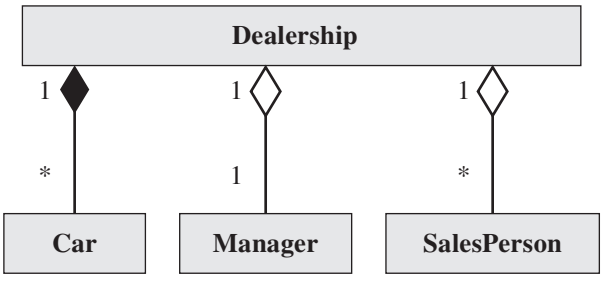
La herencia implementa una relación *es-un*. Una relación de herencia se denomina relación *es-un* porque una clase, una subclase, es una versión más detallada de otra clase. Por ejemplo, en el programa Persona/Empleado/TiempoCompleto, un empleado de tiempo completo es un empleado, razón por la cual el programa `Dealership` también implementa la relación `Persona/Empleado/TiempoCompleto` con herencia. También, un empleado es una persona, por lo cual el programa también implementa la relación `Empleado-Persona` con herencia.

Es importante tener en cuenta que lo anterior no constituye formas alternas para representar la misma relación. Son formas de representar relaciones diferentes. Las relaciones de agregación y composición ocurren cuando una clase es un todo integrado por partes constituyentes no triviales definidas en otras clases. La relación de herencia es cuando una clase es una versión más detallada de otra clase. En términos más formales, la herencia es cuando una clase, una subclase, hereda variables y métodos de otra

clase, una superclase, y luego los complementa con variables y métodos adicionales. Puesto que la composición y la herencia tratan con aspectos distintos de un problema, muchas soluciones de programación incluyen una mezcla de ambos paradigmas.

Combinación de agregación, composición y herencia

En el mundo real, suele ser común tener relaciones de agregación, composición y herencia juntas en el mismo programa. Se considerará un ejemplo en el que se usan las tres relaciones. En el programa Dealership de la sección 12.2 se usan agregación y composición, como se ilustra con este diagrama de clase UML:



¿Qué tipo de relación de herencia puede/debe agregarse al programa Dealership? Si se vuelven a observar las figuras 12.5 (clase `Manager`) y 12.6 (clase `SalesPerson`) se verá que estas dos clases declaran la misma variable de instancia, `name`, y que ambas definen el mismo método de instancia, `getName`.

Factorice el código común.

Éste es un ejemplo de duplicación indeseable, de modo que para eliminarla se usa herencia. El hecho de introducir herencia en ese programa no modifica la estructura original todo-partes. Simplemente introduce un mecanismo complementario que elimina la duplicación.

En la figura 12.14 se muestra un diagrama de clase UML mejorado y expandido para un nuevo programa Dealership2. Si este diagrama se compara con el diagrama de clase UML previo, se verá que cada clase está definida a efecto de incluir variables y métodos de instancia. El diagrama en la figura 12.14 también incluye una clase `Persona`. Las clases previas `Manager` y `SalesPerson` ahora heredan una variable, dos constructores y un método de esta clase `Persona`. La herencia reduce las clases `Manager` y `SalesPerson` a las clases más simples `Manager2` y `SalesPerson2`. Estas clases más simples no requieren una declaración explícita de `name` ni de `getName` porque heredan estos miembros de `Persona`. Lea el código de las clases más breves `Manager2` y `SalesPerson2` en las figuras 12.15 y 12.16.

La clase `Car` permanece sin cambio respecto al programa Dealership original; si el lector desea ver su código observe de nuevo la figura 12.4. Las clases `DealerShip2` y `DealerShip2Driver` son las mismas que las clases `Dealership` y `DealershipDriver` definidas en las figuras 12.3 y 12.7, respectivamente, excepto que `Dealership` ha cambiado a `Dealership2`, `Manager` ha cambiado a `Manager2` y `SalesPerson` ha cambiado a `SalesPerson2`.

En la figura 12.14, la adición de la clase `Persona` la hace ver similar a lo que hicimos en el programa DealerShip2, el cual se hizo más grande al sumarle otra clase. Pero la clase `Persona` adicional ya estaba definida en otro programa, el programa Persona/Empleado/TiempoCompleto. Al tomar prestada la clase `Persona` de ese programa se obtiene algo por nada. La clase prestada `Persona` permite acortar otras dos clases. El hecho de poder tomar prestadas clases que ya han sido escritas y luego heredarlas en otros contextos es un beneficio importante de POO. Si se observan las clases prescritas en el API de Java se verá que tienen bastante herencia una de otra, y en muchos casos, también se tiene la opción de heredar de ellas hacia el propio programa de lector.

12.10 Práctica de diseño con un ejemplo de juego de cartas

En la sección previa se aprendió a usar diferentes tipos de relaciones de clase juntas en un solo programa. La forma en que se aprendió fue sumando herencia a un programa existente. En esta sección, de nuevo se

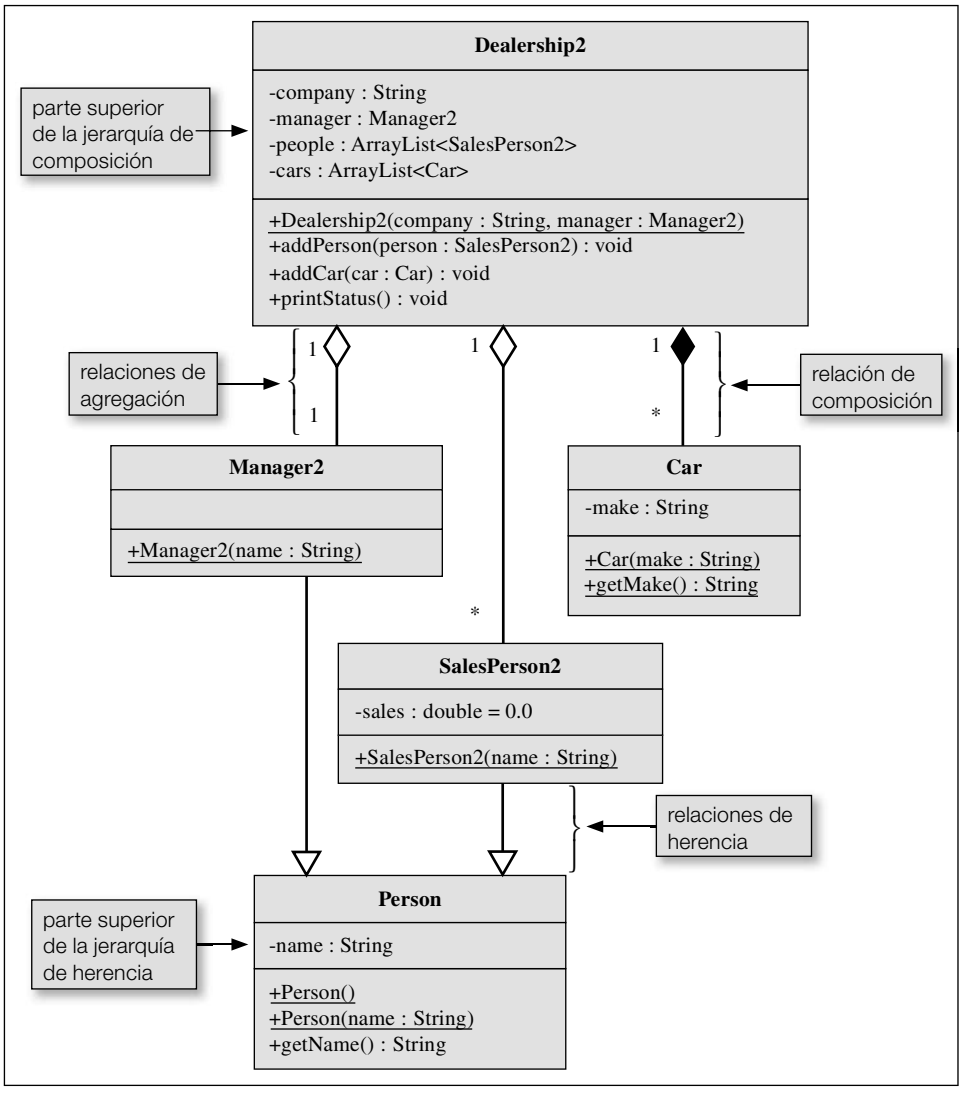


Figura 12.14 Diagrama de clase para el programa revisado del concesionario de automóviles: Dealership2.

```
/*******************************************
* Manager2.java
* Dean & Dean
*
* Esto representa al gerente del concesionario de automóviles.
*******************************************/

public class Manager2 extends Person
{
  public Manager2(String name)
  {
    super(name);
  }
} // end Manager2 class
```

Figura 12.15 Clase `Manager2` para el programa Dealership2.

```
/*******************************************
* SalesPerson2.java
* Dean & Dean
*
* Esto representa un vendedor
*******************************************/

public class SalesPerson2 extends Person
{
  private double sales = 0; // ventas a la fecha

  //*****************************************

  public SalesPerson2(String name)
  {
    super(name);
  }
} // end SalesPerson2 class
```

Figura 12.16 Clase `SalesPerson2` para el programa Dealership2.

Aprenda con la práctica. usarán diferentes tipos de relaciones de clase, pero en esta ocasión el lector diseñará el programa desde el principio. El lector hará casi todo el trabajo, en lugar de sólo comprender cómo lo hizo otra persona.

Su misión (debe elegir aceptarla)

La misión del lector consiste en diseñar e implementar un programa genérico de un juego de cartas. Para llevar a cabo su misión siga estas directrices:

- Suponga que es un juego como Guerra o gin rummy, donde hay un mazo de cartas y dos jugadores.
- Decida sobre clases idóneas. Para cada clase, dibuje un diagrama de clase UML y escriba el nombre de la clase.
- Busque relaciones de composición entre clases. Para cada par de clases relacionadas por composición, trace una línea de asociación por composición con valores de multiplicidad idóneos.
- Para cada clase, decida sobre variables de instancia idóneas.
- Para cada clase, decida sobre métodos `public` idóneos.
- Busque variables y métodos de instancia comunes. Si dos o más clases contienen un conjunto de variables y métodos de instancia, proporcione una superclase y mueva las variables y métodos de instancia comunes a la superclase. Las clases que originalmente contenían miembros comunes ahora se convierten en subclases de la superclase. Para cada par subclase-superclase, trace una línea de asociación con una flecha de herencia de la subclase a la superclase para indicar una relación de herencia.

Luego, continúe y aplique las directrices anteriores para dibujar un diagrama de clase UML para un programa genérico de un juego de cartas. Puesto que este ejercicio no es trivial, quizás el lector esté tentado a buscar la solución antes de intentar llegar a la suya. Por favor, ¡resista la tentación! Al implementar su propia solución, aprenderá más y estará alerta sobre la presencia de problemas potenciales.

Definición de las clases y las relaciones entre ellas

¿Ya terminó su diagrama de clase? En caso afirmativo, puede continuar…

Cuando hay de por medio un diagrama de clase, lo primero que hay que hacer es decidir sobre las clases mismas. Desafortunadamente, eso es algo difícil. Las clases fáciles son aquellas que corresponden directamente a algo que puede verse. Al visualizar un juego de cartas, ¿puede ver dos personas que tienen cartas en la mano y que toman cartas adicionales de un mazo situado entre ellas? El lector debe poder ver un mazo, dos manos, cartas individuales y dos personas. Para el mazo use una clase `Deck`. Para

las dos manos use una clase `Hand`. Para las cartas individuales use una clase `Card`. El lector puede decidir o no representar a las personas. Si el lector está implementando un juego de cartas elaborado donde los jugadores poseen personalidad, use una clase `Persona`. En caso contrario, no hay necesidad de una clase `Persona`. La situación se mantendrá sencilla, de modo que no se implementará la clase `Persona`.

Al pensar en la gran imagen, el lector debe preguntarse "¿Qué es un juego?" Un juego es una composición de varias partes, de modo que `Game` debe definirse como una clase completa y las otras partes del juego deben definirse como partes del juego. Un `Game` está compuesto de tres componentes/partes: un mazo y dos manos. Así, `Deck` y `Hand` son clases parte dentro de la clase de composición `Game`. En el diagrama de clase en la figura 12.17, observe la línea de asociación que une `Game` y `Deck`. La línea de asociación tiene un diamante lleno, que indica composición, y tiene valores de multiplicidad uno a uno, lo cual indica que cada juego tiene un mazo. La línea de asociación `Game` a `Hand` también tiene un diamante lleno para composición, pero tiene valores de multiplicidad uno a dos, lo cual indica que cada juego tiene dos manos.

Llegar a la idea de usar una clase `Game` probablemente es más difícil que llegar a las ideas de usar clases `Deck`, `Hand` y `Card`. ¿Por qué? Un juego no es tangible (es decir, no es posible tocarlo), de modo que es difícil verlo como una clase. ¿Por qué molestarse con tener una clase `Game`? Si se omite la clase `Game`, aún es posible implementar un juego de cartas. En lugar de declarar los objetos mazo y mano dentro de la clase `Game`, es posible declararlos dentro del método `main`. Pero es más elegante colocarlos dentro de una clase `Game`. ¿Por qué? Al colocarlos dentro de una clase `Game`, se persigue el objetivo de encapsulamiento. Así se permite también que `main` sea moderno.

Como se verá después, si ya se ha definido una clase `Game` del método `main` del controlador, simplemente necesita instanciar un objeto `Game` y luego llamar a `playAGame` y es todo. No es posible hacer más moderno (y elegante) lo anterior.

Para cada clase en el programa de juego de cartas, ¿cuáles son sus miembros (es decir, las variables y los métodos de instancia)? Primero se abordarán las clases fáciles: `Game` y `Card`. La clase `Game` requiere tres variables de instancia: una para el mazo y dos para las dos manos. Requiere un método para

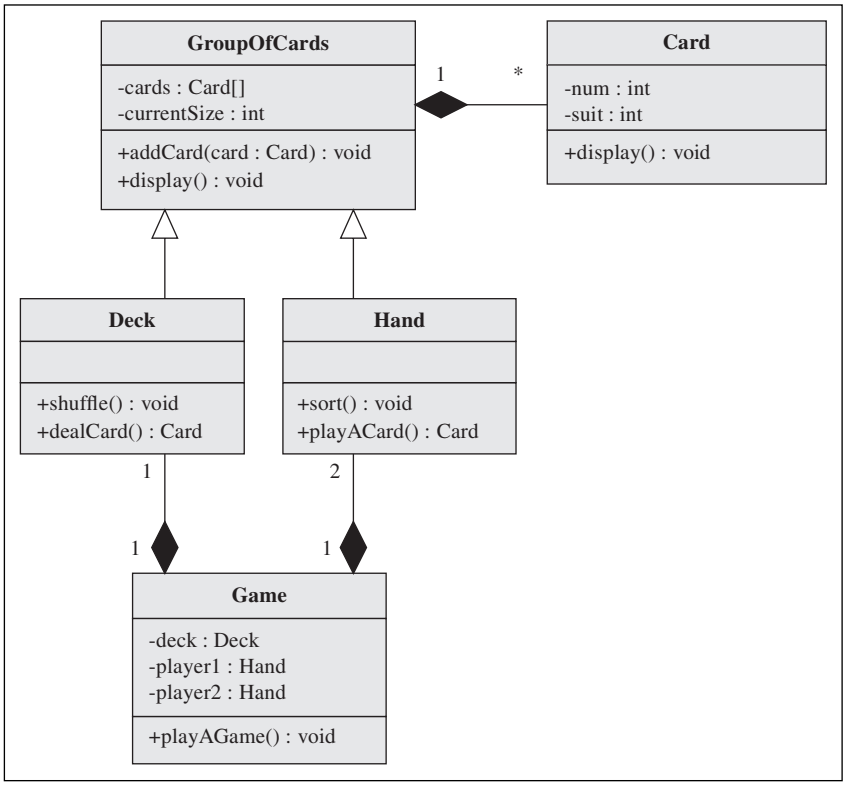


Figura 12.17 Diagrama de clase preliminar para el programa de juego de cartas.

jugar un juego. La clase `Card` requiere dos variables de instancia: una para un número (del dos hasta el as) y una para un palo (de tréboles a picas). Requiere un método para exhibir el número de carta y los valores del palo. Como una verificación de sanidad, compruebe que los miembros `Game` y `Card` en la figura 12.17 coinciden con lo que acaba de decirse.

La clase `Deck` requiere una variable de instancia para un arreglo de cartas de modo que cada carta sea un objeto `Card`. La clase `Deck` también requiere una variable de instancia para seguir la pista del tamaño actual del mazo. La clase `Deck` requiere métodos para barajar y repartir. Para ayudar a la depuración, quizá también sea necesario incluir un método para mostrar todas las cartas en el mazo.

La clase `Hand` requiere variables de instancia para un arreglo de cartas y para un valor de tamaño actual. Requiere métodos para mostrar todas las cartas, agregando una carta a la mano y jugando una carta desde la mano. Para la mayor parte de los juegos de cartas, también se requiere un método para ordenar la mano. Diferentes juegos de cartas usan métodos `Hand` diferentes y/o adicionales. La situación se mantendrá simple, de modo que no es necesario preocuparse por ellos.

El paso siguiente consiste en tratar de identificar miembros comunes y moverlos a una superclase. Las clases `Deck` y `Hand` tienen tres miembros comunes: una variable de arreglo `cards`, una variable `currentSize` y un método `display`. Al mover estos miembros a una superclase, ¿cuál sería un buen nombre para tal clase? Debe ser algo genérico que pueda usarse como la superclase tanto de `Deck` como de `Hand`. `GroupOfCards` o simplemente `Cards` suena bien. Se usará `GroupOfCards`. En el diagrama de clase de la figura 12.17 observe las líneas de asociación de herencia que unen `Deck` con `GroupOfCards` y `Hand` con `GroupOfCards`.

Ahora se han analizado los miembros de todas las cinco clases en el programa de juego de cartas, así como las relaciones entre cuatro de las clases: `Game`, `Deck`, `Hand` y `GroupOfCards`. La última pieza del rompecabezas del diagrama de clase UML es la relación entre `GroupOfCards` y `Card`. ¿Se trata de una relación es-un o una relación tiene-un? No es una relación es-un porque no tiene sentido decir que un grupo de cartas es una carta o que una carta es un grupo de cartas. En lugar de ello, se trata de una relación tiene-un porque un grupo de cartas tiene una carta (un grupo de cartas suele tener más de una carta, pero eso no niega que es una relación tiene-un). En la figura 12.17 observe la línea de asociación de composición tiene-un que une `GroupOfCards` y `Card`. La figura 12.17 sugiere implementar la composición como un arreglo denominado `cards` aunque podría ser una `ArrayList`.

Observe que la etiqueta de la figura 12.17 dice diagrama de clase "preliminar". Es preliminar porque para una aplicación de tamaño normal, resulta casi imposible obtener el diagrama de clase al 100% en el intento del primer corte. Cuando el lector ha terminado de escribir y probar (!) su programa prototipo, debe regresar y actualizar en consecuencia su diagrama de clase. El diagrama de clase sirve para dos cosas: pronto en el proceso de diseño, ayuda a organizar las ideas y mantiene a todo mundo en la misma página. En la fase de posimplementación sirve como documentación, de modo que las partes interesadas pueden obtener rápidamente el manejo de la organización de la aplicación.

El diseño es un proceso iterativo.

Herencia *versus* composición

Cuando se decide sobre la relación ente dos clases, suele ser bastante claro cuándo utilizar herencia o composición. Por ejemplo, en el programa Dealership, un `Manager` es una `Persona`, de modo que se usó herencia. En el programa del juego de cartas, un `Game` tiene un `Deck`, de modo que se usó composición.

No obstante, a veces no es tan claro. Por ejemplo, puede afirmarse que un `Deck` es un `GroupOfCards`, y también que un `Deck` tiene un `GroupOfCards`. Como regla práctica, en casos como éste en que existe la relación de herencia es-un y también existe la relación de composición tiene-un, resulta mejor decidir por la primera. Para ver por qué, se comparará el código para cada una de las dos relaciones.

Vea la clase `Deck` en la figura 12.18, que implementa la relación con herencia `Deck-GroupOfCards`. También vea la clase alterna `Deck` en la figura 12.19, que implementa la relación `Deck-GroupOfCards` con composición. Opinamos que el código de herencia en la figura 12.18 es más elegante que el código de composición en la figura 12.19. Tiene una línea menos, lo cual está bien, pero lo más importante es que no está revuelto con referencias a una variable `GroupOfCards`. En el código de composición, se requiere 1) declarar una variable `GroupOfCards`, 2) instanciar la variable `GroupOfCards` y 3) prefijar la llamada a `addCard` con el objeto que llama `GroupOfCards`. ¿No es más agradable el código de herencia donde no es necesario preocuparse de todo esto? En particular, puede llamarse

```
public class Deck extends GroupOfCards
{
  public static final int TOTAL_CARDS = 52;

  public Deck()
  {
    for (int i=0; i<TOTAL_CARDS; i++)
    {
      addCard(new Card((2 + i%13), i/13));
    }
  } // end constructor
  ...

} // end class Deck
```

Esto implementa herencia.

Con herencia, no es necesario prefijar la llamada al método con un objeto de referencia.

Figura 12.18 Implementación de herencia para la clase `Deck`.

directamente a `addCard` (no se requiere ningún objeto `GroupOfCards` que llama), lo cual resulta en un código más legible. Por cierto, el método `addCard` se define en la clase `GroupOfCards`. Con herencia, el hecho de que esté en una clase por separado de `Deck` es transparente. En otras palabras, se llama a `it` desde el constructor `Deck` de la misma forma en que se llamaría a cualquier otro método `Deck`: sin un objeto que llama.

Para algunos pares de clase (como `Deck` y `GroupOfCards`) es legal usar una relación de herencia o una de composición. Pero nunca es correcto usar tanto la herencia como la composición para la misma característica. ¿Qué ocurriría si `Deck` declara una variable local `GroupOfCards` y `Deck` también heredase de una clase `GroupOfCards`? Los objetos `Deck` contendrían dos grupos de cartas por separado, ¡lo cual está mal!

En este punto, quizá sería conveniente regresar al diagrama preliminar de clase UML y agregar más detalles. En el diagrama de clase en la figura 12.17 no aparecen constantes ni constructores. Al trabajar con la estructura de la clase `Deck` (vea la figura 12.18), ahora es evidente que es necesario 1) agregar una constante `TOTAL_CARDS` a la clase `Deck`, 2) agregar un constructor a la clase `Deck` y 3) agregar un constructor a la clase `Card`. Para adquirir práctica, alentamos al lector a actualizar el diagrama de clase

```
public class Deck
{
  public static final int TOTAL_CARDS = 52;
  private GroupOfCards groupOfCards;

  public Deck()
  {
    groupOfCards = new GroupOfCards();

    for (int i=0; i<TOTAL_CARDS; i++)
    {
      groupOfCards.addCard(new Card((2 + i%13), i/13));
    }
  } // end constructor
  ...

} // end class Deck
```

Con composición, se requiere declarar e instanciar una variable `GroupOfCards`.

Con composición, es necesario prefijar la llamada al método con un objeto de referencia.

Figura 12.19 Implementación de composición para la clase `Deck`.

```
public static void main(String[] args)
{
  Scanner stdIn = new Scanner(System.in);
  String again;
  Game game;

  do
  {
    game = new Game();
    game.playAGame();
    System.out.print("¿Juega otro juego (y/n)?: ");
    again = stdIn.nextLine();
  } while (again.equals("y"));
} // end main
```

Figura 12.20 Método `main` del programa juego de cartas.

en la figura 12.17 con estos cambios en mente. Si no está de acuerdo, no hay problema; nuestro punto importante aquí es alertarlo acerca de la naturaleza iterativa del proceso de diseño del programa. Intente organizar su pensamiento lo más claro que sea posible, aunque esté preparado para ajustarlo más tarde.

El diseño es un proceso gradual.

Codifique para empezar

Una vez que se ha terminado con el diagrama de clase del juego de cartas, normalmente el siguiente paso sería la implementación de las clases con código Java. No mostraremos los detalles de la implementación de clase, aunque nos gustaría mostrarle cómo las clases sugeridas podrían ser controladas por un método `main`. Si las directrices de diseño POO se siguieron al pie de la letra es fácil obtener un método `main` elegante: vea la figura 12.20. Observe cuán breve y comprensible es el método `main`. ¡Sí!

Otro ejemplo ilustrará aún más la forma en que las clases terminadas pueden ser utilizadas por el resto del programa. Observe la llamada de `main` a `playAGame` en la figura 12.20. En la figura 12.21 se muestra una implementación parcial para el método `playAGame`. Para barajar el mazo se llama a `deck.shuffle()`. Para repartir una carta al primer jugador, llame a `player1.addCard(deck.dealCard())`. ¿Qué opina respecto a lo directo de esto?

Se deja que el lector termine este programa. En dos ejercicios y un proyecto al final del capítulo se sugieren varias elaboraciones.

```
public void playAGame()
{
  deck.shuffle();

  // Reparte todas las cartas a los dos jugadores.
  while (deck.getCurrentSize() > 0)
  {
    player1.addCard(deck.dealCard());
    player2.addCard(deck.dealCard());
  }
  ...

} // end playAGame
```

Figura 12.21 Implementación parcial para el método `Game` de la clase `PlayAGame`.

12.11 Resolución de problema con clases de asociación (opcional)

Agregación, composición y herencia implementan algunos de los tipos más comunes de asociaciones entre clases y objetos: una asociación tiene-un para agregación y composición, y una asociación es-un para herencia. Esté alerta porque hay muchos otros tipos posibles de asociaciones, que puede conjurar fácilmente al garabatear unas cuantas frases como: "sea el próximo a. . .", "obtener . . . de. . .", "establecer. . . en. . .,", "hacer. . . con. . .,", "correr . . . hacia. . .", "vender. . . a. . .," y así sucesivamente. Típicamente, estos otros tipos de asociaciones son más complicadas que las asociaciones es-un o tiene-un. En esa sección se describe una forma poderosa para modelar otras asociaciones.

Como se ha visto, es posible implementar asociaciones simples de agregación y composición al proporcionar al objeto contenedor una referencia a cada objeto componente. Esta referencia permite que el código del objeto contenedor invoque métodos de los objetos componentes. Pero para otros tipos de asociaciones, quizá se requieran múltiples referencias y variables y métodos adicionales. En otras palabras, puede ser necesaria una clase por separado sólo para describir la asociación. Una clase así se denomina *clase de asociación*, y define un objeto asociación que representa una relación entre otros objetos. Un objeto asociación es como un contenedor agregación/composición, en cuanto a que tiene variables de instancia que refieren a otros objetos. Pero es diferente en que el objeto al que refiere también lo refiere a él, y ninguno puede contener al otro. Un objeto asociación típicamente recibe referencias a los objetos que asocia cuando es construido. Mientras un contenedor agregación/composición contiene sus objetos componentes, un objeto asociación simplemente "sabe sobre" los objetos que asocia.

Ahora se verá cómo esto podría aplicarse al programa anterior Dealership. Lo que se ha hecho hasta ahora con ese programa no es mucho de lo cual jactarse. Se creó una compañía con un gerente de ventas, algunos vendedores y algunos automóviles. Pero, ¿qué hay respecto a los clientes? ¿Sobre las ventas? Suponga que a ese programa se agrega una clase Customer. Luego suponga que algún vendedor inteligente finalmente realiza una venta con ese primer cliente. La siguiente pregunta es: ¿dónde debe colocarse la información sobre esa venta? ¿En la clase `Dealership`? ¿En la clase `SalesPerson` (como parece que se hace en la figura 12.6)? ¿En la clase `Car`? ¿En la clase `Customer`? Técnicamente esta información puede colocarse en cualquiera de estas clases, y luego colocar las referencias a esa clase en cualquiera de las clases que requieren acceder a esa información. También es posible repartir de alguna forma la información entre las clases participantes. No importa cuál de estas alternativas se elija; no obstante, desde ciertos puntos de vista, lo que se hizo podría parecer inadecuado.

Una solución más elegante consiste en encapsular toda la información de la venta en una clase de asociación, y asignar a esa clase un nombre que describa la asociación. Esto es lo que se representa en la figura 12.22, que muestra un diagrama de clase abreviado de otra versión del programa Dealership previo. Primero, observe la clase `Customer`. Puesto que un cliente es una persona, así como el gerente de ventas y los vendedores, es posible usar herencia para reducir el código y evitar redundancias en la clase `Customer` al hacer que esta clase extienda la clase `Persona`. Segundo, observe la clase `Sale`. Esta clase apa-

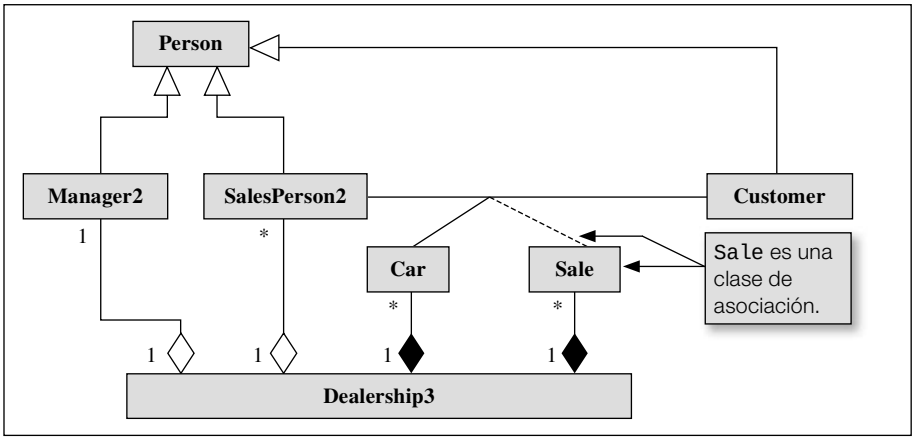


Figura 12.22 Diagrama de clase para otro programa car dealership con clientes y una asociación salesperson-car-customer.

```
// Esta clase asocia las clases SalesPerson2, Car y Customer.

public class Sale
{
  private Car car;
  private SalesPerson2 salesperson;
  private Customer customer;
  private double price;
  ...

  //*************************************************************

  public Sale(Car car, SalesPerson2 person,
    Customer customer, double price)
  {
    this.car = car;
    this.salesperson = person;
    this.customer = customer;
    this.price = price;
    ...
  } // end constructor
  ...
```

referencias a clases que están siendo asociadas

referencias pasadas al constructor

Figura 12.23 Implementación parcial de la clase `Sale` mostrada en la figura 12.22.

rece justo como otra componente en el diagrama de clase `Dealership`. La multiplicidad uno a muchos sugiere que sus objetos son elementos de una `ArrayList`, tal vez nombrada `sales`, que es instanciada en una versión mejorada del constructor `Dealership`. En cuanto al concesionario, una `Sale` podría ser justo otro tipo de componente de agregación o asociación, como una `SalesPerson2` o `Car`.

No obstante, una venta no es un ente físico, como lo son un organismo o un automóvil. Es un proceso —o evento— que asocia a un grupo de entes. Así, la clase `Sale` debe ser una clase de asociación. ¿Qué tipos de objetos participan en una asociación `Sale`? Hay un `Car`, una `SalesPerson2` y un `Customer`. Observe cómo el diagrama de clase UML en la figura 12.22 usa líneas de asociación simples para interconectar todas las clases normales que participan en una asociación. El estándar UML sugiere formas para decorar estas líneas de asociación con símbolos y nomenclatura adicionales, pero sólo las líneas mostradas transportan el mensaje: la idea de una asociación entre objetos de las clases `Car`, `SalesPerson2` y `Customer`. La línea discontinua que une gráficamente la clase `Sale` con las líneas de asociación identifica la clase `Sale` como una clase de asociación que describe la asociación relacionada. El fragmento de código en la figura 12.23 ilustra el constructor `Sale`.

Salvedad: no intente heredar de un participante en una asociación

⚠ Quizás esté tentado a tratar de usar herencia para crear una clase de asociación, porque podría pensar que así obtendría "acceso libre" a por lo menos uno de los participantes en la asociación. No intente hacer eso. Todo lo que obtendría sería la capacidad de hacer un clon mejorado de uno de los objetos que quiere asociar, y tendría que copiar todos los detalles entre el clon y la cosa real: un desperdicio de esfuerzo. Trate una asociación como una agregación, con referencias a los objetos participantes pasados hacia el constructor de asociación.

Resumen

- Los lenguajes orientados a objetos ayudan a organizar las cosas y los conceptos en dos tipos básicos de jerarquías: una jerarquía tiene-un para componentes en una agregación o en una composición, y una jerarquía es-un para tipos en una herencia.

- Una jerarquía de agregación o composición existe cuando un objeto grande contiene varios objetos más pequeños (componentes).
- Para una relación de clase todo-partes, si el contenedor contiene la única referencia a un componente, entonces el componente asociación es la composición. En caso contrario, se trata de agregación.
- En una jerarquía de herencia, las subclases heredan todas las variables y los métodos de las superclases por arriba de ellas, y suelen agregar más variables y métodos de los que heredan.
- A fin de minimizar duplicación descriptiva, las ideas deben organizarse de modo que sólo los conceptos en la base de una jerarquía de herencia (las hojas del árbol al revés) sean lo suficientemente específicos para representar objetos reales.
- Para permitir que la clase *B* herede todas las variables y métodos en la clase *A* y de todos los ancestros de la clase *A*, es necesario agregar `extends` *A* al final del encabezado de la clase *B*.
- Un constructor debe iniciar las variables que hereda al llamar de inmediato al constructor de la superclase con la declaración: `super`(<*arguments*>)`;`.
- Un método heredado puede ser sobrepuesto al escribir una versión diferente del método heredado en la clase derivada. La sobreposición ocurre automáticamente si se usan el mismo nombre del método y la misma secuencia de tipos de parámetro, pero si se hace esto, también es necesario usar el mismo tipo `return`.
- Es posible acceder a un método sobrepuesto al prefijar el nombre del método común con `super` y luego un punto.
- Un modificador de acceso `final` en un método evita que ese método sea sobrepuesto. Un modificador de acceso `final` en una clase evita que esa clase sea extendida.
- Los programadores utilizan a menudo combinaciones de agregación, composición y herencia para tratar diferentes aspectos de un problema de programación global. En un diagrama de clase UML, ambas relaciones se representan mediante líneas continuas entre clases relacionadas, y estas líneas se denominan asociaciones. En una asociación de composición/agregación hay un diamante lleno/hueco en el extremo del contenedor de cada línea de asociación. En una asociación jerárquica hay una punta de flecha hueca en el extremo de la superclase de la línea de asociación.
- La herencia permite reusar código que fue escrito para otro contexto.
- Cuando se tiene una asociación complicada entre objetos, puede ser de utilidad reunir referencias a esos objetos en una clase de asociación común.

Preguntas de revisión

§12.2 Composición y agregación

1. En un diagrama UML, ¿qué indica un asterisco (*)?

2. En un diagrama UML, ¿qué indica un diamante lleno?

§12.3 Revisión de herencia

3. Explique cómo el uso de la jerarquía de herencia conduce a reutilización del código.

4. Escriba dos sinónimos de superclase.

5. Escriba dos sinónimos de subclase.

§12.4 Implementación de jerarquía `Persona/Empleado/TiempoCompleto`

6. ¿Cómo indicaría a un compilador que una clase particular es derivada de otra clase?

7. Con base en el diagrama UML en la figura 12.9, una instancia de la clase `TiempoParcial` incluye las siguientes variables de instancia: `name` e `id`. (F/C)

§12.5 Constructores en una subclase

8. En el constructor de una subclase, ¿qué debe hacerse si se desea empezar el constructor con una llamada al constructor de parámetro cero de la superclase?

§12.6 Sobreposicion de métodos

9. Si una superclase y una subclase definen métodos que tienen el mismo nombre y la misma secuencia de tipos de parámetros, y un objeto de la subclase llama al método sin especificar la versión, Java genera un error de ejecución. (F/C)

10. Si el método de una subclase se sobrepone a un método en la superclase, ¿sigue siendo posible llamar al método de la superclase desde la subclase?

11. Si una superclase declara una variable como `private`, ¿es posible acceder a ella directamente desde una subclase?

§12.7 Utilización de jerarquía `Persona/Empleado/TiempoCompleto`

12. Si se desea llamar a un método de una superclase, siempre es necesario prefijar el nombre del método con `super`. (F/C)

12.8 El modificador de acceso `final`

13. Un método `final` se denomina así porque se le permite contener sólo constantes nombradas, no variables normales. (F/C)

§12.9 Utilización de herencia con agregación y composición

14. La composición y la herencia son técnicas de programación alternativas para representar lo que esencialmente es el mismo tipo de relación en el mundo real. (F/C)

§12.10 Práctica de diseño con un ejemplo de juego de cartas

15. Un mazo es un grupo de cartas y un mazo tiene un grupo de cartas. En nuestro ejemplo, resulta mejor elegir la relación es-un e implementar la herencia. En este caso, ¿por qué la herencia es una mejor opción que la composición?

§12.11 Resolución de problema con clases de asociación (opcional)

16. Es posible soportar una asociación con referencias, variables y métodos en clases existentes. ¿Cuál es la ventaja de utilizar una clase de asociación en lugar de lo anterior?

Ejercicios

1. [Después de §12.2] (Este ejercicio debe usarse en combinación con los ejercicios 2 y 3.) Escriba una definición de una clase `Point`. Proporcione dos variables de instancia `double`, `x` y `y`. Proporcione un constructor de dos parámetros que inicie a `x` y `y`. Proporcione un método `shiftRight` que desplace el punto en la dirección `x` por el valor del parámetro `double` del método `shiftAmount`. Proporcione un método `shiftUp` que desplace el punto en la dirección *y* por el valor del parámetro `double` del método `shiftAmount`. Haga que cada uno de estos métodos devuelva valores que permitan encadenamiento. Proporcione métodos de acceso para recuperar los valores de las dos variables de instancia.

2. [Después de §12.2] (Este ejercicio debe usarse en combinación con los ejercicios 1 y 3.) Escriba una definición de una clase `Rectangle`. Proporcione dos variables de instancia `Point`, `topLeft` y `bottomRight`, que establezcan los vértices superior izquierdo e inferior derecho del rectángulo, respectivamente. Proporcione un constructor de dos parámetros que inicie `topLeft` y `bottomRight`. Proporcione un método `shiftRight` que desplace el rectángulo en la dirección `x` por el valor del parámetro `double` del método `shiftAmount`. Proporcione un método `shiftUp` que desplace el rectángulo en la dirección `y` por el valor del parámetro `double` del método `shiftAmount`. Haga que cada uno de estos métodos devuelva valores que permitan encadenamiento. Proporcione un método `printCenter` que muestre los valores `x` y `y` del centro del rectángulo.

3. [Después de §12.2] (Este ejercicio debe usarse en combinación con los ejercicios 1 y 2.) Escriba una definición para una clase `RectangleDriver` con un método `main` que haga lo siguiente: instancie un `Point` denominado `topLeft` en `x` = −3.0 y `y` = 1.0. Instancie un `Point` denominado `bottomRigth` en `x` = 3.0 y `y` = −1.0. Instancie un `Rectangle` denominado `rectangle` usando `topLeft` y `bottomRigth` como argumentos. Llame al método `rectangle` de `printCenter`. Use una sola declaración encadenada para desplazar a la derecha el rectángulo por uno y luego lo suba por uno. Llame nuevamente al método de Instancie un `Rectangle` denominado `rectangle` usando `topLeft` y `bottomRigth` como argumentos. Llame nuevamente al método `rectangle` de `printCenter`. La salida debe ser:

```
x = 0.0 y = 0.0
x = 1.0 y = 1.0
```

4. [Después de §12.3] Suponga que se tienen tres clases: `Shape` (que define la posición de una forma en un sistema de coordenadas), `Square` (que define la posición de un cuadrado en un sistema de coordenadas más

el ancho del cuadrado) y `Circle` (que define la posición de un círculo en un sistema de coordenadas más el radio del círculo). Suponga que las tres clases forman una jerarquía de herencia idónea con dos relaciones de herencia. Para cada una de las dos relaciones de herencia, especifique la superclase y la subclase.

5. [Después de §12.3] Suponga que se desea crear una descripción por computadora de varios tipos de fuentes de energía, incluyendo estas cuatro clases: `Electrical`, `EnergySource`, `Heat`, `Mechanical`, y las seis variables: `firstCost`, `fuelUsed`, `maxRevolutionsPerMinute`, `maxTemperature`, `power-Output`, `volts`. Decida qué clase debe tener cada variable, establezca relaciones de herencia y trace un diagrama de clase UML con nombres de clase, nombres de variables y flechas de herencia. (Puede omitir las especificaciones de tipo y los métodos.)

6. [Después de §12.4] Programa Elipse:
Las clases API de Java utilizan bastante la herencia. Por ejemplo, la documentación API de Sun de Java muestra que el paquete `java.awt.geom.Ellipse2D` tiene una clase denominada `Double` con estas variables de instancia:[3]

`double`	**`height`** Altura total de la `Ellipse2D`.
`double`	**`width`** Ancho total de esta `Ellipse2D`.
`double`	**`x`** Coordenada x del vértice superior izquierdo de esta `Ellipse2D.`
`double`	**`Y`** Coordenada y del vértice superior izquierdo de esta `Ellipse2D`.

Y tiene estos constructores:

`Double()` Construye una nueva `Ellipse2D`, iniciada en la ubicación (0, 0) y de tamaño (0, 0).
`Double``(double x, double y, double w, double h)` Construye e inicia una `Ellipse2D` a partir de las coordenadas especificadas.

Tiene modificadores (accessors) para las variables de instancia y un método de inicio, aunque eso es todo. Afortunadamente, esta clase `extends` una clase denominada `Ellipse2D`, que tiene varios otros métodos útiles, incluyendo:

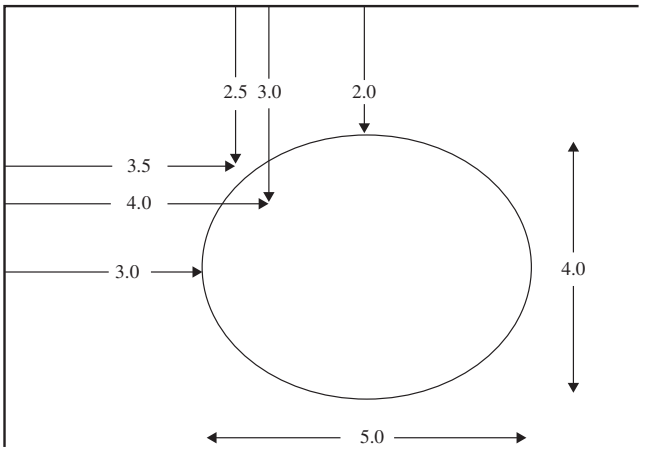
`boolean`	**`contains`**`(double x, double y)` Prueba si un punto especificado está dentro del límite de esta `Ellipse2D`.
`boolean`	**`contains`**`(double x, double y, double w, double h)` Prueba si el interior de esta `Ellipse2D` contiene por completo el área rectangular.
`boolean`	**`intersects`**`(double x, double y, double w, double h)` Prueba si el interior de esta `Ellipse2D` corta el interior de un área rectangular especificada.

Escriba un programa corto en una clase denominada `EllipseDriver`:

Importe `java.awt.geom.Ellipse2D.Double` y escriba un método `main` que llame al constructor `Double` de cuatro parámetros para instanciar una elipse como la que se muestra a continuación.[4] Luego, en las declaraciones `println`, llame al método `contains` de 2 parámetros de la superclase para mostrar si los puntos x=3.5, y=2.5 y x=4.0, y=3.0 están contenidos en la elipse especificada.

[3] Estas descripciones en los recuadros se copiaron del sitio API Sun de Java (http://java.sun.com/javase/6/docs/api/).

[4] El programa no dibuja realmente la elipse, pero la clase . . . `Ellipse2D` posee una comprensión matemática de ésta.



Salida:

```
contains x=3.5, y=2.5? false
contains x=4.0, y=3.0? true
```

7. [Después de §12.5] Defina una clase denominada `Circle` que se deriva de esta superclase API:

```
java.awt.geom.Ellipse2D.Double
```

Consulte el ejercicio 6 para ver una breve descripción de esta superclase `Double`. Su subclase debe declarar las dos variables de instancia `private`, `xCtr` y `yCtr`, iniciadas en `0.0`. Estas variables son las coordenadas `x` y `y` del centro de un círculo. Su clase debe incluir un constructor de parámetro cero, y también debe incluir un constructor de tres parámetros cuyos parámetros son las distancias `x` y `y` al centro del círculo y el diámetro del círculo. Este constructor de tres parámetros debe no sólo iniciar las nuevas variables de instancia sino también usar el constructor de cuatro parámetros de la superclase para iniciar las cuatro variables de instancia en la superclase. Su clase también debe proporcionar los siguientes métodos de acceso: `getXCtr`, `getYCtr` y `getRadius`, donde el radio es la mitad de la altura de la forma de la superclase. Compruebe el código que escribió compilándolo y ejecutándolo con valores representativos para `x` y `y` del centro y el diámetro.

8. [Después de §12.6] Suponga que se tienen dos clases relacionadas por herencia, y que ambas contienen un método de parámetro cero denominado `doIt`. Ésta es la versión de la subclase para `doIt`:

```
public void doIt()
{
  System.out.println("In subclass's doIt method.");
  doIt();
} // end doIt
```

La llamada `doIt();` es un intento por llamar a la versión de la superclase de `doIt`.

a) Describa el problema que ocurre cuando otro método llama al método anterior `doIt` y el método `doIt` se ejecuta.

b) ¿Cómo debe arreglarse el problema?

9. [Después de §12.8] ¿Qué significa que en un método se use el modificador `final`?

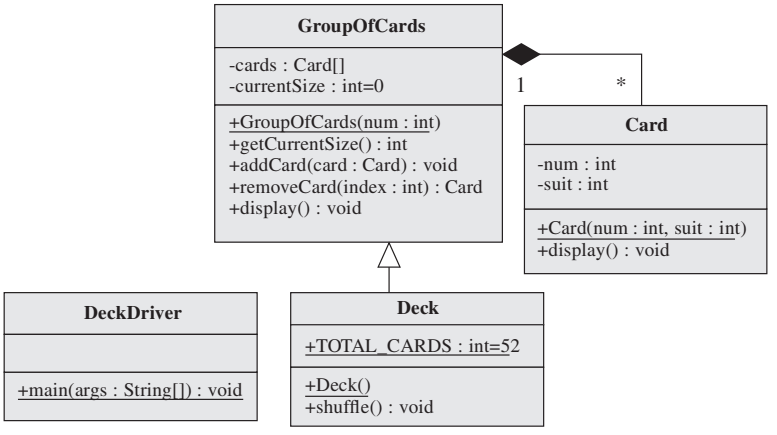
10. [Después de §12.8] ¿Qué significa que en una clase se use el modificador `final`?

11. [Después de §12.9] Llene los espacios en blanco:
Si la cosa A "tiene una" cosa B y "tiene una" cosa C, entonces hay una _____ asociación, y la definición de la clase A contiene declaraciones para variables de _____. Si A es una forma especial "es un" de B, hay una ___ asociación, y el lado derecho del encabezado de la clase A contiene las palabras _____.

12. [Después de §12.9] Identificación del tipo de asociación:
De la siguiente lista de pares de palabras, para cada par de palabras, identifique la asociación entre ambas palabras. Más específicamente, identifique si las dos palabras están relacionadas por composición o herencia. Para empezar, se han proporcionado las respuestas para los dos primeros pares de palabras. Bicicleta y rueda delantera están relacionadas por composición porque una bicicleta "tiene una" rueda delantera. Bicicleta y bicicleta de montaña están relacionadas por herencia porque una bicicleta de montaña "es una" bicicleta.

		¿herencia o composición?
bicicleta	rueda delantera	composición
bicicleta	bicicleta de montaña	herencia
elemento estructural	viga	_____
edificio	piso	_____
compañía	activos fijos	_____
empleado	vendedor	_____
bosque	árbol	_____
pájaro	petirrojo	_____
clase	método	_____
neurosis	paranoia	_____

13. [Después de §12.10] Barajar.
Suponga que está desarrollando el programa del juego de cartas sugerido en la figura 12.17 en el texto. El siguiente diagrama de clase UML parcial muestra dónde se encuentra el lector en el proceso de desarrollo:



Suponga que ya ha escrito métodos para una clase `Card` y una clase `GroupOfCards`. Suponga que el método `addCard` incrementa `currentSize` después de agregar la carta de entrada al final de la parte actualmente llena del arreglo `cards`. Suponga que el método `removeCard` recupera una referencia a la carta en `index` en el arreglo `cards`, disminuye `currentSize` del arreglo de cartas, desplaza un sitio hacia abajo todos los elementos del arreglo que están arriba de `index` y devuelve la referencia a la carta que originalmente estaba en `index`.

Para barajar el mazo, use un ciclo `for` que empiece en `unshuffled = getCurrentSize()` y se mueva hacia abajo una unidad. En cada iteración, use `Math.random` para escoger un índice en el intervalo no barajado, retire la carta en ese índice y luego agréguela al extremo alto del arreglo. Incluya toda esa funcionalidad en un método `Deck.shuffle`.

Crédito extra:
Escriba un código Java que pruebe su método `shuffle`. Para hacer lo anterior es necesario implementar todas las clases y los métodos en el diagrama de clase UML anterior. Su método `main` debe instanciar un `deck`, mostrarlo, barajarlo y volver a mostrarlo.

14. [Después de §12.10] Primera parte de la clase `Game` para el juego de Corazones:
La clase `Game` introducida en el texto contenía un mazo y exactamente dos jugadores. Mejore la clase `Game` mediante la inclusión de un arreglo de tipo `Trick[]` y una variable `numberOfTricks` que sigue la pista del número de manos jugadas hasta el momento. Incluya una variable de instancia `final` denominada `PLAYERS`, que será iniciada en un constructor cuyo valor de parámetro es el número de jugadores participantes en el juego. Sustituya las variables de instancia individuales de `player1` y `player2` por instancias en un arreglo de tipo `Hand[]`. Incluya dos variables de instancia `bolean`: `hearts` y `queenOfSpades`, cuyos valores cambien de `false` a `true` siempre que se juega el primer corazón o la reina de picas.

Escriba un código Java que defina la parte de la clase `Game` que incluye el encabezado de la clase, la declaración de la variable de instancia y el constructor a la medida de un parámetro cuyo parámetro es el número de jugadores. Este constructor debe instanciar un arreglo `Hand` con una longitud igual al número de jugadores. Debe instanciar objetos individuales `Hand` para cada jugador, usando un constructor `Hand` de dos parámetros. El primer parámetro es el número de jugador, empezando con 0. El segundo parámetro es el nú-

mero máximo de cartas que recibirá el jugador, que depende del número total de cartas en el mazo y del número de jugadores. El constructor del juego también debe instanciar un arreglo `Trick`, pero no poblarlo con ninguna mano individual.

Soluciones a las preguntas de revisión

1. Un * en un diagrama UML significa que la multiplicidad puede ser "cualquier número".
2. En un diagrama UML, un diamante lleno se coloca en una línea de asociación cerca de una clase contenedora en una asociación por composición. Indica que la clase contenedora exclusivamente contiene a la clase que se encuentra en el otro extremo de la línea de asociación.
3. Colocar código común de dos clases en una superclase es un ejemplo de reutilización del código. La reutilización del código también puede tener lugar cuando se desea agregar un trozo importante de funcionalidad a una clase existente y la solución se implementa con una nueva subclase.
4. Dos sinónimos para una superclase: clase padre, clase base.
5. Dos sinónimos para una subclase: clase hijo, clase derivada.
6. Para indicar al compilador que una clase es derivada de otra clase, se escribe `extends` <*other-class*> al final de su nueva línea superior de la clase.
7. Cierto. Una instancia de una subclase incluye las variables de instancia de esa clase y las variables de instancia de sus ancestros.
8. Nada. Esto ocurre automáticamente. No obstante, es posible adelantarse a esto al escribir `super();` como la primera línea en su constructor derivado.
9. Falso. No hay problema. La JVM selecciona el método en la subclase.
10. Sí. En la declaración de la llamada, anteponer el nombre del método común con `super`.
11. No, si una variable de instancia de una superclase es `private`, no es posible acceder a ella directamente desde una subclase. Puede accederse a ella llamando un método de acceso (en el supuesto que sea `public`). Al llamar al método de la superclase, no es necesario prefijar la llamada al método con una referencia punto.
12. Falso. El prefijo `super` sólo es necesario cuando se quiere llamar a un método de una superclase que ha sido sobrepuesto.
13. Falso. A un método `final` se le permite contener variables normales. Se denomina "final" porque es ilegal crear una versión de sobreposición del método en una subclase.
14. Falso. La composición y la herencia son relaciones de clase completamente diferentes. La composición es cuando una clase está compuesta por partes constituyentes no triviales y las partes se definen como clases. La herencia es cuando una clase es una versión más detallada de otra clase. En términos más formales, la herencia es cuando una clase, una subclase, hereda variables y métodos de otra clase, una superclase.
15. Porque con este ejemplo hay una segunda clase que también es un grupo de cartas. Puesto que hay dos clases que comparten algunas de las mismas propiedades, estas propiedades comunes deben colocarse en una superclase compartida, `GroupOfCards`. Al hacer lo anterior se promueve el reúso del software y se evita redundancia en el código.
16. Una asociación complicada puede hacerse más fácil de reconocer y comprender al organizar las referencias a todos los participantes en la asociación y otra información y métodos de la asociación en una sola clase que represente sólo a la asociación.

CAPÍTULO 13

Herencia y polimorfismo

Objetivos

- Comprender el papel de la clase `Objeto`.
- Aprender por qué es necesario redefinir los métodos `equals` y `toString`.
- Aprender cómo el polimorfismo y la vinculación dinámica mejoran la versatilidad del programa.
- Comprender lo que comprueba el compilador y lo que hace la JVM cuando una variable de referencia se asocia con el nombre de un método.
- Comprender las restricciones que afectan la asignación de un objeto de una clase a una variable de referencia de otra clase.
- Ver cómo usar un arreglo de variables de referencia ancestros para implementar polimorfismo entre métodos descendientes.
- Ver cómo una declaración de un método `abstracto` en una superclase `abstracta` elimina la necesidad de una definición de un método inválido en la superclase.
- Ver cómo puede usarse una interfaz para especificar encabezados de métodos comunes, almacenar constantes comunes e implementar múltiples polimorfismos.
- Aprender dónde usar acceso a miembros `protected`.
- Opcionalmente, aprender a dibujar un objeto en tres dimensiones.

Relación de temas

13.1 Introducción

Éste es el segundo de dos capítulos sobre herencia. En el capítulo previo se trataron ampliamente los conceptos fundamentales de la herencia. En este capítulo se reduce el enfoque y se describen a profundidad varios temas relacionados con la herencia. Se empieza con la clase `Objeto`, que es la superclase de todas las otras clases proporcionada por Sun. Luego se analiza una de las piedras angulares de la progra-

mación orientada a objetos (POO, OOP: object-oriented programming): el polimorfismo. El *polimorfismo* es la habilidad de que una llamada a un método particular ejecute varias operaciones en instantes diferentes. Esto ocurre cuando se tiene una variable de referencia que refiere a diferentes tipos de objetos durante el transcurso de la ejecución de un programa. Cuando la variable de referencia llama al método polimórfico, el tipo de objeto de la variable de referencia determina cuál método es llamado esa vez. Muy bien, ¿no? El polimorfismo proporciona programas con bastante poder y versatilidad.

Después de presentar el polimorfismo, se describe su correlato, la vinculación dinámica. La *vinculación dinámica* es el mecanismo usado por Java para implementar el polimorfismo. Luego, se proporcionan implementaciones alternativas del polimorfismo, usando clases e interfaces `abstractas` para hacer más limpio y aún más versátil el código. A continuación, se describe el modificador `protected`, que simplifica el acceso a código heredado. Por último, en una sección opcional, se presenta un problema de gráficas en tres dimensiones que ilustra el polimorfismo con el API de Java.

El material en este capítulo es relativamente difícil, pero una vez que el lector lo comprenda, verdaderamente entenderá de qué se trata la programación orientada a objetos y sabrá cómo tratar de manera elegante programas estructurados.

13.2 La clase `Objeto` y promoción automática de tipos

La clase `Objeto` es el ancestro de todas las otras clases. Es el ancestro primordial: la raíz de la jerarquía de la herencia. Cualquier clase que de manera explícita extienda una superclase usa `extends` en su definición. Siempre que alguien crea una nueva clase que no se extiende de manera explícita a alguna otra clase, el compilador automáticamente la hace extenderse a la clase `Objeto`. Por consiguiente, en última instancia todas las clases descienden de la clase `Objeto`. Esta clase no cuenta con muchos métodos, pero los pocos que tiene son importantes porque siempre son heredados por todas las otras clases. En las dos secciones siguientes se abordarán los dos métodos más importantes de la clase `Objeto`: `equals` y `toString`. Puesto que cualquier clase que se escriba incluye de manera automática estos dos métodos, es necesario estar al tanto de lo que ocurre cuando se llama a estos métodos.

Antes de entrar en los detalles de estos dos métodos, no obstante, queremos advertir al lector de un proceso en Java que es muy semejante a la promoción de tipo numérico que se estudió en los capítulos 3 y 11. Ahí se vio que en el transcurso de hacer una asignación o al copiar un argumento en un parámetro, la JVM (Java Virtual Machine) promueve automáticamente un tipo numérico, en el supuesto de que el cambio se ajuste a cierta jerarquía numérica. Por ejemplo, cuando un valor `int` se asigna a una variable `double`, la JVM promueve de manera automática el valor `int` a un valor `double`.

Una promoción automática semejante también ocurre con otros tipos. Cuando una operación de asignación o de pasar un argumento implica diferentes tipos de referencia, la JVM promueve de manera automática el tipo de referencia fuente al tipo de referencia objetivo si éste se encuentra por arriba del tipo de referencia fuente en la jerarquía de la herencia. En particular, puesto que la clase `Objeto` es un ancestro de cualquiera otra clase, cuando se presenta la necesidad, JVM promueve de manera automática cualquier tipo de clase al tipo `Objeto`. En la siguiente sección se describe una situación que estimula esta promoción de tipos.

13.3 El método `equals`

Sintaxis

El método `equals` de la clase `Objeto`, que es heredado de manera automática por todas las otras clases, tiene esta interfaz pública:

```
public boolean equals(Object obj)
```

Debido a que todas las clases heredan automáticamente este método, a menos que un método definido en forma semejante tenga prioridad, cualquier objeto, `objectA`, puede invocar este método para compararse a sí mismo con cualquier otro objeto, `objectB`, con una llamada al método como ésta:

```
objectA.equals(objectB)
```

Esta llamada al método devuelve un valor `boolean` que puede ser `true` o `false`. Observe que no se especificó el tipo de ninguno de los objetos `objectA` u `objectB`. En general, pueden ser instanciaciones de cualquier clase, y no necesariamente deben ser objetos de la misma clase. La única restricción es que el `objectA` debe ser una referencia no `null`. Por ejemplo, si existen las clases `Cat` y `Dog`, este código funciona correctamente:

```
Cat cat = new Cat();
Dog dog = new Dog();

System.out.println(cat.equals(dog));
```

Salida:
```
false
```

El método `equals` que se llama es el método `equals` que la clase `Cat` hereda automáticamente de la clase `Objeto`. El parámetro en este método heredado es del tipo `Objeto`, como se especifica en la interfaz pública del método anterior. Pero el argumento `Dog` que se pasa a este método no es del tipo `Objeto`. Es del tipo `Dog`. Entonces, ¿qué está ocurriendo? Cuando la referencia `Dog` se pasa al método heredado `equals`, el tipo de referencia automáticamente se promueve del tipo `Dog` al tipo `Objeto`. Luego, el método heredado `equals` ejecuta una prueba interna para ver si el `dog` pasado es el mismo que el `cat` que llama. Por supuesto, éste no es el caso, de modo que la salida es `false`, como puede verse.

Semántica

Observe que se acaba de mencionar "ejecuta una prueba interna". A continuación se abordará esa misteriosa "prueba interna". ¿Cómo puede decirse si dos objetos son el mismo o "iguales"? Cuando se dice "el `objetoA` es igual al `objetoB`", puede significar lo siguiente:

1. `objetoA` es simplemente otro nombre de `objetoB`, y tanto `objetoA` como `objetoB` se refieren exactamente al mismo objeto.

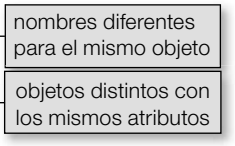
O puede significar lo siguiente:

2. `objetoA` y `objetoB` son dos objetos por separado que poseen los mismos atributos.

El método `equals` que todas las clases heredan de la clase `Objeto` implementa el significado más reducido posible de la palabra "igual". Es decir, este método devuelve `true` si y sólo si `objetoA` y `objetoB` se refieren exactamente al mismo objeto (definición 1 anterior). Este significado de "igual" es exactamente el mismo que el significado asociado con el operador == cuando se emplea para probar la igualdad de dos variables de referencia. Este operador también devuelve `true` si y sólo si ambas referencias se refieren exactamente al mismo objeto.

Suponga que se tiene una clase `Car` con tres variables de instancia: `make`, `year` y `color`, y que se tiene un constructor que inicia estas variables de instancia con valores de argumentos correspondientes. Suponga que esta clase `Car` no define un método `equals` en sí, y que el único método `equals` que hereda es el único que hereda automáticamente de la clase `Objeto`. El código siguiente ilustra que el método `equals` heredado de la clase `Objeto` hace exactamente lo mismo que hace el operador ==.

```
Car car1 = new Car("Honda", 2008, "rojo");
Car car2 = car1;
Car car3 = new Car("Honda", 2008, "rojo");

System.out.println(car2 == car1);
System.out.println(car2.equals(car1));
System.out.println(car3 == car1);
System.out.println(car3.equals(car1));
```



Salida:
```
true
true
false
false
```

Este significado reducido de la palabra "igual" no siempre es lo que se desea. Por ejemplo, suponga que su esposa decide comprar un automóvil nuevo y acude a un concesionario particular y ordena un Honda 2008 rojo, como se sugiere en la instanciación `car1` anterior. Cuando usted ve los folletos que su esposa trajo a casa, queda impresionado y decide que también quiere un automóvil nuevo. Le gustaría que fuese como el de su esposa, excepto por el color, que quiere que sea azul. De modo que se dirige al mismo concesionario y dice "quiero el mismo automóvil que mi esposa acaba de ordenar, pero lo quiero azul". Un mes después, el concesionario llama a ambos, a usted y su esposa, a sus respectivos sitios de trabajo diciendo "su automóvil está listo. Por favor, venga a recogerlo esta tarde a las 5:30 pm". Usted y su esposa se presentan a la hora indicada, y el concesionario lo lleva a usted fuera de la agencia y le dice con orgullo: "Aquí está. ¿Qué le parece?" Usted dice "¡Perfecto, es justo lo que quería!" Luego, su esposa dice "Pero ¿dónde está mi automóvil?" Y el concesionario responde: "Pero es que pensé que ambos iban a ser copropietarios del mismo automóvil, y su esposo me dijo que cambiara el color del automóvil a azul." ¡Ups! Alguien cometió un error. . .

El error ocurrió en la comunicación entre usted y el concesionario cuando usted dijo "el mismo automóvil". Aquí se refirió al segundo significado mencionado antes: `objetoA` y `objetoB` son dos objetos por separado que poseen los mismos atributos. Pero el concesionario escuchó el primer significado antes mencionado: `objetoA` es simplemente otro nombre de `objetoB`, y tanto `objetoA` como `objetoB` se refieren exactamente al mismo objeto.

Definición de su propio método `equals`

A continuación se verá cómo es posible implementar el segundo significado. Para ello, incluya en su clase una versión explícita de un método `equals` que pruebe atributos iguales. Luego, al ejecutar su programa, y cuando una instancia de su clase llame al método `equals`, su método `equals` tiene prioridad sobre el método `equals` de `Objeto`, y la JVM utiliza el método `equals` definido por usted. El método `equals` en la clase `Car` en la figura 13.1 prueba atributos iguales al comparar los valores de las tres variables de instancia; es decir, los atributos del objeto. Devuelve `true` sólo si las tres variables de instancia tienen los mismos valores, y en caso contrario devuelve `false`. Observe que este método `equals` incluye dos llamadas subordinadas al método `equals`: una hecha por la variable de instancia `make` y la otra hecha por la variable de instancia `color`. Como se explicó en el capítulo 3, estas llamadas al método `equals` de `String` verifican si dos cadenas distintas tienen la misma secuencia de caracteres.

En la expresión `return` del método `equals`, observe la subexpresión `otherCar != null`. Si esto se evalúa en `false` (indicando que `otherCar` es `null`), la evaluación de cortocircuito de Java impide que la computadora intente utilizar una referencia `null` para acceder a las otras variables de referencia `make` y `color` del automóvil. Esta evaluación de cortocircuito impide errores de tiempo de ejecución. El lector debe esforzarse siempre por robustecer su código. En este caso, lo anterior significa que debe considerar la posibilidad de que alguien pase un valor `null` para `otherCar`. Si `null` logra pasar y no hay prueba para `null`, la JVM genera un error de tiempo de ejecución cuando ve `otherCar.make`. Éste es un error bastante común: intentar acceder a un miembro desde una variable de referencia `null`, y es fácil evitarlo. Simplemente pruebe `null` antes de acceder al miembro. Para el método `equals`, si `otherCar` es `null`, entonces la subexpresión `otherCar != null` es `false`, y la declaración `return` devuelve `false`. Devolver `false` es inapropiado porque resulta evidente que un `otherCar` `null` no es lo mismo que el objeto `Car` que llama.

Adquiera la costumbre de escribir métodos `equals` para la mayor parte de sus clases definidas por el programador. Escribir métodos `equals` suele ser directo puesto que tienden a verse igual. Siéntase con confianza para usar como plantilla el método `equals` de la clase `Car`.

Recuerde que cualquier variable de referencia puede llamar al método `equals`, inclusive si la clase de la variable de referencia no define un método `equals`. El lector sabe qué ocurre en ese caso, ¿no es cierto? Cuando la JVM se da cuenta de que no hay método `equals` local, busca el método `equals` en una clase ancestro. Si no encuentra un método `equals` antes de llegar a la clase `Objeto` en la parte superior del árbol, usa el método `equals` de la clase `Objeto`. Esta operación por defecto a menudo aparece como un error. Para arreglar el error, asegúrese de que sus clases implementan sus propios métodos `equals`.

```
/*********************************************
* Car.java
* Dean & Dean
*
* Esto define y compara automóviles.
*********************************************/

public class Car
{
  private String make;  // modelo del automóvil
  private int year;     // año de lista del automóvil
  private String color; // color del automóvil

  //*******************************************

  public Car(String make, int year, String color)
  {
    this.make = make;
    this.year = year;
    this.color = color;
  } // end Car constructor

  //*******************************************

  public boolean equals(Car otherCar)
  {
    return otherCar != null &&
           make.equals(otherCar.make) &&
           year == otherCar.year &&
           color.equals(otherCar.color);
  } // end equals
} // end class Car
```

Esto sobrepone el método `equals` de la clase `Objeto`.

Figura 13.1 Clase `Car` que define `equals` para significar los mismos valores de la variable de instancia.

Métodos `equals` en clases API

Observe que los métodos `equals` están construidos en muchas clases API.[1] Por ejemplo, la clase `String` y las clases envoltorio implementan métodos `equals`. Como es de esperar, estos métodos `equals` prueban si dos referencias apuntan a datos que son idénticos (no si dos referencias apuntan al mismo objeto).

Ya ha visto antes el método `equals` de la clase `String`, de modo que el siguiente ejemplo debe ser bastante directo. Ilustra la diferencia entre el operador `==` y el método `equals` de la clase `String`. ¿Qué imprime este fragmento de código?

```
String s1 = "hola";
String s2 = "ho";

s2 += "la";
if (s1 == s2)
{
  System.out.println("mismo objeto");
}
```

[1] Para tener una idea de cuán comunes son los métodos `equals`, consulte el sitio API Sun de Java (http://java.sun.com/javase/6/docs/api/) y busque todas las ocurrencias de `equals`.

```
if (s1.equals(s2))
{
  System.out.println("mismo contenido");
}
```

El fragmento de código anterior imprime "mismo contenido". Vamos a asegurarnos de que el lector entiende por qué. El operador == devuelve `true` sólo si las dos variables de referencia que se están comparando se refieren al mismo objeto. En la primera declaración `if`, `s1 == s2` devuelve `false` puesto que `s1` y `s2` no se refieren al mismo objeto. En la segunda declaración `if`, `s1.equals(s2)` devuelve `true` puesto que los caracteres en las dos cadenas que se están comparando son los mismos.

En realidad, hay otro giro para la clase `String`. Para minimizar requerimientos de almacenamiento, el compilador de Java hace que las referencias a `String` se refieran al mismo objeto `String` siempre que una asignación se refiere a una literal repetida de la cadena. Esto se denomina *string pooling*. Por ejemplo, suponga que el código anterior incluye una tercera declaración como ésta:

```
String s3 = "hello";
```

Entonces, si la condición `if` fuese (`s1 == s3`), la salida diría "mismo objeto", porque `s1` y `s3` se referirían al mismo objeto cadena "hola".

13.4 El método `toString`

El método `toString` de la clase `Objeto`

A continuación se considerará otro método importante que todas las clases heredan de la clase `Objeto`. El método `toString` de la clase `Objeto` devuelve una cadena que es una concatenación del nombre completo de la clase del objeto que llama, un signo @, y una secuencia de dígitos y literales. Por ejemplo, considere este fragmento de código:

```
Object obj = new Object();
Car car = new Car();

System.out.println(obj.toString());
System.out.println(car.toString());
```

Cuando se ejecuta el fragmento de código, se obtiene

```
java.lang.Object@601BB1
Car@1BA34F2
```

nombre completo de la clase

Estos dígitos y literales forman un código hash.

Observe cómo `obj.toString()` genera `java.lang.Object` para el nombre completo de la clase. El nombre completo de la clase consta del nombre de la clase prefijado por el paquete del cual forma parte la clase. La clase `Objeto` es un paquete `java.lang`, de modo que su nombre completo de clase es `java.lang.Object`. Observe cómo `car.toString()` genera `Car` para el nombre completo de la clase. Puesto que la clase `Car` no forma parte de un paquete, su nombre completo de clase es simplemente `Car`.

Observe cómo `obj.toString()` genera 601BB1 para su valor en *código hash*. El valor de código hash de un objeto puede entenderse como su ubicación en la memoria, aunque en realidad es algo más complicado que eso. La JVM traduce el valor de código hash de un objeto a uno o más valores, y el último valor en la cadena de traducción especifica la ubicación real del objeto en la memoria. En Java, los valores de código hash, como 601BB1, se escriben como números hexadecimales. El sistema de numeración hexadecimal se describió en la sección opcional Unicode al final del capítulo 11. Lo que sigue es un repaso.

Números hexadecimales

Los números hexadecimales usan dígitos que pueden asumir uno de 16 valores: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F (las minúsculas de la a a la f también son aceptables). Los valores de la A a la F representan los números del 10 al 15. Con 16 dígitos únicos, los números hexadecimales constituyen lo que se

conoce como sistema de numeración base 16. Con el sistema decimal al que está acostumbrado el lector, base 10 para los números decimales, suponga que se está contando y que se llega al dígito más grande, 9. Para formar el siguiente número, 10, se requieren dos dígitos: un 1 a la izquierda y un 0 a la derecha, de modo que el resultado es 10. En forma semejante, suponga que está contando con números hexadecimales y que ya llegó al dígito más grande, F por 15. Para formar el número siguiente, 16, se requieren dos dígitos: un 1 a la izquierda y un cero a la derecha, de modo que el resultado es 10. En otras palabras, 10 es como se escribe 16 en hexadecimal. Para ayuda adicional con la forma de contar en hexadecimal, consulte el apéndice 1. Ahí encontrará una secuencia de números hexadecimales y sus números decimales asociados, en el contexto del conjunto de caracteres Unicode/ASCII.

Se sabe que el número hexadecimal A es equivalente al número decimal 10. ¿Y qué hay acerca del valor 601BB1 generado por el fragmento de código anterior? ¿Cuál es su número decimal equivalente? La conversión de números hexadecimales grandes a sus equivalentes decimales puede hacerse matemáticamente, aunque se presentará un atajo. Si el lector trabaja en una computadora basada en Windows, seleccione Start/Programs/Accesories/Calculator. En la ventana de cálculo, haga clic en el botón Hex. Introduzca 601BB1 y luego haga clic en el botón Dec. La calculadora muestra 6298545, que es el número decimal equivalente a 601BB1. Así, en el fragmento de código anterior, cuando `obj.toString()` devuelve una cadena con 601BB1 a la derecha del signo @, esto significa que la ubicación del objeto `obj` en la memoria puede encontrarse yendo a la 6 298 545-ésima posición en la *tabla hash* del objeto. La tabla hash del objeto es el ente responsable de traducir los valores en código hash en ubicaciones reales en la memoria.

Sobreposición en el método `toString`

Recuperar el nombre de la clase, un signo @ y un código hash usualmente no tiene ningún valor, de modo que casi siempre se desea evitar llamar al método `toString` de la clase `Objeto` y en lugar de eso llamar a un método `toString` de sobreposición. La razón por la que se está analizando el método `toString` de la clase `Objeto` es porque resulta fácil llamarlo accidentalmente, y cuando esto ocurre, es aconsejable saber qué está ocurriendo.

Puesto que la clase `Objeto` define un método `toString`, toda clase tiene un método `toString`, inclusive si no define o hereda uno a través de una clase que extiende en forma explícita. Muchas clases API de Java definen métodos de sobreposición `toString`. Por ejemplo, el método `toString` de la clase `String` devuelve trivialmente la cadena que está almacenada en el objeto `String`. Como se describió en el capítulo 10, el método `toString` de la clase `ArrayList` (heredado de la clase `AbstractCollection`) devuelve una lista de cadenas escrita entre corchetes y separadas por comas que representan los elementos individuales del arreglo. El método `toString` de la clase `Date` devuelve los valores del mes, día, año y segundos de un objeto `Date` como una cadena simplemente concatenada. En general, los métodos `toString` deben devolver una cadena que describa el contenido del objeto que llama.

Puesto que la recuperación del contenido de un objeto es una necesidad bastante común, es necesario adquirir la costumbre de proporcionar un método `toString` explícito para la mayor parte de las clases definidas por el programador. Típicamente, los métodos `toString` deben simplemente concatenar los datos almacenados del objeto que llama y devolver la cadena resultante. Los métodos `toString` no deben imprimir el valor de la cadena concatenada; simplemente deben devolverla. Esta cuestión se menciona porque los programadores novatos tienen la tendencia a colocar declaraciones de impresión en sus métodos `toString`, lo cual es erróneo. Un método sólo debe hacer lo que se supone que debe hacer y nada más. Se supone que el método `toString` devuelve el valor de una cadena, ¡y eso es todo!

Por ejemplo, observe el método `toString` en el programa Car2 en la figura 13.2. Devuelve una cadena que describe el contenido del objeto que llama.

Llamadas implícitas al método `toString`

En el programa Car2, el método `main` no tiene llamada explícita al método `toString`. Así, ¿cómo ilustra este programa el uso del método `toString`? Siempre que una referencia aparece sola dentro de una declaración de impresión (`System.out.print` o `system.out.println`), la JVM automáticamente llama al método `toString` del objeto referido. En la figura 13.2, esta declaración genera una llamada al método `toString` en la clase `Car2`:

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

```
/*****************************************************
* Car2.java
* Dean & Dean
*
* Esto instancia un automóvil y muestra sus propiedades.
*****************************************************/

public class Car2
{
  private String make;  // modelo del automóvil
  private int year;     // año de lista del automóvil
  private String color; // color del automóvil

  //***************************************************

  public Car2(String make, int year, String color)
  {
    this.make = make;
    this.year = year;
    this.color = color;
  } // end Car2 constructor

  //***************************************************

  public String toString()
  {
    return "make = " + make + ", year = " + year +
      ", color = " + color;
  } // end toString

  //***************************************************

  public static void main(String[] args)
  {
    Car2 car = new Car2("Honda", 1998, "silver");
    System.out.println(car);
  } // end main
} // end class Car2
```

Esto sobrepone el método `toString` de la clase `Objeto`.

Figura 13.2 Programa Car2 que ilustra la sobreposición del método `toString`.

A continuación se considerará otro ejemplo que usa el método `toString`. Observe el programa Counter en la figura 13.3. De nuevo, hay un método `toString` y ninguna llamada explícita a éste. Entonces, ¿cómo fue llamado? Cuando se concatenan una variable de referencia y una cadena (con el operador +), la JVM llama automáticamente al método `toString` de la referencia. Así, en la figura 13.3, esta referencia `counter` de la declaración genera una llamada al método `toString` de la clase `Counter`:

```
String message = "Current count = " + counter;
```

Vea que a menudo se observa el método `toString` llamado explícitamente con la sintaxis de llamada estándar inclusive cuando no es necesario. Por ejemplo, en el método `main` del programa Counter, hubiera podido usarse esta implementación alternativa para la declaración de asignación `message`:

```
String message = "Current count = " + counter.toString();
```

Algunos programadores dirían que esta implementación alternativa es mejor porque el código es más autodocumentado. Otros programadores dirían que la implementación original es mejor porque el código

```
/*****************************************************
* Counter.java
* Dean & Dean
*
* Esto crea un contador y muestra su valor de conteo.
*****************************************************/

public class Counter
{
  private int count;

  //************************************************

  public Counter(int count)
  {
    this.count = count;
  } // end constructor

  //************************************************

  public String toString()
  {
    return Integer.toString(count);
  } // end toString

  //************************************************

  public static void main(String[] args)
  {
    Counter counter = new Counter(100);
    String message = "Current count = " + counter;
    System.out.println(message);
  } // end main
} // end class Counter
```

Esto sobrepone el método `toString` de la clase `Objeto`.

Figura 13.3 Programa Counter que ilustra la llamada implícita del método `toString`.

es más reducido. Aquí no se tiene ninguna preferencia sobre cuál implementación es mejor, cualquiera está bien.

Método `toString` del programa Counter: análisis detallado

A continuación volverá a visitarse el método `toString` del programa Counter en la figura 13.3. Puesto que la clase `Counter` contiene un solo dato, `count`, no es necesario concatenar el código como parte de la implementación de `toString`. Simplemente devuelve el valor de `count` y eso es todo. Por tanto, ésa pudo ser su implementación de primer corte para `toString`:

```
public int toString()
{
  return count;
}
```

Pero así se produce un error de tiempo de compilación. ¿Sabe por qué? Un método que se sobrepone debe tener el mismo tipo de retorno que el método al que se sobrepone. Puesto que el método `toString` de la clase `Counter` es una implementación de sobreposición del método `toString` de la clase `Objeto`, ambos métodos deben tener el mismo tipo de retorno. Debido a que el tipo de retorno de la clase `Objeto`

es un `String`, el tipo de retorno `int` anterior genera un error. Con eso en mente, ésta hubiera podido ser la implementación de segundo corte para `toString`:

```
public String toString()
{
  return count;
}
```

⚠ Pero esto también produce un error. ¿Por qué? Tipos incompatibles. El valor devuelto, `count`, es un `int`, y el tipo de retorno se definió como un `String`. La solución es convertir explícitamente `count` en un `String` antes de regresarlo, como esto:

```
public String toString()
{
  return Integer.toString(count);
}
```

¿Comprende el lector el código `Integer.toString`? En el capítulo 5 se aprendió que todos los tipos primitivos tienen una clase envoltorio correspondiente. `Integer` es una de estas clases: envuelve al primitivo `int`. El método `toString` de la clase `Integer` devuelve una representación de cadena de su argumento pasado `int`. De modo que si `count` es 23, entonces `Integer.toString(count)` devuelve la cadena "23".

Prueba rápida: el método `toString` de la clase `Integer`, ¿es un método de clase o un método de instancia? Observe el prefijo de la llamada al método. La llamada al método, `Integer.toString`, usa un nombre de clase para el prefijo. Cuando una llamada a un método usa un nombre de clase para un prefijo en lugar de una variable de referencia, se sabe que el método es un método de clase. Así, `Integer.toString` es un método de clase.

Observe que todas las clases envoltorio tienen métodos `toString`. Todos hacen lo mismo: devuelven una representación en cadena de su argumento pasado. He aquí algunos ejemplos:

```
Double.toString(123.45)  : evaluates to string "123.45"
Character.toString('G')  : evaluates to string "G"
```

Método `valueOf` de `String`

Hay otra forma de convertir variables primitivas en cadenas. Use el método `valueOf` de la cadena `String`. Éste asume un valor primitivo y devuelve una cadena. Así como los métodos envoltorio `toString` definidos antes, se trata de un método de clase, por lo que debe usarse su nombre de clase, `String`, como prefijo. Por tanto, en vez de las llamadas al método previas, pueden usarse estas llamadas al método:

```
String.valueOf(123.45)     : evaluates to string "123.45"
String.valueOf('G')        : evaluates to string "G"
```

El método `valueOf` es de utilidad si de antemano no se conoce el tipo de datos. Funciona con diferentes tipos de datos porque es un método sobrecargado, y la JVM selecciona automáticamente el método particular cuyo tipo de parámetro coincide con el tipo de los datos proporcionados.

El método `valueOf`, además de usarse para convertir primitivos en cadenas, también puede usarse para convertir un arreglo de caracteres vocales en una cadena. Este fragmento de código imprime la cadena "aeiou":

```
Char[] vowels = {'a','e', 'i', 'o', 'u'};
System.out.print{String.valueOf(vowels));
```

13.5 Polimorfismo y vinculación dinámica

Repaso de polimorfismo

Si se le pide a un aficionado a la programación orientada a objetos (POO) que mencione las tres características más importantes de la POO, probablemente responda "encapsulamiento, herencia y polimor-

```
/************************************
* Dog.java
* Dean & Dean
*
* Esta clase implementa un perro.
************************************/

public class Dog
{
  public String toString()
  {
    devuelve "¡Guau! ¡Guau!";
  }
} // end Dog class
```

Figura 13.4 Clase `Dog` para el programa Pets controlado por el código en la figura 13.6.

fismo". En el capítulo previo se analizaron el encapsulamiento y la herencia. Ahora es el momento de estudiar el *polimorfismo*. La etimología de la palabra polimorfismo proviene de la expresión en griego que significa "poseer muchas formas". En química y mineralogía, el polimorfismo es cuando una sustancia puede cristalizarse en dos o más formas alternativas. En zoología, el polimorfismo es cuando una especie tiene dos o más formas diferentes, como las diversas castas de abejas desovadas por la misma reina para desempeñar funciones diferentes en un panal. En computación, el polimorfismo es cuando diferentes tipos de objetos responden en forma distinta a la llamada a un mismo método.

He aquí cómo funciona. Se declara un tipo general de variable de referencia que sea capaz de referirse a objetos de diferentes tipos. ¿Cuál es el tipo más general de variable de referencia? Es una variable de referencia `Objeto` declarada, por ejemplo, como sigue:

```
Object obj;
```

Una vez que se ha declarado una variable de referencia de tipo `Objeto`, puede usarse para referirse a cualquier tipo de objeto. Por ejemplo, suponga que se define una clase nombrada `Dog`, como en la figura 13.4, y otra clase nombrada `Cat`, como en la figura 13.5. Cada una de las dos clases derivadas contiene un método `toString` que se sobrepone al método `toString` en la clase `Objeto`. Observe que los dos métodos `toString` mostrados se sobreponen al método `toString` de `Objeto` en muchas formas. Uno devuelve lo que ladra un perro: "¡Guau! ¡Guau!", y el otro devuelve lo que maúlla un gato: "¡Miau! ¡Miau!"

```
/************************************
* Cat.java
* Dean & Dean
*
* Esta clase implementa un gato.
************************************/

public class Cat
{
  public String toString()
  {
    devuelve "¡Miau! ¡Miau!";
  }
} // end Cat class
```

Figura 13.5 Clase `Cat` para el programa Pets controlado por el código en la figura 13.6.

```
/*********************************************************
* Pets.java
* Dean & Dean
*
* Esto ilustra polimorfismo simple.
*********************************************************/

import java.util.Scanner;

public class Pets
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Object obj;

    System.out.print("¿qué tipo de mascota prefiere?\n" +
      "introduzca d para perros o c para gatos: ");
    if (stdIn.next().equals("d"))
    {
      obj = new Dog();
    }
    else
    {
      obj = new Cat();
    }
    System.out.println(obj.toString());
    System.out.println(obj);
  } // end main
} // end Pets class
```

La variable de referencia `obj` puede contener una referencia ya sea a un objeto `Dog` o a un objeto `Cat`.

Ese objeto determina cuál versión del método `toString` se está llamando aquí.

Estas dos declaraciones son equivalentes.

Sesión muestra:

```
Which type of pet do you prefer?
Enter d for dogs or c for cats: d
Woof! Woof!
Woof! Woof!
```

Figura 13.6 Controlador para el programa Pets que incluye las clases en las figuras 13.4 y 13.5.

Las diversas definiciones del método `toString` en las clases `Dog` y `Cat` permiten que el método `toString` sea polimórfico. Si `toString` se llama con una referencia a un objeto `Dog`, responde de la forma en que respondería un perro, pero si `toString` se llama con una referencia a un objeto `Cat`, responde de la forma en que respondería un gato. El controlador en la figura 13.6 demuestra este efecto. Observe cómo la variable de referencia `obj` puede contener una referencia a un objeto `Dog` o a un objeto `Cat`, y ese objeto determina cuál método `toString` se llama.

¿Por qué el programa imprime dos veces "¡Guau! ¡Guau!"? Hay dos declaraciones de impresión. La primera llama explícitamente a un método `toString`. La segunda usa una llamada implícita a un método `toString`: cuando una variable de referencia aparece sola en un contexto `String`, el compilador agrega automáticamente `.toString()` a la variable de referencia sola. Así, las dos últimas declaraciones en la clase `Pets` son equivalentes.

Dinámica vinculante

Los términos polimorfismo y *dinámica vinculante* están estrechamente relacionados, aunque no son lo mismo. Resulta útil conocer la diferencia. El polimorfismo es una forma de comportamiento. La diná-

mica vinculante es el mecanismo de ese comportamiento: cómo se implementa. Específicamente, el polimorfismo es cuando diferentes tipos de objetos responden en forma diferente a la llamada exacta al mismo método. La dinámica vinculante es lo que la JVM hace para que coincida una llamada a un método polimórfico con un método particular.

Justo antes que la JVM ejecute una llamada a un método, determina el tipo del objeto real que llama al método. Si el objeto real que llama es de la clase `X`, la JVM *vincula* el método de la clase `X` con la llamada al método. Si el objeto real que llama es de la clase `Y`, la JVM *vincula* el método de la clase `Y` con la llamada al método. Después que la JVM vincula el método idóneo con la llamada al método, la JVM ejecuta el método vinculado. Por ejemplo, observe la llamada al método `obj.toString` en la siguiente declaración cerca de la base de la figura 13.6:

```
System.out.println(obj.toString());
```

En función de a qué tipo de objeto se hace referencia por `obj`, la JVM vincula ya sea el método `toString` de `Dog` o el método `toString` de `Cat` a la llamada al método `obj.toString`. Una vez que la vinculación se lleva a cabo, la JVM ejecuta el método vinculado e imprime "¡Guau! ¡Guau!" o "¡Miau! ¡Miau!"

La dinámica vinculante se denomina "dinámica" porque la JVM ejecuta la operación de vinculación mientras se está ejecutando el programa. La vinculación se lleva a cabo en el último momento posible, justo antes que se ejecute el método. Ésta es la razón de que algunas veces la dinámica vinculante se denomine *vinculación tardía*. Por cierto, algunos lenguajes de programación vinculan llamadas a métodos en tiempo de compilación, en lugar de hacerlo en tiempo de ejecución. Ese tipo de vinculación se denomina *vinculación estática*. Los diseñadores de Java se decidieron por la dinámica vinculante, y no por la vinculación estática, porque facilita el polimorfismo.

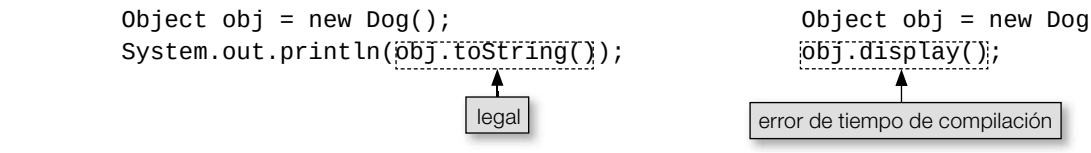
Detalles de compilación

En el programa Pets, el comportamiento polimórfico se ilustró al llamar versiones `Dog` y `Cat` del método `toString`. ¿Hubiera sido posible hacer lo mismo con las versiones `Dog` y `Cat` de un método `display`? En otras palabras, si `Dog` implementa un método `display` que imprime "Soy un perro", ¿funcionaría el código siguiente?

```
Object obj = new Dog();
obj.display());
```

Según el análisis realizado sobre la vinculación dinámica, el código debe funcionar bien. La JVM vería un objeto `Dog` en la variable de referencia `obj` y vincularía el método `display` de `Dog` con la llamada al método `obj.display`. Sin embargo, no importa que el código funcione bien en términos de la vinculación dinámica. El código no compila exitosamente porque el compilador siente que ahí puede haber un problema.

Cuando el compilador ve una llamada a un método, <*reference-variable*>.<*method-name*>(), comprueba para ver si la clase de la variable de referencia contiene una definición del método para el método llamado. Observe las llamadas a `obj.toString` y a `obj.display` en los siguientes ejemplos. En el primer ejemplo, el compilador comprueba para ver si la clase del `obj`, `Objeto`, contiene un método `toString`. La clase `Objeto` contiene un método `toString`, de modo que el código compila exitosamente. En el ejemplo de la derecha, el compilador comprueba para ver si la clase del `obj`, `Objeto`, contiene un método display. La clase `Objeto` no contiene un método `display`, de modo que el código produce un error de tiempo de compilación.



¡Espere un momento! ¿Esto significa que el polimorfismo sólo funciona para los métodos definidos en la clase `Objeto`? Afortunadamente no es así. Más adelante en este capítulo se aprenderá cómo hacer que el polimorfismo funcione para cualquier método.

El operador `instanceof`

Como se ha visto, siempre que una referencia genérica llama a un método polimórfico, la JVM usa el tipo del objeto referido para decidir cuál método llamar. Quizás el lector quiera hacer algo semejante en forma explícita en su código. En particular, podría tratar de ver si un objeto referido es una instancia de alguna clase particular. Esto puede hacerse con un operador especial denominado `instanceof` (observe que la "o" en `instanceof` es minúscula). Usando de nuevo el ejemplo Pets, suponga que se desea imprimir "Meneo de la cola" si el objeto de `obj` es una instancia de la clase `Dog` o cualquier clase descendiente de la clase `Dog`. Lo anterior puede hacerse con la declaración `if` en la parte inferior del método `main` en la figura 13.8. Así, el operador `instanceof` constituye una forma simple y directa para ordenar los diversos tipos de objeto a los que sería posible referirse por medio de una variable de referencia genérica.

13.6 Asignaciones entre clases en una jerarquía de clases

A continuación se considerará algo que es bastante común con los programas polimórficos: la asignación de un objeto a una referencia donde la clase del objeto y la clase de referencia son diferentes: en el siguiente fragmento de código, suponga que `Student` es una subclase de `Persona`. ¿Qué hace este fragmento de código?

```
Person p = new Student();
Student s = new Person();
```

Esto genera un error de tiempo de compilación.

La primera línea asigna un objeto `Student` (en realidad una referencia a un objeto `Student`) a una variable de referencia `Persona`. Está asignando un objeto de una subclase a una variable de referencia de una superclase. Esta asignación es legal porque un `Student` "es una" `Persona`. Asciende en la jerarquía de la herencia: la dirección en la cual ocurre la promoción automática de tipos. La segunda línea intenta asignar un objeto `Persona` a una variable de referencia `Student`. Está intentando asignar un objeto de una superclase a una variable de referencia de una subclase. Esto es ilegal porque una `Persona` no necesariamente es un `Student`. La segunda línea genera un error de tiempo de compilación.

El recurso mnemónico "es un" es de ayuda para recordar la regla, pero si el lector es un Curious George,[2] quizá quiera más. Tal vez quiera comprender el verdadero razonamiento detrás de la regla. De modo que aquí está: es correcto asignar un objeto de una clase descendiente a una variable de referencia de una clase ancestro porque todo el compilador se ocupa del hecho de si el objeto asignado a la clase descendiente tiene todos los miembros que debe tener cualquier objeto de la clase de la variable de referencia. Y si un objeto de una clase descendiente se asigna a una variable de referencia de una clase ancestro, lo hace. ¿Por qué? ¡Porque los objetos de la clase descendiente siempre heredan todos los miembros de la clase ancestro!

Así como ocurre con los primitivos, si hay compatibilidad, puede hacerse lo contrario usando un tipo cast. En otras palabras, puede usarse un tipo cast para obligar a que un objeto referido por una variable de referencia más genérica sea un tipo más específico: un tipo que esté abajo en la misma jerarquía de la herencia. Por ejemplo, si `p` es una variable de referencia `Persona` y `Student` hereda de `Persona`, el compilador acepta esto:

```
Student s = (Student) p;
```

Aunque el compilador acepta esta declaración, no necesariamente significa que el programa se ejecutará exitosamente. Para una ejecución exitosa, cuando ocurre vinculación dinámica, el objeto realmente referido por la variable de referencia `p` debe ser por lo menos tan específico como un `Student`. Es decir, el objeto referido debe ser una instancia de la clase `Student` o una instancia de un descendiente de la clase `Student`. ¿Por qué? Porque después de la asignación de la referencia a una variable de referencia `Student`, se espera que el objeto tenga todos los miembros que tiene un `Student`, lo cual en general es más que todos los miembros que tiene una `Persona`.

[2] Curious George es el personaje principal en una serie de libros escritos por Margret y H. A. Rey. George es un mono curioso. El niño del autor John, Caiden, es un aprendiz de Curious-George.

```
/*********************************************************
* Pets2.java
* Dean & Dean
*
* Esto ilustra el uso de un operador instanceof.
*********************************************************/

import java.util.Scanner;

public class Pets2
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Object obj;

    System.out.print("¿Qué tipo de mascota prefiere?\n" +
      "Introduzca d para perros o c para gatos: ");
    if (stdIn.next().equals("d"))
    {
      obj = new Dog();
    }
    else
    {
      obj = new Cat();
    }
    if (obj instanceof Dog)
    {
      System.out.println("Wag tail");
    }
  } // end main
} // end Pets2 class
```

Esta condición se evalúa en `true` si el objeto referido es una instancia de la clase `Dog` o una clase descendiente de la clase `Dog`.

Sesión muestra:

```
Which type of pet do you prefer?
Enter d for dogs or c for cats: d
Wag tail
```

Figura 13.7 Demostración del operador `instanceof`.

13.7 Polimorfismo con arreglos

Hasta el momento, el polimorfismo se ha visto en el contexto de fragmentos de código y de un programa Pets sencillo. Estos ejemplos cumplieron su función: ilustraron lo fundamental. Pero no ilustraron la verdadera utilidad del polimorfismo. La verdadera utilidad del polimorfismo surge cuando se tiene un arreglo o una `ArrayList` de variables de referencia genéricas y se asignan diferentes tipos de objetos a diferentes elementos. Esto permite entrar por el arreglo o la `ArrayList` y llamar a un método polimórfico para cada elemento. En tiempo de ejecución, la JVM usa vinculación dinámica para escoger el método particular que es válido para cada tipo de objeto encontrado.

Polimorfismo en una jerarquía de la herencia explícita

El programa Pets usó métodos polimórficos `toString` para las clases `Dog` y `Cat`. El compilador aceptó la variable de referencia `Object` con llamadas al método `toString` porque la clase `Object` define su propio método `toString`. Recuerde que el polimorfismo no funcionó para los métodos `display` `Dog` y `Cat` porque la clase `Object` no define su propio método `display`. Suponga que el método que se

desea hacer polimórfico no está definido en la clase `Object`. ¿Cómo puede tenerse un polimorfismo y aún así satisfacer al compilador? En realidad, hay varias formas relacionadas. Una consiste en crear una superclase para las clases que definen las diferentes versiones del método polimórfico, y definir el método dentro de la superclase. Luego, usar el nombre de esa superclase cuando se declara(n) la(s) variable(s) de referencia polimórfica(s). Otra forma de satisfacer el compilador consiste en *declarar el método* (especificar sólo el encabezado del método) en una clase ancestro `abstract` y luego usar el nombre de esa clase ancestro para el tipo de la variable de referencia. Otra forma más de satisfacer el compilador consiste en `implementar` una *interfaz* que declare el método y luego use el nombre de esa interfaz para el tipo de la variable de referencia. En esta sección se ilustrará el primer tipo y los otros dos, en secciones ulteriores.

Ejemplo de nómina

Para ilustrar el polimorfismo en una jerarquía de la herencia explícita se desarrollará un programa de nómina que usa vinculación dinámica a fin de seleccionar el método idóneo para calcular la paga de un empleado. Los empleados asalariados se vinculan automáticamente al método `getPaid` de la clase `Salaried`. Los empleados que trabajan por horas se vinculan automáticamente a un método `getPaid` de la clase `Hourly`.

Se empezará con el diagrama de clase UML en la figura 13.8. Describe la estructura de la clase del programa `Payroll`. Como puede verse, `Employee` es una superclase, y `Salaried` y `Hourly` son subclases. La cuarta clase, `Payroll`, es el controlador del programa. Su método `main` controla las clases `Salaried` y `Hourly` al instanciarlas y luego llamar a sus métodos. ¿Cuál es la asociación entre `Payroll` y las otras clases: herencia o composición/agregación? Los diamantes en el diagrama de clase UML indican una asociación de composición/agregación entre el contenedor `Payroll` y los componentes `Salaried` y `Hourly`. Esto debe tener sentido cuando uno se da cuenta de que la clase `Payroll` "tiene un" arreglo heterogéneo de objetos `Salaried` y `Hourly`. En el supuesto de que la clase `Payroll` tenga el control exclusivo sobre estos objetos, su asociación con ellos es una composición, por lo que los diamantes deben ser llenos.

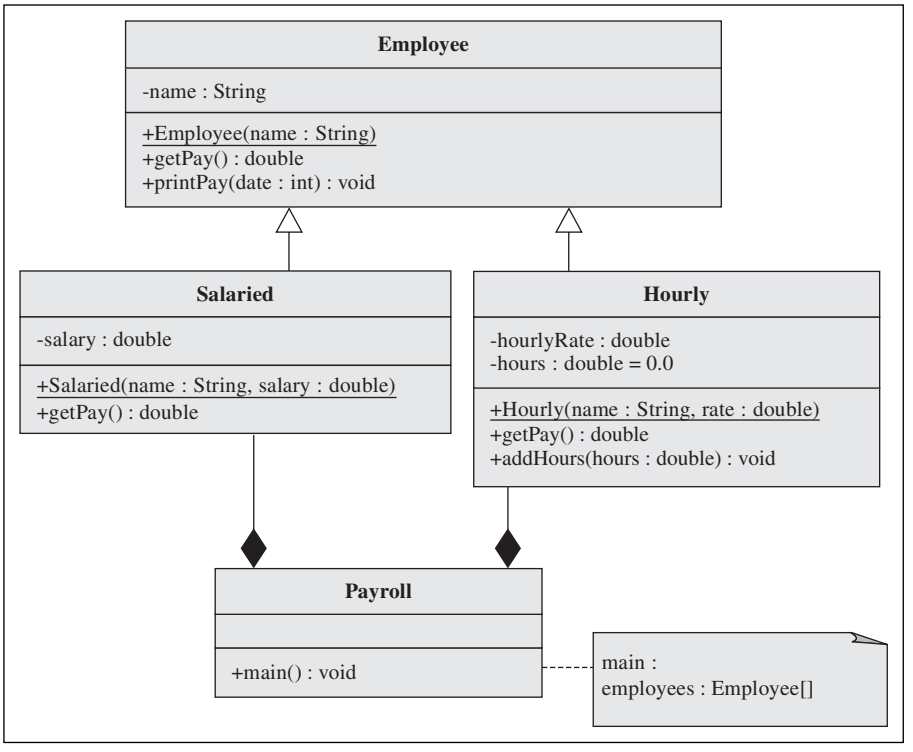


Figura 13.8 Diagrama de clase para el programa Payroll.

Suponga que Anna y Donovan son empleados por horas que ganan $25 y $20 por hora, respectivamente. Simon es un empleado asalariado que gana $4 000 al mes, y los tres empiezan a trabajar al inicio del mes, que es martes. Cuando se ejecuta el programa, debe producir la fecha del mes, el nombre del empleado y la cantidad pagada en la fecha indicada, como se muestra a continuación:

Salida:

```
 4       Anna:  800.00
 4    Donovan:  640.00
11       Anna: 1000.00
11    Donovan:  800.00
15      Simon: 2000.00
```

La implementación comenzará con el método `main` en el controlador en la figura 13.9. Observe la variable local de `main`, `employees`. Está declarada como un arreglo de 100 elementos de objetos `Employee`. Esto es lo que se ha declarado, aunque no es exactamente lo que contiene. Como puede verse a

```
/*********************************************************
* Payroll.java
* Dean & Dean
*
* Esta clase contrata y paga a empleados.
*********************************************************/

public class Payroll
{
  public static void main(String[] args)
  {
    Employee[] employees = new Employee[100];
    Hourly hourly;
    employees[0] = new Hourly("Anna", 25.0);
    employees[1] = new Salaried("Simon", 48000);
    employees[2] = new Hourly("Donovan", 20.0);

    // Esto supone arbitrariamente que el mes de la nómina
    // empieza un martes (día = 2), y que contiene 30 días.
    for (int date=1,day=2; date<=15; date++,day++,day%=7)
    {
      for (int i=0;
        i<employees.length && employees[i] != null; i++)
      {
        if (day > 0 && day < 6
          && employees[i] instanceof Hourly)
        {
          hourly = (Hourly) employees[i];
          hourly.addHours(8);
        }
        if ((day == 5 && employees[i] instanceof Hourly) ||
          (date%15 == 0 && employees[i] instanceof Salaried))
        {
          employees[i].printPay(date);
        }
      } // end for i
    } // end for date
  } // end main
} // end class Payroll
```

Esto selecciona elementos idóneos.

Esto convierte elementos en su clase nativa.

Esto selecciona el momento idóneo para imprimir cada tipo diferente.

Figura 13.9 Controlador para el programa Payroll simple.

partir de las declaraciones de asignación, los tres primeros elementos `employees` son un `Hourly`, un `Salaried` y otro `Hourly`. Este arreglo es heterogéneo. Todos los elementos en el arreglo son instancias de clases derivadas a partir de la clase del arreglo, y ninguno es una instancia de la clase `Employee` por sí mismo. Aun cuando en el arreglo puede no haber instancias de la clase del arreglo, el tipo del arreglo es el tipo correcto a usar porque es capaz de dar cabida a instancias de todas las clases descendientes de la clase del arreglo.

Al continuar con el método `main`, el ciclo externo `for` recorre 30 días, siguiendo la pista de dos variables. Observe cómo el primer compartimiento en el encabezado del ciclo `for` declara más de una variable del tipo especificado. La variable `date` representa la fecha del mes. Determina cuándo se paga a los empleados asalariados. Para no complicar las cosas, este programa supone 30 días por mes. Si el lector quiere aprender a obtener el número verdadero de días que hay en cada mes, debe consultar el sitio API Sun de Java en la red y leer la clase `Calendar`.[3] La variable `day` representa el día de la semana. Determina cuándo se paga a los empleados por hora. Suponiendo que el día 1 es lunes, puesto que el valor inicial de `day` es 2, el mes del programa empieza en martes. Observe cómo el tercer compartimiento del encabezado del ciclo `for` ejecuta más de una operación. Incrementa tanto `date` como `day` y luego usa `day%=7` para hacer que la variable `day` empiece su recorrido desde 0 siempre que llega a 7.

El ciclo interno `for` recorre el arreglo heterogéneo de empleados. El segundo componente del encabezado del ciclo `for` usa una condición de continuación compuesta. Sola, la condición `i<employees.lenght` permitiría la ejecución del ciclo a través de todos los 100 elementos del arreglo `employees`. ¿Cuál es la función de la condición `employees[1] != null` del encabezado del ciclo? El programa instancia sólo tres objetos para este arreglo, y 97 elementos siguen conteniendo el valor por defecto de `null`. Si el programa intenta llamar a un método con una referencia `null`, se cae. Más específicamente, genera un error `NullPointerException` la primera vez que intenta usar la referencia `null`. La condición `employees[1] != null` evita lo anterior al detener el ciclo cuando llega al primer elemento `null`.

Dentro del ciclo interno `for`, la primera declaración `if` acumula horas para los trabajadores por horas. Comprueba para ver si `day` es un día de la semana (distinto de 0 o 6). También comprueba si el objeto referido por el elemento del arreglo actual es una instancia de la clase `Hourly`. Esto permite al programa acumular horas sólo durante días laborales de la semana y sólo para trabajadores por horas. Una vez que se sabe que el objeto actual es una instancia de la clase `Hourly`, es seguro cambiar la referencia genérica a una referencia `Hourly`. Así, se procede a cambiar esa referencia a una tipo `Hourly`, se asigna a una variable de referencia `Hourly` y luego se usa ese tipo específico de variable de referencia para llamar al método `addHours`. ¿Por qué se saltó por estos aros? Suponga que se intenta llamar al método `addHours` con la referencia genérica en una declaración como ésta:

```
employees[i].addHours(8);
```

El compilador generaría el mensaje de error:

```
cannot find symbol
symbol  : method addHours(int)
```

Debido a que en la clase `Employee` no hay un método `addHours`, aunque hay uno en la clase `Hourly`, es necesario cambiar el elemento del arreglo explícitamente en un tipo de referencia `Hourly` y usar esa referencia para llamar al método que se necesita.

A continuación se considerará la segunda declaración `if` en el ciclo interno `for`. En lugar de acumular horas, su objetivo es generar una salida para un reporte de nómina. Esta declaración `if` se ejecuta si una de las dos condiciones es `true`. La primera condición es `true` si es viernes (`day = 5`) y si el objeto que llama es una instancia de la clase `Hourly`. La segunda condición es `true` si es la mitad del mes y si el objeto que llama no es una instancia de la clase `Hourly`. Si ninguna de esas condiciones se cumple, el elemento en el renglón del arreglo llama al método, como se muestra a continuación:

```
employees[i].printPay(date);
```

[3] He aquí un ejemplo de cómo es posible encontrar el último día en el mes actual:

```
int lastDayInCurrentMonth =
  Calendar.getInstance().getActualMaximum(Calendar.DAY_OF_MONTH);
```

Esta estrategia no hubiera funcionado con el método `addHours` en la primera declaración `if`, pero funciona con el método `printPay` en la segunda declaración `if`. ¿Por qué? Observe la especificación UML de la clase `Employee` en la figura 13.8. Esta vez se supone que el método que se está llamando, `printPay`, está definido en la clase del arreglo.

A continuación se trabajará en la implementación de ese método `printPay` en la clase `Employee`. En la figura 13.10, observe cómo `printPay` imprime la fecha y el nombre del empleado y luego llama a `getPay`. Se supone que el método `getPay` calcula el pago de un empleado. Pero el método `getPay` de la clase `Employee` simplemente devuelve 0.0. ¿Qué ocurre con esto? ¿Realmente los empleados no reciben paga? ¡Por supuesto que no! El método `getPay` de la clase `Employee` es simplemente un método inválido que no se ejecuta nunca. Los métodos `getPay` "reales" (es decir, los que se ejecutan) son las definiciones de sobreposición en las subclases `Salaried` y `Hourly`. ¡Estas definiciones de sobreposición hacen polimórfico al método `getPay`! ¿Cómo sabe la JVM utilizar estos métodos y no el método inválido en la clase `Employee`? Cuando la JVM ejecuta vinculación dinámica, busca al objeto que llama al método. Para el caso `getPay`, el objeto que llama es una instancia de la clase `Salaried` o de la clase `Hourly`. ¿Puede el lector ver por qué?

Regrese al método `main` en la figura 13.9 y observe la asignación de un objeto `Hourly` en `employees[0]`. Cuando `employees[i].printPay()` se llama con `i` igual a 0, el objeto que llama es un

```
/*********************************************
* Employee.java
* Dean & Dean
*
* Ésta es una descripción genérica de un empleado.
*********************************************/

public class Employee
{
  private String name;

  //*******************************************

  public Employee(String name)
  {
    this.name = name;
  }

  //*******************************************

  public void printPay(int date)
  {
    System.out.printf("%2d %10s: %8.2f\n",
      date, name, getPay());
  } // end printPay

  //*******************************************

  // Este método inválido satisface al compilador.

  public double getPay()
  {
    System.out.println("error! en inválido");
    return 0.0;
  } // end getPay
} // end class Employee
```

Figura 13.10 Clase `Employee`.

```
/*******************************************
* Salaried.java
* Dean & Dean
*
* Esta clase implementa un empleado asalariado.
*******************************************/

public class Salaried extends Employee
{
  private double salary;      // al año

  //*****************************************

  public Salaried(String name, double salary)
  {
    super(name);
    this.salary = salary;
  } // end constructor

  //*****************************************

  public double getPay()
  {
    return this.salary / 24;  // a la quincena
  } // end getPay
} // end class Salaried
```

Figura 13.11 Clase `Salaried`.

objeto `Hourly`. Dentro del método `printPay`, cuando se llama `getPay`, el objeto que llama sigue siendo un objeto `Hourly`. En consecuencia, la JVM usa el método `getPay` de la clase `Hourly`. Y eso es lo que se quiere: el objeto `employees[0]` es un `Hourly`, de modo que usa el método `getPay` de la clase `Hourly`. El mismo razonamiento puede aplicarse al objeto `employees[1]`. Puesto que se trata de un objeto `Salaried`, usa el método `getPay` de la clase `Salaried`. Gracias al polimorfismo y a la vinculación dinámica, la vida es bella.

Lo verdaderamente genial acerca del polimorfismo y la vinculación dinámica es poder programar en forma genérica. En el método `main`, `printPay` puede llamarse para todos los objetos en el arreglo y no preocuparse sobre el hecho de si el objeto es un `Hourly` o un `Salaried`. Simplemente se supone que `printPay` funciona correctamente para cada empleado. Esta capacidad de programar en forma genérica permite que los programadores piensen en perspectiva sin quedarse atascados en detalles.

En la clase `Employee`, ¿se molestó el lector por el método inválido `getPay`? Pensó: "¿Por qué incluir un método `getPay` en la clase `Employee` aun cuando nunca se ejecuta?" Es necesario porque si en la clase `Employee` no hubiera método `getPay`, el compilador podría generar un error. ¿Por qué? Porque cuando el compilador ve una llamada a un método sin prefijo punto, comprueba para asegurarse de que el método puede encontrarse dentro de la clase actual. El método `getPay()` (dentro del método `printPay`) no tiene prefijo punto, de modo que el compilador requiere que la clase `Employee` tenga un método `getPay`.

Ahora ya es hora de implementar los métodos `getPay` "reales". Observe las figuras 13.11 y 13.12. Los métodos en estas dos clases son simples, aunque diferentes. Para evitar que la JVM seleccione el método inválido `getPay` en la clase base durante la vinculación dinámica, todas las clases derivadas deben sobreponerse en ese método.

13.8 Métodos y clases `abstract`as

El método inválido `getPay` en la figura 13.10 es un ejemplo de un *kludge*. Un código *kludgy* es un código torpe e inelegante que proporciona una solución temporal para un problema. Por lo general, un

código no elegante es difícil de comprender. Y un código difícil de comprender es difícil de mantener. Así, intente evitar este tipo de códigos. Algunas veces no es posible hacerlo, pero en este caso sí es posible evitar, en efecto, el *kludge* en el método inválido. He aquí cómo. . .

Una clase abstracta perfila el trabajo futuro.

Si el lector está escribiendo un código inválido que será sobrepuesto por métodos definidos en todas las clases descendientes instanciables, debe detenerse y reflexionar. Hay una mejor forma de hacer lo anterior. El lector debe usar una *clase abstracta* para indicarle al compilador lo que está intentando hacer de antemano. En la clase abstracta, debe *declarar* los métodos que son inadecuados para la clase de la variable de referencia pero que estarán definidos por clases descendientes que instancian objetos. Para declarar un método, simplemente escriba el encabezado del método con el modificador adicional `abstracto`, y termine este encabezado modificado del método con un punto y coma. Por ejemplo, observe el modificador `abstracto` en el encabezado de la clase `Employee2` en la figura 13.13.

Una declaración `abstracta` no contiene suficiente información para definir el método. Simplemente especifica su interfaz con el mundo exterior e indica que la definición o las definiciones existirá(n) en alguna otra parte. ¿Dónde? ¡En todas las clases descendientes instanciables! El uso de un método `abstracto` evita la inelegante definición del método inválido, y constituye una mejor forma para implementar polimorfismo.

```
/**************************************************
* Hourly.java
* Dean & Dean
*
* Esta clase implementa un empleado pagado por hora.
**************************************************/

public class Hourly extends Employee
{
  private double hourlyRate;
  private double hours = 0.0;

  //************************************************

  public Hourly(String name, double rate)
  {
    super(name);
    hourlyRate = rate;
  } // end constructor

  //************************************************

  public double getPay()
  {
    double pay = hourlyRate * hours;
    hours = 0.0;
    return pay;
  } // end getPay

  //************************************************

  public void addHours(double hours)
  {
    this.hours += hours;
  } // end addHours
} // end class Hourly
```

Figura 13.12 Clase `Hourly`.

```
/*****************************************
* Employee2.java
* Dean & Dean
*
* Esta clase abstracta describe empleados.
*****************************************/

public abstract class Employee2
{
  public abstract double getPay();
  private String name;

  //*************************************

  public Employee2(String name)
  {
    this.name = name;
  }

  //*************************************

  public void printPay(int date)
  {
    System.out.printf("%2d %10s: %8.2f\n",
      date, name, getPay());
  } // end printPay
} // end class Employee2
```

Si hay un método `abstracto`, la clase también es `abstracta`.

La declaración del método `abstracto` sustituye la definición del método inválido.

Figura 13.13 Clase `Employee2`, usando el modificador `abstracto` para sustituir la definición de un método inválido por una declaración más simple del método.

El modificador `abstracto` posee un nombre adecuado. Algo es abstracto si es de naturaleza general, no detallada. Una declaración de un método `abstracto` es general por naturaleza. No proporciona los detalles del método. Simplemente notifica que el método existe y que es necesario precisarlo mediante definiciones del método "real" en todas las clases descendientes instanciables. En cuanto a nuestro programa, ¿se ha seguido esta regla? En otras palabras, ¿se cuenta con definiciones del método `getPay` en todas las clases descendientes `Employee2`? Sí: las clases `Salaried` y `Hourly` en las figuras 13.11 y 13.12 ya contienen las definiciones del método `getPay`. No obstante, es necesario revisar las clases `Salaried`, `Hourly` y `Payroll` al hacer estas sustituciones:

```
Employee → Employee2
Salaried → Salaried2
Hourly → Hourly2
Payroll → Payroll2
```

Luego, las clases `Salaried2`, `Hourly2` y `Payroll2` se verán como sigue:

```
public class Salaried2 extends Employee2
{
  ...

public class Hourly2 extends Employee2
{
  ...

public class Payroll2
```

```
{
  public static void main(String[] args)
  {
    Employee2[] employees = new Employee2[100];
    ...
```

Aquí hay otra cosa que observar cuando se declara un método `abstracto`. Debido a que una declaración de un método `abstracto` no proporciona una definición para ese método, la definición de clase está incompleta. Puesto que la definición de clase está incompleta, no es posible usarla para construir objetos. El compilador reconoce esto y se queja si el lector no lo reconoce en su código. Para satisfacer al compilador, es necesario agregar un modificador abstracto al encabezado de clase siempre que se tenga una clase que contiene uno o más métodos `abstractos`. Por ejemplo, observe el modificador `abstracto` en el encabezado de la clase `Employee2` en la figura 13.13.

Agregar un modificador `abstracto` al encabezado de una clase imposibilita instanciar un objeto desde esa clase. Si un programa intenta instanciar una clase `abstracta`, el compilador genera un error. Por ejemplo, puesto que `Employee2` es una clase `abstracta`, se cometería un error de compilación si se tuviese un método `main` como éste:

```
public static void main(String[] args)
{
  Employee2 emp = new Employee2("Benji");
}
```

Debido a que `Employee2` es `abstracto`, esto genera un error de compilación.

Algunas veces no es aconsejable que una clase hijo defina un método que fue declarado como `abstracto` en su padre. En lugar de eso, sería conveniente diferir la definición del método a la siguiente generación. Hacer esto es fácil. En la clase hijo, simplemente ignore ese método y también declare `abstracta` la clase hijo (puesto que por lo menos ese método sigue sin definir). Las definiciones de los métodos pueden postergarse así tanto como se quiera, en el supuesto de que el lector termine por definirlos a todos en cualquier clase descendiente no `abstracta` que se use para instanciar objetos.

Se mencionó que si cualquier método en una clase es `abstracto`, entonces esa clase debe ser `abstracta`. Pero esto no significa que todos los métodos en una clase `abstracta` deban ser `abstractos`. A menudo es de utilidad incluir una o más definiciones de métodos no `abstractos` en una clase `abstracta`. Así, las clases descendientes de una clase `abstracta` pueden heredar métodos no `abstractos` de esa clase y no es necesario redefinir estos métodos no `abstractos`.

Es ilegal usar `private` o `final` con `abstracto`

Una declaración de un método `abstracto` no puede ser `private`, y las definiciones del método que aparecen en clases descendientes tampoco pueden ser `private`. ¿Por qué? Una declaración de un método `abstracto` proporciona una ruta mínima libre de *kludges* a fin de que el compilador acepte una llamada a un método polimórfico. Si un método es polimórfico, versiones de él aparecen en más de una clase, de modo que por lo menos uno de los métodos polimórficos está inevitablemente fuera de la clase que llama. No es posible acceder a un método externo que sea `private`, y puesto que todas las definiciones de un método polimórfico deben tener interfaces idénticas con el mundo exterior, ninguna de las definiciones polimórficas puede ser `private`. Puesto que se supone que la declaración `abstracta` en la clase ancestro `abstracta` describe correctamente lo que del mundo exterior se parece al método (y al compilador), el modificador de acceso que aparece en la declaración `abstracta` tampoco puede ser `private`.

Una clase `abstracta` o un método `abstracto` no pueden ser `final`. El modificador `final` evita que una clase sea extendida e impide que un método sea sobrepuesto. Pero se supone que una clase `abstracta` es extendida y que un método `abstracto` es sobrepuesto, de modo que es ilegal usar `final` con `abstracto`.

13.9 Interfaces

Las interfaces de Java pueden hacer muchas cosas distintas, una de las cuales es ayudar a implementar polimorfismo. Pero antes de llegar a eso, conviene mencionar algunos otros usos de una interfaz de Java.

Uso de interfaces para estandarizar la comunicación entre clases

Establezca protocolos de comunicación pronto.

El uso más evidente de una interfaz de Java es el que implica su nombre: especificar los encabezados de un conjunto de métodos que debe implementar una clase. Una interfaz de Java es un contrato entre el diseñador de un programa y los implementadores del programa que estandarizan la comunicación entre clases diferentes. Este uso de las interfaces es esencial para el éxito de proyectos de programación grandes.

Suponga, por ejemplo, que está diseñando un sistema de contabilidad, y que por el momento está concentrado en la contabilidad de "activos", que siguen la pista del valor de las cosas que posee la compañía o sobre las que ésta tiene derechos. Una contabilidad típica de activos incluye lo siguiente: dinero en efectivo, cuentas por cobrar, inventario, mobiliario, equipo de manufactura, vehículos, edificios y terrenos. Estas cosas son distintas entre sí, por lo que no sería natural que las clases que las representan estén en una sola jerarquía de la herencia. Algunos de estos activos (mobiliario, equipo de manufactura, vehículos y terrenos) describen activos a largo plazo o "fijos" cuyo valor se deprecia gradualmente con el tiempo. Cada año, un contador prepara un conjunto de declaraciones financieras, como la hoja de balance y una declaración de ganancias y pérdidas. La preparación de estos documentos requiere el acceso a los objetos que representan los activos que se deprecian para obtener información como el costo original, la fecha de adquisición y la tasa de depreciación.

Para facilitar este acceso, sería conveniente que las referencias a estos objetos estuviesen en un arreglo común o en una `ArrayList`. Luego, un programa podría entrar por el arreglo o por la `ArrayList` y llamar a métodos denominados idénticamente "get" para recuperar valores de las variables de instancia `originalCost`, `acquisitionDate` y `depreciationRate` en cada objeto que representa un activo que se deprecia. Suponga que distintos programadores escriben las clases para diferentes contabilidades. La mejor forma de asegurar que todos los programadores están "leyendo la misma página" es solicitar que todas las clases que acceden a cierto conjunto de datos implementen la misma *interfaz* de Java. En el ejemplo del sistema de contabilidad, la interfaz para los métodos "get" que acceden a las variables de instancia `originalCost`, `acquisitionDate` y `depreciationRate` podría denominarse interfaz `AssetAging`. La interfaz `AssetAging` podría contener declaraciones/encabezados para sus métodos, pero no definiciones.

Si una clase particular incluye una definición de todos los métodos declarados en alguna interfaz (como `AssetAging`), puede decirse al mundo (y al compilador de Java) que esa clase proporciona tales definiciones al agregar una cláusula `implements` a su encabezado de clase, como ésta:

```
public <class-name> implements <interface-name>
{
   ...
```

Para múltiples interfaces, los nombres se separan con comas, como:

```
public <class-name> implements <interface-name1>, <interface-name2>, ...
{
   ...
```

Para herencia y una interfaz, se hace lo siguiente:

```
public <class-name> extends <parent-class-name> implements <interface-name>
{
   ...
```

Una clase dada extiende sólo una superclase, pero puede implementar (`implement`) cualquier número de interfaces. Una interfaz de Java es como una clase `abstracta` "pura". Es pura en cuanto a que nunca define ningún método. Sin embargo, es menos versátil que una clase `abstracta`. No puede declarar cualquier método `static`, no puede declarar cualquier variable y no puede declarar cualquier constante de instancia. En otras palabras, proporciona sólo constantes nombradas `public static final` y sólo declaraciones de método `public abstract`. Ésta es la sintaxis para la definición de una interfaz:

```
interface <interface-name>
{
   <type> <CONSTANT_NAME> = <value>;
   ...
```

```
    <return-type> <method-name>(<type> <parameter-name> ...);
    ...
}
```

La definición de una interfaz comienza con la palabra clave, `interface`, justo como la definición de una clase empieza con la palabra clave, `clase`. Las constantes nombradas en una interfaz se definen de la misma forma en que se definen las constantes nombradas en una clase, y los métodos en una interfaz se declaran agregando un punto y coma a los encabezados del método. Observe que la palabra clave `public` no aparece en ningún sitio en la plantilla de sintaxis. Puede incluirse el modificador `public`, aunque no es necesario, y la práctica común es omitirlo, puesto que simplemente no tiene sentido que una interfaz o cualquiera de sus componentes sea cualquier cosa menos `public`. También observe que la palabra clave `abstract` no aparece en ningún sitio en la plantilla de sintaxis. En cualquier declaración de método puede incluirse un modificador `abstract`, y el modificador `abstract` puede incluirse en el encabezado de la interfaz pero, de nuevo, no es necesario y la práctica común es omitirlo, puesto que no tiene sentido tener una interfaz que no sea completamente `abstract`. También observe que la palabra clave `static` no aparece en ningún sitio en la plantilla de sintaxis. Se entiende que cualquier constante es `static`, y se entiende que cualquier método no es `static`. Por último, observe que la palabra clave `final` no aparece en ningún sitio en la plantilla de sintaxis. Se entiende que todas las constantes son `final`, y puesto que no se ha definido ningún método, se entiende que cualquier declaración de método no es `final`.

Uso de una interfaz para almacenar constantes universales

Además de decirle al mundo que su clase define cierto conjunto mínimo de métodos, el hecho de implementar una interfaz también otorga a su clase libre acceso a todas las constantes nombradas que define esa interfaz. Colocar constantes nombradas comunes en una interfaz y luego proporcionarles acceso a múltiples clases a estas constantes nombradas al hacer que implementen esa interfaz constituye una forma práctica de proporcionar fácil acceso a un gran conjunto de constantes físicas comunes y/o factores o constantes empíricos. Se evitan las definiciones duplicadas de estas constantes y no es necesario usar un prefijo punto del nombre de la clase para acceder a estas constantes. En principio, podría usarse una jerarquía de la herencia para contar con acceso directo a constantes comunes, pero esto sería una práctica deficiente, ya que se desperdiciaría la única oportunidad de herencia en nada más que un puñado de constantes. Si para hacer lo anterior se usa una interfaz, se sigue teniendo libertad para usar herencia y/o interfaces adicionales para otros propósitos.

Uso de interfaces para implementar polimorfismos adicionales

Ahora suponga que ya ha creado una jerarquía de la herencia y que ya está usándola para implementar algún polimorfismo particular, como se hizo en el programa de nómina. Luego suponga que se desea agregar otro polimorfismo que no se ajusta a la estructura de la jerarquía de la herencia original. Por ejemplo, podría quererse que un método sea polimórfico sólo entre algunas de las clases en la jerarquía original, y/o podría quererse que un polimorfismo incluya clases que están en la jerarquía original, y/o podría desearse que el polimorfismo incluya clases que están fuera de la jerarquía. Una clase de Java no puede participar en más de una herencia: sólo puede extender una clase. Así, no es posible usar clases `abstractas` para soportar polimorfismos que generan más de una jerarquía de la herencia. Pero así como se sugiere en el ejemplo del sistema de contabilidad anterior, es posible generar más de una jerarquía de la herencia con una interfaz de Java. Y una de las razones principales para usar interfaces de Java es para implementar múltiples polimorfismos.

Hágalo polimórfico sin distorsionar la herencia.

Para ilustrar lo anterior, el programa previo de la nómina se mejorará al agregar dos clases de empleados comisionados.[4] Una de estas clases obtiene una comisión "directa". La otra clase obtiene un salario más una comisión. En cualquier caso, la comisión está basada en un porcentaje fijo común de ventas. La figura 13.14 contiene el código para una interfaz que define este porcentaje fijo como una constante nombrada y declara un método que debe definirse en todas las clases que implementan la interfaz.

[4] Vea el apéndice 7 para un diagrama UML completo del programa de nómina mejorado que se desarrolló en esta subsección.

```
/*******************************************************
* Commission.java
* Dean & Dean
*
* Esta interfaz especifica un atributo común y declara
* el comportamiento común de los empleados comisionados.
*******************************************************/

interface Commission
{
  double COMMISSION_RATE = 0.10;

  void addSales(double sales);
} // end interface Commission
```

Figura 13.14 Una interfaz para usar con una versión mejorada del programa de nómina.

La figura 13.15 muestra el código para una clase `Commissioned` que describe una clase de empleados que trabajan por una comisión directa. La clase `Commissioned` extiende la clase `Employee2` de la figura 13.13. `Employee2` es una clase `abstracta`, y como tal, la subclase `Commissioned`

```
/*************************************************************
* Commissioned.java
* Dean & Dean
*
* Esta clase representa empleados con comisión directa.
*************************************************************/

public class Commissioned extends Employee2 implements Commission
{
  private double sales = 0.0;

  //***********************************************************

  public Commissioned(String name)
  {
    super(name);
    this.sales = sales;
  } // end constructor

  //***********************************************************

  public void addSales(double sales)
  {
    this.sales += sales;
  } // end addSales

  //***********************************************************

  public double getPay()
  {
    double pay = COMMISSION_RATE * sales;

    sales = 0.0;
    return pay;
  } // end getPay
} // end class Commissioned
```

La interfaz requiere esta definición de método.

La herencia de una clase `abstracta` requiere esta definición de método.

La interfaz proporciona este valor constante.

Figura 13.15 Clase que define a los empleados por comisión directa en el programa de nómina mejorado.

debe definir todos los métodos `abstractos` de `Employee2`. El único método `abstracto` en la clase `Employee2` es la clase `getPay`, de modo que la clase `Commissioned` debe definir al método, y sí, lo hace. Esto aumenta a tres el número total de métodos `getPay` polimórficos. En el encabezado de la clase `Commissioned`, observe la cláusula, `implements interface Commission`. Ésta proporciona acceso directo a la constante nombrada `COMMISION_RATE`, que el método `getPay` de la clase `Commissioned` usa para hacer su trabajo. Cuando la clase `Commissioned` implementa la interfaz `Commission`, asume una obligación. Debe definir todos los métodos declarados en esa interfaz. El único método declarado en la interfaz `Commission` es el método `addSales`, y sí, la clase `Commissioned` también define este método.

La figura 13.16 muestra el código para una clase `SalariedAndCommissioned`. Esta clase extiende la clase `Salaried2`. La clase `Salaried2` es como la clase `Salaried` en la figura 13.11, excepto por una diferencia: mientras la clase `Salaried` extiende `Employees`, la clase `Salaried2` extiende `Employee2`. La clase `SalariedAndCommissioned` describe una clase de empleados que gana un salario y una comisión. La clase `Salaried2` define un método `getPay`, de modo que el compilador no insiste en que la clase `SalariedAndCommissioned` también define un método `getPay`. Observe cómo el método de sobreposición usa el prefijo `super` para llamar al método al que se sobrepone. Esta definición adicional del método `getPay` incrementa a cuatro el número total de métodos `getPay` polimórficos.

```
/*************************************************************
* SalariedAndCommissioned.java
* Dean & Dean
*
* Esta clase representa empleados asalariados y comisionados.
*************************************************************/

public class SalariedAndCommissioned
  extends Salaried2 implements Commission
{
  private double sales;

  //*********************************************************

  public SalariedAndCommissioned(String name, double salary)
  {
    super(name, salary);
  } // end constructor

  //*********************************************************

  public void addSales(double sales)
  {
    this.sales += sales;
  } // end addSales

  //*********************************************************

  public double getPay()
  {
    double pay =
      super.getPay() + COMMISSION_RATE * sales;

    sales = 0.0;   // reset for next pay period
    return pay;
  } // end getPay
} // end class SalariedAndCommissioned
```

La interfaz requiere esta definición de método.

La interfaz proporciona este valor constante.

Este método se sobrepone al método definido en la clase padre.

Figura 13.16 Clase que define empleados asalariados y comisionados en el programa de nómina mejorado.

La clase `SalariedAndCommissioned` también implementa la interfaz `Commission`. Lo anterior proporciona acceso directo a la constante nombrada `COMMISSION_RATE` que el método `getPay` usa para hacer su trabajo. Debido a que la clase `SalariedAndCommissioned` implementa a la interfaz `Commission`, debe definir todos los métodos declarados en esa interfaz; y sí, define el método `addSales`.

Para ejecutar estas clases adicionales, se necesita una clase `Payroll3` como la que se muestra en la figura 13.17.

La clase `Payroll3` agrega dos objetos más (Glen y Carol) al arreglo. Luego usa estos objetos para llamar a los métodos `addSales` en las nuevas clases. Para hacer estas llamadas a los métodos, los ele-

```
/*************************************************************
* Payroll3.java
* Dean & Dean
*
* Esta clase contrata y paga a cuatro tipos distintos de empleados.
*************************************************************/

public class Payroll3
{
  public static void main(String[] args)
  {
    Employee2[] employees = new Employee2[100];
    Hourly2 hourly;
    employees[0] = new Hourly2("Anna", 25.0);
    employees[1] = new Salaried2("Simon", 48000);
    employees[2] = new Hourly2("Donovan", 20.0);
    employees[3] = new Commissioned("Glen");
    employees[4] = new SalariedAndCommissioned("Carol", 24000);

    ((Commission) employees[3]).addSales(15000);
    ((Commission) employees[4]).addSales(15000);

    // Esto supone arbitrariamente que el mes de la nómina empieza
    // un martes (día = 2), y que contiene 30 días.
    for (int date=1,day=2; date<=15; date++,day++,day%=7)
    {
      for (int i=0;
        i<employees.length && employees[i] != null; i++)
      {
        if (day > 0 && day < 6
          && employees[i] instanceof Hourly2)
        {
          hourly = (Hourly2) employees[i];
          hourly.addHours(8);
        }
        if ((day == 5 && employees[i] instanceof Hourly2) ||
          (date%15 == 0 &&
            (employees[i] instanceof Salaried2 ||
            employees[i] instanceof Commissioned)))
        {
          employees[i].printPay(date);
        }
      } // end for i
    } // end for date
  } // end main
} // end class Payroll3
```

Figura 13.17 Controlador para la tercera versión del programa de nómina.

mentos del arreglo se cambian al tipo interfaz. El compilador requiere un tipo porque el método addSales no aparece en la clase Employee2. Observe que se requiere un conjunto adicional de paréntesis alrededor del operador tipo cast (Commission) y el objeto que llama. Hubiera sido posible utilizar conversiones de tipo más específicas como esto:

```
((Commissioned) employees[3]).addSales(15000);
((SalariedAndCommissioned) employees[4]).addSales(15000);
```

Pero resulta más elegante hacer lo anterior en el tipo de interfaz más genérico Commission y dejar que la JVM seleccione de entre las alternativas polimórficas como lo hace con la vinculación dinámica. A continuación se muestra lo que genera el controlador Payroll3 usando cualquier tipo de conversión:

Salida:

```
 4        Anna:      800.00
 4     Donovan:      640.00
11        Anna:     1000.00
11     Donovan:      800.00
15       Simon:     2000.00
15        Glen:     1500.00
15       Carol:     2500.00
```

En los ejemplos codificados, ¡observe la semejanza entre el uso de un nombre de interfaz y el uso de un nombre de clase! No es posible instanciar una interfaz porque es intrínsecamente abstracta, pero es posible usarla como cualquier clase normal para especificar el tipo. Por ejemplo, es posible declarar un arreglo de elementos cuyo tipo es un nombre de interfaz, es posible poblar ese arreglo con instancias de clases que implementan esa interfaz, y luego es posible retirar objetos de ese arreglo y convertirlos en cualquier tipo (clase o interfaz) al que se ajusten estos objetos. El controlador Payroll4 en la figura 13.18 y la salida ulterior ilustran estas posibilidades.

El truco consiste en pensar en lo que requiere el compilador y en lo que hace la JVM. Por ejemplo, es posible crear un arreglo de referencias a interfaces porque los elementos en el arreglo son meras referencias, no objetos instanciados. El compilador deja poblar ese arreglo con referencias a objetos desde clases que implementan esa interfaz porque sabe que esos objetos pueden llamar a cualquier método que declare la interfaz. En la llamada a un método, el compilador deja que el lector convierta una referencia en el tipo de cualquier clase que declare o defina cualquier versión de ese método porque sabe que la JVM puede encontrar por lo menos un método que vincular. En tiempo de ejecución, la JVM selecciona el método más idóneo para vincular.

13.10 El modificador de acceso protected

Hasta el momento, sólo se han analizado dos modos de accesibilidad para los miembros de una clase: public y private; a los miembros public puede accederse desde cualquier parte; a los miembros private sólo es posible acceder desde dentro de la clase de los miembros. Hay otro modificador de acceso que es una forma limitada del modificador de acceso public: el modificador de acceso protected. Especifica una accesibilidad intermedia entre public y private. A los miembros que son protected sólo puede accederse desde dentro del mismo paquete[5] o desde el *subárbol* del miembro. ¿Qué es un subárbol? Es una jerarquía de clase que consta de una clase más todas sus clases descendientes.

¿Cuándo debe usarse el modificador protected? La regla general es que debe usarse cuando se quiere tener acceso fácil a un miembro, pero no se quiere notificar al público en general. En otras palabras, se desea que esté más expuesto que un miembro private, pero menos que un miembro public.[6] Hmmm. . . esto sigue siendo vago. Esta situación se trabajará con un ejemplo.

[5] Si el lector desea aprender más sobre paquetes y cómo agrupar sus clases en un paquete definido por el programador, debe consultar el apéndice 4.

[6] Debido a que a un miembro protected puede accederse desde cualquier clase descendiente de la clase que define al miembro protected, cualquiera puede extender la clase que define al miembro protected y así acceder directamente a él. En otras palabras, el modificador protected realmente no proporciona mucha protección. Si el lector es un extraño al lenguaje, permanezca lejos de los miembros protected de cualquier otra persona. Considérelos productos irregulares que no tienen garantía.

```
/*************************************************************
* Payroll4.java
* Dean & Dean
*
* Esta clase contrata y paga a los empleados un tipo de comisión.
*************************************************************/

public class Payroll4
{
  public static void main(String[] args)
  {
    Commission[] people = new Commission[100];

    people[0] = new Commissioned("Glen");
    people[1] = new SalariedAndCommissioned("Carol", 24000);

    people[0].addSales(15000);
    people[1].addSales(15000);
    for (int i=0; i<people.length && people[i] != null; i++)
    {
      ((Employee2) people[i]).printPay(15);
    }
  } // end main
} // end class Payroll4
```

Aunque no es posible instanciar una interfaz en sí, es posible declarar referencias a la interfaz.

El compilador acepta este cambio porque `Employee2` define un método `printPay`, pero la JVM vincula los objetos con métodos en clases descendientes de `Employee2`.

Salida:

```
15    Glen:   1500.00
15   Carol:   2500.00
```

Figura 13.18 Demostración de propiedades de una interfaz semejantes a las de clase.

Programa Payroll con un método `protected`

Suponga que el programa Payroll quiere mejorarse de modo que incluya el cálculo de los impuestos FICA (FICA, del inglés Federal Insurance Contribution Act, predecesor del Programa de Seguridad Social en Estados Unidos). Este cálculo de impuestos se hace mejor en un método por separado. ¿Dónde debe ir ese método? El único momento en que se realiza este cálculo es cuando los empleados reciben su paga. Así, lógicamente, se trata de un método de ayuda llamado por el método `getPay`.

¿Dónde está el método `getPay`? Es un método polimórfico que aparece en todas las clases que directa e indirectamente extienden la clase `Employee`: `Comissioned`, `Salaried`, `Hourly` y `SalariedAndCommissioned`. Pero ¡cuidado! Este conjunto de clases juntas con la clase `Employee` en sí es el subárbol `Employee`. Así, en lugar de repetir la definición del cálculo de los impuestos FICA en todas las clases que tienen un método `getPay`, es más lógico y eficiente colocar este cálculo común en la clase `Employee` de la raíz del subárbol y hacerlo `protected`.

A fin de evitar pisotear las versiones previas del programa, en el nuevo programa de nómina mejorado FICA se usan nuevos nombres de clase. Vea la figura 13.19. Muestra el diagrama UML del programa con los nuevos nombres de clase: `Payroll5`, `Employee3`, `Commissioned2`, `Salaried3`, `Hourly3` y `SalariedAndCommissioned2`.

En la figura 13.20 se muestra la definición de `Employee3`, que incluye este método de ayuda común adicional, `getFICA`. `Employee3` también incluye algunas constantes nombradas usadas en el cálculo del FICA. Los detalles de este cálculo no son relevantes para este análisis, de modo que para ahorrar espacio, se implementa en forma ligeramente críptica usando el operador condicional. Lo que hace este pequeño `getFICA` es una representación razonable de lo que realmente ocurre con los cheques de las personas. De modo que si el lector es curioso, podría expandir el código críptico a una forma más legible. (En un ejercicio al final del capítulo se pide al lector hacer esto.)

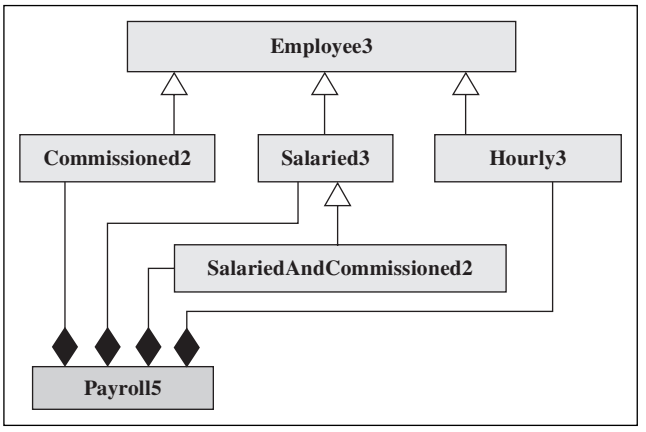


Figura 13.19 Diagrama de clase abreviado para un programa Payroll mejorado.

Cada uno de los métodos polimórficos `getPay` incluye una llamada a este nuevo método `getFICA`. El código para esta llamada es esencialmente el mismo en cada uno de los métodos `getPay`, así que sólo se mostrará una vez, en la clase `Salaried3` en la figura 13.21.

En su mayor parte, la clase `SalariedAndCommissioned2` que extiende la clase `Salaried3` es como la que se muestra en la figura 13.16, con cambios idóneos en los números de versión en los extremos de los nombres de clase. Sin embargo, en el método `getPay` no es posible usar `super.getPay()` para acceder a `salary` en la clase `Salaried3`, porque el impuesto FICA hace que el valor devuelto por el método `getPay` de `Salaried3` sea diferente al valor de `salary`.

Con el impuesto FICA, debe haber otra forma de acceder a `salary`. Aunque `Salaried3` puede incluir un método de acceso `getSalary`, el código podría ser más sencillo si `salary` fuese `public`. Pero, ¿quisiera el lector que el salario de todo mundo fuese `public`? Lo mejor que puede hacerse aquí es elevar la accesibilidad de la variable `salary` en la clase `Salaried3` de `private` a `protected`. Con esto, las clases descendientes tienen acceso directo a la variable `salary`, pero esto no expone tanto la situación como lo haría un modificador `public`.

Debido a que la clase `SalariedAndCommissioned2` extiende `Salaried3`, si hay un modificador `protected` en la variable `salary` en `Salaried3`, el método `getPay` en la clase `SalariedAndCommissioned2` puede definirse como sigue:

```
public double getPay()
{
  double pay = salary + COMMISSION_RATE * sales;
  pay -= getFICA(pay);
  sales = 0.0;   // reset for next pay period
  return pay;
} // end getPay
```

protegido en `Salaried3`

protegido en `Employee3`

De modo que aquí está. El polimorfismo permite colocar objetos heterogéneos en arreglos genéricos cuyo tipo es una clase de la que descienden las clases de los objetos o una interfaz que implementan las clases de los objetos. Luego es posible cambiar elementos del arreglo en tipos de subclase o interfaz, de modo que los elementos del arreglo pueden hacer llamadas al método que son específicas de su subclase o interfaz. La JVM encuentra el método que mejor se ajusta al objeto que llama y ejecuta ese método. El modificador `protected` permite acceso directo a variables y métodos desde cualquier sitio en el subárbol del miembro `protected`.

13.11 Apartado GUI: gráficas en tres dimensiones (opcional)

Ahora que ya sabe cómo funcionan la herencia y el polimorfismo basado en interfaz, debe poder entender algunas sutilezas que hacen funcionar el trabajo de ilustración gráfica. En esta sección se proporciona una ilustración típica de polimorfismo en el uso del API de Java.

```
/*************************************************************
* Employee3.java
* Dean & Dean
*
* Esta clase abstracta describe a los empleados e incluye
* el cálculo del impuesto por seguridad social.
*************************************************************/

public abstract class Employee3
{
  public abstract double getPay();
  private String name;
  private final static double FICA_TAX_RATE = 0.08; // fracción
  private final static double FICA_MAX = 90000;     // dólares
  private double ytdIncome;       // ingreso anual total hasta la fecha

  //***********************************************************

  public Employee3(String name)
  {
    this.name = name;
  }

  //***********************************************************

  public void printPay(int date)
  {
    System.out.printf("%2d %10s: %8.2f\n",
      date, name, getPay());
  } // end printPay

  //***********************************************************

  protected double getFICA(double pay)
  {
    double increment, tax;

    ytdIncome += pay;
    increment = FICA_MAX - ytdIncome;
    tax = FICA_TAX_RATE *
      (pay < increment ? pay : (increment > 0 ? increment : 0));
    return tax;
  } // end getFICA
} // end class Employee3
```

Esto limita la accesibilidad a las clases en el subárbol o en el mismo paquete.

Figura 13.20 Clase `Employee3` que incluye al método `getFICA` `protected`.

El API de Java proporciona varias clases que en conjunto permiten dibujar y colorear muchas formas bidimensionales. Además, la clase `Graphics2D` incluye dos métodos (`draw3DRect` y `fill3DRect`) que permiten representar una forma tridimensional sencilla. Trazan un rectángulo sombreado que hace que se vea como un rectángulo elevado ligeramente del plano de la página o ligeramente hundido con respecto al plano de la página. En los capítulos 16 y 17 se mostrará la forma en que estos dos métodos pueden usarse para simular un botón oprimido o sin oprimir. Pero esto se refiere a la amplitud de la ayuda que puede obtenerse del API de Java en la creación de lo que parecen ser imágenes tridimensionales.

La representación de una imagen tridimensional en Java requiere considerar cálculos de geometría y trigonometría. En esta sección se probará lo anterior mediante la representación de un cilindro sólido

```
/*******************************************
* Salaried3.java
* Dean & Dean
*
* Esta clase abstracta representa a los empleados asalariados.
*******************************************/

public class Salaried3 extends Employee3
{
  protected double salary;

  //*****************************************

  public Salaried3(String name, double salary)
  {
    super(name);
    this.salary = salary;
  } // end constructor

  //*****************************************

  public double getPay()
  {
    double pay = salary;

    pay -= getFICA(pay);
    return pay;
  } // end getPay
} // end class Salaried3
```

Esto permite acceso directo desde las clases descendientes.

Esto llama al método `protected` en la parte superior del subárbol.

Figura 13.21 Versión mejorada de la clase `Salaried` que incluye la deducción del impuesto.

orientado arbitrariamente. En la figura 13.22 se muestra un controlador para una clase que muestra ese objeto. En la sección de declaraciones, el constructor `JFrame` instancia una ventana denominada `frame`. Las llamadas ulteriores al método, `setSize` y `setDefaultCloseOperation`, establecen el tamaño de esa ventana en pixeles, así como lo que debe ocurrir cuando el usuario hace clic en el recuadro `X` que aparece en su ángulo superior derecho.

Ahora, observe los dos desplegados para el usuario. En este programa se usa un sistema de coordenadas esféricas. En este tipo de sistema de coordenadas, la elevación es un ángulo que es como la latitud. Una entrada de elevación cero indica que el cilindro debe ser plano, con su eje señalando hacia el ecuador. Una elevación de más o menos 90 grados indica que el cilindro debe ser vertical, con su eje apuntando ya sea al polo norte o al polo sur. El azimut es un ángulo parecido a la longitud este. Con elevación igual a cero, una entrada de azimut cero indica que el eje del cilindro debe apuntar justo hacia el observador. Una entrada de azimut positiva indica que el cilindro debe apuntar hacia la derecha, y una entrada de azimut negativa indica que el cilindro debe apuntar hacia la izquierda. La llamada al constructor `Cylinder` instancia al objeto `Cylinder`, y la llamada ulterior al método `add` coloca ese objeto en la ventana. La llamada al método `setVisible` hace visible el contenido de la ventana. En la figura 13.23 se muestra la imagen resultante para la entrada de los ángulos $-15°$ de elevación y $+60°$ de azimut.

La clase que define esta forma y que describe cómo colorearla se muestra en las figuras 13.24*a*, 13.24*b* y 13.24*c*. En la figura 13.24*a* observe que la clase `Cylinder` extiende la clase `JPanel`, que es importada de API de Java. Las variables de instancia incluyen los atributos básicos del objeto: su altura (`cylH`) y diámetro (`cylD`). Éstos son valores en pixeles, que intrínsecamente son enteros, aunque se declaran como `double`. ¿Por qué se usa `double` en lugar de `int`? Las gráficas tridimensionales suelen implicar una cantidad considerable de cálculo. El hecho de declarar a las variables como `double` obliga

```
/*************************************************************
* CylinderDriver.java
* Dean & Dean
*
* Esto controla a la clase Cylinder.
*************************************************************/

import java.util.Scanner;
import javax.swing.*;    // for JFrame and JPanel

public class CylinderDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    JFrame frame = new JFrame("Three-Dimensional Cylinder");
    Cylinder cylinder;
    double elev;    // ángulo de elevación del eje del cilindro en grados.
    double azmuth;  // ángulo de azimuth del eje del cilindro en grados.

    frame.setSize(600, 600);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    System.out.print("Enter axis elevation (-90 to +90): ");
    elev = stdIn.nextDouble();
    System.out.print("Enter axis azmuth (-90 to +90): ");
    azmuth = stdIn.nextDouble();
    cylinder = new Cylinder(elev, azmuth);
    frame.add(cylinder);
    frame.setVisible(true);
  } // end main
} // end CylinderDriver class
```

Sesión muestra:

```
Enter axis elevation (-90 to +90): -15
Enter axis azmuth (-90 to +90): 60
```

Figura 13.22 Controlador para la clase `Cylinder` en las figuras 13.24*a*, 13.24*b* y 13.24*c*.

a la promoción automática de todos los factores `int` que pudieran aparecer en las expresiones. Así se sigue la pista de información fraccionaria y se logra obtener la mejor representación visual posible.

Las otras variables de instancia describen atributos de la imagen mostrada: sus ángulos de orientación y sus extremos de iluminación. Para no complicar las cosas, el programa usa iluminación "blanca", que contiene valores idénticos para los componentes rojo, verde y azul. Las variables `c1` y `c2` representan, respectivamente, la intensidad de estos componentes para superficies sombreadas e iluminadas directamente. El programa declara que estas variables son de tipo `float` porque ése es el tipo de los parámetros en el constructor `Color` API de Java. Cero corresponde a oscuro, y la unidad corresponde a blanco puro, de modo que los valores especificados `c1` y `c2` corresponden a dos tonos distintos de gris: el más oscuro y el más claro se observan en los lados curvos del cilindro en la figura 13.23.

Observe el constructor `Cylinder` en la figura 13.24*a*. Esto transforma los ángulos de elevación y de azimut de grados a radianes y también restringe las magnitudes de los ángulos de entrada a menos de 90 grados. Así se evitan resultados ilegítimos y permite que el usuario vea sólo un extremo del cilindro, pero puesto que el otro extremo es igual, el programa sigue mostrando todo lo de interés.

Ahora observe la figura 13.24*b*, que contiene la primera parte de un gran método `paintComponent`. La JVM llama automáticamente al método `paintComponent` cuando el programa se ejecuta por primera vez y siempre que un usuario hace algo para modificar el contenido de la ventana del programa. (Por ejemplo, si un usuario maximiza una ventana, la JVM llama al método `paintComponent`

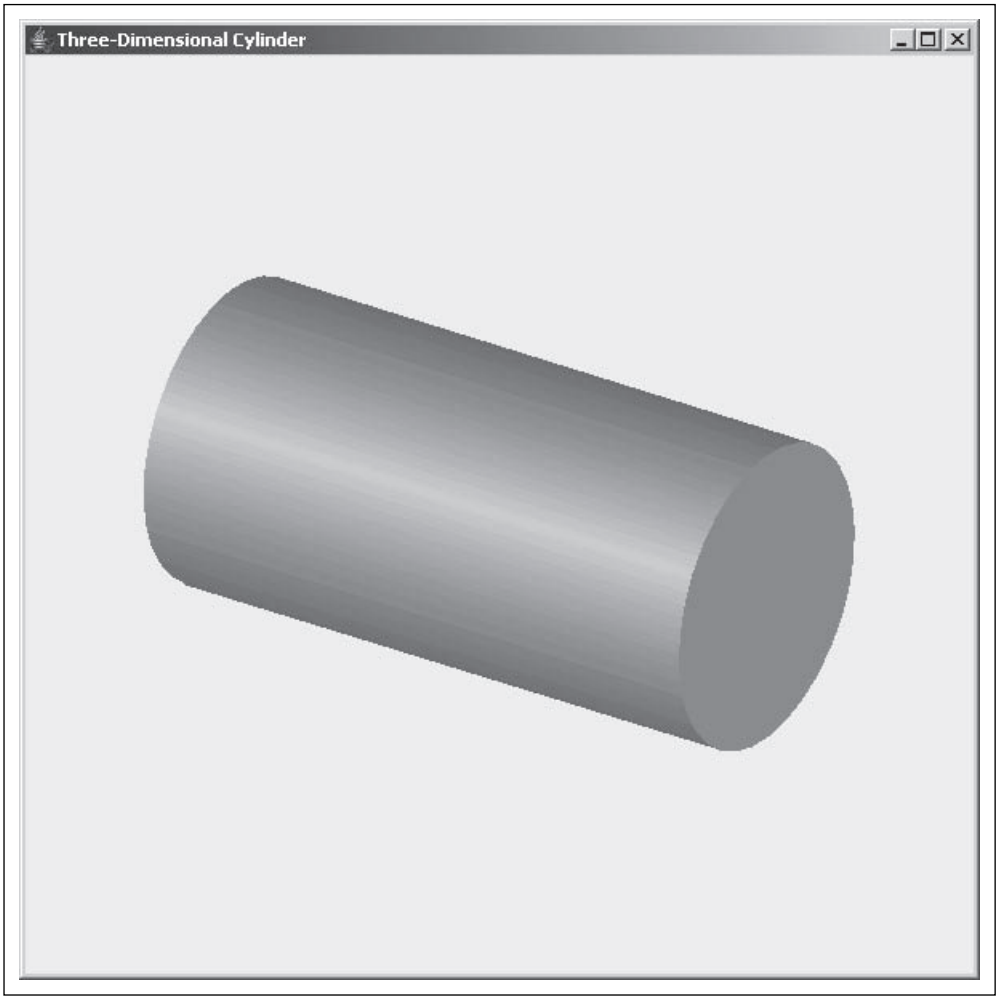


Figura 13.23 Representación tridimensional de un cilindro sólido.

del programa.) El método `paintComponent` definido aquí, se sobrepone a un método `paintComponent` definido en la clase `JComponent`, que es la superclase de la clase `JPanel` y en consecuencia un ancestro de la clase `Cylinder`. Para asegurar que los componentes gráficos estén pintados correctamente, en caso de que alguna vez se implemente un método `paintComponent` sobrepuesto, siempre debe llamarse al método `paintComponent` que la superclase define o hereda como la primera línea en el método `paintComponent` sobrepuesto. Éste es el código relevante de la figura 13.24*b*:

```
super.paintComponent(g);
```

Por definición, cuando un método se sobrepone a otro, ambos métodos deben tener exactamente la misma impronta (mismo nombre y misma secuencia de tipos de parámetros). Puesto que el método `paintComponent` en `JComponent` declara el objeto de un parámetro `Graphic`, el método `paintComponent` en `Cylinder` también declara un objeto de un parámetro `Graphic`, denominado `g`. Aun cuando `g` se declara como un objeto `Graphics`, la JVM en realidad pasa un argumento `Graphics2` al parámetro `g`. Lo anterior está bien porque la clase `Cylinder` depende del parámetro `g` para ejecutar operaciones de graficado complicadas que sólo se encuentran en la clase `Graphics2D`.

Como puede recordar el lector por el material del inicio de este capítulo, siempre es legal pasar el argumento de una clase descendiente al parámetro de una clase ancestro. Quizá también recuerde que para llamar a métodos de una clase descendiente con el parámetro de la clase ancestro, primero es necesario asignar el parámetro de la clase ancestro a una variable de la clase descendiente. Éste es el código relevante de la figura 13.24*b*:

```
Graphics2D g2d = (Graphics2D) g;
```

```
/*************************************************************
* Cylinder.java
* Dean & Dean
*
* Esto muestra un cilindro iluminado desde la dirección de visualización.
*************************************************************/

import javax.swing.JPanel;
import java.awt.*;           // for Graphics, Graphics2D, Color
import java.awt.Rectangle;
import java.awt.geom.*;      // for Ellipse2D and GeneralPath
import java.awt.GradientPaint;

public class Cylinder extends JPanel
{
  private double cylElev;    // elevación del eje del cilindro en radianes
  private double cylAzm;     // azimiuth del eje del cilindro en radianes
  private double cylH = 400; // altura del cilindro en pixeles
  private double cylD = 200; // diámetro del cilindro en pixeles
  private float c1 = 0.3f;   // brillantez mínima de la iluminación
  private float c2 = 0.7f;   // brillantez máxima de la iluminación

  //*********************************************************

  public Cylinder(double elev, double azimuth)
  {
    cylElev = Math.toRadians(elev);
    if (Math.abs(cylElev) >= Math.PI / 2.0)
    {
      cylElev = Math.signum(cylElev) * Math.PI / 2.0001;
    }
    cylAzm = Math.toRadians(azimuth);
    if (Math.abs(cylAzm) >= Math.PI / 2.0)
    {
      cylAzm = Math.signum(cylAzm) * Math.PI / 2.0001;
    }
  }
```

herencia de la clase API `JPanel`

Figura 13.24*a* Clase `Cylinder`, parte A.

Observe que todo en la figura 13.24*b* es como un tipo de declaración. Como antes, el programa usa `double` para las posiciones `MIDX` y `MIDY` en pixeles para preservar información fraccionaria. Para uso posterior, convierte el parámetro `Graphics` de entrada en una referencia `Graphics2D` más específica. Después de declarar las variables de trabajo, `imageRotAngle`, `shape` y `c`, comienza una secuencia de declaraciones iniciadas. Estas declaraciones realmente implementan la mayor parte de los cálculos del método. El uso de declaraciones para implementar pasos secuenciales en un cálculo proporciona autodocumentación. Hace más fáciles de comprender los cálculos tediosos al otorgar nombres comprensibles a las variables intermedias.

Usted puede aplicar sus conocimientos de geometría y trigonometría para comprobar los cálculos del ángulo aparente, la altura aparente del cilindro y el diámetro menor de los extremos elípticos de un cilindro que está orientado como lo especifica la entrada. La declaración bajo "//Formas de los lados curvos y extremos ovalados" usa estos valores para definir un rectángulo y dos elipses. Los objetos `rectangle` y `backEllipse` ayudan a configurar los lados del cilindro en un objeto `GeneralPath` denominado `shape`, y el objeto `frontEllipse` permite que el programa pinte el extremo visible del cilindro.

```
//*******************************************************
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  final double MIDX = 0.5 * getWidth();
  final double MIDY = 0.5 * getHeight();
  Graphics2D g2d = (Graphics2D) g;
  double imageRotAngle;                // ángulo de rotación de la imagen
  GeneralPath shape;                   // lado curvo del cilindro
  float c;                             // nivel actual de color

  // Ahusamiento aparente del cilindro
  double tipCosine = Math.cos(cylAzm) * Math.cos(cylElev);
  double tipSine = Math.sqrt(1.0 - tipCosine * tipCosine);
  double frontEndAngle =
    Math.acos(tipCosine) * 2.0 / Math.PI;

  // Diámetro menor de los óvalos extremos & altura aparente del cilindro
  double minorD = cylD * tipCosine;
  double apparentH = cylH * tipSine;

  // Formas de los lados curvos y óvalos extremos
  Rectangle rectangle = new Rectangle(
    (int) Math.round(MIDX - cylD / 2),
    (int) Math.round(MIDY - apparentH / 2),
    (int) Math.round(cylD),
    (int) Math.round(apparentH));
  Ellipse2D.Double frontEllipse = new Ellipse2D.Double(
    (int) Math.round(MIDX - cylD / 2),
    (int) Math.round(MIDY - apparentH / 2 - minorD / 2),
    (int) Math.round(cylD),
    (int) Math.round(minorD));
  Ellipse2D.Double backEllipse = new Ellipse2D.Double(
    (int) Math.round(MIDX - cylD / 2),
    (int) Math.round(MIDY + apparentH / 2 - minorD / 2),
    (int) Math.round(cylD),
    (int) Math.round(minorD));

  // Color para los lados del cilindro
  GradientPaint gradientPaint = new GradientPaint(
    (float) (MIDX - cylD / 2), 0.0f, new Color(c1, c1, c1),
    (float) (MIDX), 0.0f, new Color(c2, c2, c2), true);
```

La clase del objeto pasado debe estar en el subárbol `Graphics2D`.

Figura 13.24*b* Clase `Cylinder`, parte B.

La última declaración en la figura 13.24*b* instancia al objeto `GradientPaint`, que establece una gradación del color que hace que los lados del cilindro parezcan redondos. En el intervalo entre x = (`MIDX – clyD/2`) y x = (`MIDX`), esto crea 16 bandas verticales estrechas cuyo color varía linealmente desde una intensidad dada por `c1` a la izquierda y una intensidad dada por `c2` a la derecha. El argumento `true` hace que el método vaya de la parte sombreada hasta nuevamente el intervalo entre x = (`MIDX`) y x = (`MIDX + clyD/2`).

Observe que el rectángulo y ambas elipses están definidos con una orientación vertical, ¡pero el cilindro en la figura 13.23 no es vertical! Está orientado a un ángulo oblicuo hacia la derecha. Esto requiere una rotación. Ahora observe la figura 13.24*c*. La declaración subordinada en la parte `else` de la declaración `if` calcula la cantidad de rotación. La llamada al método `rotate` de `Graphics2D` después de la cláusula `else` indica a la computadora cómo ejecutar esta rotación alrededor del punto medio de la ventana. Sin embargo, la rotación no ocurre en realidad sino hasta después que se ha pintado el objeto.

```
      // Imagen rotada respecto al centro de rotación vertical
      if (cylElev == 0.0)
      {
        imageRotAngle =
          Math.signum(Math.sin(cylAzm)) * Math.PI / 2.0;
      }
      else
      {
        imageRotAngle =
          Math.atan(Math.sin(cylAzm) / Math.tan(cylElev));
        if (Math.tan(cylElev) < 0)
        {
          imageRotAngle += Math.PI;
        }
      }
      g2d.rotate(imageRotAngle, MIDX, MIDY);

      // Define y pinta los lados curvos del cilindro
      shape = new GeneralPath(rectangle);
      shape.append(backEllipse, false);
      g2d.setPaint(gradientPaint);
      g2d.fill(shape);

      // pinta el extremo visible del cilindro
      c = c2 - (float) ((c2 - c1) * frontEndAngle);
      g2d.setColor(new Color(c, c, c));
      g2d.fill(frontEllipse);
    } // end paint
  } // end Cylinder class
```

El tipo del parámetro es `InterfaceShape`, que implementa `Rectangle`.

El tipo del parámetro es `InterfacePaint`, que implementa `GradientPaint`.

Figura 13.24*c* Clase `Cylinder`, parte C.

El siguiente paso es instanciar un objeto `GeneralPath` y asignarlo a la variable de referencia denominada `shape`. Inicialmente, esta forma no es otra cosa que el `rectangle` definido previamente. Luego, el método `append` de `shape` agrega la componente `backEllipse`, de modo que ahora `shape` incluye todo ya sea en el rectángulo o en la elipse trasera.

Antes de pintar realmente, el programa debe especificar el esquema de coloreado con una llamada al método `setPaint`. El parámetro `setPaint` debe ser una referencia a un objeto que implementa la interfaz `Paint`. Si se consulta la documentación API para la clase `GradientPaint`, se verá que, en efecto, implementa la interfaz `Paint`. Puesto que el objeto `gradientPaint` es una instancia de la clase `GradientPaint`, y puesto que la clase `GradientPaint` implementa la interfaz `Paint`, el programa puede llamar a `setPaint` con una referencia al objeto `gradientPaint` como argumento. En este punto, el programa está listo para pintar los lados del cilindro con la llamada al método `fill` (`shape`). Esto indica a la computadora que debe ejecutar el método previamente especificado de pintura en la forma previamente especificada, y luego rotar el resultado como ya se especificó.

Las tres últimas declaraciones en el programa pintan el extremo visible del cilindro. La primera declaración calcula la intensidad con base en el ángulo del extremo visible, usando un sombreado más tenue cuando se observa directamente el extremo y un sombreado más intenso cuando se observa un ángulo de inclinación. La llamada al método `setColor` cambia el modo de pintura desde el gradiente de pintura previamente definido a un gris plano que tiene la intensidad recientemente calculada. La llamada final al método `fill` (`frontEllipse`) indica a la computadora pintar la forma de la elipse frontal y luego rotar el resultado como se había especificado previamente.

La figura 13.24*c* contiene varios ejemplos donde los tipos de parámetros de entrada API son interfaces, más que clases. El parámetro del constructor `GeneralPath` es de tipo `Shape`, donde `Shape` es una interfaz. El objeto denominado `rectangle` se ajusta a esto porque su clase, `Rectangle`, implementa la

interfaz `Shape`. El primer parámetro en el método `append` de `GeneralPath` y el parámetro en el método `fill` de `Graphics2D` también son de tipo `Shape`. Los objetos llamados `backEllipse`, `shape` y `frontEllipse` se ajustan porque sus dos clases, `Ellipse2D` y `GeneralPath`, también implementan la interfaz `Shape`. El método `setPaint` de `Graphics2D` recibe un parámetro de tipo `Paint`, donde `Paint` es otra interfaz. El objeto llamado `gradientPaint` se ajusta a lo anterior porque su clase, `GradientPaint`, implementa la interfaz `Paint`. ¡Todas estas referencias son polimórficas!

Resumen

- La clase `Object` es el ancestro de todas las otras clases.
- Para evitar el uso del método `equals` de la clase `Object`, para cada una de sus clases, debe definir un método `equals` que compare valores de variables de instancia.
- Para evitar la misteriosa respuesta de la clase `Object` a una llamada del método `toString`, cada una de sus clases, es necesario que defina un método `toString` que produzca una concatenación de cadena de valores de variables de instancia.
- En tiempo de compilación, el compilador confirma que una clase de variable de referencia es capaz de manejar de alguna forma cada una de las llamadas al método de la variable de referencia. En tiempo de ejecución, la JVM busca el tipo particular del objeto referido por la variable de referencia para determinar a cuál de los varios métodos polimórficos alternativos debe llamar realmente, y vincula el objeto con ese método.
- El operador `instanceof` permite determinar de manera explícita si el objeto referido por una variable de referencia es una instancia de una clase particular o es un descendiente de esa clase.
- Siempre es posible asignar un objeto a una variable de referencia más genérica, porque los métodos del objeto incluyen métodos heredados de la clase de la variable de referencia.
- Un tipo de referencia más genérico puede cambiarse con seguridad a un tipo de referencia más específico sólo si se sabe que el objeto actual referido es tanto o más específico que el tipo.
- El polimorfismo puede implementarse en un arreglo de objetos heterogéneos al declarar los elementos del arreglo como instancias de un ancestro de herencia común. Para satisfacer al compilador, puede escribirse un método inválido en esa clase ancestro y sobreponerlo en todas las clases instanciadas en el arreglo. O bien, es posible declarar el método en una clase ancestro `abstracta` y luego definir métodos de sobreposición en todas las clases instanciadas en el arreglo. O bien, es posible declarar el método en una interfaz e implementar esa interfaz en todas las clases instanciadas en el arreglo.
- Una clase puede extender una superclase heredada y/o implementar cualquier número de interfaces.
- Una interfaz proporciona acceso simple a constantes comunes.
- El modificador de acceso `protected` proporciona acceso directo a miembros de clases en el mismo paquete o en el subárbol de herencia cuya raíz es la clase en que se declara el miembro protegido.
- Con ayuda de cálculos trigonométricos explícitos, es posible usar las clases API JVM de Java para dibujar lo que parecen ser objetos en tres dimensiones.

Preguntas de revisión

§13.2 La clase `Objeto` y promoción automática de tipos

1. Si se desea que una clase defina heredar métodos de la clase `Objeto`, es necesario agregar el sufijo `extends Object` al encabezado de la clase. (F/C)

§13.3 El método `equals`

2. Cuando se comparan variables de referencia, el operador == funciona igual que el método `equals` de la clase `Objeto`. (F/C)
3. ¿Qué compara el método `equals` definido en la clase `String`?

§13.4 El método `toString`

4. ¿Qué devuelve el método `toString` de la clase `Objeto`?

5. ¿Qué está mal al sustituir la declaración `println` en la figura 13.2 del método `main` con estas dos declaraciones?

```
String description = car.toString();
System.out.println(description);
```

6. El tipo de retorno de un método sobrecargado debe ser el mismo que el tipo de retorno del método sobrepuesto. (F/C)

§13.5 Polimorfismo y vinculación dinámica

7. En Java las llamadas a métodos polimórficos están limitadas a definiciones de métodos en el tiempo de compilación (no en el de ejecución). (F/C)

§13.6 Asignaciones entre clases en una jerarquía de clases

8. Suponga que la clase de una variable de referencia desciende de otra clase de una variable de referencia. Para poder asignar una variable de referencia a la otra (sin usar un operador tipo cast), la clase de una variable de referencia del lado izquierdo debe ser una _____ de la clase de una variable de referencia del lado derecho.

§13.7 Polimorfismo con arreglos

9. Un arreglo dado puede contener elementos de tipo variable. (F/C)

§13.8 Métodos y clases `abstractas`

10. ¿Cuáles son las características de la sintaxis de un método `abstracto`?
11. Cualquier clase que contiene un método `abstracto` debe declararse como una clase `abstracta`. (F/C)
12. No es posible instanciar una clase `abstracta`. (F/C)

§13.9 Interfaces

13. Es posible usar una interfaz para proporcionar acceso directo a un conjunto común de constantes desde muchas clases diferentes. (F/C)
14. Es posible declarar variables de referencia para tener un tipo de interfaz y usarlas justo como se usan las variables de referencia declaradas como el tipo de una clase en una jerarquía de la herencia. (F/C)

§13.10 El modificador de acceso `protected`

15. Describa el acceso proporcionado por el modificador `protected`.
16. Es ilegal usar `private` para cualquier método que se sobreponga en un método `abstracto`. (F/C)

Ejercicios

1. [Después de §13.3] Escriba un método `sameColorAs` para la clase `Car` en la figura 13.1. Debe devolver `true` si los colores comparados de los automóviles son los mismos, sin importar sus otros atributos.
2. [Después de §13.4] Escriba la salida producida por el programa en la figura 13.2.
3. [Después de §13.4] ¿Cuál es la salida del siguiente programa? Para cada salida perro, describa cómo se genera la salida (sea específico).

```
public class Animal
{
  public static void main(String[] args)
  {
    Animal sparky = new Dog();
    Animal lassie = new Animal();

    System.out.println(
      "sparky = " + sparky + "\tlassie = " + lassie);
  } // end main
} // end Animal

class Dog extends Animal
{
  public String toString()
  {
```

```
    return "bark, bark";
  }
} // end class Dog
```

4. [Después de §13.4] ¿Qué ocurre si se agrega un objeto de una clase que no define un método `toString` a una `ArrayList`, y luego se intenta imprimir la `ArrayList`? (Suponga que la clase del objeto es una clase definida por el programador que no tiene una frase `extends` en su encabezado.)
5. [Después de §13.5] ¿Por qué la vinculación dinámica a menudo se denomina *vinculación tardía*?
6. [Después de §13.6] Dado: `Animal` = superclase, `Dog` = subclase.
   Identifique todos los errores de compilación en el fragmento de código. Proporcione un mensaje de error de compilación si así lo desea, pero no es necesario.

```
Animal animal;
Dog fido, sparky = new Dog();
animal = sparky;
fido = new Animal();
```

7. [Después de §13.6] Suponga que se tiene un objeto denominado `thing`, pero no se tiene la certeza de qué tipo es, y se desea que el programa imprima el tipo. La clase `Objeto` (¡y en consecuencia, cualquier clase!) tiene otro método, `getClass`, que devuelve un objeto especial de tipo `Class` que contiene información sobre la clase del objeto que llama al método `getClass`. La clase `Class` tiene un método denominado `getName` que devuelve el nombre de la clase descrita por el objeto que llama. Escriba una declaración que imprima el nombre de la clase de `thing`.
8. [Después de §13.7] Dado: `Animal` = superclase, `Dog` = subclase, `Cat` = subclase.
   En el siguiente fragmento de código, las dos líneas en la parte inferior generan errores de tiempo de compilación. Escriba versiones corregidas de estas dos líneas. Conserve el espíritu del código original. Por ejemplo, la línea del fondo debe asignar el segundo elemento de `animals` a la variable `fluffy`.

```
Animal[] animals = new Animal[20];
animals[0] = new Dog();
animals[1] = new Cat();
Dog lassie = animals[0];
Cat fluffy = animals[1];
```

9. [Después de §13.8] Todo método `abstracto` en una superclase debe ser sobrepuesto por un método correspondiente en toda clase `no abstracta` descendiente de ésta. (F/V)
10. [Después de §13.8] Dado el programa Pets a continuación, escriba una clase `abstracta` `Animal2` que contenga justo un artículo: una declaración `abstracta` para un método `speak`. Escriba clases `Dog2` y `Cat2` que extiendan `Animal2`, de modo que cuando se ejecute el programa Pets3 y se introduzca ‘c’ o ‘d’, el programa imprima “¡Miau! ¡Miau!” o “¡Guau! ¡Guau!”

```
import java.util.Scanner;

public class Pets3
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Animal2 animal;

    System.out.print("Which type of pet do you prefer?\n" +
      "Enter c for cats or d for dogs: ");
    if (stdIn.nextLine().charAt(0) == 'c')
    {
      animal = new Cat2();
    }
    else
    {
      animal = new Dog2();
    }
    animal.speak();
  } // end main
} // end Pets3 class
```

11. [Después de §13.9] Vuelva a escribir la interfaz `Comisión` mostrada en la figura 13.14 para mostrar explícitamente los modificadores `abstract`, `public`, `static` y `final` en todos los sitios en que se aplican. (Su definición de interfaz elaborada debe compilar).
12. [Después de §13.9] Cambie la clase `Pets3` en el ejercicio 10 anterior como sigue: Sustituya todas las instancias de `Pets3` y `Pets4`, y sustituya `Animal2` por `Animal3`. Luego, ejecute una interfaz `Animal3` y clases `Dog3` y `Cat3` que implementen `Animal3`, de modo que cuando se ejecute el programa Pets4 y se introduzca ya sea una 'c' o una 'd', el programa imprima "¡Miau! ¡Miau!" o "¡Guau! ¡Guau!"
13. [Después de §13.10] Expanda el código críptico en el método `getFICA` de la clase `Employee3` en la figura 13.20 en declaraciones "if else" de modo que el algoritmo sea más fácil de comprender.

Soluciones a las preguntas de revisión

1. Falso. Toda clase es descendiente de la clase `Objeto`, de modo que no es necesario especificar `extends Object`. De hecho, es indeseable, ya que impide la extensión de alguna otra clase.
2. Cierto.
3. El método `equals` definido en la clase `String` compara una cadena de caracteres.
4. El método `toString` de la clase `Objeto` devuelve una concatenación de cadenas de estos tres componentes:
   - nombre completo de la clase
   - carácter @
   - un valor de código hash hexadecimal
5. Nada. Es una simple cuestión de estilo: si se desea más compacidad o autodocumentación.
6. Cierto.
7. Falso. En tiempo de ejecución, la JVM determina cuál método es llamado.
8. Para poder asignar una variable de referencia a la otra (sin usar un operador tipo cast), la clase de la variable de referencia del lado izquierdo debe ser una superclase/ancestro de la clase de la variable de referencia del lado derecho.
9. Cierto, si cada tipo de elemento es el tipo de elemento definido en la declaración el arreglo o un descendiente de ese tipo (o se ajusta a la interfaz que define el tipo de arreglo: consulte la sección 13.9).
10. Las características de la sintaxis de un método `abstracto` son:
    - El encabezado del método contiene el modificador `abstracto`.
    - Al final del encabezado hay un punto y coma.
    - No hay cuerpo del método.
11. Cierto.
12. Cierto.
13. Cierto.
14. Cierto.
15. Es legal acceder a un miembro `protected`:
    - Desde dentro de la misma clase que el miembro `protected`.
    - Desde dentro de una clase que desciende del miembro `protected`.
    - Desde dentro del mismo `paquete`.
16. Cierto. Un método `abstracto` debe ser `public` o `protected` (no puede ser `private`). Un método que se sobrepone no debe ser más restrictivo que su método sobrepuesto. En consecuencia, si un método se sobrepone a un método `abstracto`, no puede ser `private`.

CAPÍTULO 14

Manejo de excepciones

Objetivos

- Comprender qué es una excepción.
- Usar bloques `try` y `catch` para validación de entrada numérica.
- Comprender cómo los bloques `catch` atrapan una excepción.
- Explicar la diferencia entre excepciones comprobadas y no comprobadas.
- Buscar detalles de excepciones en el sitio API Java de Sun en la red.
- Atrapar excepciones con la clase genérica `Exception`.
- Usar el método `getMessage`.
- Atrapar excepciones con bloques `catch` múltiples.
- Comprender los mensajes de excepción.
- Propagar excepciones de vuelta al módulo que llama con ayuda de una cláusula `throws`.

Relación de temas

14.1 Introducción
14.2 Revisión de excepciones y mensajes de excepción
14.3 Utilización de los bloques `try` y `catch` para manejo de llamadas "peligrosas" a métodos
14.4 Ejemplo de trazado lineal
14.5 Detalles de los bloques `try`
14.6 Dos categorías de excepciones: comprobadas y no comprobadas
14.7 Excepciones no comprobadas
14.8 Excepciones comprobadas
14.9 La clase `Exception` y su método `getMessage`
14.10 Bloques `catch` múltiples
14.11 Comprensión de los mensajes de excepción
14.12 Utilización de `throws` <*tipo-excepción*> para posponer el `catch`
14.13 Apartado GUI y solución de problemas: ejemplo revisitado de trazado de líneas (opcional)

14.1 Introducción

Como se sabe, los programas algunas veces generan errores. Los errores de tiempo de compilación tienen que ver con una sintaxis incorrecta, como olvidar escribir entre paréntesis una condición de declaración `if`. Los errores de tiempo de ejecución tienen que ver con códigos que se comportan de manera incorrecta, como intentar dividir entre cero. En capítulos previos, los errores de tiempo de compilación se repararon al corregir la sintaxis errónea, y los errores de tiempo de ejecución se repararon al robustecer el código. En este capítulo los errores se abordan usando una técnica diferente: el manejo de excep-

ciones. Las excepciones se describirán más formalmente después, pero por el momento considere que una excepción es un error, o simplemente algo que va mal en un programa. El manejo de excepciones es una forma elegante de tratar con tales problemas.

Este capítulo empieza con el estudio de un problema común: asegurarse de que los usuarios introducen un número válido cuando se les pide una entrada numérica. Se aprenderá a implementar esta validación de entrada usando bloques `try` y `catch`, dos de los constructos clave para el manejo de excepciones. Se trata de tipos de excepciones diferentes, y se aprenderá a tratar idóneamente con los tipos diferentes. En la sección final del capítulo, el manejo de excepciones se usará como parte de un programa GUI de trazado de líneas.

Para comprender este capítulo, el lector debe conocer los fundamentos de la programación orientada a objetos, y los fundamentos de arreglos y herencia. Así, el lector debe leer el capítulo 12. Este capítulo no depende del material cubierto en el capítulo 13.

Nos damos cuenta de que los lectores quieran leer algunas partes de este capítulo (*Manejo de excepciones*) y del capítulo siguiente (*Archivos*). Si el lector piensa leer el siguiente capítulo, primero debe leer todo éste pues el tema que se trata en el siguiente capítulo, manipulación de archivos, depende mucho del manejo de excepciones. Por otra parte, si el lector piensa omitir el siguiente capítulo y pasar directamente a los capítulos 16 y 17 (Programación GUI), entonces sólo necesita leer la primera parte de este capítulo, de la sección 14.1 a la sección 14.7.

14.2 Revisión de excepciones y mensajes de excepción

Como se define en Sun,[1] una *excepción* es un evento que trastorna el flujo normal de instrucciones durante la ejecución de un programa. El *manejo de excepciones* es una técnica para manejar con elegancia tales excepciones.

Las primeras excepciones que se analizarán tratan sobre entradas inválidas del usuario. ¿Alguna vez se le ha caído un programa al lector debido a entradas inválidas del usuario (haciendo que el programa termine grotescamente)? Si un programa llama al método `nextInt` de la clase `Scanner` y un usuario introduce un no entero, la Java Virtual Machine (JVM) genera una excepción, despliega un espantoso mensaje de error y termina el programa. A continuación se presenta una sesión de muestra que ilustra esto:


```
Enter an integer: 45.6
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:819)
        at java.util.Scanner.next(Scanner.java:1431)
        at java.util.Scanner.nextInt(Scanner.java:2040)
        at java.util.Scanner.nextInt(Scanner.java:2000)
        at Test.main(Test.java:11)
```

entrada del usuario

una excepción

mensaje de excepción

Observe la excepción `InputMismatchException` anterior. Ése es el tipo de excepción que se genera cuando un usuario introduce un no entero como respuesta a una llamada a un método `nextInt`. Observe el *mensaje de excepción*. Los mensajes de excepción pueden ser molestos, pero su función es útil. Proporcionan información sobre lo que ha estado mal. Hacia el final de este capítulo se abordan en detalle los mensajes de excepción. Pero antes hay una cuestión más importante: cómo evitar en primer lugar mensajes de excepción espantosos. Comenzamos.

14.3 Utilización de los bloques `try` y `catch` para manejo de llamadas "peligrosas" a métodos

Algunas llamadas a métodos, como `nextInt`, son peligrosas porque pueden conducir a excepciones, y las excepciones pueden ocasionar que el programa falle. Por cierto, "peligrosas" no es un término estándar para el manejo de excepciones, pero aquí se usa porque ayuda en las explicaciones. En esta sección se

[1] Sun Microsystems. "The Java Tutorial. Handling Errors with Exceptions", que puede consultarse en Internet en el sitio http://java.sun.com/docs/books/tutorial/essential/exceptions/

describe cómo usar bloques `try` y `catch` para atajar mensajes de excepción y fallas del programa. Se usará un bloque `try` para "intentar" una o más llamadas peligrosas a métodos. Si hay algún problema con la(s) llamada(s) peligrosa(s) al método, la JVM salta a un bloque `catch` y la JVM ejecuta las declaraciones encerradas por el bloque `catch`. Estableciendo una analogía, un bloque `try` es como un acto de trapecio en un circo. En un acto circense así hay uno o más saltos peligrosos (por ejemplo, un salto triple, un salto con triple giro, etc.). Estas suertes peligrosas son como las llamadas peligrosas a un método. Si algo ocurre mal en uno de los saltos y uno de los trapecistas cae, hay una red que detiene su caída. En forma semejante, si algo ocurre mal con una de las llamadas peligrosas a un método, el control pasa a un bloque `catch`. Si nada malo ocurre en los saltos ejecutados en el trapecio, la red no se utiliza en absoluto. En forma semejante, si nada malo ocurre con las llamadas peligrosas a un método, el bloque `catch` no se usa en absoluto.

Sintaxis y semántica

Ésta es la sintaxis para los `bloques try` y `catch`:

```
try
{
   <statement(s)>
}
catch (<exception-class> <parameter>)
{
   <error-handling-code>
}
```

Normalmente, una o más de estas declaraciones es una llamada "peligrosa" a un método API o a un constructor.

La clase *exception* debe coincidir con el tipo de excepción que pudiera lanzar el bloque `try`.

Como se muestra arriba, un bloque `try` y su bloque `catch` asociado (o bloques `catch` múltiples, que se estudiarán más adelante) deben ser contiguos. Es posible colocar otras declaraciones antes del bloque `try` o después del (último) bloque `catch`, pero no entre ellos. Observe el parámetro en el encabezado del bloque `catch`. Los parámetros del bloque `catch` se explicarán en el contexto del siguiente programa de ejemplo.

Observe el programa LuckyNumber en la figura 14.1. Observe cómo los bloques `try` y `catch` siguen el patrón de sintaxis mostrado arriba. Dentro del bloque `try`, la llamada al método `nextInt` intenta convertir una entrada del usuario en un entero. Para que la conversión funcione, la entrada del usuario debe contener sólo dígitos y un signo menos precedente opcional. Si la entrada del usuario se ajusta a ese formato, la JVM asigna la entrada del usuario a la variable `num`, omite el bloque `catch` y continúa con el código abajo del bloque `catch`. Si la entrada del usuario no se ajusta a ese formato, ocurre una excepción. Si ocurre una excepción, la JVM sale de inmediato del bloque `try` e instancia un *objeto excepción*: un objeto que contiene información sobre el evento excepción.

En este ejemplo, la JVM instancia un objeto `InputMismatchException`. Luego, la JVM pasa el objeto `InputMismatchException` al parámetro `e` del encabezado del bloque `catch`. Puesto que `e` se declara como una excepción `InputMismatchException`, pero `InputMismatchException` no forma parte del núcleo del lenguaje Java, en la parte superior del programa es necesario incluir:

```
import java.util.InputMismatchException;
```

Después de pasar el objeto excepción al bloque `catch`, la JVM ejecuta el cuerpo del bloque `catch`. En este ejemplo, el bloque `catch` imprime un mensaje "Entrada inválida. . ." y asigna un número aleatorio a la variable `num`. Luego, la ejecución continúa con el código abajo del bloque `catch`.

Lanzamiento de una excepción

Cuando la JVM instancia un objeto excepción, se dice que la JVM *lanza una excepción*. Aquí se preferiría decir "lanza un objeto excepción" en lugar de "lanza una excepción", puesto que la cosa que se lanza es un objeto excepción. Pero a la mayor parte de los programadores no les preocupa la diferencia entre una excepción, que es un evento, y un objeto excepción. En fin. Seguiremos el flujo y usaremos la terminología estándar: lanzamiento de una excepción.

Cuando la JVM lanza una excepción, la JVM busca un bloque `catch` que coincida. Si lo encuentra, lo ejecuta. En caso de no encontrarlo, la JVM imprime el mensaje de excepción del objeto y termina el pro-

```
/*************************************************************
* LuckyNumber.java
* Dean & Dean
*
* Este programa lee el número de la suerte del usuario como un int.
*************************************************************/

import java.util.Scanner;
import java.util.InputMismatchException;

public class LuckyNumber
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int num; // lucky number
    try
    {
      System.out.print("Introduzca su número de la suerte (un entero): ");
      num = stdIn.nextInt();
    }
    catch (InputMismatchException e)
    {
      System.out.println(
        "Entrada inválida. Usted debe proporcionar un número de la suerte.");
      num = (int) (Math.random() * 10) + 1;   // between 1-10
    }
    System.out.println("Su número de la suerte es " + num + ".");
  } // end main
} // end LuckyNumber class
```

Importa `InputMismatchException` para uso a continuación.

El parámetro `e` recibe un objeto `InputMismatchException`.

Sesión muestra 1:

```
Enter your lucky number (an integer): 27
Your lucky number is 27.
```

Sesión muestra 2:

```
Enter your lucky number (an integer): 33.42
Invalid entry. You'll be given a random lucky number.
Your lucky number is 8.
```

Figura 14.1 Programa LuckyNumber que usa bloques `try` y `catch` para una entrada numérica del usuario.

grama. ¿Qué es un "bloque `catch` que coincida"? Un bloque `catch` "coincide" si el tipo de parámetro del encabezado del `catch` es el mismo que el tipo de la excepción lanzada.[2] Por ejemplo, en el programa LuckyNumber, el parámetro `InputMismatchException` coincide con el objeto `InputMismatchException` lanzado por la llamada al método `nextInt`. Así, el bloque `catch` del parámetro `InputMismatchException` es un bloque `catch` que coincide si y cuando la llamada al método `nextInt` lanza una excepción `InputMismatchException`.

Un objeto excepción contiene información sobre el error, incluyendo el tipo del error y una lista de las llamadas al método que condujeron al error. Más adelante se usará algo de la información del objeto excepción, pero por el momento, el objeto excepción es necesario debido a su capacidad para coincidir con el bloque `catch` idóneo.

[2] En realidad, como se observa en la sección 14.9, un bloque `catch` también se considera que coincide si el tipo del parámetro del encabezado es una superclase de la clase de la excepción lanzada.

14.4 Ejemplo de trazado lineal

A continuación se verá cómo `try` y `catch` se usan en el contexto de un programa más complicado. Se empezará por presentar un programa sin bloques `try` y `catch`. Luego se analizará el programa y se determinará cómo puede mejorarse al agregar bloques `try` y `catch`.

Programa LinePlot de primer corte

El programa en la figura 14.2 traza una línea al leer las posiciones coordenadas de una serie de puntos. La mejor forma para manejar lo que hace el programa LinePlot es mostrar una sesión muestra. A continuación, el usuario escoge trazar una línea que va del origen (el punto de partida por defecto) al punto (3,1) y luego al punto (5,2):

Sesión muestra:

```
Enter x & y coordinates (q to quit): 3 1
New segment = (0,0)-(3,1)
Enter x & y coordinates (q to quit): 5 2
New segment = (3,1)-(5,2)
Enter x & y coordinates (q to quit): q
```

Como puede verse, la disposición del programa es muy primitiva: usa texto para representar cada segmento de línea. En un programa real de graficado, para representar la línea se usa el método `lineDraw` de Java. Esto es lo que se hace en el apartado GUI al final de este capítulo. Pero por ahora, las cosas permanecen sencillas y se usa una disposición basada en texto, en lugar de una basada en GUI. De esa forma, es posible mantener la atención en el tema principal de este capítulo: el manejo de excepciones.

Uso de "q" como valor centinela

En el pasado, cuando se introducían números en un ciclo, a menudo el ciclo terminaba con un valor numérico centinela. En este programa se usa una solución más elegante porque como valor centinela permite un "q" no numérico. ¿Cómo es posible leer números y la cadena "q" con la misma declaración de entrada? Utilice cadenas para ambos tipos de entrada: para la "q" y también para los números. Para cada entrada numérica, el programa convierte la cadena de números en un número al llamar al método `parseInt` de la clase `Integer`.

El método `parseInt` de la clase `Integer` se describió en el capítulo 5. El método `parseInt` intenta convertir una cadena dada en un entero. Eso debe sonar conocido: en el programa LuckyNumber, el método `nextInt` de la clase `Scanner` se usó para convertir una cadena dada en un entero. La diferencia es que el método `nextInt` obtiene su cadena de un usuario y el método `parseInt` obtiene su cadena de un parámetro que se ha pasado. Si este parámetro no contiene dígitos y un signo menos opcional, la JVM lanza un `NumberFormatException`. `NumberFormatException` está en el paquete `java.lang`. Puesto que la JVM importa de manera automática el paquete `java.lang`, el programa del lector no necesita una importación explícita para referirse a una `NumberFormatException`.

Validación de entrada

Observe cómo el programa LinePlot llama a `stdIn.next` para leer los valores de las coordenadas `x` y `y` en `xStr` y `yStr`, respectivamente. Luego, el programa intenta convertir `xStr` y `yStr` en enteros al llamar a `Integer.parseInt`. La conversión funciona bien en la medida en que `xStr` y `yStr` contengan dígitos y un signo menos opcional. Pero ¿qué ocurre si el usuario introduce un no entero para `xStr` y `yStr`? Con entrada inválida, el programa falla, como se muestra a continuación:

Sesión muestra:

```
Enter x & y coordinates (q to quit): 3 1.25
Exception in thread "main" java.lang.NumberFormatException: For
input string: "1.25"
...
```

```
/*************************************************************
* LinePlot.java
* Dean & Dean
*
* Este programa traza una línea como una serie de segmentos de línea
* especificados por el usuario.
*************************************************************/

import java.util.Scanner;

public class LinePlot
{
  private int oldX = 0;  // oldX and oldY save previous point
  private int oldY = 0;  // starting point is the origin (0,0)

  //***********************************************************

  // Este método imprime la descripción de un segmento de línea desde
  // el punto previo hasta el punto actual.

  public void plotSegment(int x, int y)
  {
    System.out.println("Segmento nuevo = (" + oldX + "," + oldY +
      ")-(" + x + "," + y + ")");
    oldX = x;
    oldY = y;
  } // end plotSegment

  //***********************************************************

  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    LinePlot line = new LinePlot();
    String xStr, yStr;   // coordenadas del punto en forma de String
    int x, y;            // coordenadas del punto

    System.out.print("Introduzca las coordenadas x & y (q para salir): ");
    xStr = stdIn.next();
    while (!xStr.equalsIgnoreCase("q"))
    {
      yStr = stdIn.next();
      x = Integer.parseInt(xStr);
      y = Integer.parseInt(yStr);
      line.plotSegment(x, y);
      System.out.print("Introduzca las coordenadas x & y (q para salir): ");
      xStr = stdIn.next();
    } // end while
  } // end main
} // end class LinePlot
```

Esto puede generar errores de tiempo de ejecución.

Figura 14.2 Programa LinePlot que traza una línea, primera versión.

Para manejar esta posibilidad, el ciclo `while` volverá a escribirse en el método `main` en la figura 14.2 de modo que incluya validación de entrada usando un mecanismo `try-catch`. El primer paso consiste en identificar el código peligroso. ¿Puede encontrar el código peligroso? Las dos llamadas al método `parseInt` son peligrosas en el sentido de que podrían lanzar una excepción `NumberFormatException`. Así, estas dos declaraciones se colocarán en un bloque *try* y se agregará un bloque `catch` que coincida, como se muestra en la figura 14.3.

Busque problemas potenciales.

```
while (!xStr.equalsIgnoreCase("q"))
{
  yStr = stdIn.next();
  try
  {
    x = Integer.parseInt(xStr);
    y = Integer.parseInt(yStr);
  }
  catch (NumberFormatException nfe)
  {
    System.out.println("Entrada inválida: " + xStr + " " + yStr
      + "\nDebe introducir entero en el espacio entero.");
  }

  line.plotSegment(x, y);
  System.out.print("Introduzca las coordenadas x & y (q para salir): ");
  xStr = stdIn.next();
} // end while
```

Esta declaración debe estar dentro de un bloque `try`.

Figura 14.3 Primer intento por mejorar el ciclo `while` del programa LinePlot.

¿Se observa algún error lógico en el ciclo `while` en la figura 14.3? ¿Qué ocurre si hay una entrada inválida? Se lanza una excepción `NumberFormatException`, misma que es atrapada, y luego se imprime un mensaje de error. A continuación se ejecuta `line.plotSegment`. Pero no es aconsejable imprimir el segmento de línea si los valores de entrada fuesen un desastre. Para evitar esta posibilidad, la línea `line.plotSegment (x, y);` se mueve a la última línea en el bloque `try`. De esta forma, se ejecuta sólo si las dos llamadas al método `parseInt` funcionan correctamente. En la figura 14.4 se muestra la versión final del ciclo `while` del programa LinePlot.

14.5 Detalles de los bloques `try`

Ahora que el lector conoce la idea detrás de los bloques `try`, es tiempo de precisar algunos detalles de estos bloques.

```
while (!xStr.equalsIgnoreCase("q"))
{
  yStr = stdIn.next();
  try
  {
    x = Integer.parseInt(xStr);
    y = Integer.parseInt(yStr);
    line.plotSegment(x, y);
  }
  catch (NumberFormatException nfe)
  {
    System.out.println("Entrada inválida: " + xStr + " " + yStr
      + "\nMust enter integer space integer.");
  }
  System.out.print("Introduzca las coordenadas x & y (q para salir): ");
  xStr = stdIn.next();
} // end while
```

Esta declaración debe estar dentro de un bloque `try`, no después de la estructura `try-catch`.

Figura 14.4 Versión final del ciclo `while` del programa LinePlot.

Tamaño del bloque `try`

Decidir sobre el tamaño de los bloques `try` tiene algo de artístico. Algunas veces es mejor usar bloques `try` pequeños, en otras conviene más usar bloques `try` grandes. Es legal rodear todo el cuerpo del método con un bloque `try`, aunque en general resulta contraproducente porque así es más difícil identificar el código peligroso. En general, debe intentarse que los bloques `try` sean suficientemente pequeños, de modo que sea fácil identificar el código peligroso.

Por otra parte, si se requiere ejecutar varias declaraciones peligrosas relacionadas, es necesario rodear las declaraciones con su propio bloque `try`. Usar bloques múltiples `try` pequeños puede conducir a un código amontonado. Un bloque `try` incluyente puede ayudar a mejorar la legibilidad. El programa LinePlot incluye ambas declaraciones `parseInt` en un solo bloque `try` porque están relacionados conceptualmente y físicamente están próximos uno de otro. Eso mejora la legibilidad.

Suponga que se omiten las declaraciones del bloque `try`

Si se lanza una excepción, la JVM salta de inmediato del bloque `try` actual. La inmediatez del salto significa que si en el bloque `try` hay alguna declaración después de la declaración de lanzamiento de la excepción, estas declaraciones se omiten. El compilador es un pesimista. Sabe que las declaraciones dentro de un bloque `try` podrían omitirse y supone lo peor; es decir, supone que todas las declaraciones dentro de un bloque `try` se omiten. En consecuencia, si hay algún bloque `try` que contenga una asignación a `x`, entonces el compilador asume que se omite la asignación. En caso de que no haya ninguna asignación a `x` fuera del bloque `try` y el valor `x` sea necesario fuera del bloque `try`, se obtiene este error de compilación:

```
variable x might not have been initialized
```

Si se obtiene este error, normalmente es posible repararlo al iniciar la variable antes del bloque `try`. A continuación se considerará un ejemplo. . .

El objetivo consiste en implementar un método `getIntFromUser` que efectúe una implementación robusta para un valor `int`. El desplegado del método pide al usuario un entero, que lea el valor introducido y luego convierta la cadena en un `int`. Si la conversión falla, el método debe volver a solicitar al usuario un entero. Si el usuario termina por introducir un valor entero válido, `getIntFromUser` debe devolverlo al módulo que llama.

La figura 14.5 es un intento de primer corte al implementar el método `getIntFromUser`. Funciona bien con la lógica, aunque contiene errores de compilación debido a las iniciaciones dentro del bloque `try`.

El bloque `try` contiene estas tres líneas:

```
valid = false;
x = Integer.parseInt(xStr);
valid = true;
```

Observe cómo el fragmento de tres líneas asigna `valid` a `false` y luego regresa y le asigna `true`. Raro, ¿no es cierto? En realidad es una estrategia bastante común suponer una cosa, tratar otra y luego cambiar la suposición en caso que esté equivocada. Y eso es lo que ocurre aquí. Este código empieza por suponer que la entrada del usuario es inválida. Llama a `parseInt` para probar si realmente es válido; es decir, comprueba para ver si la entrada del usuario es un entero. Si es válida, se ejecuta la siguiente declaración, y `valid` lleva a fijarla en `true`. Pero ¿qué ocurre que si falla `parseInt`? Pero ¿qué ocurre si falla la conversión `parseInt`? La variable `valid` nunca se establece en `true` porque se ha lanzado una excepción y la JVM salta de inmediato al bloque `try`. De modo que este código parece razonable. Desafortunadamente, "parece razonable" no basta esta vez.

Suponga una cosa y luego cambie según sea necesario.

¿Puede imaginar el lector los errores de tiempo de compilación? En caso negativo, no debe sentirse mal: aquí no los vimos hasta que el compilador nos ayudó. Como se muestra en las llamadas en la figura 14.5, el compilador se queja de que el `valid` y las variables `x` podrían no estar iniciadas. ¿Por qué tanto problema? ¿Es que el compilador no puede ver que el `valid` y el `x` son valores asignados en los bloques `try`? Sí: el compilador puede ver las asignaciones, pero recuerde que el compilador es un pesimista. Su-

```
public static int getIntFromUser()
{
  Scanner stdIn = new Scanner(System.in);
  String xStr;   // user entry
  boolean valid; // is user entry a valid integer?
  int x;         // integer form of user entry

  System.out.print("Enter an integer: ");
  xStr = stdIn.next();

  do
  {
    try
    {
      valid = false;
      x = Integer.parseInt(xStr);
      valid = true;
    }
    catch (NumberFormatException nfe)
    {
      System.out.print("Invalid entry. Enter an integer: ");
      xStr = stdIn.next();
    }
  } while (!valid);

  return x;
} // end getIntFromUser
```

error de tiempo de compilación: la entrada podría no estar iniciada

error de tiempo de compilación: x podría no estar iniciada

Figura 14.5 Método que ilustra el problema cuando se inicia dentro de un bloque `try`.

pone que todas las declaraciones dentro de un bloque `try` se han omitido. Aun cuando se sabe que la declaración `valid = false;` no corre verdadero peligro de ser omitida (se trata de una simple asignación, y es la primera línea en el bloque `try`), el compilador sigue suponiendo que se omite.

¿Cuál es la solución? 1) Mover la asignación `valid = false;` hasta la línea de la declaración `valid`. 2) Iniciar `x` en 0 como parte de la línea de la declaración `x`. La implementación correcta se muestra en la figura 14.6.

14.6 Dos categorías de excepciones: comprobadas y no comprobadas

Las excepciones se ubican en dos categorías: *comprobadas* y *no comprobadas*. Las excepciones comprobadas deben comprobarse con un mecanismo `try-catch`. Las excepciones no comprobadas pueden comprobarse opcionalmente con un mecanismo `try-catch`, aunque no es necesario.

Identificación de una categoría de excepción

¿Cómo puede decirse si una excepción particular se clasifica como comprobada o no comprobada? Una excepción es un objeto, y como tal, está asociado con una clase particular. Para determinar si una excepción particular está comprobada o no comprobada, es necesario encontrar su clase asociada en el sitio Web API de Java.[3] Una vez que se encuentra la clase, deben encontrarse sus ancestros. Si se encuentra que es un descendiente de la clase `RuntimeException`, se trata de una excepción no comprobada. En caso contrario, se trata de una excepción comprobada.

[3] http://java.sun.com/javase/6/docs/api/

```
public static int getIntFromUser()
{
  Scanner stdIn = new Scanner(System.in);
  String xStr;           // user entry
  boolean valid = false; // is user entry a valid integer?
  int x = 0;             // integer form of user entry

  System.out.print("Enter an integer: ");
  xStr = stdIn.next();

  do
  {
    try
    {
      x = Integer.parseInt(xStr);
      valid = true;
    }
    catch (NumberFormatException nfe)
    {
      System.out.print("Invalid entry. Enter an integer: ");
      xStr = stdIn.next();
    }
  } while (!valid);

  return x;
} // end getIntFromUser
```

Estas iniciaciones antes del bloque `try` satisfacen las demandas del compilador.

Figura 14.6 Versión corregida del método `getIntFromUser`.

Por ejemplo, si en el sitio Web API de Java se busca `NumberFormatException`, se verá lo siguiente:

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └ java.lang.RuntimeException
              └java.lang.IllegalArgumentException
                  └ java.lang.NumberFormatException
```

Si observa esta clase en la jerarquía, la excepción no está comprobada.

Esto demuestra que la clase `NumberFormatException` es descendiente de la clase `RuntimeException`, de modo que la clase `NumberFormatException` es una excepción no comprobada.

En la figura 14.7 se muestra la jerarquía de clase para todas las excepciones. Reitera la cuestión de que las excepciones no comprobadas son descendientes de la clase `RunTimeException`. También muestra que algunas excepciones no comprobadas son descendientes de la clase `Error`. En aras de mantener simples las cosas, antes no se mencionó la clase `Error`. Probablemente el lector no encuentre este tipo de excepciones.

Clases de excepción definidas por el programador

Es probable que los programadores definan sus propias clases de excepción. Estas clases de excepción definidas por el programador deben derivarse de la clase `Exception` o de una subclase de la clase `Exception`. En términos generales, es necesario restringirse a clases de excepción predefinidas, ya que las clases de `Exception` definidas por el programador tienden a fragmentar las actividades del manejo de errores, lo cual hace que el programa sea más difícil de comprender.

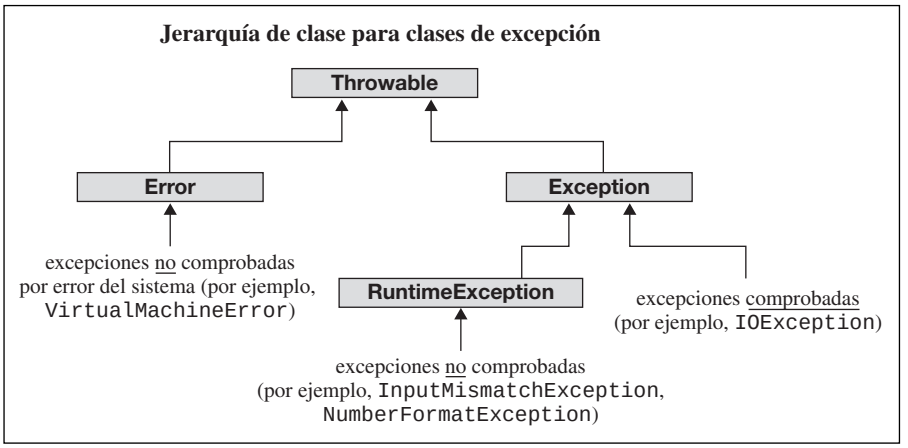


Figura 14.7 Jerarquía de clase de excepción.

14.7 Excepciones no comprobadas

Como se aprendió en la sección previa, no es necesario comprobar las excepciones no comprobadas con un mecanismo de bloque `try-catch`. No obstante, en tiempo de ejecución, si la JVM lanza una excepción no comprobada y no hay ningún bloque `catch` para atraparla, entonces el programa se cae.

Estrategias para manejar excepciones no comprobadas

Si su programa contiene código que pudiera lanzar alguna excepción no comprobada, para tratar con esta situación hay dos estrategias:

1. Usar una estrategia `try-catch`,

o

2. No intentar atrapar una excepción, sino escribir con cuidado el código a fin de evitar la posibilidad de que se lance la excepción.

En el método `getIntFromUser` de la figura 14.6, se utilizó la primera estrategia: se usó una estructura `try-catch` para manejar la peligrosa llamada al método `parseInt`. Normalmente, debe usarse una estructura `try-catch` para analizar sintácticamente las llamadas a los métodos (`parseInt`, `parseLong`, `parseDouble`, etc.) porque esto conduce a soluciones más limpias. En el siguiente ejemplo, la estrategia preferida no es tan evidente. Se usan ambas estrategias y se comparan los resultados.

Ejemplo con `StudentList`

En la figura 14.8 se presenta una clase `StudentList` que administra una lista de estudiantes. La clase administra los nombres de los estudiantes en una `ArrayList` denominada `students`. La clase contiene un constructor para iniciar la lista `students`, un método `display` para imprimir la lista `students`, y un método `removeStudent` que elimina a un estudiante especificado de la lista `students`. El centro de atención es el método `removeStudent`.

⚠ La llamada al método `students.remove` es peligrosa porque podría lanzar una excepción no comprobada, `IndexOutOfBoundsException`. Si su argumento `index` mantiene el índice de uno de los elementos `student`, entonces el elemento se elimina de la `ArrayList` de `students`. Pero si el argumento `index` mantiene un índice inválido, entonces se lanza una excepción `IndexOutOfBoundsException`. Esto ocurre, por ejemplo, si como controlador se utiliza la clase `StudentListDriver`. Observe cómo la clase `StudentListDriver` usa un valor de índice de 6 aun cuando en la lista de estudiantes sólo hay cuatro estudiantes. Las clases `StudentListDriver` y `StudentList` compilan justo bien, pero cuando se ejecuta, la llamada al método `students.remove` lanza una excepción y la JVM termina el programa e imprime el mensaje de error que se encuentra en la parte inferior de la figura 14.9.

```
/*****************************************************
* StudentList.java
* Dean & Dean
*
* Esta clase maneja una ArrayList de estudiantes.
*****************************************************/

import java.util.ArrayList;

public class StudentList
{
  ArrayList<String> students = new ArrayList<String>();

  //************************************************

  public StudentList(String[] names)
  {
    for (int i=0; i<names.length; i++)
    {
      students.add(names[i]);
    }
  } // end constructor

  //************************************************

  public void display()
  {
    for (int i=0; i<students.size(); i++)
    {
      System.out.print(students.get(i) + " ");
    }
    System.out.println();
  } // end display

  //************************************************

  public void removeStudent(int index)
  {
    students.remove(index);   <-- Ésta es una llamada peligrosa a un método.
  } // end removeStudent
} // end StudentList
```

Figura 14.8 Primera elección de la clase `StudentList` que mantiene una lista de estudiantes.

Mejora del método `removeStudent`

Haga programas robustos.

A continuación, el método `removeStudent` se robustecerá mediante el manejo elegante del caso en que se llama con un índice inválido. En las figuras 14.10*a* y 14.10*b* se muestran dos implementaciones robustas diferentes para el método `removeStudent`. La primera implementación usa un mecanismo `try-catch` y la segunda, un código cuidadoso. Éstas son las dos estrategias mencionadas antes para el manejo de excepciones no comprobadas.

¿Cuál solución es mejor: una estructura `try-catch` o un código cuidadoso? Con soluciones que son casi lo mismo en términos de legibilidad, es mejor elegir la implementación con el código cuidadoso porque es más eficiente. El código para manejar excepciones es menos eficiente porque requiere que la JVM instancie un objeto excepción y encuentre un bloque `catch` que coincida.

```
/************************************************************
* StudentListDriver.java
* Dean & Dean
*
* Éste es el controlador de la clase StudentList.
************************************************************/

public class StudentListDriver
{
  public static void main(String[] args)
  {
    String[] names = {"Caleb", "Izumi", "Mary", "Usha"};
    StudentList studentList = new StudentList(names);

    studentList.display();
    studentList.removeStudent(6);
    studentList.display();
  } // end main
} // end StudentListDriver
```

Este valor de argumento genera un error de tiempo de ejecución.

Output:

```
Caleb Izumi Mary Usha
Exception in thread "main" java.lang.IndexOutOfBoundsException:
Index: 6, Size: 4
        at java.util.ArrayList.RangeCheck(ArrayList.java:547)
        at java.util.ArrayList.remove(ArrayList.java:390)
        at StudentList.removeStudent(StudentList.java:43)
        at StudentListDriver.main(StudentListDriver.java:17)
```

Figura 14.9 Controlador de la clase `StudentList`.

14.8 Excepciones comprobadas

A continuación se considerarán las excepciones comprobadas. Si un fragmento de código tiene el potencial para lanzar una excepción comprobada, el compilador obliga al lector a asociar ese fragmento de código con un mecanismo `try-catch`. En caso de que no haya un mecanismo `try-catch` asociado, el compilador genera un error. Con excepciones no comprobadas se tiene una opción para manejarlas: un mecanismo `try-catch` o un código cuidadoso. Con excepciones comprobadas no hay opción: es necesario usar un mecanismo `try-catch`.

```
public void removeStudent(int index)
{
  try
  {
    students.remove(index);
  }
  catch (IndexOutOfBoundsException e)
  {
    System.out.println("No es posible eliminar al estudiante porque " +
      index + " es una posición de índice inválida.");
  }
} // end removeStudent
```

Figura 14.10*a* Uso de una estructura `try-catch` para el método `removeStudent`.

```
public void removeStudent(int index)
{
  if (index >= 0 && index < students.size())
  {
    students.remove(index);
  }
  else
  {
    System.out.println("No es posible eliminar al estudiante porque " +
      index + " es una posición de índice inválida.");
  }
} // end removeStudent
```

Figura 14.10*b* Uso de una estrategia de código cuidadoso para el método `removeStudent`.

Programa CreateNewFile

En la figura 14.11 el programa CreateNewFile intenta crear un archivo vacío con un nombre especificado por el usuario. Los detalles del archivo se cubrirán a fondo en el siguiente capítulo. Puesto que en este

```
/*******************************************************
* CreateNewFile.java
* Dean & Dean
*
* Esto crea un nuevo archivo.
*******************************************************/

import java.util.Scanner;
import java.io.File;
import java.io.IOException;

public class CreateNewFile
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String fileName; // user-specified file name
    File file;

    System.out.print("Introduzca el archivo a crear: ");
    fileName = stdIn.nextLine();
    file = new File(fileName);

    if (file.exists())
    {
      System.out.println("Lo siento, el archivo ya existe.");
    }
    else
    {
      file.createNewFile();
      System.out.println(fileName + " creado.");
    }
  } // end main
} // end CreateNewFile class
```

llamada al constructor API

llamada al método API

llamada al método API

Figura 14.11 Borrador del programa de CreateNewFile que se supone crea un archivo nuevo.

ejemplo los archivos se consideran "desde el exterior", en este momento no es necesario comprender los detalles del archivo. Entonces, ¿por qué se decidió usar un ejemplo de archivo antes del capítulo dedicado a los archivos? Porque se quería presentar un buen ejemplo de excepción comprobada y los programas de archivos lo proporcionan. Hubiera sido más sensato usar material previamente cubierto para el ejemplo de la excepción comprobada, aunque no constituía una opción. Los comandos cubiertos previamente no lanzan excepciones comprobadas.

El programa CreateNewFile solicita al usuario el nombre de un archivo que ha de crearse. Si el archivo ya existe, el programa imprime un mensaje "lo siento, el archivo ya existe". Si el archivo no existe, el programa lo crea. Al hacer todo lo anterior, el programa usa la clase `File` y su Application Programming Interface (API). De manera más específica, el programa llama a un constructor `File`, llama al método `exists` de `File` y llama al método `createFile` de `File`. Así como ocurre con muchas llamadas relacionadas con API, todas estas llamadas poseen el potencial de lanzar una excepción. Entonces, ¿cómo abordar este tema? Continúe leyendo. . .

Uso de documentación API al escribir un código para el manejo de excepciones

Siempre que se desea usar un método o un constructor de una de las clases API y no se tiene la seguridad de hacerlo, es necesario buscar en la documentación API para saber si se debe agregar un código para el manejo de excepciones. En la página de documentación API para el método o constructor de interés, busque una sección "throws", que identifica tipos específicos de excepciones que podrían ser lanzadas. Para manejar las excepciones, es necesario comprenderlas. Para comprender una excepción particular, haga clic en su liga en la sección throws. Así puede acceder a la documentación API para la clase de excepción.

En la página API de la clase de excepción, desplácese por la pantalla de arriba abajo y lea la descripción de la clase de excepción. Vuelva a desplazarse por la pantalla y observe la jerarquía de clase. Como ya se mencionó, si `RuntimeException` es un ancestro, entonces la excepción es una excepción no comprobada. En caso contrario, se trata de una excepción comprobada.

De vuelta al programa CreateNewFile

Si el lector usa la estrategia de buscar en API para crear el programa CreateNewFile, encontrará lo siguiente:

- La llamada al constructor `File` lanza una `NullPointerException` si su argumento es `null`. La clase `NullPointerException` se deriva de la clase `RuntimeException`, de modo que es una excepción no comprobada. El código se ha escrito de modo que no haya peligro en que el argumento del constructor `File` sea `null`, por lo que no es necesario agregar ningún código para la llamada al constructor `File`.
- La llamada al método `exists` lanza una `SecurityException` si existe un administrador de seguridad. La clase `SecurityException` se deriva de la clase `RuntimeException`, de modo que se trata de una excepción no comprobada. Si no se cuenta con un administrador de seguridad, no es necesario agregar ningún código para la llamada al método `exists`.
- La llamada al método `createNewFile` lanza una `IOException` si hay algún problema de E/S como un disco duro corrupto o un nombre de directorio inválido. La clase `IOException` se deriva de la clase `Exception`, pero no de la clase `RuntimeException`, de modo que es una excepción comprobada. Así, para manejar esta excepción se requiere un bloque `try-catch`.

Debido a la llamada al método `createNewFile`, el programa CreateNewFile en la figura 14.11 no compila exitosamente. ¿Cuál es la solución? Suponga que simplemente la llamada al método `createNewFile` se rodea con un bloque como éste:

```
else
{
  try
  {
    file.createNewFile();
  }
  catch (IOException ioe)
```

```
    {
      System.out.println("File I/O error");
    }
    System.out.println(fileName + " created.");
  }
```

Lo anterior resulta en un programa que compila y se ejecuta con éxito. Pero ¿es un buen programa? A menudo, los programadores novatos resuelven los problemas al intentar algo sin pensar exhaustivamente en la situación, y si obtienen resultados razonables, continúan con su estrategia. Intente resistir esta tentación. Aunque el código anterior compila y se ejecuta con éxito, no se comporta correctamente cuando se lanza una `IOException`. ¿Puede el lector identificar el comportamiento erróneo? Si se lanza una `IOException`, el bloque `catch` imprime su mensaje `"File I/O error"`. Pero luego también imprime el mensaje `fileName + "created."`, aun cuando no se haya creado ningún archivo. Recuerde lo siguiente: el hecho de que un programa se ejecute con éxito no significa que esté bien.

No haga nada hasta que esté seguro de la solución.

He aquí la solución preferida:

```
else
{
  try
  {
    file.createNewFile();
    System.out.println(fileName + " creado.");
  }
  catch (IOException ioe)
  {
    System.out.println("File I/O error");
  }
}
```

Esta declaración está ahora en una mejor ubicación.

Ahora el programa imprime el mensaje “creado” sólo si el archivo ha sido creado realmente. ¡Sí!

14.9 La clase `Exception` y su método `getMessage`

Hasta el momento, los ejemplos presentados han sido relativamente sencillos. Cada bloque `try` ha lanzado un solo tipo de excepción. En ese caso, la lógica `catch` es directa: atrapa cada tipo de excepción que se ha lanzado. Para casos en los que se tiene un bloque `try` que pudiera lanzar más de un tipo de excepción, la lógica `catch` puede ser algo más complicada. Es necesario elegir entre estas dos técnicas: 1) contar con un bloque `catch` genérico que maneje todo tipo de excepción que pudiera lanzarse o 2) proporcionar una secuencia de bloques `catch` específicos, uno para cada tipo de excepción que pudiera lanzarse. En esta sección se describe la técnica del bloque `catch` genérico, y en la siguiente sección se describe la técnica de la secuencia de bloques `catch`.

Bloque `catch` genérico

Para contar con un bloque `catch` genérico, es necesario definir un bloque `catch` con un parámetro tipo `Exception`. Luego, dentro del bloque `catch`, debe llamarse al método `getMessage` de la clase `Exception`, como se muestra a continuación:

```
catch (Exception e)
{
  System.out.println(e.getMessage());
}
```

Si un bloque `catch` usa un parámetro `Exception`, coincidirá con todas las excepciones lanzadas. ¿Por qué? Porque cuando se lanza una excepción, busca un parámetro `catch` que sea idéntico a la excepción lanzada o a una superclase de la excepción lanzada. La clase `Exception` es la superclase de todas

las excepciones lanzadas, de modo que todas las excepciones lanzadas consideran que un parámetro Exception catch coincide.

El método getMessage de la clase Exception devuelve una descripción textual de la excepción lanzada. Por ejemplo, si se intenta abrir un archivo usando una nueva llamada al constructor FileReader(String <*filename*>) y se pasa un nombre de archivo por un archivo inexistente, entonces la JVM lanza una excepción y la llamada al getMessage devuelve lo siguiente:

```
<filename> (The system cannot find the file specified)
```

El mensaje despliega el nombre del archivo especificado donde dice <*filename*>. Este mensaje es útil, pero esté alerta porque a veces getMessage devuelve mensajes que no son particularmente útiles.

Ejemplo PrintLineFromFile

A continuación se considerará un ejemplo de un programa completo. El programa PrintLineFromFile en la figura 14.12 abre un archivo especificado por el usuario e imprime la primera línea del archivo. Las llamadas al constructor FileReader y BufferedReader trabajan de manera conjunta para abrir el archivo especificado por el usuario. El método readLine de BufferedReader lee la primera línea del archivo. Luego, el archivo imprime la línea.

Tanto el constructor FileReader como el método readLine lanzan excepciones comprobadas, de modo que si no se hubieran colocado en un bloque try, el compilador se hubiera quejado y hubiera identificado las excepciones que debían ser atrapadas. En particular, si el usuario introduce el nombre de un archivo que no existe, entonces el constructor FileReader lanza una FileNotFoundException. Si el archivo es corrupto e ilegible, el método readLine lanza una IOException. El bloque genérico catch atrapa cualquiera de estas excepciones, y el método getMessage se usa para imprimir una descripción de la excepción lanzada.

En la primera sesión de muestra en la figura 14.12, el usuario introduce una entrada que indica al programa que lea el archivo fuente PrintLineFromFile.java. Puesto que la primera línea del programa es una línea de asteriscos, el programa imprime línea de asteriscos.

En la segunda sesión de muestra en la figura 14.12, el usuario especifica un archivo inexistente. El mal nombre del archivo hace que el constructor FileReader lance un objeto FileNotFoundException, y la llamada al getMessage genera el mensaje de error que se muestra.

Ahora el lector ha visto un ejemplo de un constructor FileReader que lanza una excepción, pero aún no ha visto un ejemplo de un método readLine que lance una excepción. Esto se debe a que es más difícil producir una excepción con la llamada al método readLine. Esto ocurre sólo si se tiene un archivo corrupto que es posible abrir pero no leer, y no es posible generar intencionalmente un archivo así. Aun cuando el método readLine rara vez puede lanzar una excepción, ésta debe colocarse dentro de una estructura try-catch porque el compilador sabe que puede lanzar una excepción comprobada.

14.10 Bloques catch múltiples

Cuando se tiene un bloque try que lanza más de un tipo de excepción. Es posible contar con un bloque genérico catch o una secuencia de bloques catch específicos: uno para cada tipo de excepción que pudiera ser lanzada. A continuación se considerará la técnica de la secuencia de bloques catch.

Otra visita al ejemplo PrintLineFromFile

En la figura 14.13 se muestra PrintLineFromFile2, una versión modificada del programa anterior PrintLineFromFile. En lugar de usar un bloque catch genérico, PrintLineFromFile2 usa una secuencia de bloques catch, un bloque catch para FileNotFoundException y un bloque catch para la IOException. Observe que debido a que este nuevo programa identifica específicamente los tipos de excepciones que pueden lanzarse, debe importar sus clases. Si el programa se ejecuta con un nombre de archivo válido como entrada, se obtiene una impresión de la primera línea de ese archivo, justo como antes. Pero si se suministra un nombre de archivo inválido para la entrada, se obtiene algo como la sesión muestra en la parte inferior de la figura 14.13.

```
/*************************************************************
* PrintLineFromFile.java
* Dean & Dean
*
* Esto abre un archivo de texto existente e imprime una de sus líneas.
*************************************************************/

import java.util.Scanner;
import java.io.BufferedReader;
import java.io.FileReader;

public class PrintLineFromFile
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String fileName;         // nombre del archivo objetivo
    BufferedReader fileIn;   // archivo objetivo
    String line;             // primera línea de fileIn

    System.out.print("Introduzca el nombre de un archivo: ");
    fileName = stdIn.nextLine();

    try
    {
      fileIn = new BufferedReader(new FileReader(fileName));
      line = fileIn.readLine();
      System.out.println("Line 1:\n" + line);
    } // end try

    catch (Exception e)
    {
      System.out.println(e.getMessage());
    }
  } // end main
} // end PrintLineFromFile class
```

Sesión muestra #1:

```
Enter a file name: PrintLineFromFile.java
Line 1:
/*************************************************************
```

Sesión muestra #2:

```
Enter a filename: garbage
garbage (El sistema no puede encontrar el archivo especificado)
```

Figura 14.12 Programa de PrintLineFromFile: un simple lector de archivos.

Orden de bloques `catch`: el orden importa

Si hay múltiples bloque `catch`, el primer bloque `catch` que coincida con el tipo de excepción lanzada es el que se ejecuta. Luego, los otros bloques `catch` se omiten. Este comportamiento es semejante al comportamiento de una declaración `switch`. Pero hay una ligera diferencia. Con una declaración `switch`, luego que se encuentra y ejecuta un bloque `catch` que coincide, el control continúa hacia el

```
/*************************************************************
* PrintLineFromFile2.java
* Dean & Dean
*
* Esto abre un archivo de texto existente e imprime una de sus líneas.
*************************************************************/

import java.util.Scanner;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;

public class PrintLineFromFile2
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String fileName;         // nombre del archivo objetivo
    BufferedReader fileIn;   // archivo objetivo
    String line;             // primera línea de fileIn

    System.out.print("Introduzca el nombre de un archivo: ");
    fileName = stdIn.nextLine();

    try
    {
      fileIn = new BufferedReader(new FileReader(fileName));
      line = fileIn.readLine();
      System.out.println("Line 1:\n" + line);
    } // end try

    catch (FileNotFoundException e)
    {
      System.out.println("Nombre de archivo inválido: " + fileName);
    }
    catch (IOException e)
    {
      System.out.println("Error de lectura del archivo: " + fileName);
    }
  } // end main
} // end PrintLineFromFile2 class
```

Secuencia de bloques `catch`

Sesión muestra con entrada de un nombre de archivo inválido:

```
Enter a filename: garbage
Invalid filename: garbage
```

Figura 14.13 Programa PrintLineFromFile2: un lector de archivos mejorado.

siguiente `case`, a menos que ocurra una declaración `break`. Con los bloques `catch`, después que se encuentra y ejecuta un bloque `catch`, los bloques `catch` ulteriores se omiten automáticamente.

Siempre que se usa más de un bloque `catch` después de un bloque `try` dado, y se deriva una clase de excepción de un bloque `catch` de otra clase de excepción de un bloque `catch`, los bloques `catch` deben ordenarse con las clases de excepción más generales en el fondo. Por ejemplo, si se observa el `FileNotFoundException` en el sitio Sun de API, se verá esta jerarquía:

```
java.lang.Object
└java.lang.Throwable
    └ java.lang.Exception
        └ java.io.IOException
            └ java.io.FileNotFoundException
```

Si se elige tener un bloque `FileNotFoundException`, y un bloque `catchIOException` en la misma secuencia de bloques `catch`, entonces el bloque `catchIOException` debe colocarse en el fondo porque la clase `catchIOException` es una versión más general de la clase `FileNotFoundException`. Si la clase `catchIOException` se coloca primero, puede coincidir con ambos tipos de excepciones, y el bloque `catch FileNotFoundException` siempre puede omitirse. Pero esto no está bien. En la medida en que se comprenda este principio, no será necesario memorizar las relaciones jerárquicas entre todos los tipos de excepciones, ya que el compilador indicará un error de compilación en caso de que bloques `catch` múltiples se ordenen de manera errónea.

Bloques genéricos `catch` contra bloques `catch` múltiples

En la sección previa se analizó la técnica del bloque `catch` genérico. En esta sección se consideró la técnica de la secuencia de bloques `catch`. ¿Cuál es mejor? La técnica del bloque `catch` genérico es ligeramente más fácil de codificar, de modo que si se tiene interés en la simpleza, debe usarse esta técnica. La técnica de la secuencia de bloques `catch` permite manejar diferentes excepciones en forma diferente, de modo que si se tiene interés en tener más control sobre el manejo de excepciones y más control sobre los mensajes de error, debe usarse esta técnica.

14.11 Comprensión de los mensajes de excepción

A menos que se tenga extremo cuidado, quizás el lector ya ha escrito programas que han generado errores de tiempo de ejecución. Pero antes de este capítulo, el lector no estaba preparado de la mejor manera para comprender a fondo estos mensajes de error. Ahora ya lo está. En esta sección, los mensajes de error se describen al mostrar los detalles de los mensajes de error en el contexto de un programa completo.

Programa NumberList

El programa en las figuras 14.14*a* y 14.14*b* lee una lista de números y calcula la media. El programa se compila y ejecuta con éxito la mayor parte del tiempo, aunque no es muy robusto. Hay tres tipos de entrada que hacen que el programa se caiga. A continuación se describirán estos tres tipos de entrada, pero antes de leer sobre éstos, intente determinarlos usted mismo.

Identifique posibles errores de entrada.

El usuario introduce un no entero

En el método `readNumbers`, observe la llamada `parseInt`. Si el usuario introduce una q, el ciclo `while` termina y no se llama a `parseInt`. Pero si el usuario introduce algo que no sea una q, se llama `parseInt`. Si `parseInt` se llama con un argumento no entero, entonces `parseInt` lanza una `NumberFormatException`. Y puesto que no hay estructura `try-catch`, la JVM imprime un mensaje de error detallado y luego termina el programa. Por ejemplo, si el usuario introduce `hi`, la JVM imprime un mensaje de error detallado y luego termina el programa como sigue:[4]

Sesión muestra:

```
Introduzca un número entero (q para salir): hi
Exception in thread "main" java.lang.NumberFormatException:
```

excepción lanzada

[4] El formato del mensaje de error puede ser ligeramente distinto, aunque la información es semejante.

```
/************************************************
* NumberListDriver.java
* Dean & Dean
*
* Éste es el controlador de la clase NumberList.
************************************************/

public class NumberListDriver
{
  public static void main(String[] args)
  {
    NumberList list = new NumberList();
    list.readNumbers();
    System.out.println("Media = " + list.getMean());
  } // end main
} // end class NumberListDriver
```

Figura 14.14*a* Controlador del programa NumberList que controla la clase en la figura 14.14*b*.

```
For input string: "hi"
    at java.lang.NumberFormatException.forInputString(
      NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:447)
    at java.lang.Integer.parseInt(Integer.java:497)
    at NumberList.readNumbers(NumberList.java:28)
    at NumberListDriver.main(NumberListDriver.java:13)
```

llama la traza de la pila

A continuación se analizará el mensaje de error. Primero, la JVM imprime la excepción que ha sido lanzada, `NumberFormatException`. Luego imprime una llamada a la *traza de la pila*, que es un listado de los métodos llamados antes de la caída, en orden invertido. ¿Qué métodos se llamaron? Primero `main`, luego `readNumbers`, luego `parseInt`. Observe los números en el lado derecho de la llamada a la traza de la pila. Son los números de línea en el código fuente para los que se llaman los métodos. Por ejemplo, el 13 en la línea del fondo indica que la 13a. línea de `main` es una llamada al método `readNumbers`.

El usuario introduce inmediatamente q para salir

En el fondo del método `getMean`, observe el operador división. Siempre que se efectúa división entera, es necesario asegurarse de evitar la división entre cero. En el programa NumberList, no se evita. La variable de instancia `size` se inicia en cero, y si el usuario introduce de inmediato q para salir, `size` permanece en cero o `getMean` ejecuta una división entre cero. La división entera entre cero provoca que la JVM lance una `ArithmeticException`. Puesto que no hay mecanismo `try-catch`, la JVM imprime un mensaje de error detallado y termina el programa como se muestra en seguida:

Sesión muestra:

```
Enter a whole number (q to quit): q
Exception in thread "main"
    java.lang.ArithmeticException: / by zero
    at NumberList.getMean(NumberList.java:47)
    at NumberListDriver.main(NumberListDriver.java:14)
```

Observe que si se realiza división de punto flotante con un denominador igual a cero, no hay excepción. Si el numerador es un número positivo, la división entre 0.0 devuelve el valor `Infinity`. Si el numerador es un número negativo, la división entre 0.0 devuelve el valor `−Infinity`. Si el numerador también es 0.0, la división entre 0.0 devuelve el valor `NaN` (que significa no es un número).

```
/*************************************************************
* NumberList.java
* Dean & Dean
*
* Esto introduce números y calcula su valor medio.
*************************************************************/

import java.util.Scanner;

public class NumberList
{
  private int[] numList = new int[100];  // arreglo de números
  private int size = 0;                  // cantidad de números

  //*********************************************************

  public void readNumbers()
  {
    Scanner stdIn = new Scanner(System.in);
    String xStr;   // número introducido por el usuario (forma String)
    int x;         // número introducido por el usuario

    System.out.print("Introduzca un número entero (q para salir): ");
    xStr = stdIn.next();

    while (!xStr.equalsIgnoreCase("q"))
    {
      x = Integer.parseInt(xStr);
      numList[size] = x;
      size++;
      System.out.print("Introduzca un número entero (q para salir): ");
      xStr = stdIn.next();
    } // end while
  } // end readNumbers

  //*********************************************************

  public double getMean()
  {
    int sum = 0;

    for (int i=0; i<size; i++)
    {
      sum += numList[i];
    }
    return sum / size;
  } // end getMean
} // end class NumberList
```

Figura 14.14*b* Clase `NumberList` que calcula la media de números de entrada.

El usuario utiliza más de 100 números

En las declaraciones de la variable de instancia del programa NumberList, observe que `numList` es un elemento arreglo de 100. En el método `readNumbers` observe que esta declaración asigna números introducidos por el usuario al arreglo `numList`:

```
numList[size] = x;
```

Si el usuario introduce 101 números, entonces la variable `size` se incrementa a 100. Esto es mayor que el índice máximo (99) en el arreglo instanciado. Si se accede a un elemento arreglo con un índice que es más grande que el índice máximo o menor que cero, la operación lanza una excepción `ArrayIndexOutOfBoundException`. Puesto que no hay bloques `try-catch`, la JVM imprime un mensaje de error detallado y luego termina el programa como sigue:

Sesión muestra:

```
...
Introduzca un número entero (q para salir): 32
Introduzca un número entero (q para salir): 49
Introduzca un número entero (q para salir): 51
Exception in thread "main"
  java.lang.ArrayIndexOutOfBoundsException: 100
  at NumberList.readNumbers(NumberList.java:29)
  at NumberListDriver.main(NumberListDriver.java:13)
```

Ahora ya se ha terminado la descripción de los tres errores de tiempo de ejecución del programa NumberList. Normalmente, cuando se observan estos errores, el código debe repararse a fin de evitar los errores de tiempo de ejecución en el futuro. Así, para el programa NumberList, es necesario agregar reparaciones para los tres errores de tiempo de ejecución. En uno de los ejercicios del capítulo se pide que el lector haga precisamente eso.

14.12 Utilización de `throws` *<tipo-excepción>* para posponer el `catch`

En todos los ejemplos hasta el momento, el lanzamiento de excepciones se ha manejado localmente; es decir, los bloques `try-catch` se han puesto en el método que contiene la declaración peligrosa. Pero algunas veces esto no es factible.

Movimiento de los bloques `try-catch` de regreso al método que llama

Cuando no es factible usar bloques `try-catch`, estos bloques no pueden moverse fuera del método con la declaración peligrosa y devolverlos al método que llama. Si se hace lo anterior y la declaración peligrosa lanza una excepción, la JVM de inmediato salta del método de la declaración peligrosa y pasa la excepción de regreso a los bloques `try-catch` en el método que llama.[5]

Así, ¿cuándo debe poner los bloques `try-catch` en el método que llama, en oposición al método con la declaración peligrosa? La mayor parte del tiempo, los bloques `try-catch` deben ponerse en el método con la declaración peligrosa porque así se promueve la modularización, lo cual está bien. Pero algunas veces resulta difícil contar con un bloque `catch` idóneo cuando se está en el interior de un método con una declaración peligrosa. Por ejemplo, suponga que se ha escrito un método de utilidad que es llamado desde muchos sitios diferentes, y que algunas veces el método lanza una excepción. Cuando se lanza una excepción, sería conveniente tener un mensaje de error a la medida del método que llama. Si el bloque `catch` está en el método de utilidad, hacer lo anterior es difícil. La solución consiste en mover los bloques `try-catch` a los métodos que llaman.

Considere otro ejemplo. Suponga que se ha escrito un método con un tipo de retorno no `vacío` que algunas veces lanza una excepción. Con un tipo de retorno no `vacío`, el compilador espera que el método devuelva un valor. Pero cuando se lanza una excepción, normalmente no se desea devolver un valor porque no hay un valor idóneo para devolver. Entonces, ¿cómo es posible tener un método no `vacío` y no devolver un valor? Mueva los bloques `try-catch` al método que llama. Luego, cuando se lanza una excepción, la JV regresa al método que llama sin devolver un valor. Los bloques `try-catch` del método manejan la excepción, lo más probable con un mensaje de error. A continuación se verá cómo funciona esto en un programa de Java.

[5] En realidad, el salto al método que llama no es inmediato si abajo del o los bloques `try` hay un bloque `finally`. En ese caso, la JVM salta al bloque `finally` antes de saltar al método que llama. El bloque `finally` se describe al final de esta sección.

```
/*************************************************************
* StudentList2.java
* Dean & Dean
*
* Este programa gestiona una ArrayList de estudiantes.
*************************************************************/

import java.util.ArrayList;

public class StudentList2
{
  private ArrayList<String> students = new ArrayList<String>();

  //*********************************************************

  public StudentList2(String[] names)
  {
    for (int i=0; i<names.length; i++)
    {
      students.add(names[i]);
    }
  } // end constructor

  //*********************************************************

  public void display()
  {
    for (int i=0; i<students.size(); i++)
    {
      System.out.print(students.get(i) + " ");
    }
    System.out.println();
  } // end display

  //*********************************************************

  public String removeStudent(int index)
    throws IndexOutOfBoundsException
  {
    return students.remove(index);
  } // end removeStudent
} // end StudentList2
```

Lanza el manejo del error al método que llama.

Figura 14.15 Clase `StudentList2` que es controlada por la clase en la figura 14.16.

Otra visita al programa StudentList

En la figura 14.15 se muestra una versión modificada de la clase `StudentList` en la figura 14.8. La diferencia más importante es que el método `removeStudent` ahora devuelve el nombre del estudiante que elimina. Esto permite que el método que llama haga algo con el elemento eliminado.

En el método `removeStudent`, observe la declaración `return`. La llamada al método `students.remove` intenta eliminar el elemento en la posición indicada por `index`. Si `index` es menor que cero o mayor que el índice del último elemento, entonces la JVM lanza una `ArrayIndexOutOfBoundsException`. En la clase `StudentList` anterior, la excepción se manejó localmente dentro del método `removeStudent`. Esta vez, puesto que se está devolviendo un valor, resulta más conveniente transferir el manejo de la excepción al método que llama. Esto se hace al colocar los bloques `try`-

`catch` en el método que llama y colocando una cláusula en el encabezado del método `removeStudent`. He aquí el encabezado:

```
public String removeStudent(int index)
  throws IndexOutOfBoundsException
```

Agregar una cláusula `throws` le recuerda al compilador que el método podría lanzar una excepción no manejada. La cláusula `throws` es necesaria si la excepción no manejada es una excepción comprobada, y se recomienda si la excepción no manejada es una excepción no comprobada. Puesto que `IndexOutOfBoundsException` es una excepción no comprobada, es legal omitir la cláusula `throws` de arriba. Pero es un buen estilo incluirla porque proporciona información de autodocumentación valiosa. Si más tarde un programador desea usar el método `removeStudent`, la cláusula `throws` advierte al programador que debe contar con un mecanismo "remoto" `try-catch` para manejar la `IndexOutOfBoundsException` cuando llame al método `removeStudent`.

Para ver cómo se implementa este mecanismo "remoto" `try-catch`, observe la clase `StudentList2Driver` en la figura 14.16. Muestra una lista de estudiantes, pregunta al usuario cuál estudiante debe eliminarse y espera para eliminar a ese estudiante. Si la llamada al método `removeStudent` falla, el bloque `catch` maneja la excepción lanzada y el programa pregunta otra vez al usuario cuál estudiante se debe eliminar.

El bloque `finally`

Al implementar un manipulador de excepciones, algunas veces es conveniente contar con un "código de limpieza" que se ejecute sin importar si se ha lanzado una excepción. Suponga que abre un archivo y trata de escribir en él. La operación de escritura puede o no lanzar una excepción. En cualquier caso, una vez que se ha terminado, es necesario cerrar el archivo. Si se olvida cerrar el archivo, entonces los recursos del sistema permanecen atendiendo al archivo abierto, lo que provoca la degradación del rendimiento del sistema. El hecho de cerrar un archivo es un ejemplo de código de limpieza. Si la limpieza se maneja localmente (por ejemplo, que las operaciones de escritura y cierre estén en el mismo método), entonces la limpieza es directa. Simplemente coloque el código debajo de los bloques `try` y `catch` y la JVM lo ejecuta sin tomar en cuenta si se ha lanzado una excepción. Pero si el trabajo de manejo de excepciones se transfiere con una cláusula `throws`, entonces la limpieza está ligeramente más implicada.

Si una excepción se maneja con una cláusula `throws`, y se requiere contar con un código de limpieza sin importar si se ha lanzado una excepción, debe usarse un bloque `finally`. Un bloque `finally` está asociado con un bloque `try` particular y, como tal, debe colocarse inmediatamente después de un bloque `try`.[6] Si la JVM lanza una excepción dentro del bloque `try`, la JVM salta de inmediato al bloque `finally`, lo ejecuta y luego lanza la excepción de regreso al módulo que llama. Si la JVM no lanza una excepción dentro del bloque `try`, la JVC termina el bloque `try` y luego ejecuta el bloque `finally`.

El método `writeToFile` en la figura 14.17 ilustra el bloque `finally`. El método abre un archivo y escribe un mensaje de texto en el archivo. Específicamente, la llamada al constructor `PrintWriter` abre un archivo denominado `testFile.txt`. La llamada al `fileOut.printf` escribe "Ésta es una prueba" en el archivo abierto. Luego, el archivo se cierra por `fileOut.close`. Puesto que la llamada al `fileOut.close` está dentro de un bloque `finally`, se ejecuta sin importar si se ha lanzado una excepción.

14.13 Apartado GUI y solución de problemas: ejemplo revisitado de trazado de líneas (opcional)

Antes en este capítulo se implementó un programa LinePlot que traza una línea definida por una secuencia de puntos definidos por el usuario. El desplegado de la gráfica de la línea fue menos que ideal. Mostró la

[6] Es legal insertar uno o varios bloques `try` entre los bloques `try` y `finally`, aunque así puede llegarse a un código confuso. Aquí se recomienda mantener sencillas las cosas. Use un bloque `try` con uno o varios bloques `catch` o un bloque `try` con un bloque `finally`, pero no las tres cosas juntas.

```
/*************************************************************
* StudentList2Driver.java
* Dean & Dean
*
* Esto controla la clase StudentList2.
*************************************************************/

import java.util.Scanner;

public class StudentList2Driver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String[] names = {"Caleb", "Izumi", "Mary", "Usha"};
    StudentList2 studentList = new StudentList2(names);
    int index;
    boolean reenter;

    studentList.display();

    do
    {
      System.out.print("Introduzca el índice del estudiante a eliminar: ");
      index = stdIn.nextInt();
      try
      {
        System.out.println(
          "removed " + studentList.removeStudent(index));
        reenter = false;
      }
      catch (IndexOutOfBoundsException e)
      {
        System.out.print("Invalid entry. ");
        reenter = true;
      }
    } while (reenter);

    studentList.display();
  } // end main
} // end StudentList2Driver
```

Si no hay error, este método devuelve el nombre del estudiante eliminado.

Si en el método `removeStudent` se lanza una excepción, este bloque `catch` la atrapa.

Sesión muestra:

```
Caleb Izumi Mary Usha
Enter index of student to remove: 6
Invalid entry. Enter index of student to remove: 1
removed Izumi
Caleb Mary Usha
```

Figura 14.16 Controlador para la clase `StudentList2`.

línea como una descripción de texto de segmentos de línea. Por ejemplo, esto es lo que produce el programa para una línea con cinco segmentos que pasa por los puntos (0,0), (1,3), (2,1), (3,2), (4,2) y (5,1):

(0,0)–(1,3), (1,3)–(2,1), (2,1)–(3,2), (3,2)–(4,2), (4,2)–(5,1)

En la figura 14.18 se muestra cómo LinePlotGUI, una versión mejorada del programa LinePlot, muestra la línea anterior que pasa por los puntos mencionados. En aras de la sencillez, no hay marcas

```
public void writeToFile() throws IOException
{
  PrintWriter fileOut = new PrintWriter("testFile.txt");
  try
  {
    fileOut.printf("%s", "Ésta es una prueba.");
  }
  finally
  {
    fileOut.close();
  }
} // end writeToFile
```

Figura 14.17 Método que usa un bloque `finally` para cerrar un archivo de salida.

hash de intervalo en los ejes *x* y *y*. Como tal vez pueda suponer el lector, el eje *x* mostrado cuenta con seis marcas hash implícitas para los valores 0, 1, 2, 3, 4 y 5. Y el eje *y* mostrado tiene cuatro marcas hash implícitas para los valores 0, 1, 2 y 3.

Software prescrito y el método `drawPolyLine`

Considere utilizar diseño ascendente.

Al resolver problemas, suele ser común asumir un enfoque arriba-abajo (*top-down*): escribir primero el método controlador (`main`), luego escribir los métodos `public` que proporcionan una interfaz con el mundo exterior, luego escribir los métodos de ayuda `private`. Este enfoque a menudo funciona a la perfección, aunque algunas veces puede llevar a reinventar la rueda. Si el lector usa un enfoque arriba-abajo puro para implementar el programa LinePlotGUI, quizás implemente la disposición como una secuencia de llamadas al método `drawLine`, una para cada segmento de línea (para un análisis del método `drawLine`, consulte el capítulo 5). Usar este método podría funcionar, aunque se requeriría un ciclo y tal vez algo de esfuerzo de depuración. El mejor enfoque consiste en desempolvar el tomo API de Java (http://java.sun.com/javase/6/docs/api/) y buscar métodos alternativos para el trazado de líneas.

He aquí que hay un método para trazar líneas que hace exactamente lo que se desea. El método `drawPolyline` traza una línea al unir una secuencia de puntos. Más específicamente, el método `drawPolyline` de la clase `Graphic` recibe tres parámetros: `xPixels`, `yPixels` y `numOfPoint`. El parámetro `numOfPoints` mantiene el número de puntos sobre la línea. El parámetro `xPixels` mantiene un arreglo de las posiciones del pixel horizontal para cada uno de los puntos. El parámetro `yPixels`

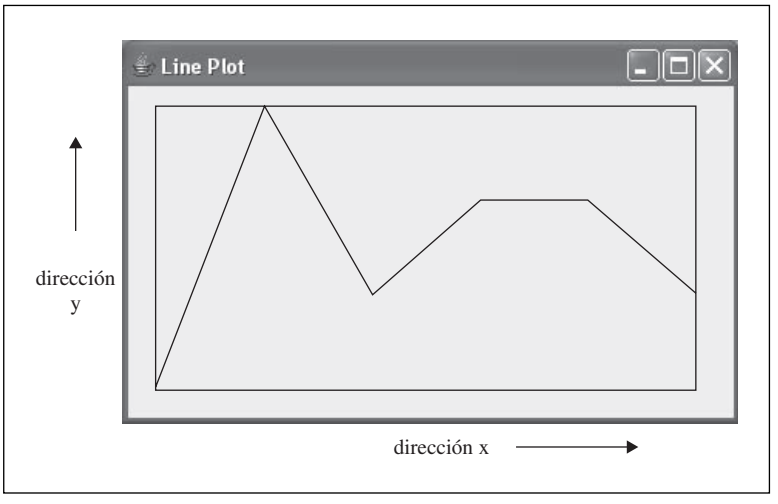


Figura 14.18 Salida de muestra para el programa LinePlotGUI.

```
public void paintComponent(Graphics g)
{
  int[] xPixels = {50, 50, 100, 100};
  int[] yPixels = {140, 40, 140, 40};
  g.drawPolyline(xPixels, yPixels, 4);
} // end paintComponent
```

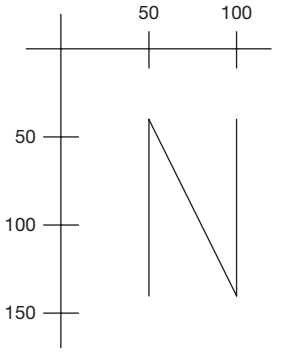


Figura 14.19 Ejemplo de una llamada al método `drawPolyline` que muestra una línea en forma de N.

mantiene un arreglo de las posiciones del pixel vertical para cada uno de los puntos. Por ejemplo, `xPixels[0]` y `yPixels[0]` mantienen posiciones de pixel para el primer punto; `xPixels[1]` y `yPixels[1]` mantienen posiciones de pixel para el segundo punto y así sucesivamente. En la figura 14.19, la llamada al método `drawPolyline` muestra una línea que une cuatro puntos en forma de N.

Desarrollo del algoritmo

Una vez que se comprende el método `drawPolyline`, el algoritmo básico del programa LinePlotGUI se clarifica: llenar los arreglos `xPixels` y `yPixels` con valores de pixeles para una secuencia de puntos. Luego, se usan esos arreglos para llamar a `drawPolyline`.

Antes de precisar con más detalle este algoritmo básico, es necesario conocer un supuesto importante. Los puntos de la línea trazada son equidistantes a lo largo del eje *x*. Más específicamente, los puntos están en las posiciones x=0, x=1, x=2, etc. Así, cuando el programa pide un punto al usuario, sólo se requiere un valor y, no un valor x (porque el valor x ya se conoce: x=0, x=1, x=2, etcétera).

He aquí una descripción de alto nivel del algoritmo:

Organice sus pensamientos como un algoritmo.

1. Solicitar al usuario un número de puntos y el valor y máximo (`maxY`).
2. Para cada punto, solicitar al usuario el valor de una coordenada *y* y almacenar el valor en `yCoords`.
3. Determinar el número de pixeles horizontales entre puntos adyacentes (`pixelInterval`).
4. Llenar el arreglo `xPixels`:

   `xPixels[i]` ← `i * pixelInterval`
5. Llenar el arreglo `yPixels`:

   `yPixels[i]` ← (`yCoords[i]/maxY`) * altura en pixeles del límite de la línea trazada
6. Llamar a `drawRect` para mostrar un límite para la línea trazada.
7. Llamar a `drawPolyline` para mostrar la línea trazada.

Estructura del programa

Ahora que ya se cuenta con una descripción de alto nivel del algoritmo quizás el lector esté tentado a traducirlo de inmediato a código fuente. Primero, es necesario determinar dónde deben ir las diversas partes del programa. Como suele ser costumbre para un programa no trivial como éste, la solución debe implementarse con dos clases: una clase `LinePlotGUI` para controlar el programa y una clase `LinePlotPanel` para trazar las figuras. Resulta evidente que las llamadas a los métodos `drawRect` y `drawPolyline` deben ir en la clase `LinePlotPanel`, puesto que implican trazos. Pero ¿qué ocurre con el código que solicita al usuario valores de las coordenadas *y*? ¿Y qué ocurre con el código que calcula los valores `xPixels` y `yPixels`? Es importante contar con el código en la clase correcta a fin de asegurar que cada clase tenga un papel claramente definido. Esto ayuda para el desarrollo, legibilidad y mantenimiento del

programa. La clase `LinePlotGUI` controla el programa y la entrada desempeña un papel importante en ese esfuerzo. En consecuencia, el código de solicitud al usuario debe colocarse en la clase `LinePlotGUI`. La clase `LinePlotPanel` traza las figuras, y los cálculos juegan un papel importante en ese esfuerzo. En consecuencia, el código para los cálculos de trazado debe colocarse en la clase `LinePlotPanel`.

Modularidad

Ahora es el momento de considerar el código fuente del programa `LinePlotGUI`. Observe la naturaleza modular de la clase `LinePlotGUI` en las figuras 14.20*a*, 14.20*b* y 14.20*c*. Las llamadas al `JFrame`

```
/*************************************************************
* LinePlotGUI.java
* Dean & Dean
*
* Este programa traza una línea como una secuencia de puntos
* unidos especificados por el usuario.
*************************************************************/

import javax.swing.*;  // for JFrame, JOptionPane

public class LinePlotGUI extends JFrame
{
  private static final int FRAME_WIDTH = 400;
  private static final int FRAME_HEIGHT = 250;
  private static final int MARGIN = 20;   // espacio entre el marco
                                          // y la línea trazada

  int numOfPoints      // los puntos van desde N=0 hasta N=numOfPoints-1
  int maxY;            // los valores de la coordenada y van desde y=0 hasta y=maxY
  double[] yCoords;    // valores de la coordenada y para todos los puntos

  //*********************************************************

  public LinePlotGUI()
  {
    setSize(FRAME_WIDTH, FRAME_HEIGHT);
    setTitle("Line Plot");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  } // end LinePlotGUI

  //*********************************************************

  int getMargin()
  {
    return MARGIN;
  }

  int getMaxY()
  {
    return maxY;
  }

  double[] getYCoords()
  {
    return yCoords;
  }
```

Figura 14.20*a* Clase `LinePlotGUI`, parte A.

```
//************************************************************

// Este método solicita al usuario las coordenadas y de los puntos
// en las posiciones x=0, x=1, etcétera.

public void readYCoordinates()
{
  String yStr;  // user's entry for a point's y coordinate
  numOfPoints = getIntFromUser("Introduzca el número de puntos: ");
  maxY = getIntFromUser("Introduzca el valor del punto máximo: ");
  yCoords = new double[numOfpoints];

  for (int i=0; i<=maxX; i++)
  {
    yStr = JOptionPane.showInputDialog(
      "En x = " + i + ", ¿cuál es el valor de y?\n" +
      "Introduzca un entero entre 0 y " +
      maxY + " inclusive:");
    try
    {
      yCoords[i] = Integer.parseInt(yStr);
      if (yCoords[i] < 0 || yCoords[i] > maxY)
      {
        JOptionPane.showMessageDialog(null,
          "Entrada inválida. El valor debe estar entre 0 y " + maxY);
        i--;
      }
    }
    catch (NumberFormatException e)
    {
      JOptionPane.showMessageDialog(null,
        "Entrada inválida. Debe introducir un entero.");
      i--;
    }
  } // end for
} // end readYCoordinates
```

Estas iniciaciones usan un método de ayuda.

Figura 14.20*b* Clase `LinePlotGUI`, parte B.

(`setSize`, `setTitle`, `setDefaultCloseOperation`) están en su propio módulo, el constructor `LinePlotGUI`. El código de solicitud al usuario está en su propio módulo, el método `readYCoordinates`. El método `readYCoordinates` solicita al usuario el número de puntos. También solicita al usuario el valor y máximo sobre el eje *y*. Las dos entradas deben experimentar el mismo tipo de validación de entrada. A fin de evitar código redundante, el código de validación de entrada está en un método de ayuda común, `getIntFromUser`. El método `readYCoordinates` también solicita al usuario el valor *y* de cada punto, que puede estar en cualquier sitio del intervalo entre cero y el valor máximo *y*.

Observe el componente `paintComponent` en la figura 14.21*b*. El método `paintComponent` es llamado automáticamente por la JVC cuando el programa se inicia y siempre que un usuario hace algo para modificar la ventana del programa (por ejemplo, cuando el usuario modifica el tamaño de la ventana o mueve una ventana fuera de otra ventana). El método `paintComponent` es responsable de trazar las imágenes de las ventanas. El código para calcular el trazado puede colocarse junto al código para el trazado de gráficas en el método `paintComponent`, pero como se muestra en las figuras 14.21*a* y 14.21*b*, el código para calcular el trazado está en su propio módulo, el constructor `LinePlotPanel`. Ésta es una buena idea por varias razones. Una, continúa el objetivo de la modularización en el sentido de

```
  //*************************************************************

  // Este método solicita al usuario un entero, efectúa validación
  // de entrada y devuelve el entero introducido.

  private static int getIntFromUser(String prompt)      <-- método de ayuda
  {
    String entry;          // entrada del usuario
    boolean valid = false; // ¿la entrada del usuario es un entero válido?
    int entryInt = 0;      // forma entera de la entrada del usuario

    entry = JOptionPane.showInputDialog(prompt);
    do
    {
      try
      {
        entryInt = Integer.parseInt(entry);
        valid = true;
      }
      catch (NumberFormatException e)
      {
        entry = JOptionPane.showInputDialog(
          "Entrada inválida. Introduzca un entero:");
      }
    } while (!valid);

    return entryInt;
  } // end getIntFromUser

  //*************************************************************

  public static void main(String[] args)
  {
    LinePlotGUI linePlotGUI = new LinePlotGUI();
    linePlotGUI.readYCoordinates();
    LinePlotPanel linePlotPanel = new LinePlotPanel(linePlotGUI);
    linePlotGUI.add(linePlotPanel);
    linePlotGUI.setVisible(true);
  } // end main
} // end class LinePlotGUI
```

Figura 14.20*c* Clase `LinePlotGUI`, parte C.

que tareas separadas se llevan a cabo en módulos separados. Dos, mejora la rapidez del programa. El constructor `LinePlotPanel` sólo se ejecuta una vez, cuando el objeto `LinePlotPanel` se instancia en `main`. El método `paintComponent` se ejecuta cada vez que el usuario hace algo para modificar la ventana del programa. No es necesario volver a hacer los cálculos para el trazado cada vez que ocurre lo anterior, de modo que mover el código al constructor `LinePlotPanel` funciona bien.

Trazar una línea con pocos puntos no sería gran cosa si se comete el error de poner todo el código de los cálculos para el trazado en el método `paintComponent`. Pero si son muchos puntos, entonces la reducción de la velocidad en el tamaño de la ventana puede ser perceptible. La lentitud es aceptable para ciertas cosas, como cargar inicialmente un programa, pero no para cuestiones relacionadas con la graphical user interface (GUI), como cambiar el tamaño de una ventana. Los usuarios suelen ser impacientes. ¿Alguna vez le ha dado un porrazo a su ratón en un ataque de impaciencia?

```
/*************************************************************
* LinePlotPanel.java
* Dean & Dean
*
* Esta clase muestra una línea como una secuencia de puntos unidos.
*************************************************************/

import javax.swing.*; // for JPanel
import java.awt.*;    // for Graphics

public class LinePlotPanel extends JPanel
{
  private int[] xPixels; // mantiene el valor x para cada punto trazado
  private int[] yPixels; // mantiene el valor y para cada punto trazado

  // La línea trazada está rodeada por un rectángulo con estas especificaciones:
  private int topLeftX, topLeftY;
  private int rectWidth, rectHeight;

  //*********************************************************

  // Calcula las dimensiones del rectángulo de la línea trazada, usando
  // el marco pasado, que contiene las dimensiones del marco y los valores
  // de las coordenadas. Llena los arreglos xPixels y yPixels.

  public LinePlotPanel(LinePlotGUI frame)
  {
    int numOfPoints = frame.getYCoords().length;
    int pixelInterval;   // distancia entre puntos adyacentes

    topLeftX = topLeftY = frame.getMargin();

    // getInsets funciona sólo si primero se llama a setVisible
    frame.setVisible(true);
    rectWidth =
      frame.getWidth() - (2 * topLeftX +
      frame.getInsets().left + frame.getInsets().right);
    rectHeight =
      frame.getHeight() - (2 * topLeftY +
      frame.getInsets().top + frame.getInsets().bottom);
```

Figura 14.21*a* Clase `LinePlotPanel`, parte A.

Escalabilidad

Un programa es *escalable* si es capaz de soportar mayores o menores cantidades de datos y más o menos usuarios. El soporte para más datos y más usuarios podría requerir cambios en el software, pero los cambios deben ser efectivos desde el punto de vista de los costos, no versiones masivas ya escritas.

Un programa profesional para el trazado de líneas debe poder manejar grandes cantidades de datos y distintos tipos de datos. El programa LinePlotGUI puede no alcanzar ese nivel de escalabilidad (sólo maneja un tipo de datos, y todos los datos se ajustan a una ventana), lo cual no está mal del todo, si se considera que se requería mantenerlo razonablemente corto a fin de darle cabida un libro de texto introductorio. El programa LinePlotGUI es escalable en el sentido de que su alcance está restringido en primer lugar por la entrada del usuario y unas cuantas constantes nombradas, no por constructos de codificación difíciles de modificar. Cuestión a considerar: el usuario especifica el número de puntos y el máximo valor y para cada punto; las constantes nombradas especifican el tamaño y el margen de la ventana.

```
      // Calcula el intervalo entero de pixeles entre puntos adyacentes
      pixelInterval = rectWidth / (numOfPoints - 1);

      // Hace que el ancho actual del rectángulo = múltiplo de pixelInterval
      rectwidth = (numOfPoints - 1) * pixelInterval;

      xPixels = new int[numOfPoints];
      yPixels = new int[numOfPoints];

      for (int i=0; i<numOfPoints; i++)
      {
        xPixels[i] = topLeftX + (i * pixelInterval);
        yPixels[i] = topLeftY + rectHeight - (int) Math.round(
          (frame.getYCoords()[i] / frame.getMaxY()) * rectHeight);
      }
    } // end LinePlotPanel constructor

    //*******************************************************

    // Esta clase muestra la línea como una secuencia de puntos unidos.

    public void paintComponent(Graphics g)
    {
      super.paintComponent(g);
      g.drawRect(topLeftX, topLeftY, rectWidth, rectHeight);
      g.drawPolyline(xPixels, yPixels, xPixels.length);
    } // end paintComponent
  } // end class LinePlotPanel
```

parámetros calculados previamente

Figura 14.21*b* Clase `LinePlotPanel`, parte B.

Fortaleza y el método `getInsets`

En el constructor `LinePlotPanel` en la figura 14.21*a*, observe cómo `frame` (la ventana de salida del programa) llama al método `getInsets`. Este método devuelve un objeto `Insets` de la ventana. El objeto `Insets` almacena el grosor de los cuatro límites de la ventana. Por ejemplo, `frame.getInsets().left` devuelve el ancho (en pixeles) del límite izquierdo de la ventana y `frame.getInsets().top` devuelve la altura (en pixeles) del límite superior de la ventana. El límite superior incluye la altura de la barra de títulos.

Si el lector no desea molestarse con el método `getInsets`, quizás esté tentado a utilizar conjeturas duramente codificadas para los tamaños de los límites. No lo haga. Diferentes plataformas para Java (por ejemplo, las plataformas Windows, UNIX y Macintosh) tienen tamaños distintos de los límites de la ventana. Entonces, inclusive si el lector conjetura correctamente para su plataforma actual de Java, sus conjeturas no necesariamente funcionan para plataformas alternativas de Java. Moraleja: sea fuerte y use `getInsets`. No use conjeturas duramente codificadas.

Resumen

- Una excepción es un evento que ocurre durante la ejecución de un programa que trastorna el flujo normal de las instrucciones durante la ejecución de un programa.
- El manejo de excepciones es una técnica para manejar excepciones en forma elegante.
- Use un bloque `try` para "intentar" una o más llamadas a métodos peligrosos. Si hay algún problema con las llamadas a métodos peligrosos, la JVM lanza una excepción y busca un bloque `catch` que "coincida".

- Un bloque `catch` coincide si el tipo de parámetro del encabezado es el mismo que o un ancestro del tipo de la excepción lanzada.
- Si se lanza una excepción, la JVM salta de inmediato del bloque `try` actual. Esto significa que si en el bloque `try` hay declaraciones después de la declaración de lanzamiento de la excepción, estas declaraciones se omiten.
- Las excepciones comprobadas deben comprobarse con un mecanismo `try-catch`.
- Las excepciones no comprobadas pueden comprobarse opcionalmente con un mecanismo `try-catch`, pero no es un requerimiento.
- Las excepciones no comprobadas son descendientes de la clase `RuntimeException`.
- Para implementar un manejador de excepciones sencillo y de objetivo múltiple, defina un bloque `catch` con un parámetro de tipo `Exception` y dentro de bloque `catch`, llame al método `getMessage` de la clase `Exception`.
- Para definir de manera más explícita un manejador de excepciones, defina una secuencia de bloques `catch`. Disponga los bloques `catch` con las clases de excepción más generales en la parte inferior.
- Si un programa falla, la JVM imprime una llamada a la traza de una pila, que consta de un listado de los métodos que fueron llamados antes de la caída, en orden inverso.
- Use una cláusula `throws` para propagar una excepción de vuelta al módulo que llama.
- Si una excepción se maneja con una cláusula `throws`, y es necesario contar con un código de limpieza sin importar si se lanza una excepción, use un bloque `finally`.

Preguntas de revisión

§14.3 Utilización de los bloques `try` y `catch` para manejo de llamadas a métodos "peligrosos"

1. Si su programa contiene una llamada a un método API, debe colocarlo dentro de un bloque `try`. Para ser completamente dócil con la práctica correcta de codificación, debe aplicar esta regla para todas las llamadas al método API. (F/C)

2. Un bloque `try` y su(s) bloque(s) `catch` asociado(s) deben ser contiguos. (F/C)

§14.5 Detalles de los bloques `try`

3. En términos generales, es necesario intentar agregar declaraciones peligrosas relacionadas en el mismo bloque `try` a fin de minimizar el desbarajuste. (F/C)

4. ¿Dónde debe colocar las declaraciones seguras que usan los resultados de operaciones peligrosas?

5. Si se lanza una excepción, la JVM salta a un bloque `catch` que coincide, y luego de ejecutar el bloque `catch`, regresa al bloque `try` en el punto en que se lanzó la excepción. (F/C)

6. Al comprobar los errores de tiempo de compilación, el compilador toma en cuenta que todas las declaraciones dentro de un bloque `try` pueden omitirse. (F/C)

§14.6 Dos categorías de excepciones: comprobadas y no comprobadas

7. Si una excepción se deriva de la clase `RunTimeException`, se trata de una excepción _____.

8. Las excepciones comprobadas son excepciones que están en o se derivan de la clase _____, pero que no están o se derivan de la clase _____.

§14.7 Excepciones no comprobadas

9. En la lista siguiente, indique si cada opción es viable para una excepción no comprobada que usted sabe que su programa podría lanzar:

a) Ignorarla.

b) Volver a escribir el código de modo que la excepción no ocurra nunca.

c) Colocarla en un bloque `try` y atraparla en un bloque `catch` siguiente.

§14.8 Excepciones comprobadas

10. Si una declaración lanzara una excepción comprobada, es posible evitar que el compilador se queje si esa declaración se coloca en un bloque `try` y el bloque `try` se sigue por un bloque `catch` cuyo tipo de parámetro sea el mismo que el tipo de la excepción. (F/C)

11. Es posible determinar si una declaración particular contiene una excepción comprobada y el tipo de esa excepción al intentar compilar sin mecanismo `try-catch`. (F/C)

§14.9 La clase `Exception` y su método `getMessage`

12. ¿Es correcto incluir en el mismo bloque `try` un código que puede lanzar excepciones comprobadas y excepciones no comprobadas?

13. ¿Qué tipo de excepción coincide con todas las excepciones comprobadas y con todas las excepciones no comprobadas, excepto aquellas derivadas de la clase `Error`?

14. ¿Qué devuelve el método `getMessage`?

§14.10 Bloques `catch` múltiples

15. Para cada tipo distinto de excepción que podría lanzarse, debe haber un bloque `catch` por separado. (F/C)

16. El compilador verifica automáticamente la existencia de bloques `catch` que no funcionan. (F/C)

§14.11 Comprensión de los mensajes de excepción

17. ¿Cuáles son los dos tipos de información mostrados por la JVM cuando ésta encuentra un error de tipo de ejecución que termina la ejecución?

§14.12 Utilización de `throws` *<tipo-excepción>* para posponer el `catch`

18. Suponga que se desea posponer atrapar una `NumberFormatException`. ¿Qué es necesario agregar al encabezado de un método a fin de alertar al compilador y a los usuarios potenciales que algo en el método pudiera lanzar ese tipo de excepción?

19. Considere un método no vacío que no contiene bloques `try` ni bloques `catch`. Si el método lanza una excepción, se sabe que la JVM transfiere el lanzamiento de la excepción de vuelta al método que llama. Pero ¿la JVM devuelve un valor (con una declaración `return`) al módulo que llama?

Ejercicios

1. [Después de §14.3] Dado el siguiente programa, ¿cuál es la salida si el usuario introduce "uno" en respuesta al desplegado?

```
/************************************************************
* FantasyFootball.java
* Dean & Dean
*
* Esto imprime los nombres de jugadores de futbol americano.
************************************************************/
import java.util.Scanner;
import java.util.ArrayList;

public class FantasyFootball
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    ArrayList<String> players = new ArrayList<String>();
    String indexStr;
    int index = 0;

    players.add("Peyton Manning");
    players.add("Ladanian Tomlinson");
    players.add("Reggie Bush");
    System.out.print("Introduzca un número entre 1 y 3: ");
    indexStr = stdIn.nextLine();
    try
    {
      index = Integer.parseInt(indexStr);
      System.out.println("Índice introducido OK.");
    }
    catch (NumberFormatException e)
    {
      System.out.println("El índice introducido no era un entero");
    }
```

```
      try
      {
        System.out.println(players.get(index - 1));
      }
      catch (IndexOutOfBoundsException e)
      {
        System.out.println(
          "No es posible acceder a jugadores[" + (index - 1) + "]");
      }
      System.out.println("hecho");
    } // end main
  } // end class FantasyFootball
```

2. Dado el programa anterior, ¿cuál es la salida si el usuario introduce 1 en respuesta al desplegado?

3. [Después de §14.4] Agregue una estructura `try-catch` al siguiente programa para hacerlo compilar y ejecutar correctamente, inclusive cuando el divisor es cero. Observe que la división entre cero lanza una `ArithmeticException`.

```
import java.util.Scanner;

public class Division
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int n, d, q;

    System.out.print("Introduzca el numerador: ");
    n = stdIn.nextInt();
    System.out.print("Introduzca el divisor: ");
    d = stdIn.nextInt();
    q = n / d;
    System.out.println(q);
  } // end main
} // end Division class
```

Para ayudar al lector, a continuación se proporciona el bloque `catch`:

```
catch (ArithmeticException e)
{
  System.out.println("Error, pero siga intentando.");
}
```

No es necesario comprobar si la entrada es correcta; es posible suponer que como entrada, el usuario introduce dos valores `int` con formato correcto.

4. [Después de §14.5] Mejora del programa:
El siguiente programa ejecuta división y no lanza ninguna excepción cuando se introduce cero como denominador. Tampoco detecta excepciones de formato de números de entrada. Minimice el total de línea de código requerido para satisfacer tales requerimientos.

```
/*********************************************************
* Division2.java
* Dean & Dean
*
* Esto intenta evitar la división entre cero.
*********************************************************/
import java.util.Scanner;

public class Division2
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
```

```
    double n;
    int d;

    System.out.print("Introduzca el numerador: ");
    n = stdIn.nextDouble();
    System.out.print("Introduzca el divisor: ");
    d = stdIn.nextInt();
    System.out.println(n / d);
  } // end main
} // end Division2 class
```

a) Primero, vuelva a escribir el programa de modo que siga empleando un numerador `double` y un denominador `int`, pero si el valor de entrada del denominador es cero, rehúsa realizar la operación división, y sigue solicitando el denominador hasta que el usuario introduce algo distinto de cero.

b) Luego, vuelva a escribir el programa de modo que si el usuario introduce un formato incorrecto para el numerador o el denominador, toda la solicitud de entrada se repite hasta que ambos formatos están correctos. Sugerencia: coloque bloques `try` y `catch` en un ciclo que ejecute `while` (`OK == false`), y establezca `OK = true` después que todas las operaciones críticas en el bloque `try` se han realizado exitosamente. Nota: si el formato analizado es incorrecto, se obtiene un ciclo infinito, a menos que vuelva a instanciarse `stdIn` en cada iteración, o use una operación en dos pasos para cada entrada (introducir una cadena y luego analizarla).

5. [Después de §14.7] ¿Qué ocurre si se lanza una excepción no comprobada y nunca es atrapada?

6. [Después de §14.9] Programa WebPageReader:

Además de probar las proezas del lector para manejar excepciones, este ejercicio también prueba su capacidad para usar en línea libros de ayuda y/o referencia. El siguiente programa intenta leer una dirección en la red e imprimir el contenido de la página en esa dirección.

En el programa dado:

Para cada clase utilizada, agregar una declaración `import`, en caso de ser necesario.

Para cada llamada a un método o a un constructor:

Si lanza una excepción no comprobada, ignorarla.

Si lanza una excepción comprobada:

Especificar la excepción específica en una cláusula `throws`.

Dentro de un bloque `catch` en `main`, atrapar la excepción e imprimir el mensaje de excepción usando su método `getMessage`.

Suponga que el siguiente código funciona bien, excepto por las cuestiones mencionadas arriba.

```
/*********************************************************
* WebPageReader.java
* Dean & Dean
*
* Esto lee una página en la red.
*********************************************************/
import java.util.Scanner;

public class WebPageReader
{
  BufferedReader reader;

  public WebPageReader(String webAddress)
  {
    URL url = new URL(webAddress);
    URLConnection connection = url.openConnection();
    InputStream in = connection.getInputStream();

    reader = new BufferedReader(new InputStreamReader(in));
  } // end constructor
  //*******************************************************

  public String readLine()
  {
    return reader.readLine();
```

```
  } // end readLine
  //*******************************************************

  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String url, line;

    System.out.print("Introduzca una dirección URL completa: ");
    url = stdIn.nextLine();
    WebPageReader wpr = new WebPageReader(url);

    while ((line = wpr.readLine()) != null)
    {
      System.out.println(line);
    }
  } // end main
} // end WebPageReader
```

Esté prevenido pues su programa puede ser correcto pero quizá no pueda acceder exitosamente a páginas en la red. Para lograr lo anterior, su computadora debe tener capacidades de acceso a Internet. Si su cortafuegos repregunta si está bien que Java acceda a Internet, haga clic en "yes" y continúe. A continuación se presentan tres sesiones de muestra:

Primera sesión muestra:

```
Introduzca una dirección URL completa: htp://www.park.edu
protocolo desconocido: htp
```

Segunda sesión muestra:

```
Introduzca una dirección URL completa: http:/www.park.edu
Conexión negada: conectar
```

Tercera sesión muestra:

```
Introduzca una dirección URL completa: http://www.park.edu

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Park University Home Page</title>
<meta name= Contenido de "TITLE"="Park University Home Page:
Programas de grados académicos de licenciatura, maestría y en línea" />
<meta name="ROBOTS" content="INDEX,FOLLOW" />
<meta name= Contenido de "Description"="Park University ofrece
programas de licenciatura, posgrado y en línea en todos los campus
nacionales. Park está acreditada por la Higher Learning Commission
de la North Central Association of Colleges and Schools." />
...
```

7. [Después de §14.10] Bloques `catch` múltiples:
Suponga que el código en un bloque `try` puede lanzar cualquiera de las siguientes excepciones:

a) `Exception`
b) `IllegalArgumentException`
c) `IOException`
d) `NumberFormatException`
e) `RunTimeException`

Identifique una secuencia aceptable de bloques `catch` múltiples de estos tipos.

8. [Después de §14.11] Corrección de problemas:
Arregle los problemas en el programa NumberList sin realizar ningún cambio en la clase `NumberListDriver`.

Si un usuario introduce de inmediato "q" para salir, imprima "NaN" mediante una pequeña corrección al programa que use un valor `NaN` `double` y evite el uso del mecanismo `try-catch` para atrapar la excepción aritmética `int`.

Sesión muestra:

```
Enter a whole number (q to quit): q
Mean = NaN
```

Si la entrada no es una "q", y si no es un entero legal, atrape la excepción, y en el bloque `catch` use el método `getClass` heredado de la clase `Object` para imprimir el nombre de la clase de excepción seguido del mensaje de error con la declaración:

```
System.out.println(e.getClass() + " " + e.getMessage());
```

Evite la posibilidad de una `ArrayIndexOutOfBoundsException` al agregar `size < numList.length` a la condición `while` y ejecutar la petición e introducir al final del ciclo `while` sólo si `size < numList.length`.

9. [Después de §14.12] TestExceptions:
¿Cuál es la salida del siguiente programa? Puesto que el programa convierte entre cadenas y valores numéricos, use comillas para denotar valores de cadena.

```
/*************************************************************
* TestExceptions.java
* Dean & Dean
*
* Esto busca el valor en el índice calculado.
*************************************************************/

public class TestExceptions
{
  private double[] value =
    new double[] {1.0, 0.97, 0.87, 0.7, 0.47, 0.17};
  private int num;
  //***********************************************************

  public double eval(String n1, String n2)
    throws IndexOutOfBoundsException
  {
    try
    {
      num = Integer.parseInt(n1) / Integer.parseInt(n2);
    }
    catch (NumberFormatException nfe)
    {
      num++;
      System.out.println("en el primer catch");
    }
    catch (ArithmeticException ae)
    {
      num++;
      System.out.println("en el segundo catch");
    }
    return value[num];
  }
  //***********************************************************

  public static void main(String[] args)
  {
    TestExceptions te = new TestExceptions();
    try
    {
      System.out.println(te.eval("5.0", "4"));
```

```
        System.out.println(te.eval("5", "0"));
        System.out.println(te.eval("22", "5"));
        System.out.println(te.eval("33", "5"));
      }
      catch (Exception e)
      {
        System.out.println("en el catch de main");
      }
      System.out.println("Bye");
    } // end main
  } // end TestExceptions class
```

Soluciones a las preguntas de revisión

1. Falso. Muchas llamadas a métodos API son seguras, y no es necesario colocar estos métodos dentro de un bloque `try`.
2. Cierto. No es posible colocar declaraciones entre bloques asociados `try` y `catch`.
3. Cierto.
4. Las declaraciones seguras que usan las operaciones peligrosas deben colocarse dentro del bloque `try` y después de tales declaraciones peligrosas.
5. Falso. Después de ejecutar el bloque `catch`, la JVM continúa hacia abajo; no salta de regreso a un bloque `try`. En consecuencia, las declaraciones del bloque `try` se omiten si están a continuación de una declaración que lanza una excepción.
6. Cierto.
7. Si una excepción se deriva de la clase `RunTimeException`, se trata de una excepción no comprobada.
8. Las excepciones comprobadas son excepciones que están en o se derivan de la clase `Exception`, pero que no están o no se derivan de la clase `RunTimeException`.
9. Opciones viables para una excepción no comprobada que usted sabe que su programa podría lanzar:
   a) ¡No es viable! El lector no desea que su programa se caiga durante el tiempo de ejecución.
   b) Viable.
   c) Viable.
10. Cierto.
11. Cierto. Si la declaración contiene una excepción comprobada, el compilador indica este hecho e identifica el tipo de excepción.
12. Sí.
13. La excepción `Exception`.
14. El método `getMessage` de la clase `Exception` devuelve una descripción de texto de la excepción lanzada.
15. Falso. Puede usarse un bloque `catch` por separado para atrapar tipos distintos de excepciones.
16. Cierto. El compilador podría quejarse si un bloque más genérico `catch` anterior se adelanta a un bloque más específico `catch` posterior.
17. Los dos tipos de información mostrados por la JVM cuando ésta encuentra un error de tipo de ejecución son:
   a) Identificación del lanzamiento de una excepción particular.
   b) Una llamada a la traza de una pila, que es un listado en orden inverso de los métodos llamados justo antes de la caída, junto con los números de línea donde ocurrió el error en cada método.
18. Es necesario agregar `throws NumberFormatException` al final del encabezado del método.
19. No. Cuando se lanza una excepción de vuelta al método que llama, la JVM no devuelve un valor (con una declaración `return`) al módulo que llama.

CAPÍTULO 15

Archivos

Objetivos

- Conocer las clases que hay en el paquete `java.io`.
- Aprender a escribir texto y datos en un archivo de texto.
- Aprender a leer texto y datos en un archivo de texto.
- Usar un archivo de texto de E/S en una actividad de traducción de datos.
- Comprender las diferencias entre formatos de archivos de texto y binarios.
- Aprender a escribir y leer valores primitivos desde y hacia archivos binarios.
- Aprender a usar un encabezado de un archivo de datos.
- Aprender a escribir y leer objetos de y hacia archivos objeto de Java.
- Usar la clase `File` de API para reunir información sobre un archivo específico.
- Implementar la funcionalidad del selector de archivo GUI con la clase `JFileChooser` de API.

Relación de temas

15.1 Introducción

Hasta ahora, toda la entrada al programa ha provenido del teclado y toda la salida se ha ido a la pantalla de la computadora. Pero este tipo de entrada/salida (E/S) es temporal. Cuando la salida se envía a la pantalla de la computadora, no se guarda. Un día después, si se desea verla de nuevo, es necesario volver a ejecutar el programa. En forma semejante, cuando la entrada se introduce desde el teclado, la entrada no se guarda. Un día después, si se desea usar la misma entrada, es necesario introducirla otra vez.

Para E/S permanente o reusable, los datos de entrada o de salida pueden almacenarse en un *archivo*. Un archivo es un grupo de datos relacionados que usualmente se almacenan en un bloque contiguo o en un dispositivo de almacenamiento no volátil (como un disco duro). Los archivos de datos son fundamentalmente los mismos que los archivos de programa `.java` y `.class` que el lector ya ha utilizado todo el tiempo para guardar sus programas de Java. Pero en lugar de contener programas, los archivos de datos

contienen datos que los programas leen como entrada o escriben como salida. Para evitar que el lector o su computadora se confundan, debe nombrar sus archivos de datos con una extensión que los identifique como datos, como `.txt` o `.data`.

En este capítulo se aprenderá a escribir código que elabore archivos de datos de salida y almacene datos en estos archivos, y se aprenderá a escribir código que lea datos de archivos de entrada preexistentes. Todo esto se aprenderá con *archivos de texto* simples, que guardan sus datos como caracteres. Se aprenderá a hacer más eficiente lo anterior con *archivos binarios*, que guardan sus datos en formato *nativo*: el formato que usa la computadora cuando procesa datos durante la ejecución del programa. Y se aprenderá a almacenar objetos completos en *archivos objeto.*

15.2 Clases API de Java que se precisa importar

En los programas que manipulan archivos se verán varias clases preconstruidas de Java. Para acceder a estas clases es necesario un paquete `import` que las contiene. En la figura 15.1 se muestra el subconjunto de clases API de Java que se analizarán en este capítulo. Se podrá identificar la clase `Scanner` porque ya se ha utilizado para obtener información de entrada desde un teclado. También, en este capítulo, se aprenderá a usarla para obtener información de un archivo. Como ya se sabe, esta clase se puede importar con esta declaración:

```
import java.util.Scanner;
```

Todo el resto de las clases en la figura 15.1 están en el paquete `java.io`. Por lo general, se requiere más de una y es posible importar cualquier combinación con esta declaración comodín:

```
import java.io.*;
```

Los tres agrupamientos que aparecen en la figura 15.1, "texto", "binaria" y "objeto", identifican tres estrategias de E/S distintas. Cada una tiene su sitio. La E/S de texto es una forma práctica de almacenar tipos de datos primitivos. Resulta relativamente fácil leer o escribir archivos de texto en Java, y para leer o escribir archivos de texto es posible usar casi cualquier otro tipo de programa de computadora (como los procesadores de texto y las hojas de cálculo). Los archivos binarios guardan datos de manera más eficiente que los archivos de texto o los archivos objeto. Cuando los datos a almacenar son complejos, implican objetos o una combinación de objetos y primitivos, los archivos objeto son más fáciles de usar. Tenga en mente que es difícil inspeccionar un archivo binario o un archivo objeto porque estos archivos no pueden leerse con programas que procesan texto. Para leer un archivo binario, es necesario saber cómo está organizado. Para leer un archivo objeto, es necesario conocer el tipo de objetos que contiene, aunque es posible ignorar su estructura interna porque la información estructural insertada por la JVM cuando se escribe un archivo objeto, indica a la JVM cómo está organizado el archivo cuando se lee.

Además de las clases que aparecen en la figura 15.1, también hay muchas otras clases que tratan con archivos de E/S, y sus operaciones se traslapan. En otras palabras, algunas clases manejan la misma clase de operación de archivos que otras clases. ¿Por qué hay tantas clases? Cada clase posee características diferentes, de modo que algunas clases funcionan mejor en ciertas situaciones. También hay una razón histórica. Las clases de archivos de E/S se fueron agregando gradualmente al lenguaje Java. Siempre que los diseñadores de Java se percataban de que las clases de E/S eran deficientes de alguna manera, no modificaban las clases existentes porque hacerlo hubiera creado un caos en el código existente que dependía de esas clases. Así, decidieron agregar nuevas clases. Desafortunadamente, el resultado es que hay muchas clases de archivos de E/S, y es difícil recordar lo que hace cada una. Aquí sólo se analizarán las clases más útiles y directas.

La forma más sencilla de almacenar texto y/o representaciones de números como texto es en archivos de texto. En estos archivos, todo se representa en términos de caracteres ASCII, lo cual se describió en el capítulo 11. Como se muestra en la figura 15.4, cada carácter ASCII se codifica en una secuencia de ocho bits de ceros y unos. Ese esquema de codificación se analizará con mayor detalle después en este capítulo, en la sección 15.6. Para escribir texto en un archivo de texto, se recomienda usar la clase `PrintWriter`, que tiene métodos `println`, `print` y `printf`. Estos métodos imitan los métodos con el mismo nombre que ya se han llamado desde `System.out` desde hace mucho tiempo. Los métodos `System.out` imprimen hacia el monitor de la computadora. Los métodos `PrintWriter` imprimen

Archivo de texto de E/S. Para datos primitivos. Fácil de comprender.
La computadora transforma datos primitivos de formato nativo en un formato de texto legible para archivos.
El lector puede crear o ver archivos de texto con casi cualquier editor de texto.

para salida hacia un archivo de texto:
`PrintWriter`
`FileWriter`
para entrada desde un archivo de texto:
`Scanner`
`FileReader`

Archivo binario de E/S. Para datos primitivos. Eficiente.
Los archivos obtienen datos primitivos en formato nativo.
No es posible crear ni ver archivos binarios con un editor de texto, aunque los archivos binarios son bastante compactos.

para salida hacia un archivo binario:
`DataOutputStream`
`FileOutputStream`
para entrada desde un archivo binario:
`DataInputStream`
`FileInputStream`

Archivo objeto de E/S. Para objetos completos. Fácil de usar.
La computadora descompone objetos en datos primitivos para archivos, aunque también obtiene encabezados descriptivos.
No es posible crear ni ver archivos objeto con un editor de texto, aunque la codificación se minimiza.

para salida hacia un archivo objeto:
`ObjectOutputStream`
`FileOutputStream` <lo mismo que para salida binaria>
para entrada desde un archivo objeto:
`ObjectInputStream`
`FileInputStream` <lo mismo que para entrada binaria>

Figura 15.1 Clases que se recomienda usar para archivo de E/S.
La clase `Scanner` está en el paquete `java.util`. Las otras están en el paquete `java.io`.

hacia un archivo. Para leer texto desde un archivo de texto se recomienda usar la clase `Scanner`, que tiene los métodos `nextLine`, `next`, `nextInt`, `nextLong`, `nextFloat` y `nextDouble` que ya se han estado llamando desde hace tiempo. Pero ahora el lector utilizará estos métodos para obtener entrada desde un archivo, y no desde un teclado. La clase `Scanner` funciona de manera conjunta con la clase `FileReader`. El constructor `Scanner` se instanció con un argumento `FileReader`. En las secciones 15.3 y 15.4 se presentan ejemplos de archivos de texto que utilizan `PrintWriter`, `Scanner` y `FileReader`.

La forma más eficiente de almacenar arreglos homogéneos de datos primitivos es en archivos binarios. Para escribir datos primitivos en un archivo binario, un objeto de la clase `DataOutputStream` se instancia con un argumento que se refiere a un objeto de la clase `FileOutputStream`. (Una *corriente* es un flujo secuencial de datos.) Para leer datos primitivos de un archivo binario, un objeto de la clase `DataInputStream` se instancia con un argumento que se refiere a un objeto de la clase `FileInputStream`. El uso de `DataOutputStream`, `FileOutputStream`, `DataInputStream` y `FileInputStream` se ilustra en la sección 15.7.

Si se tiene una cantidad considerable de datos en forma de objetos que es necesario transferir a otros programas de Java, es conveniente usar archivos objeto. Para escribir objetos (y cualquier combinación

de primitivos) en un archivo objeto, un objeto de la clase `ObjectOutputStream` se instancia con un argumento que se refiere a un objeto de la clase `FileOutputStream`. Para leer objetos desde un archivo objeto, un objeto de la clase `ObjectInputStream` se instancia con un argumento que se refiere a un objeto de la clase `FileInputStream`. En la sección 15.8 se presentan ejemplos que usan `ObjectOutputStream`, `FileOutputStream`, `ObjectInputStream` y `FileInputStream`.

Los lectores interesados en las clases API de Java de archivos de E/S, pueden consultar el sitio API de Java en la dirección http://java.sun.com/javase/6/docs/api/.

15.3 Salida a archivos de texto

Los lectores que deseen usar un archivo de E/S, tienen la opción de leer esta sección y la siguiente antes de terminar la sección 3.23 del capítulo 3. Si el lector decide saltar desde el capítulo 3 hasta aquí, debe saber que algo del material en las secciones 15.3 y 15.4 no tienen sentido. Pero si el lector considera el programa de la figura 15.2 como una receta, le mostrará cómo sacar de un archivo cualquier cosa que pueda desplegar a la pantalla de la computadora. En forma semejante, si el lector considera el programa de la figura 15.5 como una receta, le mostrará cómo introducir desde un archivo cualquier cosa que pueda introducir desde el teclado.

En esta sección se mostrará cómo usar un objeto `PrintWriter` para sacar texto en un archivo. En la sección 15.4 se mostrará cómo usar un objeto `Scanner` para introducir texto desde un archivo. Para todo archivo de E/S, hay tres pasos básicos:

- Abrir el archivo para instanciar la(s) clase(s) idónea(s).
- Escribir hacia o leer desde el archivo al llamar al método idóneo.
- Cerrar el archivo al llamar al método `close`.

A continuación se considerarán estos tres pasos en relación con la clase `PrintWriter`.

Apertura de un archivo de texto para salida

Para abrir un archivo de texto para salida, la clase `PrintWriter` debe instanciarse como se muestra a continuación:

```
PrintWriter <reference-variable>;
...
<reference-variable> = new PrintWriter(<filename>);
```

Observe que en esta declaración no hay ningún comando "open" explícito. Simplemente se instancia un objeto `PrintWriter`, con lo cual se abre automáticamente el archivo especificado por el argumento del nombre del archivo del constructor `PrintWriter`. Si el nombre del archivo es inválido, la JVM lanza una `FileNotFoundException`. Hay varias formas de que el nombre del archivo sea inválido: 1) El nombre del archivo puede contener un carácter inválido en el nombre del archivo, como un asterisco, 2) el nombre del archivo puede especificar un directorio en lugar de un archivo, 3) el nombre del archivo puede especificar un directorio inexistente seguido del nombre de un archivo (por ejemplo, `javaFiles/Mouse.java`). La `FileNotFoundException` es una excepción comprobada, de modo que la llamada al constructor `PrintWriter` debe estar en un bloque `try`, y el parámetro del bloque `catch` correspondiente debe coincidir con la `FileNotFoundException`.

Para abrir un archivo para salida de texto, todo lo que se requiere es escribir la declaración especificada arriba. No es necesario saber cómo funciona. Pero si el lector es curioso, he aquí una explicación: siempre que se instancia un objeto `PrintWriter`, también se obtiene automáticamente un objeto `FileWriter`. El objeto `FileWriter` usa uno de los métodos `write` que hereda de su padre, `OutputStreamWriter`, para transformar una *corriente* (o una secuencia) de caracteres Unicode en una corriente de bytes: el formato de datos natural del archivo. Además, el `FileWriter` proporciona un *buffer* para esa corriente final de bytes. Un buffer es un arreglo de longitud variable que absorbe variaciones del flujo de datos, de modo que el flujo de datos fuera del procesador (donde está el programa) no tiene que estar perfectamente sincronizado con el flujo de datos hacia la memoria (donde está el archivo). Suponga que el hardware que presta servicio a la memoria está ocupado. Si no hubiese buffer, el pro-

grama tendría que detenerse y esperar hasta que el hardware estuviese disponible. Pero con un buffer, el procesador puede simplemente colocar los datos en el buffer y dejar que se apilen ahí. Luego, cuando el hardware que presta servicio a la memoria esté libre, puede sacar todo lo que se haya acumulado en el buffer y transferirlo hacia la memoria en un trozo relativamente grande. Debido a su capacidad para absorber variaciones en los ritmos de flujo de datos, un buffer constituye un gran mejorador de rendimiento: hace que el programa se ejecute más rápido. Así, puede verse que abrir un archivo es una gran operación. Constituye una "infraestructura" sustancial para el transporte de datos desde el procesador hasta el archivo siempre que se requiera.

Programa de ejemplo

Observe el programa WriteTextFile en la figura 15.2. Ilustra cómo usar la clase `PrintWriter` para salida a archivos de texto. El constructor `PrintWriter` provoca que la JVM abra el archivo especificado por el usuario justo como acaba de describirse. En caso de que el archivo especificado no exista, el programa crea uno nuevo. Si el archivo especificado ya existe, el programa limpia el contenido de ese archivo antes de escribir los nuevos datos.

Puesto que el constructor `PrintWriter` lanza una excepción comprobada, el programa WriteTextFile inserta la llamada al constructor `PrintWriter` en un bloque `try`, y proporciona un bloque correspondiente `catch FileNotFoundException`.[1] El bloque `catch` llama a `e.getMessage`, que devuelve el mensaje de error generado automáticamente por la excepción.

Para escribir en el archivo de texto abierto, el programa llama al método `println` de `PrintWriter`. Como ya se indicó, funciona como `System.out.println`, que en forma automática agrega una terminación de línea. La clase `PrintWriter` también cuenta con métodos `print` y `printf`, que funcionan como los métodos `print` y `printf` de `System.out`. Estos dos últimos métodos no proporcionan terminaciones de línea en forma automática, de modo que si se desean más líneas con ellos, es necesario tener la seguridad de suministrar explícitamente caracteres `\n`. Todos estos métodos aceptan argumentos `String` y convierten estas cadenas en corrientes de caracteres. En el fondo, el objeto `FileWriter` transforma las corrientes de caracteres en corrientes de bytes para el archivo.

Para cerrar el archivo de texto abierto, el programa llama al método `close` de `PrintWriter` usando este formato:

<`PrintWriter`-reference-variable>`.close();`

⚠ Esto hace salir cualquier corriente parcialmente llena y desmonta la "infraestructura" de salida. Recuerde cerrar el archivo una vez que haya terminado lo que esté haciendo. Si escribe un archivo `PrintWriter` y olvida cerrar el archivo, todos los datos de la o las operaciones en el archivo de escritura pueden no guardarse en el archivo. También, si se olvida cerrar un archivo, los recursos del sistema permanecen asignados al archivo abierto, lo que provoca la degradación del rendimiento del sistema.

Agregación de datos a un archivo existente (Append)

Suponga que ya se tienen datos en un archivo existente y que es necesario agregar datos al archivo. Para agregar datos nuevos a un archivo existente, se llama al constructor `PrintWriter`, como antes, pero esta vez es necesario obtener ayuda de un objeto `FileWriter`. Específicamente, es necesario pasar un objeto `FileWriter` al constructor `PrintWriter`. Para saber de qué se está hablando, analice el código de instanciación de `PrintWriter` en la figura 15.3.

Para abrir un archivo de salida de texto en modo de agregación, todo lo que se tiene que hacer es escribir la declaración `fileOut =` que se muestra en la figura 15.3. No es necesario saber por qué esto es así. Pero si el lector es curioso, he aquí una explicación: ninguno de los constructores sobrecargados `PrintWriter` incluye un parámetro `append`, aunque hay un constructor `PrintWriter` que sí lo hace. Entonces, en lugar de usar el constructor `PrintWriter` que instancia en forma automática un

[1] Como se indicó en el capítulo previo, los bloques `try` y `catch` de un método pueden eliminarse al agregar una cláusula `throws` al encabezado del método. Sin embargo, aquí no se recomienda esta práctica, porque separa en forma innecesaria el manejo de excepciones a partir del momento en que ocurre la excepción, lo cual dificulta la comprensión y depuración del programa.

```
/*************************************************
* WriteTextFile.java
* Dean & Dean
*
* Esto escribe una cadena en un archivo de texto.
*************************************************/

import java.util.Scanner;
import java.io.*;

public class WriteTextFile
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    PrintWriter fileOut;
    String text = "¡Hola, mundo!";

    try
    {
      System.out.print("Introduzca el nombre del archivo: ");
      fileOut = new PrintWriter(stdIn.nextLine());
      fileOut.println(text);
      fileOut.close();
    }
    catch (FileNotFoundException e)
    {
      System.out.println("Error: " + e.getMessage());
    }
  } // end main
} // end WriteTextFile class
```

Abre el archivo.

Escribe en el archivo.

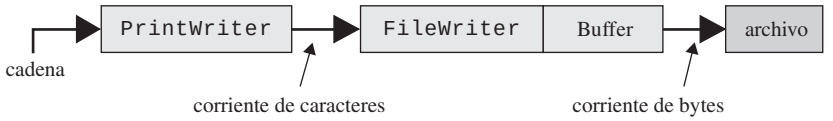
Cierra el archivo.

Llamada para el constructor `PrintWriter`

Figura 15.2 Programa WriteTextFile para escribir texto en un nuevo archivo o rescribir un archivo anterior.

objeto `FileWriter` que nunca agrega, se puede construir explícitamente un objeto `FileWriter` que pueda agregar. Luego, este objeto puede pasarse a un constructor sobrecargado `PrintWriter` que tenga un tipo de parámetro `Writer`. Puesto que la clase `FileWriter` desciende de la clase `Writer`, este otro constructor `PrintWriter` acepta un objeto `FileWriter` como su argumento.

El segundo parámetro en el constructor `FileWriter` es un `boolean` que indica a la computadora si se desea agregar datos o no. El uso de `true` indica que se desea agregar datos. En ese caso, se puede usar esta declaración de apertura de archivo más elaborada para crear un nuevo archivo o rescribir por completo en un archivo de salida al usar `false` para el segundo argumento en el constructor `FileWriter`. Observe cómo esta forma alternativa de abrir un archivo de texto para salida requiere un `catch` más genérico para atrapar una `FileNotFoundException` o una `IOException` lanzada por el constructor `FileWriter`. (El constructor `PrintWriter` que se usa en la figura 15.2 atrapa en forma interna a la `IOException` de `FileWriter`.) El `FileWriter` en la figura 15.3 realiza de manera explícita la misma operación básica de canalización de corrientes que el `FileWriter` en la figura 15.2 realiza de manera implícita. En cualquier caso, he aquí a lo que se parece el proceso de salida de texto:[2]



[2] Opcionalmente, la ejecución puede acelerarse al insertar un objeto `BufferedWriter` entre los objetos `PrintWriter` y `FileWriter`.

```
/*****************************************************************
* WriteTextFile2.java
* Dean & Dean
*
* Esto agrega datos en un archivo de texto existente.
*****************************************************************/

import java.util.Scanner;
import java.io.*;

public class WriteTextFile2
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    PrintWriter fileOut;
    String text = "¡Hola, mundo!";

    try
    {
      System.out.print("Introduzca el nombre del archivo: ");
      fileOut =
        new PrintWriter(new FileWriter(stdIn.nextLine(), true));
      fileOut.println(text);
      fileOut.close();
    }
    catch (IOException e)
    {
      System.out.println("IO: " + e.getMessage());
    }
  } // end main
} // end WriteTextFile2 class
```

valor pasado al parámetro `boolean append`

La `IOException` necesaria por el `FileWriter` también atrapa la `FileNotFoundException`.

Figura 15.3 Programa `WriteTextFile2` para agregar texto en un archivo existente.

En la figura 15.3 hay una cuestión adicional que conviene mencionar. Observe la asignación `fileOut`, que se repite a continuación:

```
fileOut = new PrintWriter(new FileWriter(stdIn.nextLine(), true));
```

objeto anónimo

El código `new FileWriter`, insertado en la llamada al constructor `PrintWriter`, es un ejemplo de objeto anónimo. Los objetos anónimos usualmente son bastante comunes cuando se trabaja con archivos porque los constructores de archivos a menudo usan como argumentos otros objetos del archivo. En tales casos, no es necesario guardar el archivo del argumento recientemente instanciado en una variable por separado. Simplemente se le usa en forma anónima.

15.4 Lectura de archivos de texto

Suponga que se dispone de una gran cantidad de datos de entrada que es necesario utilizar más de una vez. En lugar de introducirlos directamente desde un teclado en forma repetida, es más eficiente y confiable introducirlos en un archivo una sola vez. Un archivo de texto legible por Java puede crearse con casi cualquier editor de texto o procesador de palabras, en el supuesto de que se guarde como "texto llano". Luego, los datos se leen desde el archivo cada vez que es necesario. En esta sección se mostrará cómo leer datos de entrada de un archivo de texto.

Apertura de un archivo de texto para entrada

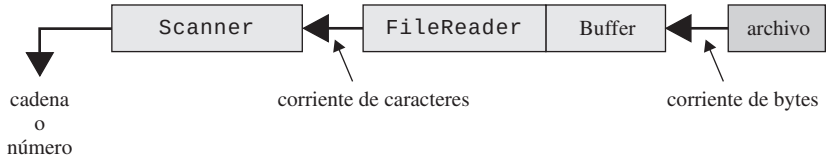
Es posible abrir un archivo para entrada usando una instancia de la clase `FileReader` con el argumento del nombre del archivo. Eso permite leer un carácter a la vez. Algunas veces eso es lo que se quiere hacer. Aunque en general se lee una línea completa, una palabra completa o algún tipo de número. `Scanner` cuenta con métodos que ejecutan estas otras operaciones. Así, para abrir un archivo de texto para entrada de cadenas, se recomienda instanciar las clases `FileReader` y `Scanner` juntas en una simple declaración, como se muestra a continuación:

```
Scanner <reference-variable>;
...
<reference-variable> = new Scanner(new FileReader(<filename>));
```

⚠ Si el archivo especificado por el argumento del nombre del archivo del constructor `FileReader` es inválido, la JVM lanza una `FileNotFoundException`. Esto debe sonar conocido: lo mismo se dijo sobre el constructor `PrintWriter` para salida a archivos de texto. Pero hay una diferencia importante. El constructor `PrintWriter` permite que el argumento del nombre del archivo especifique un archivo inexistente. El constructor `FileReader` requiere que el argumento del nombre del archivo especifique un archivo existente. La `FileNotFoundException` es una excepción comprobada; así, a menos que el lector quiera usar una cláusula `throws`, es necesario colocar la llamada al constructor `FileReader` en un bloque `try`, y el parámetro del bloque `catch` correspondiente debe coincidir con la `FileNotFoundException`.

Para abrir un archivo para entrada de texto, todo lo que se requiere es usar `new Scanner(new FileReader(<filename>))` como se muestra arriba. No es necesario saber por qué esto es así. Pero si el lector es curioso, he aquí una explicación: ninguno de los constructores `Scanner` acepta un argumento de un nombre de un archivo como lo hace el constructor `PrintWriter` en la figura 15.2.[3] Así, en caso de que el lector quiera usar `Scanner`, debe usarlo en combinación con otra clase que acepte como argumento el nombre de un archivo y que también coincida con el tipo de parámetro en uno de los constructores `Scanner` disponibles. Uno de los constructores `Scanner` tiene un tipo de parámetro `Readable`. Puesto que la clase `FileReader` implementa la interfaz `Readable`, es posible instanciar un objeto `FileReader` y proporcionarlo como un argumento en este constructor `Scanner`.

El objeto `FileReader` amortigua la entrada desde el archivo y transforma los bytes del archivo en caracteres. El objeto `Scanner` convierte la corriente de caracteres en cadenas y números. He aquí cómo se ve el proceso de entrada de texto:[4]



Programa de ejemplo

A continuación se verá cómo funciona la lectura de archivos de texto en el contexto de un programa completo. Suponga que se tiene un archivo de texto denominado `markAntony.txt` que contiene la cita de la obra *Julio César* de Shakespeare en la figura 15.4*a*. También, suponga que se tiene un texto denominado `randomNumbers.txt` que contiene la lista de números aleatorios que se muestra en la figura 15.4*b*.

La figura 15.5 contiene un programa ReadTextFile que puede leer los datos en cualquiera de estos archivos. Tenga en cuenta que los números que aparecen en la figura 15.4*b* no van hacia el programa como números. Al programa van como representaciones en cadena de números. El programa ReadTextFile pide al usuario el nombre de un archivo (como `markAntony.txt` o `randomNumbers.txt`), lee el archivo especificado e imprime el contenido del archivo. La mayor parte del código del programa es directa, aunque otras partes merecen atención. . .

[3] El constructor `Scanner` con un parámetro `String` lee una cadena llana anterior, no un archivo.
[4] Opcionalmente, la ejecución puede acelerarse al insertar un objeto `BufferedReader` entre los objetos `Scanner` y `FileReader`.

```
Friends, Romans, countrymen,
Lend me your ears;
I come to bury Caesar,
not to praise him.
```

Figura 15.4*a* Contenido del archivo de texto `markAntony.txt`.

```
0.9709900750891582    0.3874009922012617    0.1262329780823327
0.7782696919307651    0.15480236215303655    0.9756100238518657
```

Figura 15.4*b* Contenido del archivo de texto `randomNumbers.txt`.

Ya de un tiempo para acá se ha utilizado la clase `Scanner` para entrada desde el teclado. Cuando la clase `Scanner` se usa para lo anterior, sigue siendo posible usar a los viejos conocidos `nextInt`, `nextDouble`, `nextLine`, etc., de la misma forma en que se hizo para entrada desde el teclado. Pero

```
/**************************************************************
* ReadTextFile.java
* Dean & Dean
*
* Esto lee datos desde un archivo de texto.
**************************************************************/
import java.util.Scanner;
import java.io.*;

public class ReadTextFile
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Scanner fileIn;
    String line;

    try
    {
      System.out.print("Introduzca el nombre del archivo: ");
      fileIn = new Scanner(new FileReader(stdIn.nextLine()));
      while (fileIn.hasNextLine())
      {
        line = fileIn.nextLine();
        System.out.println(line);
      }
      fileIn.close();
    }
    catch (FileNotFoundException e)
    {
      System.out.println("Error: " + e.getMessage());
    }
  } // end main
} // end ReadTextFile class
```

Abre el archivo.

Lee una línea desde el archivo y la imprime.

Cierra el archivo.

Figura 15.5 Programa ReadTextFile que lee texto desde el teclado y también desde un archivo de texto.

con entrada a archivos es necesario tener en cuenta varios métodos `Scanner` adicionales. Cuando se ha terminado de leer desde el archivo `Scanner`, debe llamarse su método `close`. Vea `fileIn.close()` en la figura 15.5. También, cuando desde un archivo se lee una serie de líneas, a menudo es aconsejable usar el método `hasNextLine` de `Scanner` en el encabezado de un ciclo `while`, como se hace en la figura 15.5.

El método `hasNextLine` de `Scanner` proporciona una señal de terminación de ciclo idónea cuando se están leyendo datos con el método `NextLine` de `Scanner`. La ejecución del ciclo termina en forma automática al final del archivo cuando ya no hay líneas por leer. El programa en la figura 15.4 puede usarse para leer ya sea el texto en la figura 15.4*a* o los números en la figura 15.4*b*. Funciona para los números y para el texto porque el programa simplemente imprime los números y los espacios entre ellos a medida que los lee, como una secuencia de caracteres, sin molestarse en interpretar los significados de estas secuencias de caracteres.

En lugar de leer una línea completa a la vez, el método `hasNext` de `Scanner` se puede usar como una señal de terminación de ciclo y luego leer datos con el método `next` de `Scanner`. El método `next` lee un *token* a la vez. Un *token* es una secuencia de caracteres separados de secuencias precedentes y subsiguientes por espacio en blanco. En otras palabras, un *token* es una palabra aislada o un número aislado. Siempre que se lee una línea o un token a la vez, ¡resulta difícil no leer texto puro!

Es otra historia, sin embargo, si el programa requiere conocer los valores numéricos de los números que hay en un archivo de texto. Si el archivo de texto tiene espacios en blanco entre números adyacentes, la computadora puede leer tales números y de manera simultánea *analizarlos* (determinar sus valores numéricos) usando métodos `Scanner` como `nextInt` o `nextDouble`. Estos métodos lanzan excepciones no comprobadas en caso de que ocurran errores de análisis. Si se quiere mejorar el programa en la figura 15.5 de modo que incluya análisis numérico, es conveniente atrapar las excepciones no comprobadas lanzadas por los errores de análisis.

Lectura de datos formateados en un archivo de texto

Use sus conocimientos sobre formato.

Si se quiere usar `nextInt` o `nextDouble` de `Scanner` para analizar datos numéricos a medida que se introducen, se debe usar un espacio en blanco para hacer de cada número un *token* por separado. Sin embargo, en un archivo de texto no es absolutamente necesario usar un espacio en blanco para separar datos. En lugar de lo anterior, puede seguirse la pista de la posición del carácter al inicio y al final de cada dato. Esto es lo que se hacía en la edad de hielo. Por ejemplo, suponga que cada línea en el archivo está formateada en tres campos, como esto:

las columnas 0-20 contienen un `String` que puede contener más de una palabra
las columnas 21-28 contienen la representación de texto de un `int`
las columnas 30-42 contienen la representación de texto de un `double`

Para leer los tres campos, es necesario declarar las siguientes variables:

```
String text;
int iNum;
double dNum;
```

Cada línea se lee como texto puro y luego se analiza como sigue:

```
...
line = in.nextLine();
text = line.substring(0,21);
iNum = Integer.parseInt(line.substring(21,29).trim());
dNum = Double.parseDouble(line.substring(30,43).trim());
```

Como quizá se recuerde, el segundo argumento de `substring` es uno más grande que el índice del último carácter en la subcadena que debe extraerse. Por tanto, en la tercera declaración del fragmento de código anterior, 29 es uno más grande que la última columna del campo `int`.

15.5 Generador de archivos HTML

A continuación se considerará un ejemplo que ilustra tanto la entrada a un archivo de texto como la salida desde un archivo de texto. El programa de las figuras 15.6*a* y 15.6*b* lee el contenido de un archivo de texto especificado por el usuario. Traduce esos datos en un formato de página Web. Luego, pasa la traducción a un nuevo archivo generado por HTML.

En la figura 15.6*a*, el programa empieza por leer un nombre de archivo especificado por el usuario hacia la variable `filenameIn`. Luego, usa el nombre de archivo introducido a fin de abrir el archivo de entrada y crear un objeto `Scanner` denominado `fileIn` para administrar las operaciones de lectura en el archivo.

El nombre del archivo de salida debe ser el mismo que el nombre del archivo de entrada, excepto por la extensión, que debe ser `.html`. Para componer el nombre del archivo de salida, el método `lastIn-`

```
/*******************************************************
* HTMLGenerator.java
* Dean & Dean
*
* Este programa copia el contenido de un archivo especificado
* por el usuario y lo pasa a un nuevo archivo generado por HTML.
*******************************************************/

import java.util.Scanner;
import java.io.*;

public class HTMLGenerator
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String filenameIn;     // nombre del archivo original
    Scanner fileIn;        // conexión al archivo de entrada
    int dotIndex;          // posición del punto en el nombre del archivo
    String filenameOut;    // nombre del archivo HTML
    PrintWriter fileOut;   // conexión al archivo HTML
    String line;           // línea desde el archivo de entrada

    System.out.print("Introduzca el nombre del archivo: ");
    filenameIn = stdIn.nextLine();

    try                                          Esto abre un archivo para entrada.
    {
      fileIn = new Scanner(new FileReader(filenameIn));

      // Compose the new filename
      dotIndex = filenameIn.lastIndexOf(".");
      if (dotIndex == -1) // no dot found
      {
        filenameOut = filenameIn + ".html";
      }
      else // dot found
      {
        filenameOut =
          filenameIn.substring(0, dotIndex) + ".html";
      }
      fileOut = new PrintWriter(filenameOut);    Esto abre un archivo para salida.
```

Figura 15.6*a* Programa HTMLGenerator, parte A.

dexOf de String encuentra el índice del último punto en filenameIn. En caso de que no haya punto, el método lastIndexOf devuelve un valor de −1 y el programa simplemente agrega .html al nombre original del archivo. Si hay un punto, el método substring de String devuelve la parte de la cadena hasta el carácter que está inmediatamente antes del punto, y el programa agrega .html. Este proceso sustituye la extensión del archivo original con una extensión .html y asigna el resultado a filenameOut.

Antes de analizar el resto del programa, se hará una digresión para revisar brevemente el HTML (el lenguaje de computación usado para crear páginas de Internet). Este libro no es sobre HTML, aunque la

```
        // La primera línea se usa para elementos del título y del encabezado.
        line = fileIn.nextLine();
        if (line == null)
        {
          System.out.println(filenameIn + " está vacío.");
        }
        else
        {
          // Write the top of the HTML page.
          fileOut.println("<html>");
          fileOut.println("<head>");
          fileOut.println("<title>" + line + "</title>");
          fileOut.println("</head>");
          fileOut.println("<body>");
          fileOut.println("<h1>" + line + "</h1>");

          while (fileIn.hasNextLine())
          {
            line = fileIn.nextLine();

            // Blank lines generate p tags.
            if (line.isEmpty())
            {
              fileOut.println("<p>");
            } // end if
            else
            {
              fileOut.println(line);
            }
          } // end while

          // Write ending HTML code.
          fileOut.println("</body>");
          fileOut.println("</html>");
        } // end else
        fileIn.close();
        fileOut.close();
      } // end try

      catch (FileNotFoundException e)
      {
        System.out.println("Error: " + e.getMessage());
      } // end catch
    } // end main
  } // end class HTMLGenerator
```

Figura 15.6*b* Programa HTMLGenerator, parte B.

siguiente revisión es de ayuda para comprender el programa GeneratorHTML. También, conviene aprender algo de HMTL porque lo que hace marchar al lenguaje Java son las páginas de Internet.

Revisión de HTML:

- Las *etiquetas* HTML se escriben entre corchetes, y describen el objetivo de su texto asociado.
- Las etiquetas sin *slashes* (como las etiquetas `<html>`, `<head>` y `<title>`) se denominan *etiquetas iniciales*. Las etiquetas con *slashes* (como las etiquetas `</title>`), `</head>` y `</html`) se denominan *etiquetas finales*.
- Las etiquetas `<html>` y `</html>` rodean toda la página Web.
- El contenido entre las etiquetas `<head>` y `</head>` es el encabezado de una página HTML. El encabezado contiene información que describe la página HTML. Esta información es usada por el buscador y por los mecanismos de búsqueda, pero no es visible en la página Web.
- Las etiquetas `<title>` y `</title>` rodean la página que aparece en la barra del título de la página Web. Los mecanismos de búsqueda de Internet usan el contenido de `<title>` para encontrar páginas Web.
- El contenido entre las etiquetas `<body>` y `</body>` es el cuerpo para la página Web. El cuerpo contiene el texto mostrado en la página Web.
- Las etiquetas `<h1>` y `</h1>` rodean el texto que aparece como un encabezado dentro de una página Web. Los buscadores de la red usan fuentes grandes para mostrar texto rodeado por las etiquetas `<h1>`.
- Las etiquetas `<p>` indican el inicio de un nuevo párrafo. Los buscadores de la red generan una línea en blanco por cada etiqueta `<p>`, lo cual ayuda a separar los párrafos.

A continuación se analizará la figura 15.6*b*, la segunda mitad del programa GeneratorHTML. El código empieza por comprobar si hay un archivo de entrada vacío. Si está vacío, imprime un mensaje de advertencia. En caso contrario, hace lo siguiente. Imprime las etiquetas `<html>` y `<head>` en el archivo de salida. Imprime la primera línea del archivo de entrada en el archivo de salida, entre las etiquetas `<title>` y `</title>`. Luego, termina la sección del encabezado de la página Web al imprimir `</head>` en el archivo de salida. A continuación, reutiliza la primera línea del archivo de entrada y la imprime en el archivo de salida, rodeada de las etiquetas `<h1>` y `</h1>`. Luego ejecuta un ciclo a través de las líneas subsiguientes en el archivo de entrada. Para cada línea en blanco, imprime una etiqueta `<p>` en el archivo de salida, indicando un nuevo párrafo. Imprime cada línea que no esté en blanco hacia el archivo de salida tal como está.

Para ver cómo trabaja el programa HTMLGenerator cuando se aplica a un verdadero archivo de entrada, es necesario estudiar los archivos de entrada y de salida en la figura 15.7. Si se quiere comprobar que el programa HTMLGenerator genera una página Web que funciona, se debe crear el archivo `family.txt` de la figura 15.7 y luego ejecutar el programa HTMLGenerator con `family.txt` como entrada. Eso debe generar el archivo `family.html` de la figura 15.7. Se abre una ventana de búsqueda y en ésta se abre un archivo `family.html`. Por ejemplo, se abre una ventana de búsqueda Microsoft Internet Explorer y se ejecuta un comando File/Open. Ya está: ¡el archivo `family.html` debe verse desplegado como una pagina Web!

Observe la segunda llamada en la figura 15.7. Para cumplir las estrictas normas HTML, es necesario insertar una etiqueta `</p>` al final de cada párrafo. No obstante, muchas páginas Web reales sólo se ajustan a normas relajadas HTML, no a normas estrictas HTML. Los buscadores reales manejan páginas Web estrictas y relajadas, aunque buscadores futuros probablemente sólo manejen páginas Web estrictas. En un ejercicio al final del capítulo se pide que el lector mejore el programa HTMLGenerator de modo que genere etiquetas finales `</p>` al final de cada párrafo.

15.6 Formato de archivo de texto *versus* formato de archivo binario

En esta sección se compara el formato de texto usado en archivos de texto con el formato nativo binario usado en archivos binarios y archivos objeto.

Formato de texto

Los datos de archivos de texto se almacenan usando valores de ocho bits del American Standard Code for Information Interchange (ASCII). Puesto que el conjunto de caracteres ASCII constituye una norma

Archivo de entrada ejemplo, `family.txt`:

```
Nuestra familia

Tenemos una perrita, Barkley. Barkley es una buena perrita. Duerme mucho
y escarba en el césped. La alimentamos dos veces al día.

Tenemos dos hijas, Jordan y Caiden. Son niñas y les gusta comer, gritar
y jugar. Las queremos mucho.
```

Archivo de salida resultante, `family.html`:

```
<html>
<head>
<title>Our Family</title>
</head>
<body>
<h1>Our Family</h1>
<p>
Tenemos una perrita, Barkley. Barkley es una buena perrita. Duerme mucho
y escarba en el césped. La alimentamos dos veces al día.
<p>
Tenemos dos hijas, Jordan y Caiden. Son niñas y les gusta comer, gritar
y jugar. Las queremos mucho.
</body>
</html>
```

La primera línea del archivo de salida aparece dos veces en el archivo de salida.

Esto funciona, pero según las estrictas normas HTML, las páginas Web deben tener una etiqueta </p> al final de cada párrafo.

Figura 15.7 Archivo de entrada ejemplo para el programa HTMLGenerator y su archivo de salida resultante.

universal, los caracteres ASCII pueden ser leídos por casi todos los editores de texto[5] o procesadores de palabras, y pueden ser leídos por programas escritos en cualquier lenguaje, no sólo por Java.

Los archivos de texto están orientados por líneas. Cuando se escribe un archivo de texto, el método `println` de `PrintWriter` inserta automáticamente un símbolo de terminación de línea al final de la línea. En Microsoft Windows, inserta `\r\n`. (`\r` es el símbolo de regreso de carro y `\n` es el símbolo de nueva línea.) En UNIX, sólo inserta `\n`. Cuando el método `nextLine` de `Scanner` lee desde un archivo de texto, lee una línea completa de caracteres y como terminación de línea acepta `\r\n` o `\n` en sí, pero no incluye la terminación en la cadena recuperada. Puesto que `\n` es más simple, se usará en las ilustraciones que se presentan aquí. A continuación se verá un archivo de texto. Suponga que se tienen los datos:

```
Bob 2222
Paul5555
```

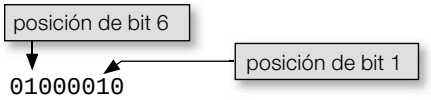
En la figura 15.8 se muestra cómo se almacenan estos datos en un archivo de texto.

A continuación se analizará el almacenamiento real para los caracteres en este archivo de texto. El valor del código ASCII para 'B' es el decimal 66. (El valor de 'B', junto con todos los demás valores en código ASCII se muestran en la figura 11.4.) Para encontrar el valor binario equivalente, se identifican todas las potencias de 2 que suman 66. 2^6 y 2^1 son las potencias de 2 que suman 66 ($2^6 = 64$, $2^1 = 2$ y $64 + 2 = 66$). Para cada potencia de 2 identificada, su exponente se usa como un indicador de lugar para 1 en el valor binario equivalente. Para el ejemplo 66, las potencias de 2, 2^6 y 2^1, tienen exponentes 6 y 1, de modo que las posiciones de los bits 6 y 1 son 1 en la siguiente representación binaria de 66. Observe que las posiciones de bits empiezan en 0 desde el lado derecho.

[5] Excepción: Microsoft Notepad sólo reconoce `\r\n` como terminación de línea. Entonces, si Notepad intenta leer un archivo de texto de Java que use `\n` como terminación de línea, muestra cada `\n` como un carácter no reconocido (□) y no genera una nueva línea.

B	o	b		2	2	2	2	\n	P
01000010	01101111	01100010	00100000	00110010	00110010	00110010	00110010	00001010	01010000
a	u	l	5	5	5	5	\n		
01100001	01110101	01101100	00110101	00110101	00110101	00110101	00001010		

Figura 15.8 Forma cruda de un formato de texto.
Los caracteres ASCII se muestran en letra negra (bold) arriba de cada byte. Cada línea lógica se termina con el carácter `\n`, pero en un archivo todo está relacionado entre sí en forma de secuencia continua. Aquí la secuencia se acomoda, de tal forma que todo se ajusta al ancho de la página disponible.



He aquí la explicación matemática de por qué el decimal 66 es equivalente al binario 01000010:

$$66 = (64 + 2) = (2^6 + 2^1) = (\mathbf{0}*2^7 + \mathbf{1}*2^6 + \mathbf{0}*2^5 + \mathbf{0}*2^4 + \mathbf{0}*2^3 + \mathbf{0}*2^2 + \mathbf{1}*2^1 + \mathbf{0}*2^0) = (\mathbf{01000010})$$

El valor del código ASCII para un carácter espacio es el decimal 32. El valor binario es:

$$32 = (2^5) = (\mathbf{0}*2^7 + \mathbf{0}*2^6 + \mathbf{1}*2^5 + \mathbf{0}*2^4 + \mathbf{0}*2^3 + \mathbf{0}*2^2 + \mathbf{0}*2^1 + \mathbf{0}*2^0) = (\mathbf{00100000})$$

El valor del código ASCII para '2' es el decimal 50. El valor binario es:

$$50 = (32 + 16 + 2) = (2^5 + 2^4 + 2^1) =$$
$$(\mathbf{0}*2^7 + \mathbf{0}*2^6 + \mathbf{1}*2^5 + \mathbf{1}*2^4 + \mathbf{0}*2^3 + \mathbf{0}*2^2 + \mathbf{1}*2^1 + \mathbf{0}*2^0) = (\mathbf{00110010})$$

El valor del código ASCII para un carácter de línea nueva es el decimal 10. El valor binario es:

$$10 = (8 + 2) = (2^3 + 2^1) = (\mathbf{0}*2^7 + \mathbf{0}*2^6 + \mathbf{0}*2^5 + \mathbf{0}*2^4 + \mathbf{1}*2^3 + \mathbf{0}*2^2 + \mathbf{1}*2^1 + \mathbf{0}*2^0) = (\mathbf{00001010})$$

Observe que el carácter de línea nueva no coloca los datos en una "línea" por separado en un archivo. `Bob 2222` y `Paul 5555` se imprimen en líneas por separado, pero dentro de un archivo, los datos se almacenan secuencialmente: un byte después de otro.

Formato binario

Cuando valores primitivos se escriben hacia un archivo binario o un archivo objeto, Java usa el tipo nativo de cada formato de almacenamiento. Aquí no se está de acuerdo en llamarlo "archivo binario", porque esto implica que los archivos binarios usan números binarios y los archivos de texto en realidad no lo hacen. En realidad, toda la información de los archivos de la computadora es "binaria" en el sentido de que todo en una computadora está representado por unos y ceros. Aquí se prefiere denominar los "archivos binarios" como "archivos de almacenamiento nativos", pero ¡lástima!, "binario" es el término usado por todo mundo. Así, cuando se habla de un formato binario, se entiende que es el formato de almacenamiento nativo reconocido por el procesador. Por ejemplo, en un archivo binario, un `char` usa el *Unicode de 16 bits*, un `int`, *32 bits de complemento a 2*, un `double`, el *estándar IEEE de punto flotante de 64 bits*, y así sucesivamente. Consulte el capítulo 11 para un análisis de *Unicode*. Complemento a 2 significa lo siguiente:[6] si (valor binario >= 2^{bits-1}), entonces valor = valor binario − 2^{bits}. IEEE significa Instituto de Ingenieros Eléctricos y Electrónicos (*Institute of Electrical and Electronic Engineers*).

Los archivos binarios no están orientados a líneas. Los métodos de lectura de un archivo binario no reconocen los caracteres de final de línea como si tuviesen alguna función especial. Estos caracteres pueden estar presentes —justo como cualquier otro carácter— pero no afectan la medida de lo que se lee, y

[6] He aquí un ejemplo `int` (donde bits = 32): si valor binario = (10000000000000000000000000000000) = 2^{31} = 2147483648, entonces valor = 2147483648 − 4294967296 = −2147483648. Para una explicación más amplia del complemento a 2, consulte la página http://en.wikipedia.org/wiki/Two's_complement.

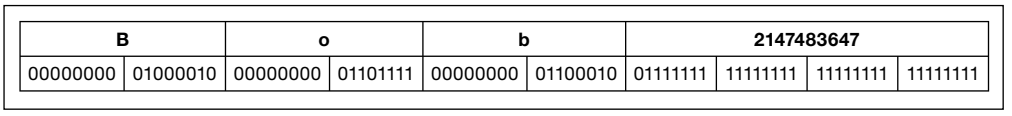


Figura 15.9 Forma cruda de un formato binario.
Los caracteres Unicode y el número `int` se muestran en letra negra (bold) arriba del carácter de 16 bits y la secuencias de números de 32 bits.

los métodos que escriben en archivos binarios jamás agregan en forma automática caracteres al final de la línea. En consecuencia, los programas que accesan datos primitivos en archivos binarios no leen o escriben todas las líneas. Es decir, no usan métodos `nextLine` y `println` para leer o imprimir una línea. En lugar de lo anterior, usan métodos como `readChar`, `writeChar`, `readInt`, `writeInt`, `readDouble`, `writeDouble`, etc., para leer y escribir valores de variables primitivas individuales.

Por ejemplo, suponga que se tienen estos datos:

```
Bob2147483647
```

En la figura 15.9 se muestra cómo se almacenan estos datos en un archivo binario o en un archivo objeto.

Los caracteres en un programa Java usan un esquema de almacenamiento Unicode de 16 bits. En consecuencia, una 'B' suele almacenarse en un archivo binario usando el esquema de almacenamiento Unicode de 16 bits. En este esquema, el primer byte tiene ocho ceros, y la secuencia del segundo byte coincide con el valor ASCII para 'B' que se muestra en la figura 15.8. Todos los ocho bits a la izquierda para 'B', 'o' y 'b', son ceros y no proporcionan ninguna información útil. Entonces, ¿qué hacen ahí esos ocho bits extra? Como se describió en el capítulo 11, están ahí para manejar caracteres Unicode que no están en el conjunto de caracteres ASCII. Esos otros caracteres requieren los ocho bits extra a la izquierda para mantener sus valores de código completos.

¿Cómo se almacena el 2147483647? Si fuese un archivo de texto, los dígitos se almacenarían como 10 caracteres ASCII por separado, lo que requiere 10 bytes. Pero con un archivo binario, 2147483647 se almacena como un simple número `int`. Puesto que un `int` ocupa 32 bits, los archivos binarios usan 32 bits para almacenar `int`, lo cual sólo requiere 4 bytes. El bit más significativo indica el signo del número. Un 0 en la posición más significativa indica que el número es positivo. Un 1 en la posición más significativa indica que el número es negativo. Ocurre que el número 2147483647 es el mayor número positivo que puede contener un `int`. Puesto que se trata de un número positivo, el bit más significativo debe ser 0. Puesto que es el mayor número positivo, todos los otros bits son 1. Con toda seguridad, si el lector usa su calculadora de mano, encontrará lo siguiente:

$$
\begin{aligned}
&(01111111\ 11111111\ 11111111\ 11111111) = \\
&(0*2^{31} + 1*2^{30} + 1*2^{29} + 1*2^{28} + 1*2^{27} + 1*2^{26} + 1*2^{25} + 1*2^{24} + \\
&\ 1*2^{23} + 1*2^{22} + 1*2^{21} + 1*2^{20} + 1*2^{19} + 1*2^{18} + 1*2^{17} + 1*2^{16} + \\
&\ 1*2^{15} + 1*2^{14} + 1*2^{13} + 1*2^{12} + 1*2^{11} + 1*2^{10} + 1*2^{9} + 1*2^{8} + \\
&\ 1*2^{7} + 1*2^{6} + 1*2^{5} + 1*2^{4} + 1*2^{3} + 1*2^{2} + 1*2^{1} + 1*2^{0}) = \\
&2147483647
\end{aligned}
$$

Soluciones intermedias

Beneficios de los archivos de texto:

- Creación independiente: uso de UNIX vi, Microsoft Notepad, Workpad, Word Text File, etcétera.
- Visualización independiente: uso de casi todos los procesadores de texto u otros lenguajes de computación.

Beneficios de los archivos binarios u archivos objeto:

- Pueden manejar todos los caracteres Unicode
- Almacenamiento de números más eficiente
- Pueden almacenar objetos complejos

15.7 Archivo binario

En Java, almacenar datos en archivos primitivos es directo. Desde una perspectiva de hardware, constituye la estrategia de almacenamiento más eficiente.

Salida

A fin de abrir un archivo binario para salida de datos primitivos, se instancia un objeto `FileOutputStream`. Esto manda una corriente de bytes al archivo. Para transformar tipos de datos primitivos en bytes, se instancia un objeto `DataOutputStream`. La clase `FileOutputStream` desciende de la clase `OutputStream`, y el único constructor `DataOutputStream` tiene un tipo de parámetro `OutputStream`, de modo que el nuevo objeto `FileOutputStream` puede pasarse directamente hacia el constructor `DataOutputStream` como un argumento como éste:[7]

```
DataOutputStream fileOut;
...
fileOut =
  new DataOutputStream(new FileOutputStream(stdIn.nextLine(), true));
...
```

Lo anterior se ve más bien como la declaración `fileOut =` en la figura 15.3. Aquí la instanciación `FileOutputStream` se parece a la instanciación `FileWriter` en la figura 15.3 en que abre el archivo e implementa un buffer. Así como en la figura 15.3, el segundo argumento indica si agregar o rescribir un archivo existente. Este objeto difiere del objeto `FileWriter` en el sentido de que no realiza ninguna transformación fundamental de tipo de datos. Simplemente recibe bites en bruto o un arreglo de bytes en bruto y los pasa sin modificar al archivo. La transformación de datos la realiza el objeto `DataOutputStream`, que usa uno de sus métodos `write` para convertir datos primitivos o una cadena en una secuencia idónea de `char`. Para escribir un `char`, `int`, `double` o `String` individual hacia un archivo, puede usarse uno de estos métodos `DataOutputStream`:

```
void writeChar(int ch)
void writeInt(int i)
void writeDouble(double x)
void writeChars(String s)
```

Por ejemplo, suponga que se tiene un arreglo de valores `double` denominado `doubleValues`. En un bloque `try` después de la declaración de apertura del archivo escrita arriba, estos valores pueden escribirse hacia el archivo binario usando un código como éste:

```
for (int i=0; i<doubleValues.length; i++)
{
  fileOut.writeDouble(doubleValues[i]);
}
```

El método `writeChars` escribe un cadena "como es", no agrega ningún carácter de terminación de línea. Sin embargo, el objeto `String` original puede incluir cualquier número de caracteres `'\n'` en cualquier sitio. Así, es posible referirse a un documento con multilíneas con una sola variable `String`, y todo ese documento puede escribirse hacia un archivo binario con un solo ciclo `for` que contiene justo una declaración `writeChars`.

Entrada

Para abrir un archivo binario para entrada de datos primitivos, deben instanciarse un `FileInputStream` y un `DataInputStream` como sigue:[8]

[7] Opcionalmente, la ejecución puede acelerarse al insertar un objeto `BufferedOutputStream` entre los objetos `DataOutputStream` y `FileOutputStream`.

[8] Opcionalmente, la ejecución puede acelerarse al insertar un objeto `BufferedInputStream` entre los objetos `DataInputStream` y `FileInputStream`.

```
DataInputStream fileIn;
...
fileIn =
  new DataInputStream(new FileInputStream(stdIn.nextLine()));
...
```

Lo anterior se ve más bien como la declaración `fileIn =` en la figura 15.5. Aquí la instanciación `FileInputStream` se parece a la instanciación `FileReader` en la figura 15.5 en que abre el archivo y proporciona un buffer. No obstante, `FileInputStream` difiere del objeto `FileReader` en que no realiza ninguna transformación fundamental de tipo de datos. Simplemente recibe bytes en bruto y los pasa sin modificar al archivo. Toda la transformación de datos la realiza el objeto `DataInputStream`, que usa uno de sus métodos `read` para convertir los bytes que recibe en datos primitivos. Para leer un `char`, `int` o `double` individual hacia un archivo, puede usarse uno de estos métodos `DataInputStream`:

```
char readChar()
int readInt()
double readDouble()
```

Por ejemplo, suponga que se ha declarado un arreglo de valores `double` denominado `doubleData`. En un bloque `try` después de la declaración de apertura del archivo escrita arriba, este arreglo puede llenarse con datos del archivo binario usando un código como éste:

```
for (int i=0; i<doubleData.length; i++)
{
  doubleData[i] = fileIn.readDouble();
}
```

Propiedades comunes

Los constructores `FileOuputStream` y `FileInputStream` lanzan excepciones `FileNotFoundExceptions`, y los métodos en las clases `DataOuputStream` y `DataInputStream` lanzan excepciones `IOExceptions`. Así, es posible usar un bloque `IOException catch` para atraparlas a todas. Por supuesto, una vez que se termina es necesario cerrar cada archivo, y para realizar esta operación pueden usarse métodos `close` heredados por las clases `DataOuputStream` y `DataInputStream`. Resulta más fácil incluir la declaración `close` en las declaraciones de apertura y cierre y transferir las declaraciones en el mismo bloque `try`.

La clase `DataInputStream` no incluye ningún método viable de entrada de líneas. Aunque un método así existe,[9] a menudo es mejor leer a través de caracteres `'\n'` y acumular texto de entrada en trozos más grandes que en líneas. Para hacer lo anterior, es necesario usar un carácter `null` de dos bytes cuyo valor de código sea cero a fin de terminar una operación de lectura de una cadena. Este carácter puede agregarse a la cadena cuando se escribe un archivo binario con esta declaración:

```
fileOut.writeChar(0);
```

Luego, en el programa de lectura de un archivo, se puede verificar cómo llega cada carácter para ver si su valor de código es igual a cero. En caso afirmativo, se está al final de la cadena.

Archivos binarios estructurados

El archivo binario de E/S es más fácil cuando el tipo de datos es el mismo. Con algo de trabajo, no obstante, se pueden mezclar tipos de datos y colocar la estructura en un archivo binario. Al hacer esto se dota de sentido a lo que el código integrado de Java hace en el siguiente objeto de E/S de la sección. Así, se considerará un ejemplo sencillo de un archivo binario estructurado. Suponga que se tiene un archivo binario que contiene algunas entradas de una tabla de una base de datos. Para leer esta información, es necesario saber cómo está estructurado ese archivo. Suponga que se sabe que el archivo posee esta organización relativamente simple:

[9] Si su archivo binario sólo contiene texto, en lugar de `DataInputStream` es mejor usar un `BufferedReader`, y la clase `BufferedReader` incluye un método `readLine`.

1. El primer bloque de datos es para un texto de título y/o descripción. Está compuesto por un número indefinido de `char` de 2 bytes. Este bloque de texto termina con el carácter `null`, cuyo valor de código es una secuencia de bytes con dos ceros.
2. El segundo bloque de datos es un `int` simple de cuatro bytes. Proporciona el número de sub-bloques que hay en el tercer bloque.
3. El tercer bloque contiene un número idéntico de sub-bloques. Cada uno de estos sub-bloques tiene dos campos:
   a) El primer campo es un `int` de cuatro bytes.
   b) El segundo campo es un `double` de ocho bytes.

En su programa de lectura de un archivo, suponga que desea colocar los dos valores de cada sub-bloque en las dos variables de instancia de un nuevo objeto `Record`. Y suponga que `Record` es un constructor de dos parámetros que inicia estas dos variables.

El siguiente fragmento de código muestra lo que podría verse en un programa que lee datos de un archivo binario que tiene la organización mencionada. Después de abrir el archivo para entrada binaria, el código almacena los caracteres del primer bloque en un `StringBuffer` denominado `table`. (La clase `StringBuffer` implementa un tipo flexible de `String`.) Observe cómo se ve la condición del ciclo `while` para un carácter `null` que señala el fin de la corriente de caracteres de entrada. Una vez que termina la operación de lectura de caracteres, el código lee el entero en el segundo bloque y lo almacena en un `int` denominado `numRecords`. Por último, el código instancia objetos `Record` a fin de mantener los pares de valores en los sub-bloques, y agrega estos objetos a una `ArrayList` denominada `table`.

```
Scanner stdIn = new Scanner(System.in);
DataInputStream fileIn;
char ch = 0;
int numRecords = 0;
ArrayList<Record> table = new ArrayList<Record>();
StringBuffer tableName = new StringBuffer();

System.out.print("Enter filename: ");
try
{
  fileIn = new DataInputStream(new FileInputStream(
    stdIn.nextLine()));
  while((ch = fileIn.readChar()) != 0)
  {
    tableName = tableName.append(ch);
  }
  numRecords = fileIn.readInt();
  for (int i=0; i<numRecords; i++)
  {
    table.add(new Record(
      fileIn.readInt(), fileIn.readDouble()));
  }
  fileIn.close();
} // end try
...
```

un “arreglo” flexible

una “cadena” flexible

Acumula caracteres en un `StringBuffer`.

Acumula objetos en una `ArrayList`.

Un `StringBuffer` es más flexible que un `String` porque tiene métodos `append` e `insert` que se agregan al final o se insertan en cualquier sitio en la parte de en medio. El código anterior usa el método `append` para acumular caracteres a medida que fluyen desde el archivo. Por supuesto, `StringBuffer` también tiene un método `toString` que permite la conversión de un `StringBuffer` en un `String` en cualquier tiempo posterior.

El argumento del constructor `Record` en la llamada al método `table.add` cerca del final de este ejemplo indica cómo los valores de cada variable de instancia del objeto están dispuestos en el archivo. Están alineados en una secuencia, uno después de otro. Observe que el código de lectura en un archivo

debe conocer de antemano el esquema básico del formato del archivo, aunque algunos detalles del formato del archivo —el número de registros que contiene el archivo— están insertados en el archivo en sí.

El código de lectura en un archivo determina este detalle de formato en el último momento, a medida que los datos fluyen. Esta combinación de un *protocolo* estándar (un acuerdo formal sobre la forma en que debe hacerse algo) más una variación insertada es típica del formateo en archivos binarios del mundo real. La diferencia es que los protocolos del mundo real y sus variaciones insertadas son cientos de veces más complicados que este ejemplo sencillo. A menudo, los programadores emplean algún tipo de software escrito previamente para realizar las transformaciones de datos necesarias para un archivo binario de E/S.

15.8 Archivo objeto

Cuando es posible permanecer en un entorno de programación de Java, es decir, usar programas Java para toda la escritura hacia archivos y toda la lectura desde archivos, se tiene una ventaja. Es posible usar software construido en el lenguaje Java para realizar la conversión estructural entre objetos del programa y corrientes de datos primitivos. En esta sección se explica cómo usar ese software preconstruido.

Almacenamiento de objetos de Java en un archivo

El lenguaje Java cuenta con un software preconstruido que *serializa* los datos de cada objeto a medida que los objetos llegan a un archivo y *deserializa* esos datos a medida que salen de un archivo y regresan a forma de objeto. Siempre que un programa escribe datos serializados en un archivo, también escribe la receta que utiliza para serializar esos datos. Esta receta incluye el tipo del objeto, el tipo de cada dato y la secuencia en la que se almacenan los datos. Cuando otro programa lee datos serializados desde un archivo, también lee la receta para aprender a reconstruir el objeto a partir de los datos serializados. Para permitir que una clase use software de serialización preconstruido, se debe agregar la siguiente cláusula al encabezado de esa clase:

```
implements Serializable
```

Esto hace que se vea como si esa clase estuviese implementando una interfaz. Pero esta interfaz no define ninguna constante nombrada, y no requiere que la clase implemente ningún método particular. Simplemente *etiqueta* (identifica) los objetos de la clase como objetos que requieren servicios de serialización. Por ejemplo, vea la clase `TestObject` en la figura 15.10. Observe que esta clase `implementa` la interfaz `Serializable`.

Escritura de un objeto `serializable` en un archivo

La figura 15.11 contiene un programa sencillo que escribe un objeto `TestObject` en un archivo especificado por el usuario. La primer declaración en el bloque `try` usa instanciaciones `ObjectOutputStream` y `FileOutputStream` para abrir el archivo. Observe que el constructor `FileOutputStream` sólo tiene un parámetro. El constructor crea un nuevo archivo o agrega objetos a datos que ya están en un archivo existente. No hay opción de agregar nada. (Uno de los proyectos de este capítulo muestra cómo agregar objetos a datos que ya están en un archivo existente.) La segunda declaración en el bloque `try` escribe un objeto en el archivo abierto. El objeto escrito es una instancia de la clase en la figura 15.10. La tercera declaración cierra el archivo.

El parámetro `catch` es una `IOException` porque todos los métodos `writeObject` y `close` del constructor `ObjectOutputStream` y de `ObjetOutputStream` lanzan una `IOException`. Como ya se indicó, el constructor `FileOutputStream` lanza una `FileNotFoundException`, pero esta excepción se deriva de la `IOException`. En consecuencia, el uso de `IOException` como parámetro `catch` permite que el bloque `catch` atrape todas las excepciones que pudieran lanzarse desde el bloque `try`.

Lectura de un objeto `Serializable` desde un archivo

El programa `ReadObject` en la figura 15.12 intenta leer datos para dos objetos de la clase `TestObject` desde un archivo especificado por el usuario. El código del programa debe ser conocido puesto que

```
/*****************************************************
* TestObject.java
* Dean & Dean
*
* Éste es un objeto heterogéneo típico.
*****************************************************/

import java.io.*;

public class TestObject implements Serializable
{
  private int id;
  private String text;
  public double number;

  //*************************************************

  public TestObject(int id, String text, double number)
  {
    this.id = id;
    this.text = text;
    this.number = number;
  } // end constructor

  //*************************************************

  public void display()
  {
    System.out.print(this.id + "\t");
    System.out.print(this.text + "\t");
    System.out.println(this.number);
  } // end display
} // end TestObject class
```

Para que un objeto pueda escribirse hacia un archivo y sea legible desde un archivo, debe ser una instancia de una clase que implemente esta interfaz.

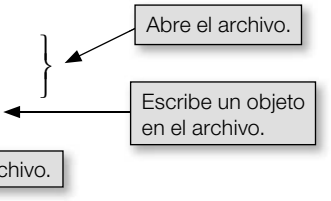
Figura 15.10 Definición típica de un objeto `Serializable`.

imita el programa WriteObject en la figura 15.11. Sin embargo, en lugar de usar los constructores `FileOutputStream` y de `ObjectOutputStream` y el método `writeObject`, usa los constructores `FileIntputStream` y de `ObjectInputStream` y el método `readObject`. La única cuestión engañosa es que debe incluirse un tipo cast (`TestObject`) para convertir la referencia devuelta por el método `readObject` en el tipo específico que define los métodos que se quieren usar. Esa conversión puede lanzar una `ClassNotFoundException`, de modo que resulta conveniente incluir un bloque `catch` adicional para esa excepción.

Observe la segunda entrada en la sesión de muestra en la figura 15.12. ¿Puede imaginar qué ocurrió? El programa WriteObject envía sólo un objeto al archivo `objectFile.data`, pero el programa ReadObject intenta leer dos objetos desde ese archivo. Puesto que la JVM no puede encontrar un segundo objeto, lanza una `IOException` que genera la declaración de impresión en la última línea de salida.

Si una clase es `Serializable`, todas las clases derivadas a partir de ésta también son `Serializable`. Suponga que su clase `Serializable` posee variables de instancia que refieren a otros objetos. Estas clases de objetos también deben ser `Serializable`. Esto debe ser cierto a través de todos los niveles en una jerarquía de composición. Lo anterior, ¿suena problemático? No, realmente. Simplemente debe tenerse la seguridad de incluir `implements Serializable` en la definición de todas las clases que definen objetos que le gustaría almacenar como objetos. No obstante, la alternativa sería un problema. En caso de no ser posible almacenar un objeto completo, es necesario contar con un código explícito para escribir y leer cada dato primitivo en el objeto contenedor, y cada dato primitivo en todos los

```
/*************************************************************
* WriteObject.java
* Dean & Dean
*
* Esto escribe un objeto en un archivo binario.
*************************************************************/

import java.io.*;
import java.util.Scanner;

public class WriteObject
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    ObjectOutputStream fileOut;
    TestObject testObject = new TestObject(1, "test", 2.0);
    String filename;

    System.out.print("Introduzca el nombre del archivo: ");
    filename = stdIn.nextLine();
    try
    {
      fileOut = new ObjectOutputStream(
        new FileOutputStream(filename));
      fileOut.writeObject(testObject);
      fileOut.close();
    } // end try
    catch (IOException e)
    {
      System.out.println("Error: " + e.getMessage());
    }
  } // end main
} // end WriteObject class
```



Sesión muestra:

```
Enter filename: objectFile.data
```

Figura 15.11 Programa WriteObject que escribe un objeto `Serializable` en un archivo.

objetos componente en ese objeto contenedor, y así sucesivamente, recorriendo el árbol de composición hasta llegar a todas las hojas primitivas.

Salida de una versión actualizada de un objeto de salida previo

Si se solicita que un método `writeObject` de `ObjetOutputStream` produzca de nuevo exactamente el mismo objeto cuando el archivo aún está abierto, el software de serialización reconoce la repetición y produce justo una referencia al objeto de salida previo. Esto es como lo que ocurre cuando se instancia una nueva `String` que es exactamente la misma que la cadena `String` previamente instanciada.

Ésta es una buena característica para ahorrar espacio, aunque puede resultar un problema si se está simulando el comportamiento de un objeto particular y se desea que un archivo acumule un registro de cómo cambia el estado de ese objeto a medida que avanza la simulación. Para ver este problema, se sustituye la declaración `writeObject` en el programa WriteObject en la figura 15.11 por estas tres declaraciones:

```
fileOut.writeObject(testObject);
testObject.number *= 1.1;
fileOut.writeObject(testObject);
```

```
/*************************************************************
* ReadObject.java
* Dean & Dean
*
* Esto lee dos objetos desde un archivo binario.
*************************************************************/

import java.io.*;
import java.util.Scanner;

public class ReadObject
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    ObjectInputStream fileIn = null;
    TestObject testObject;

    System.out.print("Introduzca el nombre del archivo: ");
    try
    {
      fileIn = new ObjectInputStream(
        new FileInputStream(stdIn.nextLine()));
      testObject = (TestObject) fileIn.readObject();
      testObject.display();
      testObject = (TestObject) fileIn.readObject();
      testObject.display();
      fileIn.close();
    }
    catch (IOException e)
    {
      System.out.println("IO Error: " + e.getMessage());
    }
    catch (ClassNotFoundException e)
    {
      System.out.println("ClassNotFound " + e.getMessage());
    }
  } // end main
} // end ReadObject class
```

Abre el archivo.

Lee un objeto desde el archivo.

Cierra el archivo.

Sesión muestra:

```
Enter filename: objectFile.data
1       test    2.0
IO Error: null
```

Figura 15.12 Programa ReadObject que intenta leer dos objetos desde un archivo.

Entonces, al ejecutar el programa revisado WriteObject y el programa ReadObject, se obtiene esto:

```
Enter filename: objectFile.data
1       test    2.0
1       test    2.0
```

El segundo registro del estado del objeto es justo una copia del primero. No refleja el cambio en el valor de la variable `number`. Para hacer que Java almacene el último estado de un objeto en lugar de una simple referencia al estado original, es necesario invocar al método `reset` de `ObjectOutputStream`

antes de sacar la versión actualizada del objeto. Para ver cómo funciona esto, se sustituyen las tres declaraciones anteriores por estas cuatro declaraciones:

```
fileOut.writeObject(testObject);
fileOut.reset();
testObject.number *= 1.1;
fileOut.writeObject(testObject);
```

Esto permite una versión actualizada de salida del mismo objeto.

Entonces, al ejecutar el programa revisado WriteObject y el programa ReadObject se obtiene el resultado que se busca:

```
Enter filename: objectFile.data
1       test    2.0
1       test    2.2
```

15.9 La clase `File`

En esta sección se describe la clase `File`. Difiere de las demás clases en este capítulo en que no tiene que ver con el contenido de un archivo. Trata con el archivo en sí, y describe el entorno del archivo: el sitio de la computadora en que está el archivo.

Instanciación de un objeto `File`

Para usar la clase `File`, primero es necesario instanciar un objeto que representa un archivo. Éste es el encabezado API para el constructor `File`:

```
public File(String filename)
```

Por ejemplo, para instanciar un objeto `File` para un archivo denominado `dalaiLamaEssay.doc`, debe hacerse lo siguiente:

```
File paper = new File("daliLamaEssay.doc");
```

Un objeto `File` no es un archivo en sí. Es simplemente un contenedor de información concerniente al archivo. Si el lector desea saber si un archivo con un nombre particular existe realmente, debe instanciar un objeto `File` con el nombre del archivo de interés y luego usar ese objeto para llamar al método `exist` de `File` como sigue:

```
File paper = new File("daliLamaEssay.doc");
if (paper.exists())
{
  System.out.println("Sí existe.");
}
```

El objeto `paper` de arriba representa un archivo en el *directorio actual*. El directorio actual es el directorio en que reside el programa que se está ejecutando en ese momento. Para especificar un archivo en un directorio distinto al directorio actual, es necesario incluir la *ruta* del archivo como parte del argumento del nombre del archivo. Una ruta es la ubicación de un archivo dentro de la estructura de directorios de la computadora. Más específicamente, una ruta es una serie de uno o más nombres de directorio separados por slashes hacia delante (/) que conduce a un directorio particular. Opcionalmente, en máquinas Windows es posible usar backslashes (\), aunque éstos originan más confusión porque en Java un backslash es el carácter de escape y se necesitan dos backlashes siempre que se quiere indicar uno solo. Así, para los nombres de las rutas se recomienda usar slashes hacia delante. Hay dos tipos de rutas: *ruta relativa* y *ruta absoluta*. Una ruta relativa va desde el directorio actual hasta el archivo especificado. Una ruta absoluta va desde el directorio raíz hasta el archivo especificado. El *directorio raíz* es el directorio que está en la parte superior de la estructura de directorios de la computadora. Un slash inicial representa el directorio raíz.

Suponga que un archivo dalaiLamaEssay.doc está en un directorio re101 y que el directorio re101 está en el directorio raíz. A fin de instanciar un objeto File para el archivo dalaiLamaEssay.doc, debe usarse una ruta absoluta como ésta:

```
File paper = new File("/re101/daliLamaEssay.doc");
```

Suponga que un archivo checkers.class está en un subdirectorio del directorio. A fin de instanciar un objeto File para el archivo checkers.class, debe usarse una ruta relativa como ésta:

```
File paper = new File("checkers/checkers.class");
```

Métodos File

Una vez que se tiene una referencia a un objeto File, puede usarse para llamar a cualquiera de los métodos útiles de la clase File. Como ya se indicó, el método boolean exists devuelve true si el archivo del objeto que llama está presente. El método boolean isfile devuelve true si el archivo del objeto que llama es un archivo normal. El método boolean isDirectory devuelve true si el archivo del objeto que llama es un directorio. (Un directorio se considera un archivo, aunque en realidad es un tipo especial de archivo.) El método boolean delete borra el archivo del objeto que llama. El método boolean mkdir crea un nuevo directorio y le asigna el nombre especificado por el argumento que le fue dado. El método boolean renameTo cambia el nombre de la ruta especificada por el argumento que le fue dado. Los métodos delete, mkdir y renameTo devuelven true si tienen éxito.

Cuando se desea transferir datos hacia o desde un archivo, a menudo es de utilidad ver cuáles archivos ya existen, y algunas veces es útil ver de qué tamaño son. El programa en la figura 15.13 muestra este tipo de información.

En el programa FileSizes, observe el argumento "." en la llamada al constructor File. El punto es un símbolo especial que representa el directorio actual. Así, new File (".") instancia un objeto File para el directorio actual. En forma semejante, new File ("..") instancia un objeto File para el directorio padre.

Observe esta declaración en el programa FileSizes:

```
File[] files = currentDirectory.listFiles();
```

La llamada currentDirectory.listFiles devuelve un arreglo de objetos File, con un objeto File por cada archivo en el directorio actual. También observe esta declaración:

```
System.out.printf("%-25s%6d bytes\n",
  files[i].getName(), files[i].length());
```

La cadena de formato indica la impresión de una cadena alineada a la izquierda en 25 espacios y luego la impresión de un número de punto flotante (alineado a la derecha) en seis espacios. La variable files[i] identifica el archivo i en el directorio actual. La llamada getName devuelve el nombre del archivo i. La llamada lengt devuelve el tamaño del archivo i en bytes.

15.10 Apartado GUI: la clase JFileChooser (opcional)

En el programa FileSizes en la sección previa, los nombres y los tamaños de los archivos se mostraron para todos los archivos en el directorio actual. Pero suponga que lo que se está buscando es otro directorio. ¿No sería mejor mostrar los nombres y los tamaños de los archivos de cualquier directorio, y no sólo los que están en el directorio actual? En esta sección se presenta un programa que hace precisamente eso. Usa un formato GUI para mostrar los nombres y los tamaños de los archivos de un directorio especificado por el usuario. El programa obtiene la selección del directorio del usuario con ayuda de la ventana de diálogo JFileChooser. Antes de abordar el programa, se considerará la ventana de diálogo JFileChooser definida en el API de Java.

Interfaz de usuario

Una ventana de diálogo de un selector de archivo permite que el usuario seleccione un archivo o un directorio desde una estructura interactiva gráfica. Los selectores de archivos son ubicuos en el software

```
/*********************************************
* FileSizes.java
* Dean & Dean
*
* Este programa muestra los nombres y tamaños de
* los archivos en el directorio actual.
*********************************************/

import java.io.*;

public class FileSizes
{
  public static void main(String[] args)
  {
    File currentDirectory = new File(".");
    File[] files = currentDirectory.listFiles();

    for (int i=0; i<files.length; i++)
    {
      System.out.printf("%-25s%6d bytes\n",
        files[i].getName(), files[i].length());
    }
  } // end main
} // end FileSizes class
```

Sesión muestra:

```
.classpath                  226 bytes
.project                    384 bytes
FileSizes.class            1135 bytes
FileSizes.java              645 bytes
FileSizesGUI.class         2058 bytes
FileSizesGUI.java          2185 bytes
HTMLGenerator.class        2182 bytes
HTMLGenerator.java         2659 bytes
```

Figura 15.13 Programa `FileSizes` con salida de muestra.

moderno. Por ejemplo, un procesador de palabras emplea un selector de archivos siempre que el usuario selecciona Open desde el menú File. En la figura 15.14 se muestra cómo un usuario selecciona un archivo con ayuda de la ventana de diálogo `JFileChooser`.

Uso de `JFileChooser`

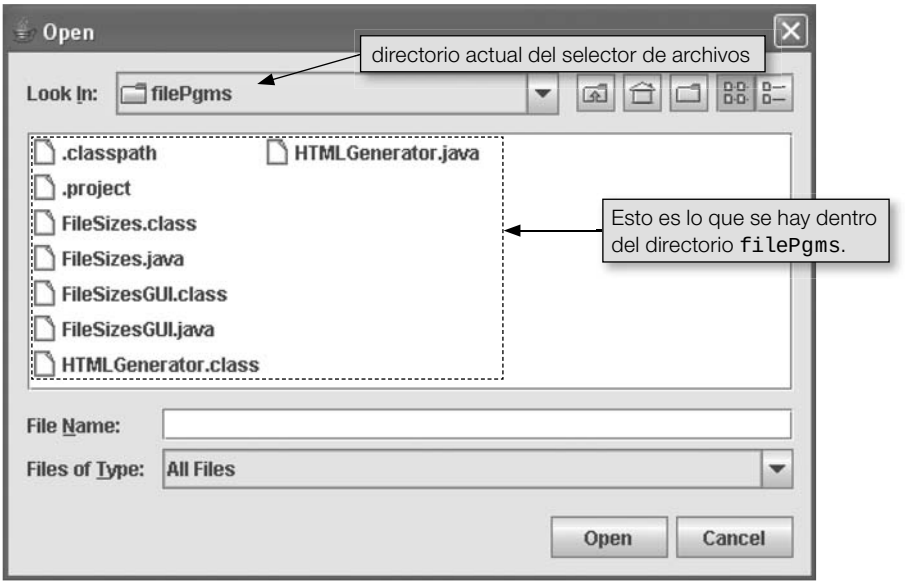
La clase `JFileChooser` está definida en el paquete `javax.swing`, de modo que es necesario `importar` ese paquete para acceder a esta clase. Para crear una ventana de diálogo `JFileChooser`, es necesario llamar al constructor `JFileChooser` como sigue:

```
JFileChooser chooser = new JFileChooser(<current-directory>);
```

El argumento *current-directory* especifica el nombre del directorio que aparece inicialmente en la parte superior de la ventana de diálogo del selector de archivos. Ése es el directorio actual del selector de archivos. Esta declaración muestra cómo se creó el selector de archivos en la figura 15.14:

```
JFileChooser chooser = new JFileChooser(".");
```

Desplegado inicial cuando el directorio actual del selector de archivos es `filePgms`:



Desplegado que aparece después que el usuario selecciona el archivo `FileSizes.java`:

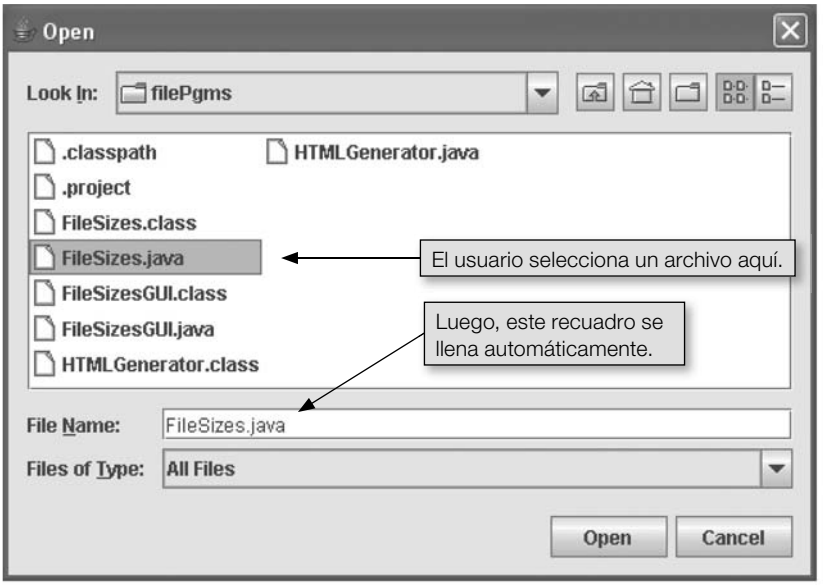


Figura 15.14 Selección de un archivo con el componente `JFileChooser`.

Como ya se mencionó, el argumento "." representa el directorio actual de la computadora. Tenga en mente que el directorio actual del selector de archivos y el directorio actual de la computadora no siempre son los mismos. Si se ha llamado al constructor `JFileChooser` con un argumento `"C:/spreadsheets"`, el directorio actual del selector de archivos sería `C:/spreadsheets`, pero el directorio actual de la computadora permanecería sin cambios.

La clase `JFileChooser` posee muchos métodos. Aquí sólo se considerarán tres de éstos: los métodos `setFileSelectionMode`, `showOpenDialog` y `getSelectedFile`. A continuación se proporcionan sus encabezados y descripciones API:

`public void setFileSelectionMode(int mode)`

Especifica el tipo de archivo que puede escoger el usuario: un archivo, un directorio o cualquiera de los dos.

`public int showOpenDialog(null)`

Muestra la ventana de diálogo del selector de archivos. Devuelve una constante con nombre, que indica si el usuario ha seleccionado Open o Cancel.

`public File getSelectedFile()`

Devuelve el archivo o directorio seleccionado.

Cuando se llama `setFileSelectionMode`, se pasa un argumento `mode` para especificar el tipo de archivo que puede escoger el usuario. Si el modo es `JFileCHooser.FILES_ONLY` sólo se permite al usuario escoger un archivo. Si el modo es `JFileChooser.DIRECTORIES_ONLY`, al usuario se permite escoger sólo un directorio. Si el modo es `JFileChooser.FILES_AND_DIRECTORIES`, se permite al usuario escoger un archivo o un directorio.

Se llama a `showOpenDialog` para mostrar una ventana de diálogo del selector de archivos. Luego, la ventana de diálogo muestra, si el usuario hace clic en el botón Open del selector de archivos, la llamada al método `showOpenDialog` devuelve la constante nombrada `JFileChooser.APPROVE_OPTION`. Si el usuario hace clic en el botón Cancel, el método `showOpenDialog` devuelve la constante nombrada `JFileChooser.CANCEL_OPTION`.

Después que el usuario selecciona un archivo con la ventana de diálogo `JFileChooser`, el programa llama a `getSelectedFile` para recuperar el archivo o directorio seleccionado. En ese momento, probablemente el programa quiera hacer algo con el archivo o directorio. Pero antes de hacerlo, debe usar el método `exists` de `File` para determinar si la entrada del usuario es válida, y debe usar el método `isFile` o `isDirectory` de `File` para determinar el tipo de archivo seleccionado.

Uso de `JOptionPane`

Quizá también sea necesario usar algunos de los métodos de la clase `JOptionPane`. Esta clase también está en el paquete `javax.swing`, de modo que el lector ha importado este paquete para la clase `JFileChooser`, también tiene acceso a la clase `JOptionPane`. La clase `JOptionPane` proporciona muchos métodos de clase útiles a los que es posible acceder directamente con el nombre de la clase. Aquí sólo se abordarán dos de ellos: el método `showConfirmDialog` y el método `showMessageDialog`. Éstos son los encabezados y las descripciones:

```
public static int showConfirmDialog(Component parentComponent,
  Object message, String title, int optionType)
```

Esto trae una ventana de diálogo donde el número de opciones está determinado por `optionType`.

```
public static void showMessageDialog(Component parentComponent,
  Object message, String title, int messageType)
```

Esto trae una ventana de diálogo que muestra un mensaje.

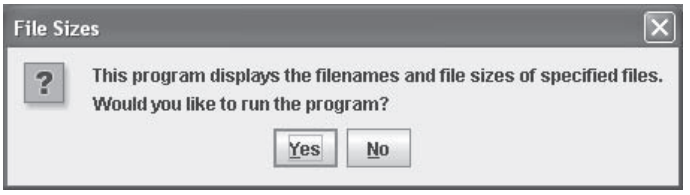
Cuando se llama a `showConfirmDialog`, es posible usar una referencia a otro marco dentro del que se desea mostrar el recuadro, o puede usarse justo un `null` para posicionarlo con respecto a toda la pantalla. Además, quizá sea conveniente proporcionar un mensaje de texto, por lo que es necesario proporcionar un título del texto. Para el parámetro `optionType`, es necesario introducir `JOptionPane.YES_NO_OPTION` o `JOptionPane.YES_NO_CANCEL_OPTION`. El valor devuelto es la opción escogida por el usuario, como `JOptionPane.YES_OPTION` o `JOptionPane.NO_OPTION`.

Cuando se llama a `showMessageDialog`, puede usarse una referencia a otro marco dentro del que se desea mostrar el recuadro, o puede usarse justo un `null` para posicionarlo con respecto a toda la pantalla. Además, quizá sea conveniente proporcionar un mensaje de texto, por lo que es necesario proporcionar un título del texto. Para el parámetro `messageType`, es necesario introducir `JOptionPane.ERROR_MESSAGE`, `JOptionPane.INFORMATION_MESSAGE`, `JOptionPane.WARNING_MESSAGE`, `JOptionPane.QUESTION_MESSAGE` o `JOptionPane.PLAIN_MESSAGE`.

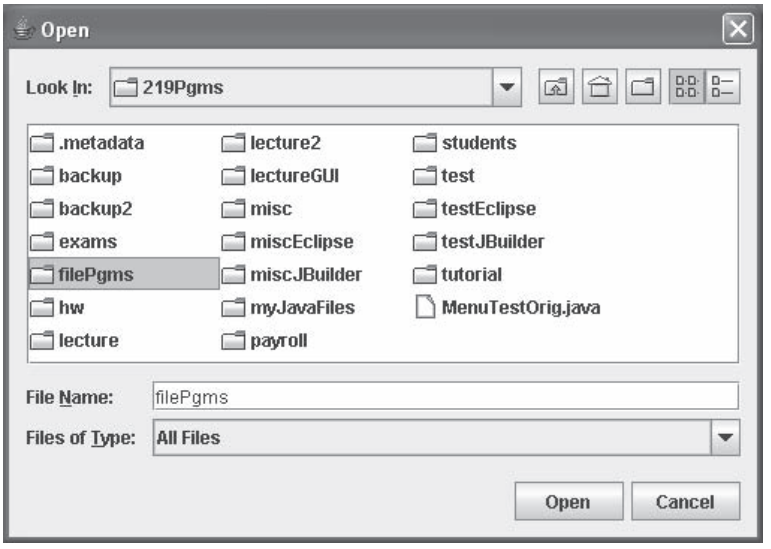
Programa FileSizesGUI

Ahora ya es posible incorporar estas ideas en el programa mejorado FileSizes mencionado antes. El programa FileSizesGUI usa una ventana de diálogo `JFileChooser` para recuperar un archivo o un directorio especificado por el usuario. Si el usuario selecciona un archivo, el programa muestra el nombre y el tamaño del archivo. Si el usuario selecciona un directorio, el programa muestra los nombres y los tamaños de todos los archivos en el directorio. Para tener una mejor idea de cómo opera el programa, vea la sesión muestra en la figura 15.15. El programa en sí se muestra en la figura 15.16.

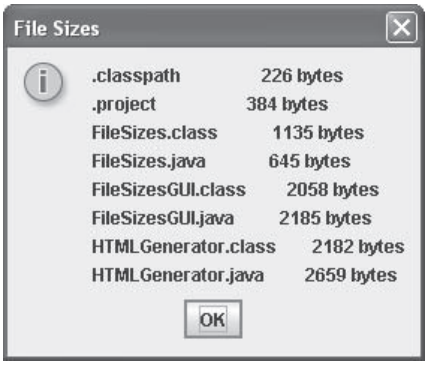
Cuando se ejecuta el programa, una llamada al método `JOptionPane.showConfirmDialog` muestra esto:



Al hacer clic en Yes se produce una llamada al método `showOpenDialog` de `JFileChooser` para mostrar otra ventana, y después de escoger filePgms se ve esto:



Al hacer clic en Open se origina una llamada al método `showMessageDialog` de `JOptionPane` para mostrar esto:



Y al hacer clic en OK se termina el programa.

Figura 15.15 Sesión muestra para el programa FileSizesGUI.

```
import java.io.File;
import javax.swing.*;   // para JFileChooser y JOptionPane;

public class FileSizesGUI
{
  public static void main(String[] args)
  {
    File fileDir;       // archivo o directorio especificado por el usuario
    int response;       // respuesta del usuario a los desplegados GUI
    File[] files;       // arreglo de archivos en el directorio especificado
    String output = ""; // lista de nombres de archivos y directorios
    JFileChooser chooser = new JFileChooser(".");

    response = JOptionPane.showConfirmDialog(null,
      "Este programa muestra los nombres y tamaños de archivos de" +
      " archivos especificados. \n ¿Le gustaría ejecutar el programa?",
      "File Sizes", JOptionPane.YES_NO_OPTION);
    if (response == JOptionPane.YES_OPTION)
    {
      chooser.setFileSelectionMode(
        JFileChooser.FILES_AND_DIRECTORIES);
      response = chooser.showOpenDialog(null);
      if (response == JFileChooser.APPROVE_OPTION)
      {
        fileDir = chooser.getSelectedFile();
        if (fileDir.isFile())
        {
          output += String.format("%-25s%12s%n",
            fileDir.getName(), fileDir.length() + " bytes");
        }
        else if (fileDir.isDirectory())
        {
          files = fileDir.listFiles();
          for (int i=0; i<files.length; i++)
          {
            output += String.format("%-25s%12s%n",
              files[i].getName(), files[i].length() + " bytes");
          } // end for
        } // end else
        else
        {
          output = "Entrada inválida. No es un archivo o un directorio.";
        }
        JOptionPane.showMessageDialog(null, output,
          "File Sizes", JOptionPane.INFORMATION_MESSAGE);
      } // end if
    } // end if
  } // end main
} // end FileSizesGUI class
```

Figura 15.16 Programa FileSizesGUI.

En la figura 15.16 observe las siguientes operaciones:

- Una llamada al constructor `JFileChooser`.
- Una llamada al método `showConfirmDialog` de `JOptionPane`, y el uso del valor devuelto.

- Una llamada al método `setFileSelectionMode` de `JFileChooser`, y el uso del valor devuelto.
- Una llamada al método `showOpenDialog` de `JFileChooser`, y el uso del valor devuelto.
- Una llamada al método `getSelectedFile` de `JFileChooser`, y el uso del valor devuelto.
- Una llamada al método `showMessageDialog` de `JFileChooser`.

Después de llamar a `getSelectedFile`, el programa necesita determinar el tipo de la selección del usuario: archivo o directorio. Las llamadas a `isFile` y `isDirectory` se encargan de esto. Dentro del código de procesamiento del directorio, observe cómo `fileDir`, un objeto `File`, llama a `listFiles`. El método `listFiles` devuelve los archivos y los directorios que están en el directorio `fileDir`. Los archivos y directorios devueltos se almacenan como objetos `File` en un arreglo denominado `files`. Después de llenar el arreglo `files`, el programa ejecuta un ciclo a través de cada uno de sus elementos `File`. Para cada elemento, imprime el nombre y el tamaño del archivo al llamar a `getName` y `length`, respectivamente.

En el programa FileSizesGUI, observe las llamadas al método `String.format`. El método `String.format` funciona igual que el método `printf`, excepto que en lugar de imprimir un valor formateado, devuelve un valor formateado. Las llamadas al método `String.format` se realizan en un intento por mostrar valores con anchos uniformes. Específicamente, se quiere mostrar los nombres y los tamaños de los archivos con anchos uniformes, de modo que los valores del tamaño del archivo aparezcan en forma alineada. Pero la ventana de diálogo inferior en la figura 15.15 muestra que los valores del tamaño del archivo no están alineados. El problema es que con salida GUI, caracteres distintos imprimen anchos diferentes. Por ejemplo, puede verse que el "HTML" en `HTMLGenerator.class` es más ancho que el "File" en `FileSizesGUI.java`. Por tanto, la línea `HTMLGenerator.class` es más larga. El hecho de contar con caracteres diferentes que imprimen con anchos distintos es elegante la mayor parte del tiempo, pero en el programa FileSizesGUI es molesto. Para arreglar este problema, puede insertarse un componente `JTextArea` en la ventana de diálogo `JOptionPane` y establecer la fuente del componente `JTextArea` como una fuente con un espacio (con esta fuente, todos los caracteres imprimen con el mismo ancho). En el capítulo 17 se aprenderá sobre el componente `JTextArea`.

Use fuente con espacio para alinear texto.

Resumen

- La mayor parte de las clases de transferencia de archivos pueden encontrarse en el paquete `java.io`.
- Para enviar texto hacia un nuevo archivo, el archivo debe abrirse al instanciar un objeto `PrintWriter` con un nombre de archivo `String` como el argumento del constructor. Hacia el archivo se escribe llamando al método `println`, `print` o `printf` de `PrintWriter`, y el archivo se cierra llamando al método `close` de `PrintWriter`.
- Para agregar texto a un archivo existente, el archivo debe abrirse al instanciar un objeto `PrintWriter` con un objeto anónimo `FileOutputStream` como el argumento del constructor. Para los argumentos del constructor `FileOutputStream` se usa el nombre del archivo y `true`.
- Para introducir texto desde un archivo, el archivo debe abrirse al instanciar un objeto `Scanner` con un objeto anónimo `FileReader` como argumento del constructor. Para el argumento del constructor `FileReader`, use el nombre del archivo. Desde el archivo se lee llamando a uno de los métodos `Scanner`, y el archivo se cierra usando el método `close` de `Scanner`.
- Es posible usar un archivo de texto de E/S para traducir información en texto llano a formato HTML para una página Web.
- Los datos en un archivo de texto aparecen como una secuencia de bytes, donde cada byte corresponde a un carácter, y las líneas están delimitadas por símbolos `\r\n` o `\n`. Puede usarse un editor de texto o un procesador de palabras con almacenamiento de texto llano para crear un archivo de texto que pueda leer un programa de Java. Puede usarse un procesador de palabras para leer cualquier archivo de texto escrito por cualquier programa de Java.
- Los datos en un archivo binario aparecen como una secuencia de datos, cada uno codificado en el formato nativo para ese tipo de datos. No es posible crear o leer archivos binarios con editores de

texto o procesadores de palabras. No obstante, los archivos binarios pueden almacenar una variedad mucho mayor de caracteres, y pueden almacenar números de alta precisión de manera más eficiente.

- Para extraer (o introducir) datos desde (o hacia) un archivo binario, el archivo debe abrirse al instanciar un objeto `DataOutputStream` (o `DataInputStream`) con un objeto anónimo `FileOutputStream` (o `FileInputStream`) como argumento del constructor. Para el argumento de `FileOutputStream` (o `FileInputStream`), use el nombre del archivo. Luego, use los métodos `writeInt` o `readInt` para leer o escribir datos primitivos.
- En un archivo es posible almacenar objetos completos en formato binario, en el supuesto de que estos objetos y todos sus objetos componentes `implementen` la interfaz `Serializable`.
- Para extraer (o introducir) datos desde (o hacia) un archivo, el archivo debe abrirse al instanciar un `ObjectOutputStream` (u `ObjectInputStream`) con un objeto anónimo `FileOutputStream` (o `FileInputStream`) como argumento del constructor. Como argumento del constructor de `FileOutputStream` (o `FileInputStream`), use el nombre del archivo. Luego, use los métodos `writeObject` y `readObject` para transferir objetos completos hacia y desde el archivo.
- La clase `File` manipula archivos completos y describe sus entornos.
- Opcionalmente, con la clase `JFileChooser` de Java es posible permitir al usuario de uno de sus programas que encuentre cualquier archivo en su computadora al interactuar con una interfaz gráfica conocida por el usuario.

Preguntas de revisión

§15.2 Clases API de Java que se precisa importar

1. Es posible crear o ver el contenido de archivos binarios o archivos objeto con muchos editores de texto. (F/C)
2. Escriba una declaración `import` que proporciona acceso a cualquiera de las clases en el paquete `java.io`.

§15.3 Salida a archivos de texto

3. ¿Cuáles son los tres pasos básicos para un archivo de E/S?
4. Use un método `PrintWriter` para escribir una declaración de Java que saque una `String` denominada `name` seguida por un espacio y un `int` denominado `id`, de modo que una operación `nextLine` ulterior en un archivo de texto reconozca la combinación como una cadena distinta.
5. Escriba una declaración sencilla que abra un archivo de texto existente denominado `mydata.txt` para salida, tal que los nuevos datos de salida se agreguen a los datos que ya están en el archivo.

§15.4 Lectura de archivos de texto

6. En el supuesto de que `fileName` es una `String` que identifica correctamente un archivo de texto en el directorio actual, ¿cuál es el error en la siguiente declaración de apertura de archivo?

```
Scanner fileReader = new Scanner(fileName);
```

7. Suponga que se tiene un archivo de texto con las dos siguientes líneas de datos:

```
55.6 hi
there
```

Suponga que este archivo de texto se ha abierto exitosamente para entrada y que a la conexión se ha dado el nombre `fileIn`. Suponga que las siguientes líneas del código son:

```
double num = fileIn.nextDouble();
String name = fileIn.nextLine();
```

¿Cuál es la longitud de la cadena en `name`? Explique su respuesta.

8. Dado este arreglo:

```
int[] number = new int[] {2, 3, 4};
```

Suponga que se usa el código siguiente para escribir los tres elementos del arreglo `number` en un archivo de texto:

```
for (i=0; i<3; i++)
{
  fileOut.print(Integer.toString(number[i]));
}
```

Entonces, si se usa el programa ReadTextFile modificado para leer valores `int`, ¿qué valor tendría `number[0]`?

§15.5 Generador de archivos HTML

9. En el supuesto de que el objeto que administra la salida se denomina `writer`, escriba una declaración que abra un archivo de texto denominado `dogs.html` para salida por declaraciones `println`.
10. En un archivo HTML, ¿dónde van las etiquetas `<h1>` y `<\h1>`?
11. Escriba una declaración que rompa una conexión de salida a un archivo de texto denominada `writer`.

§15.6 Formato de archivo de texto *versus* formato de archivo binario

12. En un archivo de texto, cada carácter consume sólo un byte (ocho bits) de memoria. (F/C)
13. En un archivo de texto, cada carácter consume normalmente sólo dos bytes (16 bits) de memoria. (F/C)

§15.7 Archivo binario E/S

14. Escriba una declaración que abra un nuevo archivo binario para salida y asigne una referencia a la conexión hacia `binaryOut`. Ponga el archivo en el directorio actual, con el nombre de `windSpeed.data`.

§15.8 Archivo objeto E/S

15. Escriba una declaración que abra un archivo para entrada de objetos y asigne una referencia a la conexión hacia `objectIn`. Suponga que el archivo está en el directorio actual, con el nombre de `automobiles.data`.

§15.9 La clase `File`

16. Escriba un fragmento de código que enumere todos los archivos en el directorio que contienen el programa que se está ejecutando en ese momento.

Ejercicios

1. [Después de §15.2] ¿Cuál es la ventaja más importante de cada uno de los tres tipos de archivo de E/S: de texto, binario y objeto?
2. [Después de §15.3] Si olvida cerrar un archivo de texto, puede provocar que el rendimiento de su computadora se degrade. (F/C)
3. [Después de §15.3] Escriba los fragmentos de código que faltan en el siguiente programa de Java, de modo que éste escriba exitosamente la cadena `churchill` que se indica en un archivo denominado `elAlamein.txt`. Elabore `fileOut` de modo que los nuevos datos se sobrescriban en cualquier dato previo en un archivo existente con el nombre especificado.

```
/**********************************************************
* TextWriter.java
* Dean & Dean
*
* Esto escribe dos líneas de texto hacia un archivo de texto.
**********************************************************/

<fragment>

public class TextWriter
{
  public static void main(String[] args)
  {
    String[] churchill =
      {"Antes del Alamein nunca habíamos tenido una victoria.",
      "Después del Alamein nunca tuvimos una derrota."};
    PrintWriter fileOut;

    try
    {
      <fragment>
      for (String line : churchill)
```

```
      {
        <fragment>
      }
      <fragment>
    }
    catch (FileNotFoundException e)
    {
      System.out.println(e.getMessage());
    }
  } // end main
} // end TextWriter class
```

4. [Después de §15.3] Modifique el código en el ejercicio previo de modo que agregue al texto que ya está en `elAlamein.txt` el año en que ocurrió la batalla, 1942. En el programa elabore esta pieza de información adicional como un entero, como esto:

```
int year = 1942;
```

En el archivo, coloque esta pieza de información adicional en la línea de texto, después del texto previo.

5. [Después de §15.4] Se supone que el programa a continuación abre un archivo cuyo nombre de ruta completo es proporcionado por el usuario mediante un teclado. Luego, se supone que cuenta el número de palabras en el archivo, donde cualquier tipo de espacio en blanco es un delimitador de palabra. El programa está completo, excepto por un fragmento de código de varias líneas en el bloque `try`. Escriba el fragmento de código que falta. Use un objeto anónimo `File` como argumento de `Scanner` cuando instancie el objeto `File`. Por supuesto, puede usar los mismos métodos `hasNext` y `next` de la clase `Scanner` para el objeto `fileIn` que ya usó antes con el objeto `fileIn` cuando lea desde el teclado.

```
/*************************************************************
* WordsInFile.java
* Dean & Dean
*
* Esto cuenta las palabras en un archivo de texto.
*************************************************************/
import java.io.*;
import java.util.*;

public class WordsInFile
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Scanner fileIn;
    int numWords = 0;

    try
    {
      <fragment>
    } // end try
    catch (FileNotFoundException e)
    {
      System.out.println("Nombre de archivo inválido.");
    }
    catch (Exception e)
    {
      System.out.println("Error de lectura desde el archivo.");
    }
  } // end main
} // end WordsInFile class
```

Si el archivo es `family.txt` que se muestra en el siguiente ejercicio, debe obtenerse algo como esto:

Sesión muestra:

```
Enter full pathname of file:
e:/myJava/problems/chapter15/family.txt
Number of words = 63
```

6. [Después de §15.5] Como se explicó en el texto, las estrictas normas de HTML requieren que todas las etiquetas iniciales p ( ) tengan una etiqueta acompañante final p ( ). Edite el programa `HTMLGenerator.java` proporcionado en el texto, de modo que etiquetas finales p ( ) se inserten correctamente en el archivo HTML generado. Las etiquetas finales p deben insertarse en la parte inferior de cada párrafo.
Nota:

- En el archivo `family.text` a continuación, suponga que al final de cada línea hay un carácter de línea nueva.
- No permita que se genere una etiqueta final p cuando no haya etiqueta inicial p (las etiquetas iniciales y finales siempre deben ir por pares).
- Su programa debe ser robusto (es decir, manejar los casos extraños). En particular, debe manejar el caso en que sólo hay un título y ningún párrafo.

`family.txt` (el archivo de entrada):

```
Nuestra familia

Somos Stacy y John y vivimos en una casa
rodante cerca del río.

Tenemos una perrita, Barkley.
Barkley es una buena perrita.
Duerme mucho y escarba en el césped.
La alimentamos dos veces al día.

Tenemos dos hijas, Jordan y Caiden.
Son niñas y les gusta comer, gritar y jugar.
Las queremos mucho.
```

`family.html` (el archivo de salida resultante):

```
<html>
<head>
<title>Our Family</title>
</head>
<body>
<h1>Our Family</h1>
<p>
Somos Stacy y John y vivimos en una casa rodante
cerca del río.
</p>
<p>
Tenemos una perrita, Barkley.
Barkley es una buena perrita.
Duerme mucho y escarba en el césped.
La alimentamos dos veces al día.
</p>
<p>
Tenemos dos hijas, Jordan y Caiden.
Son niñas y les gusta comer, gritar y jugar.
Las queremos mucho.
</p>
</body>
</html>
```

7. [Después de §15.6] Suponga que cada una de las dos líneas (registros) siguientes consta de nueve caracteres visibles. En el supuesto de que se escriben hacia un archivo de texto mediante declaraciones `println` por una computadora Windows, muestre el patrón de bits para estos datos en un archivo.

```
Nik:  x88
Josh: x24
```

8. [Después de §15.6] En Windows, el símbolo de línea nueva es _____. En UNIX, el símbolo de línea nueva es _____.

9. [Después de §15.7] En un archivo binario, el regreso de carro y los caracteres de línea nueva no tienen ninguna función especial en las operaciones de lectura o escritura: simplemente son como cualquier otro carácter. (F/C)

10. [Después de §15.8] Inventario de una tienda de comestibles:
Mejore el programa del inventario de una tienda de comestibles creado en un proyecto del capítulo 13 al proporcionar una clase `FileHandler` que contenga métodos de lectura y escritura para escribir un objeto hacia un archivo en el directorio local o leer un objeto desde un archivo en el directorio local. Complete la siguiente estructura al proporcionar los fragmentos de código necesarios. Para hacer lo anterior, realmente no es necesario el programa del inventario de una tienda de comestibles del capítulo 13. Puede probar esta nueva clase con cualquier objeto instanciado correctamente. La clase que define ese objeto debe, no obstante, incluir una característica especial. ¿Cómo modificaría la clase `Inventory` a fin de incorporar esa característica especial?

```
/************************************************************
* FileHandler.java
* Dean & Dean
*
* Esto almacena y recupera un objeto.
************************************************************/

import java.util.*;
import java.io.*;

public class FileHandler
{
  public static void write(Object object, String filename)
  {
    ObjectOutputStream fileOut;

    try
    {
      <provide code fragment here>
    }
    catch (IOException e)
    {
      System.out.println(e.getMessage());
    }
  } // end write

  //***********************************************************

  public static Object read(String filename)
  {
    ObjectInputStream fileIn;
    Object object;

    try
    {
      <provide code fragment here>
    }
    catch (Exception e)
    {
      System.out.println(e.getMessage());
      return new Object();                // para satisfacer al compilador
    }
  } // end read
} // end FileHandler class
```

Luego, con la clase `FileHandler` agregada al programa ligeramente modificado del inventario de una tienda de comestibles, el siguiente controlador muestra cómo es posible eliminar modificaciones indeseadas al restituir los datos previamente guardados:

```
/**********************************************************
* InventoryDriver2.java
* Dean & Dean
*
* Esto muestra el llenado del inventario de una tienda de comestibles.
**********************************************************/

public class InventoryDriver2
{
  public static void main(String[] args)
  {
    Inventory store = new Inventory("groceries");

    store.newItem("bread", 15, 9.99);
    store.newItem("SunnyDale", "milk", 2, 2.00);
    store.newItem("eggs", 3, 1.50);
    store.newItem("bread", 2, 1.25);        // cuidado: hay en el almacén
    store.stockReport();
    FileHandler.write(store, "Inventory.data");

    store.update("SunnyDale", "milk", .25); // subir el precio en 25%
    store.update("eggs", -1);               // disminuir la calidad por 1
    store.update("beer", 3);                // cuidado: no hay en el almacén
    store.newItem("BrookSide", "milk", 4, 1.95);
    store.stockReport();

    store = (Inventory) FileHandler.read("Inventory.data");
    store.stockReport();
  } // end main
} // end InventoryDriver2 class
```

Salida:

```
Item already exists - bread
bread - in stock: 15, price: $9.99
SunnyDale milk - in stock: 2, price: $2.00
eggs - in stock: 3, price: $1.50
Total value: $158.35

Cannot find specified item - beer
bread - in stock: 15, price: $9.99
SunnyDale milk - in stock: 2, price: $2.50
eggs - in stock: 2, price: $1.50
BrookSide milk - in stock: 4, price: $1.95
Total value: $165.65

bread - in stock: 15, price: $9.99
SunnyDale milk - in stock: 2, price: $2.00
eggs - in stock: 3, price: $1.50
Total value: $158.35
```

11. [Después de §15.9] Suponga que se usó un programa como Notepad de Microsoft o de UNIX vi para crear un archivo de texto, `alphabet.txt`, que contiene esta única línea de texto: “abcdefg”. Escriba los fragmentos de código faltantes en el siguiente programa de Java, de modo que lea y muestre exitosamente los datos en ese archivo.

```
/*************************************************************
* TextReader.java
* Dean & Dean
*
* This reads a line of text from a text file.
*************************************************************/

<fragment>

public class TextReader
{
  public static void main(String[] args)
  {
    File file = <fragment>;
    Scanner fileIn;

    try
    {
      fileIn = <fragment>;
      System.out.println(fileIn.nextLine());
      <fragment>                  // close the file
    }
    catch (FileNotFoundException e)
    {
      System.out.println(e.getMessage());
    }
  } // end main
} // end TextReader class
```

12. [Después de §15.9] Suponga que un archivo de texto denominado `myDates.txt` contiene esta línea de texto:

```
1999 2000 2001 2002
```

Modifique el programa en el ejercicio previo de modo que lea como enteros los datos de este archivo y los imprima de inmediato en la pantalla, como se muestra:

Salida:

```
1999
2000
2001
2002
```

13. [Después de §15.9] Observe el ejemplo de texto en la figura 15.13 y la documentación Sun de Java para la clase `File` y explique lo que hacen los siguientes métodos y constructor:

a) `File(".")`
b) `getAbsoluteFile()`
c) `getParentFile()`
d) `list()`

Soluciones a las preguntas de revisión

1. Falso. No es posible ver el contenido de archivos binarios o archivos objeto con editores de texto.
2. `import java.io.*;`
3. 1) Abrir el archivo. 2) Hacer la transferencia y transformar el formato de los datos. 3) Cerrar el archivo.
4. Use el método `println` para colocar la cadena distinta en una línea por separado, como esto:

```
fileOut.println(name + " " + id);
```

5.
```
fileOut = new PrintWriter(
  new FileOutputStream("mydata.txt", true));
```

6. El parámetro `String` del constructor `Scanner` opera sobre la cadena de entrada en sí, no en el archivo que identifica.

7. El valor final de `name.length() = 3`. El método `nextDouble` lee todo hasta el último dígito numérico. El método `nextLine` lee el siguiente espacio más los dos caracteres en "hi".
8. `number[0] es igual a 234`. Escribir enteros sin un espacio en blanco a continuación combina los números de salida por separado en lo que se ve como un número en el archivo, y una lectura subsiguiente interpreta la combinación como un número.
9. `writer = new PrintWriter("dogs.html");`
10. Las etiquetas `<h1>` y `</h1>` encierran el encabezado visible de la página Web.
11. `writer.close;`
12. Cierto. En un archivo de texto, cada carácter consume sólo un byte de memoria.
13. Cierto. En un archivo binario, cada carácter consume normalmente dos bytes (16 bits) de memoria.
14.
```
DataOutputStream binaryOut = new DataOutputStream(
  new FileOutputStream("windSpeed.data"));
```
15.
```
ObjectInputStream objectIn = new ObjectInputStream(
  new FileInputStream("automobiles.data"));
```
16. Esto enumera todos los archivos en el directorio que contiene el programa que se está ejecutando en ese momento:

```
String[] listing =
  (new File(".")).getAbsoluteFile().getParentFile().list();
for (int i=0; i<listing.length; i++)
{
  System.out.println(listing[i]);
}
```

CAPÍTULO 16

Fundamentos de programación GUI

Objetivos

- Comprender el paradigma de la programación de manejo de eventos. En particular, comprender qué significa activar un evento, así como los términos oyente y manipulador de eventos.
- Usar la clase `JFrame` para implementar la funcionalidad de ventanas.
- Crear y usar los componentes `JLabel`, `JTextField` y `JButton`.
- Implementar un oyente para los componentes `JTextField` y `JButton`.
- Comprender lo que es una interfaz e implementar la interfaz `ActionListener`.
- Comprender lo que es una clase interna e implementar un oyente como una clase interna.
- Conocer la diferencia entre una clase interna anónima y una clase interna estándar.
- Crear y usar ventanas de diálogo `JOptionPane`.
- Distinguir entre múltiples eventos.
- Describir los paquetes primarios GUI.
- Describir las diferencias entre componentes de peso ligero y de peso pesado.

Relación de temas

Figura 16.1 Ejemplo de ventana que usa una ventana de diálogo y un texto.

16.1 Introducción

Esperamos que el lector haya permanecido sentado en el borde de su silla al leer los nombres de los capítulos previos. Si no, debe prepararse para hacerlo a partir de este momento. Ya es hora de presentar el verdadero material: la programación *graphical user interface* (*GUI*).

Quizá ya ha escuchado el término GUI, pero, las tres palabras de GUI, Graphical User Interface, ¿tienen sentido? "Graphical" se refiere a imágenes; "User", a una persona, e "Interface", a comunicación. Así, la programación GUI usa imágenes —como ventanas, ventanas de diálogo, botones, etc.— para comunicarse con los usuarios. Por ejemplo, en la figura 16.1 se muestra una ventana con un recuadro (o pestaña) de diálogo y un botón. Las ventanas, los cuadros de diálogo y los botones se describirán en detalle más adelante.

En otra época, las interfaces del programa eran sólo de texto. Los programas podían solicitar algo al usuario mediante un texto, y el usuario podía responder con un texto. Eso es lo que se ha estado utilizando en todos los programas hasta el momento. La entrada/salida (E/S) de texto funciona bien en muchas situaciones, pero es posible evitar el hecho de que hay quien considera aburrido el despliegue de un texto. Muchos de los usuarios actuales esperan que los programas sean más animados. Esperan ventanas, botones, colores, etc., para entrada y salida. Esperan GUI.

Aunque las empresas continúan escribiendo muchos programas basados en texto para uso interno, normalmente escriben programas basados en GUI porque los programas externos van a los clientes, y los clientes no suelen comprar programas a menos que estén basados en GUI. Entonces, si el lector desea escribir programas que compre la gente, es mejor que aprenda programación GUI.

Este capítulo empieza con una revisión de los conceptos y terminología básicos en GUI. Luego se aborda un programa rudimentario donde se introduce la sintaxis básica de GUI. En seguida se estudian los oyentes, las clases internas y varios *componentes GUI* rudimentarios, que son objetos ubicados dentro de una ventana, incluyendo `JLabel`, `JTextField` y `JButton`. Por último, se aborda la clase `JOptionPane` (para generar una ventana de diálogo) y la clase `Color` (para generar un color).

Quizá haya notado secciones opcionales GUI al final y casi en la mitad de los capítulos previos. El material GUI en este capítulo y en el siguiente es diferente del material GUI de los capítulos previos, y no depende de ese material. Por tanto, si ha omitido el material GUI anterior, no debe preocuparse.

Para comprender este capítulo, es necesario estar familiarizado con programación orientada a objetos, arreglos, herencia y con manejo de excepciones. Para ello, es necesario haber leído hasta el capítulo 14. Este capítulo no depende del material cubierto en el capítulo 15.

16.2 Fundamentos de programación de manejo de eventos

Usualmente los programas GUI utilizan técnicas de *programación de manejo de eventos*. La idea básica detrás de la programación de manejo de eventos es que el programa espera que ocurran eventos y responde a ellos si y cuando ocurren.

Terminología

Entonces, ¿qué es un evento? Un *evento* es un mensaje que indica al programa que algo ha ocurrido. Por ejemplo, si el usuario hace clic en un botón, entonces se genera un evento, y le indica al programa que en

un botón particular se hizo clic. Más formalmente, cuando un usuario hace clic en un botón, se dice que el objeto de botón *activa un evento.* Observe estos ejemplos adicionales de eventos:

Acción del usuario	Qué ocurre
Oprimir la tecla Enter mientras el cursor está dentro de una ventana de diálogo.	El objeto en la ventana de diálogo activa un evento e indica al programa que la tecla Enter fue oprimida dentro de la ventana de diálogo.
Hacer clic en el botón de un menú.	El objeto en el menú activa un evento, e indica al programa que se seleccionó ese elemento del menú.
Cerrar una ventana (hacer clic en el botón "X" en el ángulo superior derecho).	El objeto en la ventana activa un evento, e indica al programa que se hizo clic en el botón para cerrar la ventana.

Si un evento se activa, y se desea que el programa maneje ese evento activado, entonces es necesario crear un *oyente* del evento. Por ejemplo, si se desea que el programa haga algo cuando el usuario hace clic en un botón particular, es necesario crear un oyente para el botón. Por ahora, considere que un oyente es un oído. Si se activa un evento y no hay oyente que lo escuche, entonces el evento activado nunca es "escuchado" y no hay respuesta a eso. Por otra parte, si hay un oyente escuchando un evento activado, entonces el oído "escucha" el evento y entonces el programa responde al evento activado. La forma en que responde el programa es ejecutando un trozo de código conocido como *manipulador de eventos.* Observe la figura 15.2. Ahí se muestra que se está oprimiendo un botón (vea el indicador del ratón), un evento que se activa (vea las ondas sonoras), un oyente que escucha el evento (vea la oreja) y la ejecución de un manipulador de eventos (vea la flecha dirigida hacia abajo al lado del código del manipulador de eventos). Este sistema de utilizar oyentes para el manejo de eventos se conoce como *modelo de delegación de eventos*: la manipulación de eventos se "delega" en un oyente particular.

Marco de programación de manejo de eventos

Con base en la descripción de arriba, la programación de manejo de eventos puede percibirse como un nuevo tipo de programación. En particular, lo concerniente a la activación de un evento y el escuchar un evento activado. Mucha gente está de acuerdo con la idea de que la programación de manejo de eventos constituye un nuevo tipo de programación. Sin embargo, lo cierto en la cuestión es que en realidad se trata simplemente de programación orientada a objetos con un aderezo de ventanas. Haga mucho de este aderezo de ventanas. Sun proporciona una colección exhaustiva de clases GUI que, juntas, constituyen un marco en el cual construir las aplicaciones GUI. Y el marco está integrado por clases, métodos, herencia, etc.; es decir, está compuesto de componentes de la programación orientada a objetos. Como programador, no es necesario comprender todos los detalles de cómo funciona el marco; basta comprenderlo lo suficiente para utilizarlo. Por ejemplo, el lector debe saber cómo conectar correctamente sus manipuladores de eventos. En la figura 16.3 se proporciona una ilustración gráfica de alto nivel de lo que se está hablando.

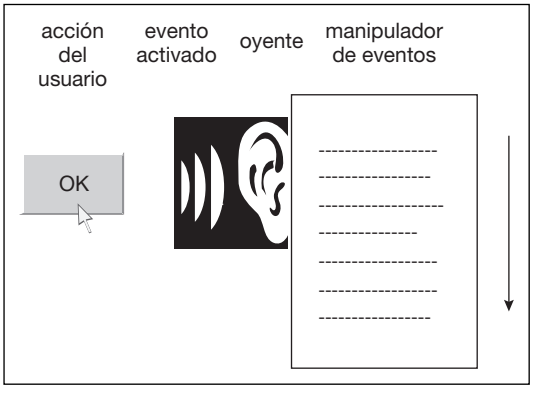


Figura 16.2 Lo que ocurre cuando se oprime un botón.

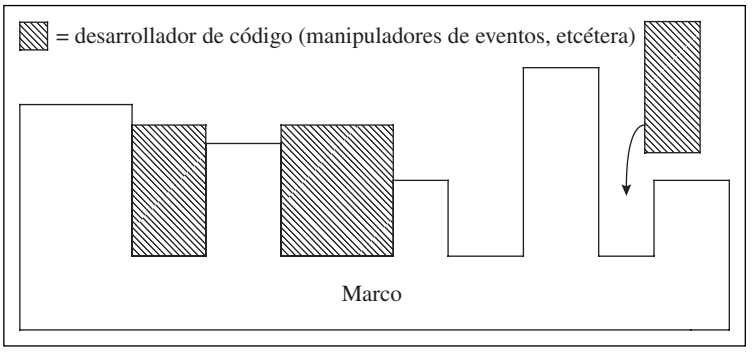


Figura 16.3 Marco de programación de manejo de eventos.

¿Por qué Sun se molesta en proporcionar el marco de programación de manejo de eventos? Satisface el objetivo de obtener beneficio máximo a partir de un código mínimo. Con ayuda del marco, los programadores de Java pueden levantar y ejecutar un programa GUI con una cantidad de esfuerzo relativamente pequeña. Inicialmente, el esfuerzo puede no ser pequeño, pero cuando se considera todo lo que hace el programa GUI (inicio automático de eventos, escucha de eventos activados, etc.) se encuentra que el rendimiento de su inversión es bastante bueno.

16.3 Un sencillo programa de ventanas

Está bien. Suficiente charla de conceptos. Es hora de arremangarse y ensuciarse las manos con algo de código. Para tener una idea panorámica, se comenzará con un sencillo programa GUI y los comandos GUI se analizarán a alto nivel. Después, los comandos GUI se cubrirán con más detalle.

En la figura 16.4 se presenta un programa sencillo de ventanas que muestra una línea de texto dentro de una ventana. Observe las dos declaraciones `import` en la parte superior del programa. Importan los paquetes `javax.swing` y `java.awt`. Al escribir programas GUI, se usan muchas clases preconstruidas GUI de Java de la biblioteca API de Sun. Para usar clases GUI preconstruidas es necesario importarlas en su programa GUI. Las clases deben importarse individualmente, aunque hay una mejor forma de hacerlo. Recuerde que un paquete es una colección de clases preconstruidas. Puesto que la mayor parte de las clases preconstruidas GUI provienen de los paquetes `javax.swing` y `java.awt`, si se importan estos dos paquetes se importan las clases preconstruidas GUI más críticas. Acostúmbrese a importar estos dos paquetes en cada uno de sus programas GUI. Recuerde que para importar un paquete es necesario usar un asterisco; es decir, `import javax.swing.*;` El * es un comodín, y permite importar todas las clases dentro de un paquete particular.

En el encabezado de la clase `SimpleWindows`, observe la cláusula `extends Jframe`. La clase `JFrame` forma parte del marco GUI antes mencionado. La clase `JFrame` proporciona características estándar de Windows, como la barra de título, un botón minimizado, etc Abajo del encabezado de la clase, observe las constantes nombradas `WIDTH` y `HEIGHT`. Son usadas por la llamada al método `setSize` para especificar las dimensiones de la ventana.

A continuación se analizará el método `main`. Los programas GUI típicamente crean una ventana con componentes GUI, y luego simplemente se instalan en espera de que el usuario haga algo, como clic en un botón, seleccionar una opción de menú, etc. Así, `main` es muy corto: simplemente instancia la ventana y eso es todo. En este ejemplo simple, ni siquiera nos molestamos en asignar un objeto de ventana instanciado a una variable de referencia. Repaso: ¿cómo se llama un objeto que no está almacenado en una variable de referencia? Objeto anónimo.

Al efectuar la instanciación del objeto anónimo, `main` llama al constructor `SimpleWindows`. El constructor `SimpleWindows` 1) llama a `setTitle` para asignar el título de la ventana, 2) llama a `setSize` para asignar el tamaño de la ventana, 3) llama a `setLayout` para asignar el esquema de la disposición de la ventana y 4) llama a `setDefaultCloseOperation` para hacer que el *botón close-window* (la "X" en el ángulo superior derecho) funcione correctamente.

```
/*********************************************************
* SimpleWindow.java
* Dean & Dean
*
* Este programa muestra una etiqueta en una ventana.
*********************************************************/

import javax.swing.*; // for JFrame, JLabel
import java.awt.*;    // for FlowLayout

public class SimpleWindow extends JFrame
{
  private static final int WIDTH = 250;
  private static final int HEIGHT = 100;

  //*******************************************************

  public SimpleWindow()
  {
    setTitle("Simple Window");
    setSize(WIDTH, HEIGHT);
    setLayout(new FlowLayout());
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents();
    setVisible(true);
  } // end SimpleWindow constructor

  //*******************************************************

  private void createContents()
  {
    JLabel label = new JLabel("Hi! I'm Larry the label!");
    add(label);
  } // end createContents

  //*******************************************************

  public static void main(String[] args)
  {
    new SimpleWindow();
  } // end main
} // end class SimpleWindow
```

Esto agrega la etiqueta a la ventana.

Esto instancia un objeto anónimo window.

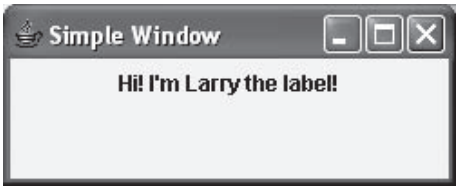


Figura 16.4 Sencillo programa de ventanas y su salida.

En aras de la modularización, el constructor `SimpleWindows` llama después a un método de ayuda, `createContents`, para crear los componentes que van dentro de la ventana. El método `createContents` contiene sólo dos líneas. Con sólo dos líneas, realmente no hay necesidad de un método de ayuda, pero queremos inculcar buenos hábitos al lector. Para este ejemplo trivial, sólo hay un componente y no hay manipulador de eventos para el componente. Así, dos líneas es todo lo que se requiere.

Pero los programas GUI normales tienen múltiples componentes y múltiples manipuladores de eventos. Para eso, se requieren unas cuantas líneas. Si estas líneas se colocan en el constructor se tendría un constructor largo. Resulta mejor descomponer las cosas y colocarlas en el método de ayuda.

El método `createContents` instancia una componente `JLabel` y luego llama al método `add` para agregar la componente `JLabel` a la ventana. Un componente `JLabel` es el tipo más simple de componente GUI. Se trata de una pieza de texto que el usuario puede leer pero no modificar.

Después de ejecutar `createContents`, la JVM regresa al constructor `SimpleWindows`. Luego, el constructor `SimpleWindows` llama a `setVisible` para hacer visible la ventana.

16.4 Clase `JFrame`

En la sección previa, se presentó la clase `JFrame`. En esta sección se describe con mayor detalle `JFrame`. Más específicamente, se analizan sus características y sus métodos.

Fundamentos de la clase `JFrame`

Hoy en día, casi todo el software que puede comprarse está basado en ventanas. Cuando se carga este software, se observa una ventana que contiene una barra de título, límites, un botón para minimizar, un botón para cerrar la ventana, la función de modificar el tamaño de la ventana, etc. Estas características pueden implementarse en el aula a partir de nada, pero, ¿por qué "reinventar la rueda"? La clase `JFrame` implementa las características estándar de las ventanas que llegan a conocerse y quererse. Para adquirir gratis toda esta funcionalidad genial de las ventanas, simplemente implemente sus clases al extender la clase `JFrame`. ¡Qué cosa!

La clase `JFrame` debe ser la superclase para la mayor parte de las aplicaciones GUI de ventanas, de modo que una ventana definida por el programador normalmente cuenta con `extends JFrame` en el encabezado de su clase. Para que `extends JFrame` funcione, es necesario importar la clase `JFrame` o importar el paquete `javax.swing` de `JFrame`. Como ya se explicó, suele ser común importar el paquete `javax.swing` para todos los programas GUI.

La clase `JFrame` se denomina *contenedor* porque contiene componentes (como etiquetas, botones, menús, etc.). Hereda la capacidad de contener componentes de su superclase, la clase `Container`.

Métodos `JFrame`

Al extender la clase `JFrame`, automáticamente se obtiene la funcionalidad estándar de la ventana antes mencionada. Además, se heredan bastantes métodos relacionados con la ventana. En el programa SimpleWindow, se usan estos métodos heredados: `setTitle`, `setSize`, `setLayout`, `setDefaultCloseOperation`, `add` y `setVisible`. Los métodos `setLayout` y `setDefaultCloseOperation` provienen directamente de la clase `JFrame`. Los otros métodos provienen directamente del antepasado de la clase `JFrame`: `set Title` de la clase `Frame`, `add` proviene de la clase `Container`; y `setSize` y `setVisible` provienen de la clase `Component`.

El método `setTitle` muestra una cadena especificada en la barra de título de la ventana actual. Si no se llama a `setTitle`, entonces la barra de título de la ventana permanece vacía.

El método `setSize` asigna el ancho y la altura de la ventana actual. Vea la figura 16.4 y observe cómo el programa SimpleWindow asigna el ancho de 300 y la altura de 200. Los valores del ancho y la altura se especifican en términos de *pixeles*. Un pixel es la menor unidad que puede mostrarse en el monitor de una computadora, y en la pantalla aparece como un punto. Si el método `setSize` se llama con un ancho de 300 y una altura de 200, entonces la ventana consta de 200 renglones, donde cada renglón contiene 300 pixeles. Cada pixel muestra un color específico. Los pixeles forman una imagen con varios colores para los distintos pixeles. Por ejemplo, la ventana que se muestra en la figura 16.4 podría contener pixeles azules en el perímetro (para el límite de la ventana) y pixeles negros en el centro (para el mensaje en la ventana).

Para darle una idea de cuán grande es una ventana de 300 por 200 pixeles, es necesario conocer las dimensiones, en pixeles, de toda una pantalla de computadora. Las dimensiones de la pantalla de una computadora se denominan *resolución de la pantalla*. Los valores de la resolución son ajustables. Dos

valores de resolución comunes son de 800 por 600 y de 1 024 por 768. Las dimensiones de 800 por 600 muestran 600 renglones, donde cada renglón contiene 800 pixeles.

⚠ Si se olvida llamar al método `setSize`, la ventana es realmente pequeña. Sólo muestra el inicio del título y los tres botones estándar de ajuste de la ventana: minimizar, maximizar y cerrar la ventana. No muestra el contenido de la ventana, a menos que el tamaño de la ventana se ajuste manualmente. Esto es lo que el programa SimpleWindow muestra en caso de que se omita llamar al método `setSize`:

El método `setLayout` asigna un *gestor de disposición* a la ventana actual. Este gestor es un software preconstruido de Sun que determina la ubicación de los componentes. En la llamada al método `setLayout` del programa SimpleWindow se especifica el gestor `FlowLayout`, mismo que provoca que los componentes se ubiquen en la parte central superior. La clase `FlowLayout` está definida en el paquete `java.awt`, de modo que no olvide importar ese paquete. En el siguiente capítulo se describen con más detalle el gestor `FlowLayout` y otros gestores de disposición. En este capítulo se usa el gestor `FlowLayout` (en contraposición a otros gestores de disposición) porque el gestor `FlowLayout` es el más sencillo de usar, y por el momento se está intentando mantener simples las cosas.

⚠ Por defecto, el botón para cerrar la ventana del programa (la X en el ángulo superior derecho de la ventana) no funciona muy bien. Cuando el usuario hace clic en este botón, la ventana se cierra, aunque el programa sigue ejecutándose al fondo. Para remediar esta situación, es necesario llamar a `setDefaultCloseOperation(EXIT_ON_CLOSE)`. Luego, cuando el usuario hace clic en el botón para cerrar la ventana, ésta se cierra y el programa termina. Tener un programa cerrado en el fondo suele ser imperceptible, razón por la cual muchos programadores tienen dificultades para llamar a `setDefaultCloseOperation(EXIT_ON_CLOSE)`. A pesar de lo anterior, es necesario tratar de recordar llamarlo. En caso de olvidar llamarlo, la computadora del usuario tiene memoria limitada y en el fondo hay muchos programas en ejecución, entonces el rendimiento de la computadora se degrada.

El método `add` agrega un componente especificado a la ventana actual. Una vez que se agrega el componente, permanece en la ventana durante la vida del programa. Esto se menciona para que el lector se sienta cómodo al usar una declaración de variable local como componente. En el siguiente ejemplo, aun cuando `label` se ha identificado localmente dentro de `createContents`, el componente instanciado `JLabel` permanece en la ventana después que termina `createContents`:

```
private void createContents()
{
  JLabel label = new JLabel("Hi! I'm Larry the label!");
  add(label);
} // end createContents
```

⚠ Las ventanas son invisibles por defecto. Para hacer visibles una ventana y su contenido, es necesario agregar los componentes a la ventana y después llamar `setVisible (true)`. Lo anterior debe hacerse en ese orden: primero agregar el componente y luego llamar a `setVisible`. En caso contrario, no aparecen los componentes agregados. Para hacer invisible una ventana, debe llamarse a `setVisible (false)`.

La clase `JFrame` contiene muchos métodos adicionales, demasiados para mencionarlos aquí. Si el lector tiene tiempo, le recomendamos averiguar lo que está disponible al consultar la clase `JFrame` en el sitio Sun API de Java: http://java.sun.com/javase/6/docs/api/

16.5 Componentes en Java

A continuación se considerarán los objetos que se encuentran en una ventana: los componentes. Éstos son algunos ejemplos de componentes en Java:

- `JLabel`, `JTextField`, `JButton`,
- `JTextArea`, `JCheckBox`, `JRadioButton`, `JComboBox`
- `JMenuBar`, `JMenu`, `JMenuItem`

Éstos no son todos los componentes en Java, sino solamente algunos de los más utilizados. En este capítulo se describirán los tres primeros componentes y los tres últimos, en el siguiente capítulo.

Todas las clases de los componentes anteriores están en el paquete `javax.swing`, por lo que es necesario importar este paquete a fin de usarlo. Sin embargo, recuerde que ese paquete ya fue importado para acceder a la clase `JFrame`. No es necesario importarlo dos veces.

Las clases de componentes usualmente se derivan de la clase `JComponent`, que soporta muchas características heredables. Junto con muchos otros métodos, la clase `JComponent` contiene métodos que manejan estas características de los componentes:

- colores en primer plano y de fondo
- fuente de texto
- apariencia de los límites
- sugerencias de herramientas
- foco

Para información detallada sobre estas características, consulte la clase `JComponent` en el sitio Sun API de Java.

16.6 Componente `JLabel`

Interfaz de usuario

El componente `JLabel` no hace mucho. Simplemente muestra una sola línea de texto que se ha especificado. Se considera un componente de sólo lectura porque el usuario puede leerlo, pero no puede interactuar con él.

Normalmente, el componente `JLabel` muestra una sola línea de texto, no varias líneas. Si el lector desea ver múltiples líneas, debe usar el componente `JTextArea`, que se analiza en el siguiente capítulo.

Implementación

Para crear un objeto `JLabel`, el constructor `JLabel` se llama así:

```
JLabel <JLabel-reference> = new JLabel(<label-text>);
```

opcional

El *label-text* es el texto que aparece en el componente `JLabel`. Si el argumento de label-text contiene carácter de línea nueva, `\n`, se ignora (recuerde que el componente `JLabel` sólo muestra una línea de texto). Si se omite el argumento de label-text, entonces el componente `JLabel` no muestra nada. ¿Por qué instanciar una etiqueta vacía? Porque así es posible llenarla más tarde con texto que depende de la misma condición.

Para agregar un objeto `JLabel` a la ventana `JFrame`, se usa esta sintaxis:

```
add(<JLabel-reference>);
```

JLabel-reference proviene de la declaración de la iniciación de arriba

La clase `JLabel` requiere el paquete `javax.swing`, que ya debe estar disponible desde antes, porque es necesario para la clase `JFrame`.

Métodos

La clase `JLabel`, como todos los componentes de las clases GUI, tiene muy pocos métodos. Aquí sólo se mencionan dos de ellos: los métodos de acceso y regulador `getText` y el `setText`. A continuación se presentan sus encabezados y descripciones API:

```
public String getText()
```

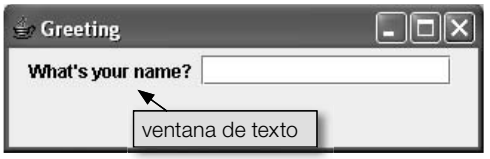
Devuelve el texto de la etiqueta.

`public void setText(String text)`
Asigna el texto de la etiqueta. Observe que los programadores pueden actualizar el texto de la etiqueta, pero el usuario no puede hacerlo.

16.7 Componente `JTextField`

Interfaz del usuario

El componente *`JTextField`* muestra un rectángulo y permite que el usuario introduzca texto en el rectángulo. Éste es un ejemplo:



Implementación

Para crear un objeto `JTextField`, el constructor `JTextField` se llama así:

`JTextField` <*JTextField-reference*> `= new JTextField(`<*default-text*>`,` <*width*>`);`

opcional

El *default-text* es el texto que aparece en la ventana de diálogo por defecto. El *ancho* es el número de caracteres que es posible mostrar en la ventana de diálogo a la vez. Si el usuario introduce más caracteres de los que es posible mostrar a la vez, entonces los caracteres más a la izquierda recorren el desplegado. Si se omite el texto por defecto, entonces la cadena vacía se usa como el valor por defecto. Si se omite el argumento del ancho, entonces el ancho de la ventana es ligeramente mayor que el ancho del texto por defecto.

Para agregar un objeto `JTextField` a la ventana `JFrame`, se usa esta sintaxis:

`add(`<*JTextField-reference*>`);`

La clase `JTextField` requiere el paquete `javax.swing`, que ya debe estar disponible desde antes, porque es necesario para la clase `JFrame`.

Métodos

La clase `JTextField` tiene muy pocos métodos. A continuación se presentan los encabezados y descripciones API de algunos de los más útiles:

`public String getText()`
Devuelve el contenido de la ventana de texto.

`public void setText(String text)`
Asigna el contenido de la ventana de texto.

`public void setEditable(boolean flag)`
Hace editable o no editable la ventana de texto.

`public void setVisible(boolean flag)`
Hace visible o invisible la ventana de texto.

`public void addActionListener(ActionListener listener)`
Agrega un oyente a la ventana de texto.

Estas ventanas son editables por defecto, lo que significa que los usuarios pueden escribir en su interior. Si se quiere evitar que los usuarios editen una ventana de texto, es necesario llamar a `setEditable`

con un valor de argumento `false`. Al llamar a `setEditable(false)` se impide que los usuarios actualicen una ventana de texto. Los programadores pueden llamar al método `setText` sin importar si la ventana de texto es editable o no.

Los componentes son visibles por defecto, aunque hay algunas instancias en las que es aconsejable llamar a `setVisible(false)` y hacer desaparecer el componente. Una vez que se calcula el resultado, tal vez se quiera que sólo el resultado aparezca sin los demás componentes. Cuando se desaparece un componente, su espacio es reclamado de inmediato por la ventana, de modo que otros componentes puedan usarlo.

Cuando un componente `JTextField` llama a `addActionListener`, la JVM vincula un objeto oyente a la ventana de texto, lo cual permite que el programa responda cuando el usuario presiona Enter dentro de la ventana de texto. Los oyentes se estudiarán con mayor detalle dentro de poco, pero primero se abordará un programa ejemplo que pone en práctica lo aprendido hasta el momento. . .

16.8 Programa Saludo

En las figuras 16.5*a* y 16.5*b* se presenta un programa Saludo que muestra un saludo personalizado. Lee el nombre del usuario de una ventana de texto (componente `JTextField`) y muestra el nombre introducido en una etiqueta (un componente `JLabel`).

La mayor parte del código en el programa Saludo debe ser conocido, puesto que imita de cerca el código en el programa SimpleWindow. Por ejemplo, observe el breve método `main` con el objeto anónimo instanciación. Observe el constructor que contiene llamadas a `setTitle`, `setSize`, `setLayout`, `setDefaultCloseOperation` y `setVisible`. Por último, observe el método de ayuda `createContents` que crea los componentes y los agrega a la ventana. A continuación, la atención se dirigirá a lo que hay de nuevo sobre el programa Saludo: una ventana de texto y un manipulador de eventos.

El programa Saludo usa una ventana de texto, `nameBox`, para almacenar el nombre del usuario. Observe cómo el método `createContents` instancia `nameBox` con un ancho de 15. Observe cómo el método `createContents` llama al método `add` para agregar `nameBox` a la ventana. Ese código es directo. Pero algo que no es tan directo es la declaración de `nameBox`. Se ha declarado como una variable de instancia en la parte superior de la clase. ¿Por qué una variable de instancia en lugar de una variable local `createContents`? ¿No que se prefieren las variables locales? Sí, pero en este caso es necesario acceder a `nameBox` no sólo en `createContents`, sino también en el manipulador de eventos `actionPerformed` (que se abordará en breve). Es posible usar una variable local dentro de `createContents` y aún así poder acceder a ella desde el manipulador de eventos, aunque tiene un precio.[1] Por ahora, las cosas se mantienen sin complicación y `nameBox` se declara como una variable de instancia. El mismo razonamiento es válido para la etiqueta `greeting`. Es necesario acceder a ella en `createContents` y también en el manipulador de eventos `actionPerformed`, de modo que se hace una variable de instancia.

El manipulador de eventos `actionPerformed` del programa Saludo especifica lo que ocurre cuando el usuario oprime Enter dentro de la ventana de texto. Observe que el método `actionPerformed` está dentro de la clase `Listener`. En la siguiente sección se estudiarán los oyentes y la mecánica de los manipuladores de eventos.

16.9 Componente oyentes

Cuando el usuario interactúa con un componente (por ejemplo, cuando el usuario hace clic en un botón u oprime Enter mientras está en una ventana de texto), el componente activa un evento. Si el componente está conectado a un oyente, el evento activado es "escuchado" por el oyente. En consecuencia, el oyente manipula el evento al ejecutar su método `actionPerformed`. En esta sección se aprenderá a hacer que todo esto funcione al crear un oyente y un método `actionPerformed` asociado.

[1] Si dentro de `createContents` se declara una variable localmente, ésta puede recuperarse desde un manipulador de eventos al llamar a `getSource`. El método `getSource` se analiza en la sección 16.14.

```
/*************************************************************
* Greeting.java
* Dean & Dean
*
* Este programa muestra ventanas de texto y etiquetas. Cuando el usuario
* oprime Enter después de escribir algo en la ventana de diálogo, el valor
* de la ventana de diálogo muestra la etiqueta que se muestra a continuación.
*************************************************************/

import javax.swing.*;    // for JFrame, JLabel, JTextField
import java.awt.*;       // for FlowLayout
import java.awt.event.*; // for ActionListener, ActionEvent

public class Greeting extends JFrame
{
  private static final int WIDTH = 325;
  private static final int HEIGHT = 100;
  private JTextField nameBox; // contiene el nombre del usuario
  private JLabel greeting;    // saludos personalizados

  //*********************************************************

  public Greeting()
  {
    setTitle("Greeting");
    setSize(WIDTH, HEIGHT);
    setLayout(new FlowLayout());
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents();
    setVisible(true);
  } // end constructor

  //*********************************************************

  // Create components and add them to window.
  private void createContents()
  {
    JLabel namePrompt = new JLabel("¿Cómo te llamas?");
    nameBox = new JTextField(15);
    greeting = new JLabel();
    add(namePrompt);
    add(nameBox);
    add(greeting);
    nameBox.addActionListener(new Listener());
  } // end createContents
```

4. Este paquete se importa para manipulación de eventos.

3. Registra un oyente.

Figura 16.5*a* Programa Saludo, parte A.

Cómo implementar un oyente

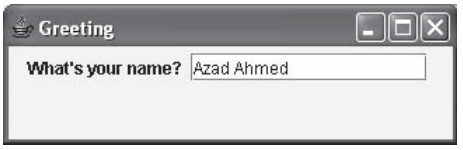
A continuación se muestran los pasos necesarios para implementar un oyente para una ventana de texto. Estos pasos corresponden a las llamadas numeradas en las figuras 16.5*a* y 16.5*b*:

1. Definir una clase con una cláusula `ActionListener` agregada a la derecha del encabezado de la clase. Para ver un ejemplo, observe la llamada 1 en la figura 16.5*b*. La cláusula `implement ActionListener` significa que la cláusula es una implementación de la interfaz `ActionListener`. Las interfaces se analizan en la siguiente subsección.

```
  //*********************************************************

  // Inner class for event handling.

  private class Listener implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      String message; // the personalized greeting
      message = "Glad to meet you, " + nameBox.getText()+ "!";
      nameBox.setText("");
      greeting.setText(message);
    } // end actionPerformed
  } // end class Listener

  //*********************************************************

  public static void main(String[] args)
  {
    new Greeting();
  } // end main
} // end class Greeting
```

1. encabezado de la clase interna.

2. manipulador de eventos



Después de oprimir Enter en la ventana de diálogo:

Figura 16.5*b* Programa Saludo, parte B y su salida asociada.

2. Incluir un método manipulador de eventos `actionPerformed` en su clase oyente. Ésta es una estructura de un método `actionPerformed` dentro de una clase oyente:

```
private class Listener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    <do-something>
  }
}
```

Inclusive si su método `actionPerformed` no usa un parámetro `ActionEvent` (`e`, arriba), sigue siendo necesario incluir ese parámetro en el encabezado del método para hacer que su método se ajuste a los requerimientos de un oyente.

Para ver un ejemplo de un método `actionPerformed` completamente ejecutado, observe la llamada 2 en la figura 16.5*b*. Esta llamada se refiere a una clase oyente denominada `Listener`. `Listener` no es una palabra reservada: simplemente es un buen nombre descriptivo para la clase listener en el programa Saludo.

3. *Registre* su clase oyente. Más específicamente, esto significa agregar su clase oyente a un componente de ventana de texto al llamar al método `addActionListener`. Ésta es la sintaxis:

<*text-box-component*>`.addActionListener(new` <*listener-class*>`());`

Para ver un ejemplo, observe la llamada 3 en la figura 16.5*a*.

Lo importante del proceso de registro consiste en que su ventana de texto pueda encontrar un oyente cuando se activa un *introducir evento*. Un evento así se activa siempre que el usuario oprime Enter desde el interior de la ventana de texto.

Registrar un oyente es como registrar un automóvil. Cuando se registra un automóvil, nada ocurre en ese momento. Pero después, cuando ocurren algunos eventos, el registro del automóvil entra en juego. ¿Qué evento puede ocasionar el uso del registro del automóvil? Si el automóvil es multado por exceso de velocidad, la policía puede usar el registro del automóvil para hacer la multa. Si el automóvil es parte de un accidente, la compañía de seguros puede usar el registro del automóvil para aumentar la tasa de su seguro.

4. Importe el paquete `java.awt.event`. La manipulación de eventos requiere el uso de la interfaz `ActionListener` y de la clase `ActionEvent`. Estas entradas están en el paquete `java.awt.event`, de modo que para que la manipulación de eventos funcione es necesario importar el paquete. Para ver las declaraciones `import` dentro de un programa completo, observe la llamada 4 en la figura 16.5*a*.

La interfaz `ActionListener`

En el programa Saludo, `implements ActionListener` se implementó en el encabezado de la clase oyente. `ActionListener` es una *interfaz*. Quizás el lector recuerde las interfaces estudiadas en el capítulo 13. Una interfaz es algo así como una clase que contiene variables y métodos. Pero a diferencia de una clase, sus variables deben ser constantes (es decir, variables `final`) y sus métodos deben estar vacíos (es decir, encabezados de métodos). Si un programador usa una interfaz para derivar una clase nueva, el compilador requiere que la nueva clase implemente métodos para todos los encabezados del método de la interfaz.

Así, ¿de qué se trata el tener una interfaz con todos los métodos vacíos? La respuesta es que lo anterior puede usarse como una plantilla o patrón al crear una clase que caiga en cierta categoría. Más específicamente, ¿de qué se trata la interfaz `ActionListener`? Puesto que todos los oyentes de eventos de acción deben implementarla, significa que todos los eventos de acción son semejantes y, por tanto, comprensibles. Esto significa que todos los oyentes de eventos de acción implementarán un método de `ActionListener`: el método `ActionPerformed`. Y al implementar ese método, están obligados a usar este encabezado prescrito:

```
public void actionPerformed(ActionEvent e)
```

El uso del encabezado prescrito asegura que los eventos activados serán recibidos de manera idónea por el oyente.

16.10 Clases internas

A continuación se muestra una reimpresión del programa Saludo, en forma estructural:

```
public class Greeting extends JFrame
{
  ...
  private class Listener implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      String message; // el mensaje de saludo personalizado
      message = "Mucho gusto, " + nameBox.getText();
      nameBox.setText("");
      greeting.setText(message);
    } // end actionPerformed
  } // end class Listener
  ...
} // end class Greeting
```

¿Observa algo raro sobre la posición de la clase `Listener` en el programa Saludo? ¿Ve la sangría de la clase `Listener` y cómo su paréntesis de llave de cierre está antes del paréntesis de llave de cierre de la clase `Greeting`? ¡La clase `Listener` está dentro de la clase `Greeting`!

Si el alcance de una clase está limitado de modo que es necesario sólo por otra clase, se debe definir la clase como una *clase interna* (una clase que está dentro de otra clase). Puesto que un escucha normalmente está limitado a escuchar justo una clase, los oyentes usualmente se implementan como clases internas.

Aunque no es necesario para el compilador, las clases internas normalmente deben ser `private`. ¿Por qué? Porque la cuestión importante de usar una clase interna es proseguir el objetivo del encapsulamiento, y usar `private` significa que el mundo exterior no puede acceder a la clase interna. Observe el modificador `private` en el encabezado de la clase `Listener` de arriba.

Además de proseguir el objetivo del encapsulamiento, hay otra razón para usar una clase interna en contraposición a una clase de *nivel superior* (una clase de nivel superior es la expresión formal de una clase regular: una clase que no está definida dentro de otra clase). Una clase interna puede acceder directamente a sus variables de instancia de la clase que la contiene. Puesto que los oyentes normalmente requieren acceder a sus variables de instancia de la clase que la contiene, éste es un beneficio importante.

16.11 Clases internas anónimas

Observe el programa GreetingAnonymous en las figuras 16.6*a* y 16.6*b*. Es virtualmente igual al programa Saludo previo. ¿Puede identificar la diferencia entre el programa GreetingAnonymous y el programa Saludo?

En el programa Greeting se implementó una clase oyente denominada `Listener`, usando este código:

```
private class Listener implements ActionListener
{
```

Ese código se omitió en el programa GreetingAnonymous: no hay clase denominada `Listener`. Pero sigue siendo necesario un objeto oyente de modo que el evento de entrada en la ventana de texto sea aceptado y se actúe sobre él. Esta vez, en lugar de declarar una clase oyente con un nombre (por ejemplo, `Listener`), se implementa una clase oyente en forma anónima (sin nombre).

Los objetos anónimos ya se han analizado. Es cuando se instancia un objeto sin almacenar su referencia en una variable. En el programa Greeting previo, se instanció un objeto anónimo `Listener` con esta línea:

```
nameBox.addActionListener(new Listener());
```

La razón de usar un objeto anónimo es evitar que se enrede el código con el nombre de una variable cuando se requiere usar un objeto sólo una vez. La misma idea puede aplicarse a clases. La razón de usar

```
/*********************************************************
* GreetingAnonymous.java
* Dean & Dean
*
* Este programa demuestra una clase interna anónima.
*********************************************************/

import javax.swing.*;    // for JFrame, JLabel, JTextField
import java.awt.*;       // for FlowLayout
import java.awt.event.*; // for ActionListener, ActionEvent

public class GreetingAnonymous extends JFrame
{
  private static final int WIDTH = 325;
  private static final int HEIGHT = 100;
  private JTextField nameBox; // contiene el nombre del usuario
  private JLabel greeting;    // saludo personalizado

  //*******************************************************

  public GreetingAnonymous()
  {
    setTitle("Greeting Anonymous");
    setSize(WIDTH, HEIGHT);
    setLayout(new FlowLayout());
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents();
    setVisible(true);
  } // end constructor
```

Figura 16.6*a* Programa GreetingAnonymous que tiene una clase interna anónima, parte A.

una *clase interna anónima* es evitar enredar el código con el nombre de una clase que sólo es necesario usar una vez. Por ejemplo, si una clase oyente particular escucha sólo un objeto, entonces la clase oyente sólo requiere usarse una vez como parte de una llamada al método `addActionListener`. En consecuencia, para desenredar el código, conviene usar una clase oyente interna anónima para el oyente.

Usar una clase interna anónima no es un requisito del compilador. Es una cuestión de elegancia. En la industria, hay personas que afirman que las clases internas anónimas son elegantes y hay otras que sostienen que éstas son confusas. El lector es libre de decidir entre ambas posiciones. Mejor aún, haga como su profesor le aconseje.

A continuación se muestra la sintaxis para una clase interna anónima. Por supuesto, no hay nombre de clase. Pero hay un nombre de interfaz. De modo que las clases internas anónimas no surgen de la nada: se construyen con ayuda de una interfaz.[2] Observe el operador `new`. En términos formales, este operador forma parte de la clase interna anónima. Pero en términos prácticos, como no tiene caso contar con una clase interna anónima sin instanciarla, este operador `new` puede considerarse como si fuese parte de la sintaxis de la clase interna anónima.

```
new <interface-name> ()
{
 <class-body>
}
```

Éste es un ejemplo de una clase interna anónima, tomada del programa GreetingAnonymous:

[2] Como alternativa, es legal definir una clase anónima con una superclase en lugar de con una interfaz. Los detalles rebasan el alcance de este libro.

```
  //*********************************************************

  // Crea componentes y las agrega a la ventana.

  private void createContents()
  {
    JLabel namePrompt = new JLabel("¿Cómo te llamas?");
    nameBox = new JTextField(15);
    greeting = new JLabel();
    add(namePrompt);
    add(nameBox);
    add(greeting);
    nameBox.addActionListener(

      // anonymous inner class for event handling
      new ActionListener()
      {
        public void actionPerformed(ActionEvent e)
        {
          String message; // the personalized greeting
          message = "Mucho gusto, " + nameBox.getText();
          nameBox.setText("");
          greeting.setText(message);
        } // end actionPerformed
      } // end anonymous inner class
    ); // end addActionListener call
  } // end createContents

  //*********************************************************

  public static void main(String[] args)
  {
    new GreetingAnonymous();
  } // end main
} // end class GreetingAnonymous
```

Figura 16.6*b* Programa GreetingAnonymous que tiene una clase interna anónima, parte B.

```
nameBox.addActionListener(
  new ActionListener()
  {
    public void actionPerformed(ActionEvent e)
    {
      ...
    } // end actionPerformed
  } // end inner-class constructor
);
```

`ActionListener` es una interfaz

Con el propósito de comparar, a continuación se presenta un ejemplo de una clase interna (no anónima) nombrada. Se tomó del programa GreetingAnonymous:

```
private void createContents()
{
  ...
  nameBox.addActionListener(new Listener());
} // end createContents

private class Listener implements ActionListener
```

```
{
  public void actionPerformed(ActionEvent e)
  {
    ...
  } // end actionPerformed
} // end class Listener
```

Entre los dos fragmentos de código sólo hay dos diferencias sintácticas: la llamada a `addActionListener` y el encabezado de la clase oyente. Entre los dos fragmentos de código no hay diferencias semánticas, de modo que el programa Saludo y el programa GreetingAnonymous se comportan igual.

16.12 Componente `JButton`

Ya es hora de aprender otro componente GUI: un componente botón.

Interfaz del usuario

Si se oprime un botón en un aparato eléctrico, normalmente ocurre algo. Por ejemplo, si se oprime el botón de encendido de un televisor, éste se enciende o se apaga. En forma semejante, si un componente *botón* GUI se oprime o se hace clic en él, normalmente ocurre algo. Por ejemplo, en la ventana TrustyCredit en la figura 16.1, si se hace clic en el botón OK, entonces los números de tarjetas de crédito son procesados por la compañía TrustyCredit.

Implementación

Para crear un componente botón, el constructor *`JButton`* se llama así:

```
JButton helloButton = new JButton("Oprímeme");
```

Cuando se despliega este botón, indica “Oprímeme” en el centro del botón. El argumento de la etiqueta es opcional. Si se omite, la etiqueta obtiene la cadena vacía por defecto y el botón muestra una cara en blanco (sin escritura o iconos ahí).

Después que se crea el botón `helloButton`, debe agregarse a la ventana como se muestra a continuación:

```
add(helloButton);
```

Para que el botón sea útil, es necesario implementar el oyente. Así como con los oyentes de las ventanas de texto, los oyentes botón deben implementar la interfaz `ActionListener`. Esta interfaz determina que es necesario contar con un método para manipulación de eventos `actionPerformed`. La estructura del código es algo como esto:

```
private class Listener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    <do-something>
  }
}
```

Para la clase oyente se está usando `private` en lugar de `public` porque un oyente normalmente se implementa como una clase interna, y estas clases usualmente son `private`. Se está usando una clase interna nombrada en lugar de una clase interna anónima porque las clases internas nombradas son ligeramente más flexibles. Permiten crear un oyente que se utiliza en más de un componente. Se mostrará un ejemplo en un programa futuro.

Para registrar el oyente de arriba con el componente `helloButton`, se hace lo siguiente:

```
helloButton.addActionListener(new Listener());
```

La clase `JButton` necesita el paquete `javax.swing`, que ya debe estar disponible desde antes, porque es necesario para la clase `JFrame`. La interfaz `ActionListener` y la clase `ActionEvent` necesitan el paquete `java.awt.event`, de modo que es necesario importarlo.

Métodos

Éstos son los encabezados y las descripciones API para algunos de los métodos `JButton` más útiles:

`public String getText()`
Devuelve la etiqueta del botón.

`public void setText(String text)`
Asigna la etiqueta del botón.

`public void setVisible(boolean flag)`
Hace visible o invisible el botón.

`public void addActionListener(ActionListener listener)`
Agrega un oyente al botón. El oyente "escucha" cuando se hace clic en el botón.

Programa FactorialButton

Ya es hora de poner en práctica toda esta sintaxis `JButton` al mostrar cómo se usa dentro de un programa completo. Se ha escrito un programa FactorialButton que usa un componente `JButton` para calcular el factorial de un número introducido por el usuario.[3] Para dar una mejor idea de cómo funciona el programa, consulte la sesión de muestra en la figura 16.7.

Las figuras 16.8*a* y 16.8*b* contienen el listado del programa FactorialButton. La mayor parte del código ya debe tener sentido, puesto que la estructura del programa imita la estructura de los programas GUI previos. Se omitirá el código más conocido y la atención se dedicará al código más difícil.

La mayor parte de las variables GUI se declaran localmente dentro de `createContents`, aunque las dos ventanas de texto se declaran como variables de instancia en la parte superior del programa. ¿Por

[3] El factorial de un número es el producto de todos los enteros positivos menores o iguales que el número. El factorial de n se escribe como n! Ejemplo: el factorial de 4 se escribe como 4! y es igual a 24 porque $1 \times 2 \times 3 \times 4$ es igual a 24.

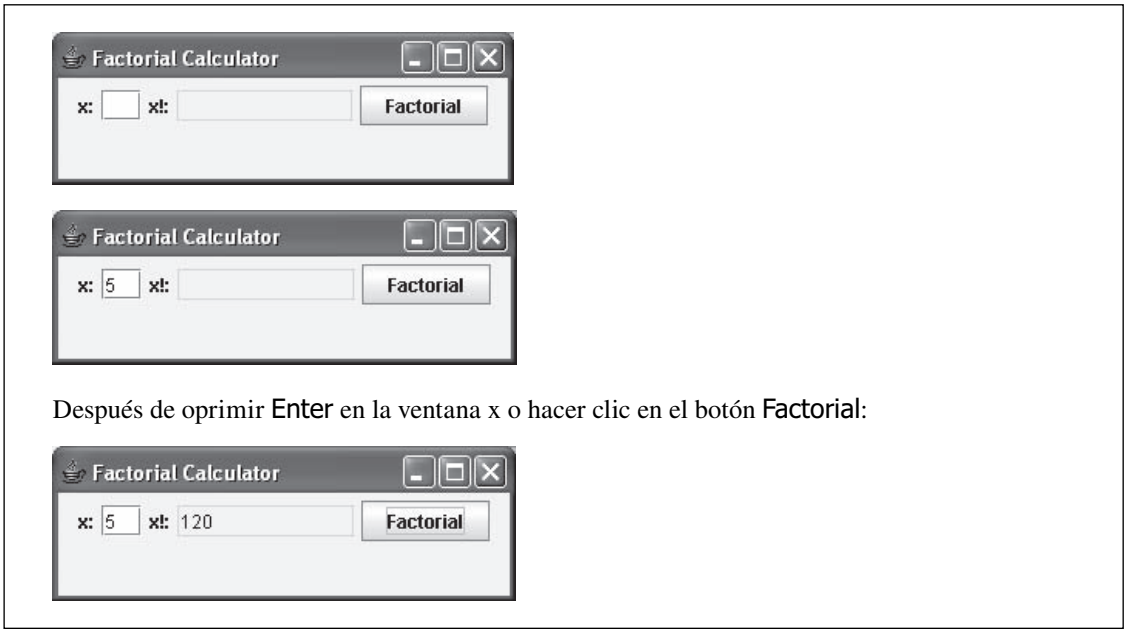


Figura 16.7 Sesión muestra para el programa FactorialJButton.

qué la diferencia? Como ya se analizó, normalmente es necesario crear componentes como variables locales a fin de ayudar con el encapsulamiento. Pero si es necesario un componente en `createContents`, y también en un manipulador de eventos, está bien declararla como una variable de instancia donde puede ser compartida de manera más fácil. En el programa FactorialButton, se declaran dos ventanas de texto como variables de instancia porque es necesario usarlas en `createContents` y también en el manipulador de eventos `actionPerformed`.

```
/*********************************************************
* FactorialButton.java
* Dean & Dean
*
* Cuando el usuario hace clic en el botón u oprime Enter en la ventana de texto de
* entrada, el factorial del número introducido aparece en la ventana de texto de salida.
*********************************************************/

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

public class FactorialButton extends JFrame
{
  private static final int WIDTH = 300;
  private static final int HEIGHT = 100;
  private JTextField xBox;  // contiene la entrada del usuario
  private JTextField xfBox; // contiene el factorial generado

  //*******************************************************

  public FactorialButton()
  {
    setTitle("Factorial Calculator");
    setSize(WIDTH, HEIGHT);
    setLayout(new FlowLayout());
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents();
    setVisible(true);
  } // end FactorialButton constructor

  //*******************************************************

  private void createContents()
  {
    JLabel xLabel = new JLabel("x:");
    JLabel xfLabel = new JLabel("x!:");
    JButton btn = new JButton("Factorial");
    Listener listener = new Listener();

    xBox = new JTextField(2);
    xfBox = new JTextField(10);
    xfBox.setEditable(false);
    add(xLabel);
    add(xBox);
    add(xfLabel);
    add(xfBox);
    add(btn);
    xBox.addActionListener(listener);
    btn.addActionListener(listener);
  } // end createContents
```

Aquí se registra el mismo oyente con dos componentes distintos.

Figura 16.8*a* Programa FactorialButton, parte A.

```
    //*************************************************

    // Clase interna para manipulación de eventos.

    private class Listener implements ActionListener
    {
      public void actionPerformed(ActionEvent e)
      {
        int x;         // valor numérico para x introducido por el usuario
        int xf;        // x factorial

        try
        {
          x = Integer.parseInt(xBox.getText());
        }
        catch (NumberFormatException nfe)
        {
          x = -1;   // indica una x inválida
        }

        if (x < 0)
        {
          xfBox.setText("undefined");
        }
        else
        {
          if (x == 0 || x == 1)
          {
            xf = 1;
          }
          else
          {
            xf = 1;
            for (int i=2; i<=x; i++)
            {
              xf *= i;
            }
          } // end else

          xfBox.setText(Integer.toString(xf));
        } // end else
      } // end actionPerformed
    } // end class Listener

    //*************************************************

    public static void main(String[] args)
    {
      new FactorialButton();
    } // end main
  } // end class FactorialButton
```

Convierte el número introducido por el usuario de una cadena en un número.

Figura 16.8*b* Programa FactorialButton, parte B.

Observe esta línea del método `createContents`:

```
xfBox.setEditable(false);
```

Esto hace que la ventana de texto, `xfBox`, no sea editable (es decir, el usuario no puede actualizar la ventana de texto). Esto debe tener sentido porque `xfBox` contiene el factorial, y depende del programa

(no del usuario) generar el factorial. Observe en la figura 16.7 que la ventana de texto factorial es de color gris. Esta característica visual se obtiene gratis siempre que se llama a `setEditable(false)` desde una componente de ventana de texto. ¡Perfecto!

De nuevo, a partir del método `createContents`:

```
Listener listener = new Listener();
...
xBox.addActionListener(listener);
btn.addActionListener(listener);
```

Observe que el mismo oyente está registrándose con dos componentes diferentes. Al hacer esto, se proporcionan al usuario dos formas de activar una respuesta. El usuario puede oprimir Enter cuando el cursor esté en la ventana de texto de entrada (`xBox`), o puede hacer clic en el botón (`btn`). Cualquiera de ambos casos provoca la reacción del oyente. Siempre que el mismo oyente se registra con dos componentes distintos, debe contarse con un nombre para el oyente. Ésta es la razón de que para este programa se use una clase interna nombrada (una clase interna anónima no funcionaría).

El método `actionPerformed` en la figura 16.8*b* está lleno de código interesante. De importancia primordial es la llamada al método`Integer.parseInt`. Si alguna vez es necesario leer o mostrar números desde la ventana de texto, primero se leen en una cadena, y luego la cadena se convierte en un número. Para lograr lo anterior, la cadena se lee usando `xBox.getText()`, y luego se convierte en un número usando `Integer.parseInt`.

Idealmente, siempre debe comprobarse la entrada del usuario para tener la certeza de que es válida. En el método `actionPerformed` se comprueban dos tipos de entrada inválida: una entrada no entera y la entrada de un número negativo. Estas entradas son inválidas porque el factorial no está definido matemáticamente para estos casos. El caso del número negativo es más fácil, por lo que se empezará en éste. Observe este código en la parte de en medio del método `actionPerformed`:

```
if (x < 0)
{
  xFbox.setText("undefined");
}
```

`x` es la entrada del usuario después que se ha convertido en un entero. Si `x` es negativo, el programa muestra `undefined` en el componente `xfBox`.

Ahora, para el caso en que se introduce un número no entero. Observe este código cerca de la parte superior del método `actionPerformed`:

```
try
{
  x = Integer.parseInt(xBox.getText());
}
catch (NumberFormatException nfe)
{
  x = -1;   // indicates an invalid x
}
```

El método `Integer.parseInt` trata de convertir el valor introducido por el usuario en `xBox` en un entero. Si el valor introducido en `xBox` no es un entero, el `parseInt` lanza una `NumberFormatException`. Para manejar esta posibilidad, la llamada al método `Integer.parseInt` se coloca dentro de un bloque `try`, y se incluye un bloque `catch` asociado. Si `parseInt` lanza una excepción, se quiere que en la componente `xBox` aparezca `undefined`. Para hacer esto, podría llamarse a `xBox.setText("undefined")` en el bloque `catch`, pero entonces se tendría un código redundante: `xfBox.setText("undefined")` en el bloque `catch` y también la declaración `if` ulterior. Para evitar redundancia de código y sus problemas de mantenimiento inherentes, se asigna -1 a `x` en el bloque `catch`. Esto provoca que la declaración `if` ulterior sea `true`, lo cual a su vez provoca que se llame a `xfBox.setText("undefined")`.

Después de validar la entrada, el método `actionPerformed` calcula el factorial. Primero se ocupa del caso especial cuando `x` es igual a 0 o 1. Luego se ocupa del caso $x \geq 2$ al usar un ciclo `for`. Funciona

Escriba un código compacto.

bien, pero ¿puede el lector ver una forma de hacerlo más reducido? Es posible omitir el bloque de código que empieza con `if (x == 0 || x == 1)` porque el caso es manipulado por el bloque `else`. Más específicamente, es posible eliminar las seis líneas que están arriba de la segunda línea `xf = 1;`.

16.13 Ventanas de diálogo y clase `JOptionPane`

Una *ventana de diálogo*, a menudo referida simplemente como *diálogo*, es un tipo especializado de ventana. La diferencia más importante entre una ventana de diálogo y una ventana estándar es que la primera está más restringida en términos de lo que puede hacer. Mientras una ventana estándar usualmente permanece en la pantalla del usuario bastante tiempo (a menudo mientras dura el programa) y ejecuta muchas tareas, una ventana de diálogo permanece en la pantalla sólo el tiempo suficiente para realizar una tarea específica. Mientras una ventana estándar puede ajustarse, una ventana de diálogo típicamente está asegurada en un formato particular.

Interfaz del usuario

Hay tres tipos de diálogos `JOptionPane`: un *diálogo de mensaje*, un *diálogo de entrada* y un *diálogo de confirmación*. Cada uno lleva a cabo una tarea específica. El diálogo de mensaje exhibe la salida. El diálogo de entrada muestra una pregunta y un campo de entrada. El diálogo de confirmación presenta una pregunta de sí/no y opciones de botón sí/no/cancelar. En la figura 16.9 se presentan estos tipos distintos de diálogos. En este capítulo, la atención se centra sólo en uno de los tres diálogos: el diálogo de mensaje. Si el lector desea aprender más sobre el diálogo de entrada, debe consultar el apartado GUI en el capítulo 3. Si desea aprende más sobre el diálogo de confirmación, debe consultar el sitio Sun API de Java en la red.

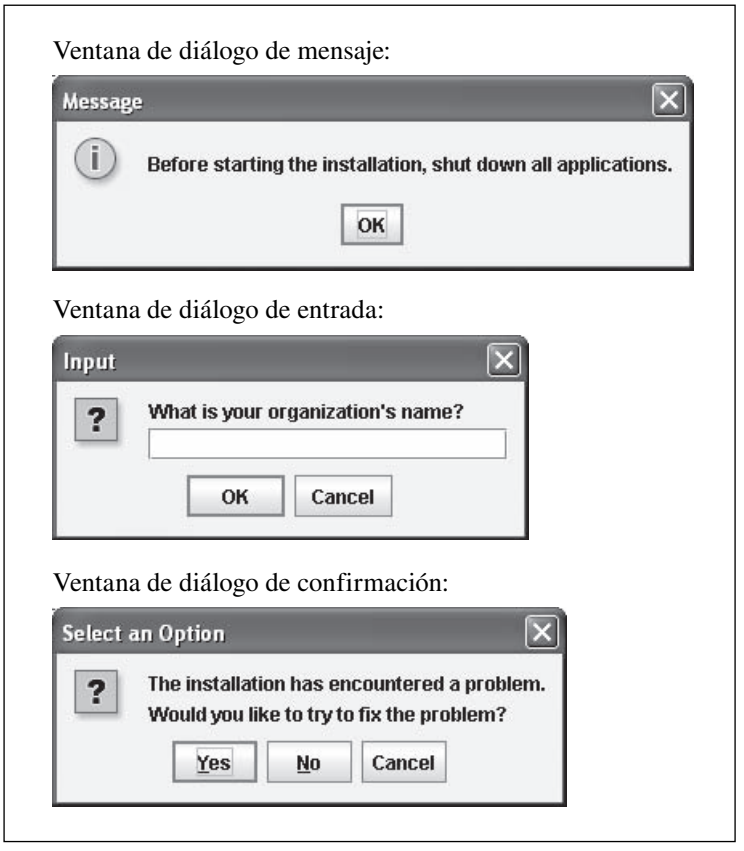


Figura 16.9 Tres tipos de ventanas de diálogo `JOptionPane`.

Implementación

Para crear una ventana de diálogo de mensaje, el método `showMessageDialog` se llama así:

```
JOptionPane.showMessageDialog(<container>, <message>);
```

Es necesario prefijar la llamada al `showMessageDialog` con "`JOptionPane` punto" porque `showMessageDialog` es un método de clase en la clase `JOptionPane`. Recuerde llamar a los métodos usando la sintaxis "<*reference-variable*> punto", y llamar a los métodos de clase usando la sintaxis "<*class-name*> punto".

En la llamada al `showMessageDialog` anterior, el argumento del mensaje especifica el texto que aparece en la ventana de diálogo. El argumento contenedor especifica el contenedor que rodea la ventana de diálogo. La ventana de diálogo aparece en el centro de ese contenedor.

Observe la llamada al `showMessageDialog` en el programa `HelloWithAFrame` en la figura 16.10. Para el argumento del contenedor `showMessageDialog` se usa `helloFrame`. ¿Qué tipo de contenedor es? Como puede verse a partir del código, `helloFrame` es una instancia de la clase `HelloWithAFrame`, y la clase `HelloWithAFrame` extiende el contenedor `JFrame`. En consecuencia, por herencia, `helloFrame` es un contenedor `JFrame`. Y en consecuencia, la ventana de diálogo aparece en el centro del contenedor `JFrame` del programa. Compruebe esto observando la salida de la figura 16.10.

Suponga que no quiere molestarse centrando la ventana de diálogo dentro de un contenedor particular. En ese caso, debe usarse el argumento `null` `showMessageDialog` del contenedor. Esto origina

```
import javax.swing.*;
public class HelloWithAFrame extends JFrame
{
  public HelloWithAFrame()
  {
    setTitle("Hello");
    setSize(400, 200);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
  } // end HelloWithAFrame constructor

  //*********************************************************

  public static void main(String[] args)
  {
    HelloWithAFrame helloFrame = new HelloWithAFrame();
    JOptionPane.showMessageDialog(helloFrame, "Hello, world!");
  } // end main
} // end class HelloWithAFrame
```

Figura 16.10 Programa HelloWithAFrame y su salida.

Constantes nombradas de JOptionPane (Para especificar un icono de la ventana de diálogo)	Icono	Cuándo usarlo
INFORMATION_MESSAGE		Para una ventana de diálogo que proporcione texto de información.
WARNING_ MESSAGE		Para una ventana de diálogo que advierta al usuario sobre un problema.
ERROR_MESSAGE		Para una ventana de diálogo que advierta al usuario sobre un error. Normalmente, un error se considera más grave que una advertencia.
QUESTION_MESSAGE		Para una ventana de diálogo que plantee una pregunta al usuario. Normalmente, el icono del signo de interrogaciones se usa para confirmar una ventana de diálogo de confirmación o una ventana de diálogo de entrada. Pero también es legal usarlo con una ventana de diálogo de mensaje.
PLAIN_MESSAGE	No hay icono	Para ventana de diálogo de texto llano. La ventana de diálogo contiene un mensaje, pero ningún icono.

Figura 16.11 Opciones de iconos con una ventana de diálogo `JOptionPane`.

que la ventana de diálogo aparezca en el centro de la pantalla. Por ejemplo, este código genera una ventana de diálogo centrada en la pantalla:

```
JOptionPane.showMessageDialog(
  null, "Antes de iniciar la instalación,\n" +
  "cierre todas las aplicaciones.");
```

Por cierto, es bastante común usar `null` para `showMessageDialog` como argumento del contenedor, tal vez más común que usar un valor `non-null`.

La clase `JOptionPane` necesita el paquete `javax.swing`. Si este paquete ya se ha importado para la clase `JFrame` no es necesario importarlo de nuevo.

Detalles del método

En la figura 16.10, observe el mensaje en la barra de título de la ventana de diálogo: es un "Message". ¡Qué aburrido!, ¿no? Para animar las cosas, debe agregarse un tercer argumento a la llamada a `showMessageDialog` que especifique el título de la ventana de diálogo. En la misma figura, observe también el icono de la ventana de diálogo: es una **i** dentro de un círculo. Éste es el icono por defecto. Para especificar de manera explícita un icono, debe agregarse un cuarto argumento a la llamada a `showMessageDialog` que especifique una de las constantes nombradas en la figura 16.11.

He aquí cómo llamar la versión con el cuarto parámetro de `showMessageDialog`:

```
JOptionPane.showMessageDialog(
  <null-or-container>, <message>, <title>, <icon_constant>);
```

He aquí un ejemplo de la llamada a `showMessageDialog` versión con el cuarto parámetro, así como la ventana de diálogo resultante:

```
JOptionPane.showMessageDialog(null, "A virus has been detected.", "Warning",
  JOptionPane.WARNING _ MESSAGE);
```

16.14 Distinción entre múltiples eventos

Ahora que el lector ha comprendido las piedras angulares de la programación GUI (los componentes `JLabel`, `JTextField` y `JButton` de las ventanas `JFrame` y `JOptionPane`), ya está preparado para considerar situaciones más complicadas que encuentran los programadores de GUI. En esta sección se aprenderá a usar un simple oyente para distinguir entre dos eventos componentes distintos.

El método `getSource`

Suponga que se registra un oyente con dos componentes. Cuando el oyente escucha un evento, probablemente sea conveniente determinar cuál componente activó el evento. Así es posible ajustar la manipulación de eventos: haga una cosa si el componente X activó el evento y haga otra si el componente Y activó el evento.

Desde el interior de un oyente, ¿puede determinar la fuente de un evento? En otras palabras, ¿cómo es posible identificar el componente que activó un evento? ¡Llamando a `getSource`, por supuesto! Más específicamente, dentro del método `actionPerformed`, use el parámetro `ActionEvent` del método `actionPerformed` para llamar a `getSource`. El método `getSource` devuelve una referencia al componente cuyo evento fue activado. Para ver cuál componente fue, use == para comparar el valor devuelto con el componente en cuestión. Por ejemplo, en el siguiente fragmento de código se compara el valor devuelto con un componente botón denominado `okButton`.

```
public void actionPerformed(ActionEvent e)
{
  if (e.getSource() == okButton)
  {
    ...
```

Programa FactorialButton mejorado

¿Recuerda el programa FactorialButton en la figura 16.8? Calculaba el factorial de un número introducido por el usuario. Los cálculos eran activados por el usuario al oprimir el botón factorial o al presionar Enter en la ventana de diálogo de texto. Con el simple programa FactorialButton de primer corte, no es necesario molestarse en distinguir entre el evento de hacer clic en el botón y el evento de oprimir Enter. A continuación se mejorará el programa al hacer que eventos distintos activen resultados diferentes. Al hacer clic en el botón sigue apareciendo el factorial, aunque la ventana de texto muestra la siguiente ventana de diálogo:

Vea la figura 16.12. Muestra la clase `Listener` para el nuevo programa FactorialButton mejorado. Sólo se muestra la clase `Listener` porque el resto del programa no ha cambiado. Si se quiere ver el

```
private class Listener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    int x;   // valor numérico para x introducido por el usuario
    int xf;  // x factorial

    if (e.getSource() == xBox)
    {
      JOptionPane.showMessageDialog(null,
        "Haga clic en el botón factorial para realizar un cálculo.");
    }

    else // debe haber hecho clic en el botón
    {
      try
      {
        x = Integer.parseInt(xBox.getText());
      }
      catch (NumberFormatException nfe)
      {
        x = -1;   // indica una x inválida
      }

      if (x < 0)
      {
        xfBox.setText("undefined");
      }
      else
      {
        if (x == 0 || x == 1)
        {
          xf = 1;
        }
        else
        {
          xf = 1;
          for (int i=2; i<=x; i++)
          {
            xf *= i;
          }
        } // end else

        xfBox.setText(Integer.toString(xf));
      } // end else
    } // end else button was clicked
  } // end actionPerformed
} // end class Listener
```

Ésta es la ventana de texto de entrada.

Figura 16.12 Clase `Listerner` modificada para el programa FactorialButton.

resto del programa, consulte la figura 16.8. En la nueva clase `Listener`, observe la forma en que se llama a `getSource` y compare su valor devuelto con `xBox`. `xBox` es el componente de la ventana de texto que contiene la entrada del usuario para `x`. Si `getSource` devuelve `xBox`, se llama a `showMessageDialog` y aparece la ventana de mensaje de diálogo anterior.

16.15 Utilización de `getActionCommand` para distinción entre múltiples eventos

En esta sección se continúa el análisis de distinción entre múltiples eventos. Pero en lugar de llamar a `getSource`, esta vez se llamará a `getActionCommand`.

`getSource` es algo limitado

En la clase `Listener` de la figura 16.12 se llamó a `getSource` para identificar la componente cuyo evento fue activado. Esto funciona bien la mayor parte del tiempo, pero no siempre. Observe los casos siguientes en los que llamar a `getSource` es inadecuado:

1. Si los componentes que activan el evento están en una clase distinta a la clase `Listener`.

   El método `getSource` de la clase `Listener` puede recuperar exitosamente el componente responsable del evento activado, aunque no hay forma de identificar el tipo del componente devuelto porque para eso se requiere comparar el componente devuelto con los componentes originales (usando ==). Si los componentes originales están en una clase diferente y son `private`, entonces al usarlos en la clase `Listener` generan un error de tiempo de compilación.

2. Si hay necesidad de contar con un componente *modal*.

   Un componente modal es un componente que tiene más de un estado o estados. Por ejemplo, suponga que hay un botón cuya etiqueta se activa entre "Mostrar detalles" y "Ocultar detalles". Estas dos etiquetas corresponden a dos modos de operación distintos: en un modo se muestran los detalles, y en otro se ocultan. Si se hace clic en un botón modal, `getSource` puede recuperar el botón, pero no puede determinar directamente si el modo del botón es para mostrar u ocultar los detalles.

Comando `getAction` al rescate

Si se requiere identificar un evento desde un oyente y `getSource` es inadecuado, debe intentarse `getActionCommand`. El método `getActionCommand` devuelve el "comando acción" asociado con el componente cuyo evento fue activado. Típicamente, el comando acción es la etiqueta del componente. Por ejemplo, el comando acción por defecto para un botón es la etiqueta del botón.

A continuación volverá a revisarse el caso en que la etiqueta de un botón se activa entre "Mostrar detalles" y "Ocultar detalles". En el siguiente fragmento de código, suponga que `instructions` es un componente etiqueta, `detailedInstructions` y `briefInstructions` son variables locales de cadena, y `btn` es el botón "Mostrar detalles/Ocultar detalles". Observe cómo `getActionCommand` determina el modo del botón al recuperar la etiqueta del botón.

```
public void actionPerformed(ActionEvent e)
{
  if (e.getActionCommand().equals("Show Details"))
  {
    instructions.setText(detailedInstructions);
    btn.setText("HideDetails");
  }
  else
  {
    instructions.setText(briefInstructions);
    btn.setText("ShowDetails");
  }
} // end actionPerformed
```

16.16 Color

Hasta ahora en este capítulo, todos los componentes han sido simples en términos de color: texto negro sobre fondo blanco o texto negro sobre fondo ligeramente gris. Ya es hora de agregar algo de color. El

lector debe acostumbrarse al color en la mayor parte de las aplicaciones GUI. Después de todo, el color puede mejorar la experiencia del usuario con un programa al proporcionar elementos y atractivos visuales. ¡Recuerde que el color es divertido!

Métodos `Color`

La mayor parte de los componentes GUI están compuestos por dos colores. El *color de primer plano* es el color del texto, y el *color de fondo* es el color del área detrás del texto. A continuación se mostrará directamente un ejemplo que muestra cómo establecer los colores. Este fragmento de código crea un botón azul con texto blanco:

```
JButton btn = new JButton("Click Me");
btn.setBackground(Color.BLUE);
btn.setForeground(Color.WHITE);
```

Y así se ve más o menos el botón blanco-azul:

Los métodos `setBackground` y `setForeground` son métodos reguladores (*mutators*). Éstos son los encabezados y las descripciones API para sus métodos de acceso asociados:

`public Color getBackground()`
Devuelve el color de fondo del componente.

`public Color getForeground()`
Devuelve el color del primer plano del componente.

Éste es un ejemplo en el que se usan los métodos `getBackground` y `getForeground` con una ventana de texto:

```
JTextField nameBox = new JTextField();
Color originalBackground = nameBox.getBackground();
Color originalForeground = nameBox.getForeground();
```

¿Por qué querrían guardarse los colores originales de una ventana de texto? Como elemento visual, quizás es conveniente volver a los colores originales cuando el usuario introduce algo inválido. Y una vez que el usuario fija la entrada, se regresa a los colores originales. Para hacer lo anterior, es necesario recuperar y guardar los colores originales la primera vez que se carga la ventana.

Ahora ya se han visto ejemplos de color con un botón y una ventana de texto. El color funciona de la misma forma para la mayor parte de los otros componentes. Una excepción es el componente `JLabel`. Por defecto, su fondo es transparente, de modo que si se desea aplicarle color, éste no se ve. Para cambiar el color de una etiqueta, primero es necesario hacerla opaca al llamar a `label.setOpaque(true)`. Después de eso, si se llama a `setBackground` (<*color*>), se verá el color especificado.

Constantes `Color` nombradas

A continuación se hablará sobre los valores de los colores. Los valores de los colores pueden especificarse con constantes nombradas o con objetos `Color` instanciados. Se empieza con las constantes nombradas.

La clase `Color` define este conjunto de constantes nombradas:

```
Color.BLACK              Color.GREEN        Color.RED
Color.BLUE               Color.LIGHT _ GRAY Color.WHITE
Color.CYAN               Color.MAGENTA      Color.YELLOW
Color.DARK _ GRAY        Color.ORANGE
Color.GRAY               Color.PINK
```

Como es costumbre, las constantes nombradas son miembros de clase. Como ocurre con todos los miembros de clase, para acceder a ellos se usa la sintaxis <*class name*>. En otras palabras, a ellos se accede mediante un prefijo "`Color`".

La clase `Color` está en el paquete `java.awt`, de modo que no olvide importar ese paquete cuando trabaje con colores.

Objetos `Color`

Para obtener un color que no esté en la clase `Color` de la lista de constantes nombradas, es necesario instanciar un objeto `Color` con una mezcla especificada de rojo, verde y azul. Ésta es la sintaxis para llamar al constructor `Color`:

`new Color(`*<red 0–255>*`,` *<green 0–255>*`,` *<blue 0–255>*`)`

Cada uno de los tres argumentos del constructor `Color` es un valor `int` entre 0 y 255. El valor `int` representa una cantidad de color, donde 0 indica ningún color y 255 indica la cantidad máxima de color. Por ejemplo, la siguiente línea establece el color de fondo de un botón en magenta oscuro:

```
button.setBackground(new Color(128, 0, 128));
```

El objeto `Color` instanciado usa la mitad del máximo para rojo (128), nada de verde (0) y la mitad del máximo para azul (128). En el magenta más brillante, los valores del rojo y del azul aumentan de 128 a 255.

La luz blanca es una combinación de todos los colores,[4] de modo que `new Color(255, 255, 255)` produce blanco. El negro es la ausencia de color, de modo que `new Color(0, 0, 0)` produce negro.

Esta técnica para crear color al mezclar cantidades especificadas de rojo, verde y azul la utilizan muchos lenguajes de programación. La tripleta rojo, verde, azul suele denominarse *valor RGB*. Cuando use valores RGB para sus programas, es perfectamente aceptable hacerlo al tanteo, pero para ahorrar tiempo quizá sea conveniente consultar una tabla de color RGB en línea. Por ejemplo, http://www.pitt.edu/~nisg/cis/web/cgi/rgb.html

Color de fondo `JFrame`

Establecer el color de fondo para una ventana `JFrame` es ligeramente más engañoso que hacerlo para un componente. Primero debe contar con el panel de contenido (*content pane*) de `JFrame`, luego debe aplicarle el color de fondo. Como se muestra a continuación, el panel de contenido está en la parte interna de `JFrame`.

Mientras la clase `JFrame` manipula características del perímetro como las dimensiones de la ventana, la barra de título y el botón de apertura y cierre, el panel de contenido manipula características internas como los componentes, la disposición y el color de fondo. Así, cuando se agregan componentes, se establece la disposición y se establece el color de fondo, se le realiza al panel de contenido, no a `JFrame`. Estas tres declaraciones ilustran lo anterior:

```
getContentPane().add(btn);
getContentPane().setLayout(new FlowLayout());
getContentPane().setBackground(Color.YELLOW);
```

En versiones de Java anteriores a Java 5.0, se requería el método `getContentPane` para las tres tareas: agregar un componente, establecer la disposición y establecer el color de fondo de la ventana. Con la llegada de Java 5.0, el personal en Sun facilitó las cosas para las dos primeras tareas. Ahora, si se desea

[4] En 1666, Isaac Newton descubrió que la luz blanca está compuesta por todos los colores del espectro cromático. Demostró que cuando la luz blanca pasa a través de un prisma triangular, se descompone en varios colores; y que cuando estos colores pasan por un segundo prisma triangular, vuelven a integrarse en el haz de luz blanca original.

agregar un componente o establecer la disposición, en forma opcional puede omitirse la llamada a `getContentPane`. En otras palabras, esto funciona así:

```
add(btn);
setLayout(new FlowLayout());
```

La razón de que funcione este código es que con la versión actual de Java, los métodos `add` y `setLayout` de `JFrame` obtienen automáticamente el panel de contenido tras bambalinas. Y el panel de contenido rescrito se usa para las operaciones `add` y `setLayout` subsiguientes. Así, ¿qué es mejor: `add(btn)` o `getContentPane().add(btn)`? Desde un punto de vista funcional, son equivalentes, aunque el primero suele preferirse porque es menos revuelto. Lo mismo para la llamada al método `setLayout`.

Para establecer el color de fondo de la ventana, la versión actual de Java sigue requiriendo la llamada a `getContentPane` antes de llamar a `setBackground`. Si `setBackground` se llama sin llamar a `getContentPane`, establece el color de fondo de `JFrame`, no el color de fondo del panel de contenido. Y puesto que el panel de contenido se encuentra en la parte superior de `JFrame`, el color de `JFrame` aparece cubierto, por lo que no es visible.

Ahora se sabe que para establecer el color de fondo de una ventana se requiere `getContentPane`. En forma semejante, para obtener un color de fondo de la ventana se requiere `getContentPane`. Por ejemplo:

```
Color saveColor = getContentPane().getBackground();
```

Programa ColorChooser

A continuación se pondrá en práctica lo que se ha aprendido sobre el color al utilizarlo dentro de un programa completo. En el programa ColorChooser se implementan los botones gris claro y azul claro, que establecen el color de fondo de la ventana en gris o azul, respectivamente. Observe la figura 16.13 para tener una idea de lo que se está hablando.

Observe el listado del programa ColorChooser en las figuras 16.14*a* y 16.14*b*. La mayor parte del código debe tener sentido puesto que su estructura es una imagen especular de la estructura en los programas GUI previos.

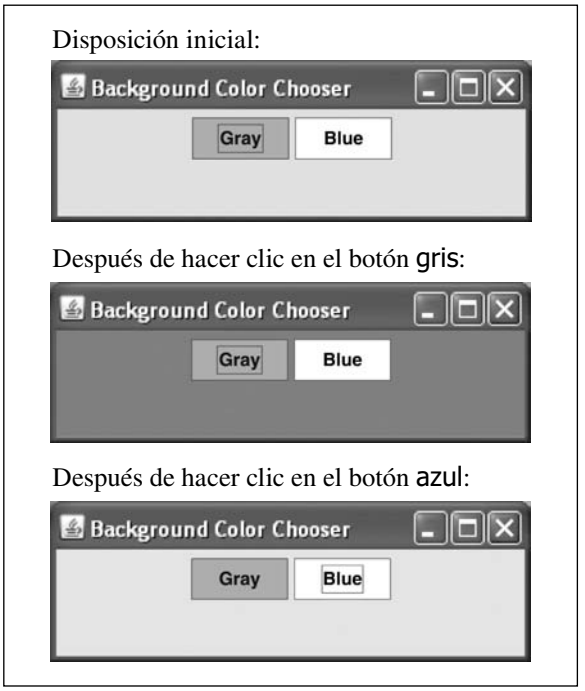


Figura 16.13 Sesión muestra para el programa ColorChooser.

```
/*********************************************************
* ColorChooser.java
* Dean & Dean
*
* Estos botones del programa permiten al usuario establecer
* el color de fondo de la ventana en gris o azul.
*********************************************************/

import javax.swing.*;      // para JFrame & JButton
import java.awt.*;         // para FlowLayout, Color, & Container
import java.awt.event.*;   // para ActionListener & ActionEvent

public class ColorChooser extends JFrame
{
  private static final int WIDTH = 300;
  private static final int HEIGHT = 100;

  private JButton grayButton;  // cambia el fondo a gris
  private JButton blueButton;  // cambia el fondo a azul
```

Figura 16.14*a* Programa ColorChooser, parte A.

Observe la diferencia entre las llamadas a `setBackground` en el método `createContents` y las llamadas a `setBackground` en el método `actionPerformed`. En `createContents`, se está tratando con los componentes de los botones gris y azul, de modo que no es necesario llamar a `getContentPane` antes de llamar a `setBackground`. En `actionPerformed` se está tratando con la ventana `JFrame`, por lo que es necesario llamar a `ContentPane` antes de llamar a `setBackground`.

Observe la línea siguiente del método `createContents`. Establece el color del botón azul en azul claro:

```
blueButton.setBackground(new Color(135, 206, 250));
```

No hay constante nombrada para el azul claro, de modo que es necesario instanciar el objeto `Color` azul claro usando un valor RGB. Se usa casi la cantidad máxima de azul (250), así como una cantidad sustancial de rojo (135) y verde (206). ¿Sabe el lector por qué se usa tanto rojo y verde? Para obtener una luz de sombra, se requiere una cantidad sustancial de los tres valores de color. Esto debe tener sentido una vez que se percata de que el blanco es `Color(255, 255, 255)`.

16.17 ¿Cómo se agrupan las clases GUI?

A lo largo de este capítulo se han usado las clases preconstruidas GUI de la biblioteca Sun de API. Por ejemplo, la clase `JFrame` se usó para crear una ventana, la clase `JButton`, para crear un botón, y la clase `Color`, para crear un color. En esta sección se describe cómo se agrupan y organizan las clases GUI.

Subpaquetes

API de Java es una gran biblioteca de clases que agrega funcionalidad al lenguaje Java. Para simplificar las cosas, las clases están organizadas en una jerarquía de paquetes, donde cada paquete contiene un grupo de clases. Para evitar tener demasiadas clases en un paquete, a menudo los paquetes se separan en *subpaquetes*. Un subpaquete es un grupo de clases provenientes de un grupo más grande de clases. Por ejemplo, en lugar de colocar todas las clases GUI (¡y hay bastantes!) en el paquete `java.awt`, el perso-

```
//*********************************************************

public ColorChooser()
{
  setTitle("Background Color Chooser");
  setSize(WIDTH, HEIGHT);
  setLayout(new FlowLayout());
  setDefaultCloseOperation(EXIT _ ON _ CLOSE);
  createContents();
  setVisible(true);
} // end ColorChooser constructor

//*********************************************************

private void createContents()
{
  grayButton = new JButton("Gray");
  grayButton.setBackground(Color.LIGHT _ GRAY);
  grayButton.addActionListener(new ButtonListener());
  add(grayButton);

  blueButton = new JButton("Blue");
  blueButton.setBackground(new Color(135,206,250));
  blueButton.addActionListener(new ButtonListener());
  add(blueButton);
} // end createContents

//*************************************************************

// Clase interna para manipulación de eventos.

private class ButtonListener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    Container contentPane = getContentPane();
    if (e.getSource() == grayButton)
    {
      // Cambia el color de fondo de la ventana a gris.
      contentPane.setBackground(Color.GRAY);
    }
    else
    {
      // Cambia el color de fondo de la ventana a azul.
      contentPane.setBackground(Color.BLUE);
    }
  } // end actionPerformed
} // end class ButtonListener
```

Esto establece el botón de color gris.

Esto establece el botón de color azul.

Estas líneas cambian el color de fondo de la ventana.

Figura 16.14*b* Programa ColorChooser, parte B.

nal de Sun dividió las clases de manipulación de eventos y colocó ahí su propio paquete, `java.awt.event`. Para importar todas las clases en el paquete `java.awt`, y en el subpaquete `java.awt.event`, se hace lo siguiente:

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

```
    //*************************************************************

    public static void main(String[] args)
    {
      new ColorChooser();
    }
  } // end class ColorChooser
```

Figura 16.14*c* Programa ColorChooser, parte C.

Puesto que el subpaquete `java.awt.event` contiene `java.awt` en su nombre, ¿es correcto omitir la declaración `import java.awt.event` y simplemente hacer lo siguiente?

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

Esto importa clases que están sólo en el paquete `java.awt.`

No, `java.awt` y `java.awt.event` deben importarse por separado. Considere que el paquete `java.awt` y el subpaquete `java.awt.event` son entes completamente ajenos. El hecho de que compartan el nombre común *"java.awt.event"* es irrelevante en cuanto al compilador toca. El compilador los trata como paquetes por separado. Entonces, ¿por qué comparten el nombre? El nombre compartido ayuda a que el programador recuerde que las clases en `java.awt.event` están relacionadas conceptualmente con las clases en `java.awt`.

Aquí se está haciendo referencia a `java.awt.event` como un "subpaquete". Pero es muy común llamarlo "subpaquete" o "paquete". Puesto que ambos términos son válidos, se usarán los dos.

Las bibliotecas AWT y Swing

En el primer compilador Sun de Java, todas las clases GUI estaban ubicadas en una biblioteca conocida como Abstract Windowing Toolkit /AWT). Los comandos GUI de AWT generaban componentes GUI que parecían diferentes en plataformas distintas. En otras palabras, si su programa instancia un componente de botón de AWT, el botón sigue teniendo un toque de Macintosh y siente si el programa se ejecuta en una computadora Macintosh, pero también tiene un "*look and feel*" de Windows si el programa se ejecuta en una computadora Windows.[5] Lo anterior lleva a cuestiones de portabilidad. Sus programas siguen siendo portátiles en el sentido de que pueden ejecutarse en plataformas distintas. Pero se ejecutan de manera distinta en plataformas diferentes. Si usted tiene un cliente fastidioso que solicita una apariencia precisa en todas las plataformas, entonces probablemente los componentes de AWT no sean satisfactorios.

Uno de los atractivos de venta más fuertes de Java era (y es) su portabilidad, de modo que poco después del lanzamiento inicial de Java, el personal de Sun se dedicó a desarrollar un conjunto más portátil de componentes GUI. Colocaron sus nuevos componentes más portátiles en una biblioteca totalmente nueva denominada Swing. Para hacer clara la relación entre los nuevos componentes de Swing y los componentes de AWT, usaron los mismos nombres de componentes, excepto que antepusieron los nuevos componentes de Swing con una "J". Por ejemplo, la AWT tiene un componente `Button`, de modo que Swing tiene un componente `JButton`.

Los componentes GUI de AWT se conocen como *componentes de peso pesado,* mientras los componentes GUI se conocen como *componentes de peso ligero.* Los componentes GUI son de peso pesado porque están construidos con comandos gráficos que forman parte de la plataforma de la computadora. Al ser parte de la plataforma de la computadora, son demasiado "pesados" para moverse a otras plataformas. Los componentes GUI son de peso ligero porque están construidos con código de Java. Esto significa que son suficientemente "ligeros" para moverse a diversas plataformas.

La biblioteca Swing incluye bastante más que las clases componentes GUI. Añade bastante funcionalidad a la AWT, aunque sin sustituir a la AWT por completo. Actualmente, los programadores de apli-

[5] *Look and feel* es un término GUI estándar, y se refiere a la apariencia de alguna cosa y a la forma en que el usuario interactúa con ésta.

caciones GI de Java usan ambas bibliotecas: la AWT y la Swing.[6] Los paquetes primarios AWT son `java.awt` y `java.awt.event`. El paquete primario Swing es `javax.swing`. La "x" en `javax` significa "extensión" porque los paquetes `javax` (`javax.swing` es uno de varios paquetes `javax`) se consideran como las extensiones más importantes de la plataforma Java.

16.18 Oyentes de ratón y de imágenes (opcional)

Sun proporciona varios tipos distintos de oyentes. Ya antes en este capítulo se aprendió sobre el oyente más común: el `ActionListener`. El oyente `ActionListener` debe usarse para eventos en los que el usuario modifica algún componente al hacer clic en un botón o al oprimir Enter dentro de una ventana de texto. En esta sección se aprenderá sobre oyentes de ratón. Como lo dice su nombre, estos oyentes deben usarse para eventos en los que el usuario hace algo con el ratón. También en esta sección se aprenderá sobre imágenes. Se aprenderá a mostrar una imagen y arrastrarla con el ratón.

Oyentes de ratón

Al crear los oyentes de ratón, se usan los mismos pasos básicos que se usan para `ActionListener`: se define una clase `Listener`, dentro de la clase oyente se definen uno o varios métodos para manipulación de eventos y la clase oyente se registra con un componente. Aunque se usan los mismos pasos básicos, los oyentes de ratón son algo más complicados que el `ActionListener`. Hay varios tipos de oyentes de ratón, y cada tipo manipula múltiples tipos de eventos de ratón.

A continuación se describen dos tipos de oyentes de ratón, definidos por sus dos interfaces: `MouseListener` y `MouseMotionListener`. En la figura 16.15 se muestran los encabezados y las descripciones API para los métodos definidos por las dos interfaces. Lea los encabezados y las descripciones API para tener una idea de lo que es posible en términos de la manipulación de eventos con ratón.

Como programador, no tiene que preocuparse de llamar a los métodos de manipulación de eventos con ratón. Éstos son llamados automáticamente una vez que ocurren sus eventos relacionados con el ratón. Por ejemplo, si el usuario oprime el botón del ratón mientras el cursor del ratón está en un componente registrado como `MouseListener`, la JVM llama automáticamente al manipulador de eventos `mousePressed`.

En el programa que se presenta a continuación, el objetivo es permitir que el usuario arrastre una imagen por la ventana usando el ratón. Para hacer esto, es necesario detectar que el ratón se ha oprimido y movido (es decir, arrastrado) mientras el cursor del ratón está en la imagen. Y para hacer esto, es necesario registrar un oyente de ratón. El lector ya conoce algunos componentes, pero ninguna imagen. Así, ¿cuál es la solución? El lector ya está familiarizado con algunos componentes: los componentes `JLabel`, `JTextField` y `JButton`. Estas clases son clases de componentes porque son descendientes de la clase `Component`. Constituyen otra clase de componentes algo distintos. No sienten como un componente en el sentido normal de la palabra, aunque a pesar de ello siguen siendo componentes de Java, y funcionan bien para manipular eventos con ratón. Entonces, ¿cuál es el componente misterioso? ¡`JPanel`!

Considere un objeto `JPanel` como una gran área de almacenamiento para otros objetos. Más formalmente, la clase `JPanel` es una descendiente de la clase `Container`, y como tal, se trata de un contenedor al cual es posible agregar objetos. En el siguiente capítulo, se agregarán componentes de Swing (`JLabel`, `JTextField`, etc.) a contenedores `JPanel`. En el siguiente programa ejemplo, se agrega un objeto de imagen a un contenedor `JPanel`. Al rodear la imagen con un contenedor `JPanel`, se cuenta con una plataforma a la que es posible conectar los oyentes de ratón. En el siguiente programa ejemplo, los oyentes `JPanel` permiten detectar eventos de ratón en el objeto de imagen.

[6] Los programadores de *applet* de Java suelen usar sólo la AWT, inclusive para los componentes GUI, y no usan en absoluto la biblioteca Swing. ¿Por qué? Porque los *applets* dependen de los buscadores y, lamentablemente, muchos de los buscadores actuales siguen utilizando versiones viejas de Java, versiones que no incluyen a Swing.

Manipuladores de eventos de la interfaz `MouseListener`
`public void mouseClicked(MouseEvent event)` Se llama cuando el botón del ratón se oprime y se suelta mientras el cursor del ratón permanece estacionario en un componente registrado como `MouseListener`.
`public void mouseEntered(MouseEvent event)` Se llama cuando el cursor del ratón entra a los límites de una componente registrada como `MouseListener`.
`public void mouseExited(MouseEvent event)` Se llama cuando el cursor del ratón sale de los límites de una componente registrada como `MouseListener`.
`public void mousePressed(MouseEvent event)` Se llama cuando el usuario oprime el botón del ratón mientras el cursor del ratón está en una componente registrada como `MouseListener`.
`public void mouseReleased(MouseEvent event)` Se llama cuando el usuario suelta el botón del ratón, pero sólo si la vez anterior que se oprimió ese botón fue sobre una componente registrada como `MouseListener`.
Manipuladores de eventos de la interfaz `MouseMotionListener`
`public void mouseDragged(MouseEvent event)` Se llama cuando el usuario mantiene oprimido el botón del ratón al mover el cursor del ratón, pero sólo si la primera vez que se oprimió el botón del ratón fue sobre un componente registrado como `MouseMotionListener`.
`public void mouseMoved(MouseEvent event)` Se llama cuando el usuario mueve el ratón mientras el cursor del ratón está en un componente registrada como `MouseMotionListener`.

Figura 16.15 Encabezados y descripciones API para los métodos en las interfaces `MouseListener` y `MouseMotionListener`.

El programa DragSmiley

Observe la figura 16.16. Contiene una clase de activación y una sesión muestra para un programa `DragSmiley`. Como se indica en la sesión muestra, el programa muestra inicialmente una carita sonriente en la parte superior izquierda de la ventana del programa. Si el usuario oprime el botón del ratón, la carita sonriente cambia a una carita con miedo (tal vez porque la carita sonriente tiene miedo de lo que pueda hacer el usuario). Cuando el usuario suelta el botón del ratón, la carita con miedo cambia de vuelta a la carita sonriente. Si el cursor del ratón está sobre la imagen y el usuario arrastra el ratón, la imagen sigue el cursor del ratón.

Estudie el constructor `DragSmiley` en la figura 16.16. En ella, las dos declaraciones siguientes instancian un contenedor `JPanel` denominado `smileyPanel` y agrega el contenedor `JPanel` a la ventana `DragSmiley`.

```
smileyPanel = new SmileyPanel();
add(smileyPanel);
```

Observe la clase `SmileyPanel` en las figuras 16.16*a* y 16.16*b*. La clase `SmileyPanel` es donde se encuentra el grueso de la lógica del programa. La clase `SmileyPanel` se describirá primero centrando la atención en los oyentes. Observe cómo el constructor `SmileyPanel` crea los oyentes de ratón y los

```
/*******************************************************
* DragSmiley.java
* Dean & Dean
*
* Este programa muestra la imagen de una carita sonriente.
* Cuando el usuario oprime el ratón, la imagen cambia a una carita
* con miedo. El usuario puede arrastrar la imagen.
*******************************************************/

import javax.swing.*;

public class DragSmiley extends JFrame
{
  private static final int WIDTH = 250;
  private static final int HEIGHT = 250;
  private SmileyPanel smileyPanel;      // drawing panel

  //*****************************************************

  public DragSmiley()
  {
    setTitle("Drag Smiley");
    setSize(WIDTH, HEIGHT);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    smileyPanel = new SmileyPanel();
    add(smileyPanel);
    setVisible(true);
  } // end DragSmiley constructor

  //***********************************

  public static void main(String[] args)
  {
    new DragSmiley();
  }
} // end class DragSmiley
```

Desplegado inicial: Mientras se está arrastrando la carita feliz: Después de soltar el botón del ratón:

Figura 16.16 Clase controladora y muestra de salida para el programa DragSmiley.

agrega al contenedor `JPanel`. Observe los encabezados de la clase oyente, que se repiten aquí por conveniencia:

```
private class ClickListener extends MouseAdapter
private class DragListener extends MouseMotionAdapter
```

La cláusula `extends` indica herencia de las clases `MouseAdapter` y `MouseMotionAdapter`. Para cada interfaz de manipulación de eventos con más de un método, Sun proporciona una clase asociada que ya implementa el método de la interfaz, en lugar de que lo haga el usuario. Esas clases se denominan *clases adapter*. La clase `MouseAdapter` implementa los métodos de la interfaz `MouseListener`, y la clase `MouseMotionAdapter` implementa los métodos de la interfaz `MouseMotionListener`. Las clases adapter no hacen mucho. Simplemente implementan sus métodos de interfaz asociados como métodos inválidos, como esto:

```
public void mousePressed(MouseEvent event)
{ }
```

Para implementar un oyente que detecte el ratón que se está oprimiendo, la clase `MouseAdapter` se extiende y se proporciona un método de sobreposición `mousePressed`. Como un ejemplo, vea la figura 16.17*a*. Como alternativa, es posible implementar, y cuando se implementa una interfaz, se requiere proporcionar métodos para todos los métodos de la interfaz. Si se desea sustituir a los adaptadores de la clase `SmileyPanel` con interfaces, es necesario proporcionar métodos inválidos para los métodos que ya no se usen.

Despliegue de una imagen

Ya es hora de ver cómo la clase `SmileyPanel` traza sus imágenes. En la parte superior de la clase, las constantes nombradas `SMILEY` y `SCARED` se inician como sigue:

```
final private ImageIcon SMILEY = new ImageIcon("smiley.gif");
final private ImageIcon SCARED = new ImageIcon("scared.gif");
```

El constructor `ImageIcon` crea un objeto de imagen a partir de su parámetro de archivo pasado. Así, en el fragmento de código de arriba, a partir de los archivos `smiley.gif` y `scared.gif` se crean dos objetos de imagen, respectivamente.[7]

En el constructor `SmileyPanel`, en el manipulador de eventos `mousePressed` y en el manipulador de eventos `mouseReleased`, observe cómo `SMILEY` y `SCARED` se asignan a la variable de instancia `image`. Estas asignaciones son lo que provocan que la imagen cambie cuando el usuario oprime el botón del ratón y luego lo suelta.

La clase `JPanel` tiene un método `paintComponent` a cargo de dibujar los componentes Swing (por ejemplo, ventanas y botones de texto) dentro del contenedor `JPanel`. Pero no manipula el trazo de líneas, formas o imágenes. Para trazar estos objetos, es necesario contar con un método de sobreposición `paintComponent` que llame a métodos gráficos. Por ejemplo, éste es el método de sobreposición `paintComponent` de `SmileyPanel`:

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  image.paintIcon(this, g,
    (int) imageCorner.getX(), (int) imageCorner.getY());
} // end paintComponent
```

Observe el parámetro `g` del método `paintComponent`. Se trata de un objeto `Graphics` y se usa para llamar a métodos gráficos dentro del método `paintComponent`. Por ejemplo, el método `image.PaintIcon` llama a *draws* `image` (una carita sonriente o una carita con miedo) y requiere un objeto `Graphics`, `g`, como su segundo argumento. Al llamar al método `paintIcon`, se proporcionan tres argumentos además del argumento `Graphics`: 1) la ventana en la que aparece la imagen (en el ejemplo de arriba, la referencia `this` se refiere a la ventana `JPanel`), 2) la coordenada x del ángulo superior izquierdo de la imagen y 3) la coordenada y del ángulo superior izquierdo de la imagen. Observe la llamada al método `super.PaintComponent(g)`. Esta llamada siempre debe incluirse como la primera

[7] *`gif`* significa Formato de Intercambio Gráfico (Graphics Interchange Format). Se usa para una representación exacta de una imagen sencilla dibujada.

```
/*************************************************************
* SmileyPanel.java
* Dean & Dean
*
* Esta clase contiene una carita sonriente y oyentes
* que permiten el arrastre y el intercambio de la imagen.
*************************************************************/

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

public class SmileyPanel extends JPanel
{
  private final ImageIcon SMILEY = new ImageIcon("smiley.gif");
  private final ImageIcon SCARED = new ImageIcon("scared.gif");
  private final int WIDTH = SMILEY.getIconWidth();
  private final int HEIGHT = SMILEY.getIconHeight();

  private Point imageCorner;  // ubicación de la imagen en el ángulo superior izquierdo
  private Point prevPt;       // ubicación del ratón para un evento previo
  private ImageIcon image;    // se activa entre sonriente y con miedo.

  //*********************************************************

  public SmileyPanel()
  {
    image = SMILEY;
    imageCorner = new Point(0, 0); // image starts at top left
    ClickListener clickListener = new ClickListener();
    DragListener dragListener = new DragListener();
    this.addMouseListener(clickListener);
    this.addMouseMotionListener(dragListener);
  } // end SmileyComponent constructor

  //*********************************************************

  private class ClickListener extends MouseAdapter
  {
    // Cuando se oprime el ratón, cambia a imagen con miedo.

    public void mousePressed(MouseEvent e)
    {
      image = SCARED;
      prevPt = e.getPoint();  // guardar la posición actual
      repaint();
    } // end mousePressed
```

Agrega oyentes de ratón al contenedor `JPanel`.

Figura 16.17*a* La clase `SmileyPanel` del programa DragSmiley, parte A.

declaración dentro de un método de sobreposición `paintComponent`. Sin ella, el fondo para el objeto asociado con `paintComponent` puede aparecer incorrectamente.

Observe que no hay ninguna llamada explícita al método `paintComponent` del programa DragSmiley. El método `paintComponent` nunca debe llamarse directamente. En lugar de ello, debe llamarse al método `repaint` y dejar que éste llame al método `paintComponent`. El método `repaint`

```
    // Cuando se oprime el ratón, regresa la imagen sonriente.

    public void mouseReleased(MouseEvent e)
    {
      image = SMILEY;
      repaint();
    } // end mouseReleased
  } // end class ClickListener

  //*******************************************************

  private class DragListener extends MouseMotionAdapter
  {
    // Permite que el ratón arrastre la imagen.

    public void mouseDragged(MouseEvent e)
    {
      Point currentPt = e.getPoint(); // posición actual

      // Make sure mouse was pressed within the image.
      if (currentPt.getX() >= imageCorner.getX() &&
          currentPt.getX() <= imageCorner.getX() + WIDTH &&
          currentPt.getY() >= imageCorner.getY() &&
          currentPt.getY() <= imageCorner.getY() + HEIGHT)
      {
        imageCorner.translate(
          (int) (currentPt.getX() - prevPt.getX()),
          (int) (currentPt.getY() - prevPt.getY()));
        prevPt = currentPt;  // save current position
        repaint();
      }
    } // end mouseDragged
  } // end class DragListener

  //*******************************************************

  // Dibuja la ventana, incluyendo la imagen actualizada.

  public void paintComponent(Graphics g)
  {
    super.paintComponent(g);
    image.paintIcon(this, g,
      (int) imageCorner.getX(), (int) imageCorner.getY());
  } // end paintComponent
} // end class SmileyPanel
```

Llama al `icono paint` para mostrar la imagen.

Figura 16.17*b* La clase `SmileyPanel` del programa DragSmiley, parte B.

espera hasta que la ventana del programa esté idóneamente preparada para manipular al método `paintComponent`. Observe en la clase `SmileyPanel` cómo se llama a `repaint` en la parte inferior de los tres manipuladores de eventos. Es ahí que se requiere volver a dibujar la imagen. Por cierto, además de llamar a `paintComponent` siempre que se llama a `repaint`, la JVM llama automáticamente a `paintComponent` cuando empieza el programa y siempre que un usuario hace algo para modificar la ventana del programa (por ejemplo, cuando el usuario cambia el tamaño de la ventana, o mueve otra ventana fuera de la ventana).

Resumen

- La clase `JFrame` debe usarse como la superclase para casi todas las ventanas de aplicación GUI.
- La clase `JFrame` implementa todas las características de las ventanas, como límites, barra de título, botón para minimizar, botón de cierre (la "X"), la capacidad de ajustar la ventana, etcétera.
- `JLabel` es un componente de sólo lectura; el usuario simplemente lee el mensaje de la etiqueta.
- El componente `JTextField` permite que el usuario introduzca texto en una ventana de texto.
- Cuando el usuario interactúa con un componente (por ejemplo, al hacer clic en un botón u oprimir `Enter` mientras está en una ventana de texto), el componente activa un evento.
- Si un componente tiene conectado un oyente, el evento activado es "escuchado" por el oyente y en consecuencia, es manipulado por el oyente.
- Un oyente manipula un evento al ejecutar su método `actionPerformed` manipulador de eventos.
- A menudo, los oyentes se implementan con la interfaz `ActionListener`. Una interfaz es un ente semejante a una clase cuyos métodos, todos, están vacíos. Si un programador usa una clase para derivar una nueva clase, el compilador requiere que la nueva clase implemente métodos para todos los métodos de la interfaz.
- Si una clase está limitada en su alcance de modo que sólo es necesaria para otra clase, entonces es necesario definir la clase como una clase interna (una clase dentro de otra clase).
- Una clase interna anónima es una clase interna sin nombre.
- Para mostrar una simple ventana con un mensaje, es necesario llamar al método `showMessageDialog` de `JOptionPane`.
- Para identificar el componente cuyo evento fue activado, utilice el parámetro `ActionEvent` del método `actionPerformed` para llamar a `getSource` o a `getActionCommand`.
- Para ajustar el color del texto de un componente GUI, llame a `setForeground`. Para ajustar el color detrás del texto, llame a `setBackground`.
- Para ajustar el color de fondo de una ventana, llame al método `setBackground` del panel de contenido.
- Para detectar y manipular eventos de ratón, es necesario usar las clases `MouseAdapter` y `MouseMotionAdapter`, que implementan las interfaces `MouseListener` y `MouseMotionListener`, respectivamente.

Preguntas de revisión

§16.2 Fundamentos de programación de manejo de eventos

1. ¿Qué es un oyente?

2. ¿Qué es un manipulador de eventos?

§16.3 Un sencillo programa de ventanas

3. Escriba una declaración que agregue funcionalidad a un botón de cierre de una ventana de modo que al hacer clic en ese botón se provoque la terminación del programa.

§16.4 Clase `JFrame`

4. ¿Cuál es el nombre de la superclase para las clases que contienen componentes?

§16.5 Componentes en Java

5. ¿Qué paquetes son `JButton` y cuántos otros componentes tienen el prefijo J?

§16.6 Componente `JLabel`

6. Proporcione una declaración de inicio que declare una variable de referencia nombrada `hello` y que asigne "Hello World" a la variable de referencia.

§16.7 Componente `JTextField`

7. Proporcione una declaración de inicio que instancie un objeto en una ventana de texto de 10 caracteres de ancho. Como parte de la iniciación, asigne el objeto de la ventana de texto a una variable de referencia denominada `input`.

§16.9 Componente oyentes

8. Escriba una declaración que registre una variable de referencia oyente denominada `responder` con un componente nombrado `component`.

9. Si se desea que una clase manipule un evento, ¿qué cláusula debe agregarse al lado derecho del encabezado de la clase?

10. ¿Cuál es el encabezado de un método especificado por la interfaz `ActionListener`?

§16.10 Clases internas

11. Si una clase está limitada en alcance de modo que sólo es necesaria internamente dentro de otra clase, es necesario declarar que la clase sea una _____.

§16.11 Clases internas anónimas

12. Si se desea implementar un manipulador de eventos con una clase interna anónima, ¿qué argumento debe proporcionarse al método `addActionListener` para registrar al oyente?

§16.12 Componente `JButton`

13. En el método `createContents` del programa FactorialButton de la figura 16.8*a*, ¿qué tipo de objeto llama a los métodos `add`?

14. En el programa `FactorialButtom` de las figuras 16.8*a* y 16.18*b*, ¿qué componente activa el evento que manipula el oyente?

§16.13 Ventanas de diálogo y clase `JOptionPane`

15. ¿Qué paquete contiene la clase `JOptionPane`?

16. Escriba una declaración que despliegue una ventana de diálogo en el centro de la pantalla. La ventana de diálogo debe mostrar "Éste es sólo un texto" en la zona de mensajes, "TEST" en la zona del título, y no debe mostrar ningún icono.

§16.14 Distinción entre múltiples eventos

17. Suponga que se tienen varios componentes registrados con el mismo oyente, y que los componentes y el oyente están definidos en la misma clase. Dentro del oyente, ¿qué método `ActionEvent` debe llamarse para determinar el componente que activa un evento?

§16.15 Utilización de `getActionCommand` para distinción entre múltiples eventos

18. Suponga que hay un oyente que se ha registrado para varios botones diferentes. Suponga que el oyente utiliza un método `actionPerformed` con un parámetro `ActionEvent` denominado `action`. Suponga que el usuario hace clic en un botón. Escriba una declaración que recupere la etiqueta de texto desde el botón en que se hizo clic y asigne la etiqueta recuperada a una variable `String` denominada `buttonLabel`.

§16.16 Color

19. Escriba una declaración que establezca el color del texto en azul para un objeto `JButton` denominado `button1`.

20. ¿Cómo se obtiene una referencia al contenedor que rodea todos los componentes en un objeto `JFrame`?

§16.17 ¿Cómo se agrupan las clases GUI?

21. Si un programa requiere al subpaquete `java.awt.event`, es posible importarlo implícitamente al importar el paquete `java.awt`. (F/V)

Ejercicios

1. [Después de §16.2] Proporcione tres ejemplos de cómo un usuario podría ocasionar la activación de un evento.

2. [Después de §16.3] Para cada una de las opciones siguientes, ¿qué paquete Java es necesario importar?
   a) `JFrame` y `JLabel`
   b) `FlowLayout`

3. [Después de §16.4] Para los programas GUI previos, se ha realizado trabajo de organización (establecer el título, agregar componentes, etc.) dentro de un constructor. Esto suele preferirse aunque no es un requerimiento del compilador. Para efectos de práctica, escriba un programa mínimo aunque completamente funcional, que despliegue lo siguiente:

El programa no debe incluir ningún constructor. Sólo debe incluir un método: un método `main` con sólo cinco declaraciones (o cuatro declaraciones si se encuentra un atajo para establecer el título del marco).

4. [Después de §16.6] Escriba un programa completo que despliegue este mensaje `Hello World`:

Observe estas características de la etiqueta: 1) límite más alto del bisel; 2) cursivas; 3) fuente grande (30 puntos), y 4) pestaña que indica Life is Great! Use la estructura de este programa como punto de partida:

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

//*********************************************************

public class BigHello extends JFrame
{
  public BigHello()
  {
    JLabel label = <instantiation> ;
    setSize(200, <height> );
    setLayout(new FlowLayout());
    add(label);

    <3-statement code fragment>

    setVisible(true);
  } // end constructor

  //*********************************************************

  public static void main(String[] args)
  {
    BigHello hello = new BigHello();
  } // end main
} // end BigHello class
```

Para tener una idea de cómo hacer esto, es necesario consultar, en API de Java, los métodos `setFont`, `setBorder` y `setToolTipText` que `JLabel` hereda de `JContainer`. Para el argumento de `setFont`, use `getFont` de `JContainer` para obtener la fuente por defecto, y luego modificarla usando el método de dos parámetros `deriveFont` de `Font`, donde el primer parámetro especifica un estilo de fuente en cursivas y el segundo parámetro especifica un tamaño de 30 puntos. Use el método `setBorder` de `JContainer`, y para su argumento `Border`, use el método de clase idóneo de la clase `BorderFactory`.

5. [Después de §16.7] El parámetro del ancho en el constructor `JTextField` especifica el ancho de la ventana de texto en pixeles. (F/C)

6. [Después de §16.7] ¿Qué puede hacer el lector para impedir que los usuarios actualicen el componente `JTextField`?

7. [Después de §16.9] Escriba el encabezado del método que el lector debe definir en una clase que implemente un `ActionListener`.

8. [Después de §16.9] ¿En qué paquete API de Java están la interfaz `ActionListener` y la clase `ActionEvent`?

9. [Después de §16.9] Una interfaz es algo semejante a una clase cuyos métodos, todos, están vacíos. Si una interfaz se aplica a una clase, entonces la interfaz actúa como una plantilla a la que debe ajustarse la clase. (F/C)

10. [Después de §16.10] Una clase interna puede acceder directamente a sus variables de instancia de clase envolventes. (F/C)

11. [Después de §16.12] Si se piensa utilizar una sola vez una clase, es idóneo utilizar una clase interna anónima. En el programa Factorial en las figuras 16.8*a* y 16.8*b*, el objeto `listener` se usa dos veces, de modo que el objeto `listener` debe tener un nombre. No obstante, la clase del objeto sólo se usó una vez para instanciar ese objeto. En consecuencia, esa clase del objeto no necesita tener un nombre, y hubiera podido usarse una clase anónima para crear el objeto `listener`. Para este ejercicio, modifique el programa Factorial para que use una clase `ActionListener` anónima en lugar de la clase nombrada `Listener`. [Sugerencia: El programa ya está instalado para facilitar este cambio: sólo se trata de cortar y pegar.]

12. [Después de §16.13] Para usar una ventana de diálogo `JOptionPane`, ¿es necesario crear una ventana `JFrame`?

13. [Después de §16.13] Para contestar esta pregunta, tal vez sea necesario consultar los métodos `showInputDialog` y `showConfirmDialog` de `JOptionPane` en el sitio Web Sun de Java. ¿Qué hace este programa?

```
import javax.swing.JOptionPane;
public class UncertainHello
{
  public static void main(String[] args)
  {
    String name;
    int response;
    do
    {
      name = JOptionPane.showInputDialog("What's your name? ");
      response = JOptionPane.showConfirmDialog(null, "Are you sure?");
      if (response == JOptionPane.NO_OPTION)
      {
        name = "there";
        break;
      }
    } while (response == JOptionPane.CANCEL_OPTION);

    System.out.println("Hello " + name);
  } // end main
} // end class UncertainHello
```

14. [Después de §16.14] Al llamar a `setEnabled(false)`, es posible desactivar un botón, darle una apariencia muda y hacer que su oyente sea insensible a hacer clic en él. Habilítelo sólo después que el usuario introduzca un carácter en la ventana de texto `xBox` y tenga la llave de la llamada `setEnabled(true)` al evento `keyTyped`. Use la siguiente estructura del código oyente:

```
private class KeyListener extends KeyAdapter
{
  public void keyTyped(KeyEvent e)
  {
    ...
  }
} // end class KeyListener
```

Observe el `extends KeyAdapter` en el encabezado de la clase de arriba. Una clase *adapter* implementa una interfaz al proporcionar un método con cuerpo vacío para cada método en la interfaz. En este caso, la clase `KeyAdapter` de API implementa la interfaz `KeyListener` de API.

15. [Después de §16.16] Para establecer un color de fondo de `JFrame`, ¿qué método debe llamarse justo antes de llamar `setBackground`?

16. [Después de §16.17] ¿Qué significan las siglas “awt”?

Soluciones a las preguntas de revisión

1. Un oyente es un objeto que espera la ocurrencia de eventos.

2. Un manipulador de eventos es un método que responde a un evento.

3. `setDefaultCloseOperation(EXIT_ON_CLOSE);`

4. La superclase para objetos que contienen otros objetos es la clase `Container`.
5. Muchos componentes que tienen el prefijo J están definidos en el paquete `javax.swing`.
6. `JLabel hello = new JLabel("Hello World!");`
7. `JTextField input = new JTextField(10);`
8. `component.addActionListener(responder);`
9. Para que una clase manipule un evento, es necesario agregar esto al lado derecho del encabezado de la clase:
   ```
   implements ActionListener
   ```
10. El encabezado del método especificado por la interfaz `ActionListener` es:
    ```
    public void actionPerformed(ActionEvent e)
    ```
11. Si una clase está limitada en alcance de modo que sólo es necesaria internamente dentro de otra clase, la clase debe definirse como una clase interna.
12. El argumento que debe darse al método `addActionListener` para registrar una clase oyente anónima es:
    ```
    new ActionListener()
    {
      <implementation-of-ActionListener-interface>
    }
    ```
13. El objeto que llama a los métodos `add` es un objeto `JFrame`.
14. Es ambiguo. Puede ser `xBox` o `btn`.
15. El paquete que contiene la clase `JOptionPane` es el paquete `javax.swing`.
16. Este código genera la ventana de diálogo solicitada:
    ```
    JOptionPane.showMessageDialog(null,
      "This is only a test.", "TEST", JOptionPane.PLAIN_MESSAGE);
    ```
17. Para identificar el componente que activa un evento, es necesario llamar al método `getSource`.
18. `buttonLabel = action.getActionCommand();`
19. `button1.setForeground(Color.BLUE);`
20. Llamando al método `getContentPane` de `JFrame`.
21. Falso. Los paquetes `java.awt` y `java.awt.event` contienen clases por separado. Para importar clases de `java.awt.event`, este paquete debe importarse implícitamente así:
    ```
    import java.awt.event.*;
    ```

CAPÍTULO 17

Programación GUI: distribución de componentes, componentes GUI adicionales

Objetivos

- Conocer los fundamentos del diseño GUI.
- Conocer los beneficios de usar gestores de distribución.
- Comprender los detalles del gestor `FlowLayout`.
- Comprender los detalles del gestor `BorderLayout`.
- Poder usar la interfaz `SwingConstants`.
- Comprender los detalles del gestor `GridLayout`.
- Usar gestores de distribución insertados y `JPanel` para ventanas que tienen un número sustancial de componentes.
- Implementar componentes `JTextArea` para texto que abarca más de una línea.
- Implementar un componente `JCheckBox` para entrada sí/no del usuario.
- Implementar componentes `JRadioButton` y `JComboBox` cuando el usuario necesita escoger un valor de entre una lista de valores predefinidos.
- Familiarizarse con los componentes Swing adicionales como menús, paneles de desplazamiento y controles deslizantes.

Relación de temas

17.1 Introducción

Éste es el segundo capítulo de los dos dedicados a la programación GUI. En el capítulo previo se estudiaron los fundamentos de GUI. Se estudiaron ventanas, componentes y oyentes. Casi todos los programas GUI requieren lo anterior. En este capítulo se aprenderá a hacer que los programas GUI sean más funcionales y más atractivos visualmente. Se mejorará su funcionalidad al implementar algunos componentes GUI adicionales: `JTextArea`, `JCheckBox`, `JRadioButton` y `JComboBox`. Se mejorará el atractivo visual mediante la aplicación de varias técnicas de distribución a los componentes de las ventanas. Más específicamente, se aprenderá a aplicar estos gestores de distribución: `FlowLayout`, `BorderLayout` y `GridLayout`. Asimismo, se aprenderá a aplicar diferentes gestores de distribución a distintas zonas de sus ventanas.

Como ejemplo de lo que se aprenderá, vea la figura 17.1. Observe los cuadros combo, los botones de radio y los componentes de casillas de verificación. También, observe la forma en que los botones de radio están agrupados en el centro, los botones de verificación están agrupados a la derecha y los botones Next y Cancel están agrupados en la parte central inferior. En este capítulo se aprenderá a hacer estos agrupamientos, y también a posicionarlos correctamente.

17.2 Diseño GUI y gestores de distribución

Con los programas basados en texto, es relativamente fácil decir a los usuarios qué hacer. Como programador, usted simplemente proporciona instrucciones de texto y el usuario introduce lo que se le solicita. Con los programas GUI es más difícil decir a los usuarios qué hacer. Como programador, usted presenta una ventana con varios componentes, establece los oyentes y luego espera que el usuario haga algo. Es importante que su desplegado sea fácil de comprender; en caso contrario, los usuarios no sabrán qué hacer. Para que su desplegado sea fácil de comprender, siga estas directrices:

- Escoja los componentes correctos.
- Sea consistente.
- Posicione correctamente los componentes.

Fundamentos del diseño GUI

En la figura 17.1 observe los pequeños círculos cerca de Visa, MasterCard y Other. Estos círculos son componentes de botones de radio (los botones de radio se describen en la sección 17.13). El uso de estos botones para las opciones de tarjetas de crédito es un ejemplo de cómo escoger el componente correcto.

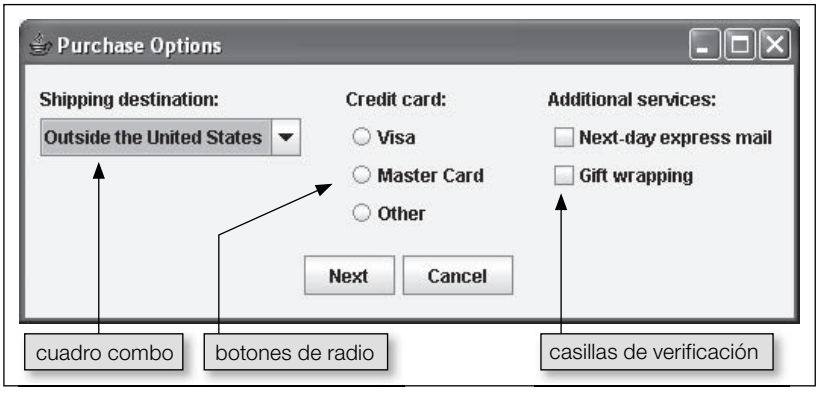


Figura 17.1 Ejemplo de ventana que usa botones de radio, casillas de verificación y un cuadro combo.

La mayor parte de los usuarios reconocen los círculos pequeños como botones de radio, y cuando los ven, saben hacer clic en uno de ellos con su ratón.

En la figura 17.1 observe los botones Next y Cancel en la parte central inferior de la ventana. Suponga que la ventana es una de muchas en una aplicación de compra. Suponga que otras ventanas en la aplicación también despliegan botones Next y Cancel en la parte central inferior de la ventana. Colocar los Next y Cancel en la misma posición es un ejemplo de consistencia. La consistencia es importante porque los usuarios están más cómodos con cosas que ya han visto antes. Como otro ejemplo, sea consistente con los esquemas de color. En una aplicación dada, si escoge rojo para un mensaje de advertencia, debe usar rojo para todos los mensajes de advertencia.

En la figura 17.1 observe cómo los tres componentes de los botones de radio (Visa, MasterCard y Other) y el componente de etiqueta "Credit card:" están juntos como un grupo. Más específicamente, están alineados en una columna vertical y están físicamente próximos entre sí. Éste es un ejemplo de posicionar correctamente los componentes. Al colocarlos juntos como grupo se proporciona el elemento visual de que están relacionados lógicamente. Como otro ejemplo de posicionamiento correcto de los componentes, observe que hay espacios mensurables que separan los grupos de componentes izquierdo, centro y derecho. Por último, observe cómo las etiquetas "Shipping destination:", "Credit card" y "Additional services:" están alineadas en el mismo renglón. Esa alineación, los espacios y los agrupamientos en grupos ya mencionados conducen a una presentación más atractiva y comprensible.

Gestores de distribución

Como ya sabe el lector, el posicionamiento correcto de los componentes es una parte importante del diseño GUI. En otra época, esta tarea era un proceso manual tedioso. Los programadores pasaban horas calculando el espacio necesario para cada componente y las posiciones de coordenadas en pixeles de cada componente. Hoy, los programadores se han liberado de ese tedio al disponer de gestores de distribución que hacen esos cálculos. Como quizás el lector recuerde del capítulo previo, un *gestor de distribución* es un objeto que controla el posicionamiento de los componentes dentro de un contenedor. En general, el objetivo del gestor de distribución es disponer los componentes en forma clara. Por lo general, el objetivo de la claridad es igual a asegurarse de que los componentes estén alineados y espaciados correctamente dentro del contenedor del gestor de distribución. Por ejemplo, en la figura 17.1, los gestores de distribución son responsables del alineamiento de los componentes a la izquierda, de los componentes en el centro, de los componentes a la derecha y de la separación de los tres componentes a lo ancho de la ventana.

Si un usuario ajusta el tamaño de una ventana, la JVM (Java Virtual Machine) consulta al gestor de distribución y éste vuelve a calcular las posiciones de coordenadas de los pixeles para cada componente. Todo esto se lleva a cabo en forma automática sin intervención del programador. ¡Cuán conveniente! ¡Gloria al gestor de distribución!

Hay varios tipos de gestores de distribución que tienen diversas estrategias para posicionar los componentes dentro de un contenedor. Observe la tabla en la figura 17.2. Describe varios gestores de distribución de la biblioteca Sun de API.

Gestor de distribución	Descripción
`BorderLayout`	Divide al contenedor en cinco regiones: norte, sur, este, oeste y centro.
`BoxLayout`	Permite que los componentes sean dispuestos en una sola columna o en un solo renglón.
`FlowLayout`	Permite que los componentes sean agregados de izquierda a derecha, pasando al siguiente renglón en caso de ser necesario.
`GridLayout`	Divide al contenedor en una retícula rectangular cuyas celdas son del mismo tamaño. Permite un componente por celda.
`GridBagLayout`	Ésta es una versión más flexible y complicada de `GridLayout`. Permite la variación del tamaño de las celdas.

Figura 17.2 Algunos de los gestores de distribución más conocidos.

En el capítulo previo se usó el tipo más sencillo de gestor de distribución: el `FlowLayout`. Este gestor es útil en algunas situaciones, pero para otras a menudo se requieren gestores de distribución alternativos. En este capítulo se describe el gestor `FlowLayout` con más detalle, y también se describen los gestores `BorderLayout` y `GridLayout`. Éstos son los tres tipos más importantes de gestores de distribución, de modo que es necesario conocerlos bien.

Asignación de un gestor de distribución

Para asignar un gestor de distribución particular a una ventana `JFrame` desde una clase interna que extienda `JFrame`, el método `setLayout` se llama así:

```
setLayout(new <layout-manager-class>());
```

En esta plantilla de código sustituya <*layout-manager-class*> por una de la clases del gestor de distribución (por ejemplo, `FlowLayout`, `BorderLayout`, `GridLayout`). Si no se llama a `setLayout`, entonces se usa el gestor `BorderLayout`, ya que es el gestor por defecto de una ventana `JFrame`.

Las clases del gestor de distribución están en el paquete `java.awt`, de modo que es necesario importar este paquete. Por supuesto, si este paquete ya se ha importado para otro fin, no es necesario importarlo otra vez.

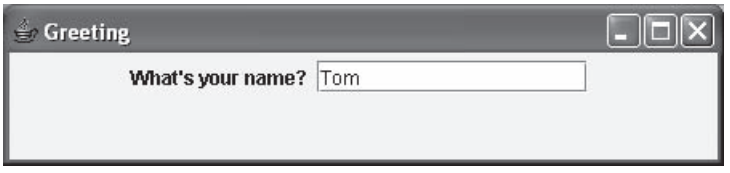
17.3 Gestor `FlowLayout`

En el capítulo previo se quiso presentar lo básico de GUI sin quedarse atascado en los detalles del gestor de distribución. Por eso se escogió un gestor de distribución sencillo, `FlowLayout`, que no requería muchas explicaciones. Sencillamente se usó y no se incursionó en detalles particulares. Ahora es el momento de explicar los detalles particulares, de modo que sea posible sacar más provecho de su funcionalidad.

Mecanismo de la distribución

La clase `FlowLayout` implementa un sencillo esquema de distribución de un compartimiento que permite la inserción de múltiples componentes en el compartimiento. Cuando un componente se agrega al compartimiento, se coloca a la derecha de los componentes que fueron agregados previamente al compartimiento. En caso de que no haya suficiente espacio para agregar un componente a la derecha de los componentes que fueron agregados previamente al compartimiento, el nuevo componente se coloca en la línea siguiente (es decir, "fluye" a la siguiente línea). Observe el siguiente ejemplo.

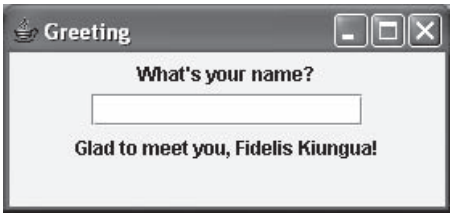
Suponga que se ha implementado un programa que solicita al usuario introducir su nombre e imprime un saludo personalizado después que el usuario oprime Enter. Se mostrará una sesión muestra que empieza con una ventana ancha y un nombre corto. Esto es lo que despliega el programa después que el usuario introduce Tom:



Y he aquí lo que despliega el programa después que el usuario oprime Enter:

Si el usuario introduce un nombre más largo, como Fidelis Kiungua, la etiqueta de saludo no cabe en la primera línea, por lo que pasa a la línea siguiente:

Si el usuario ajusta manualmente la ventana para hacerla más estrecha, la ventana de texto no cabe en la primera línea, por lo que se ajusta a la línea siguiente:

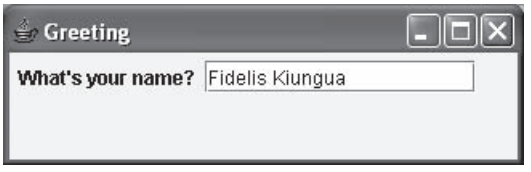


Alineamiento

Por defecto, el gestor `FlowLayout` coloca sus componentes usando un alineamiento central. Por ejemplo, en la ventana de arriba, observe cómo la etiqueta "What's your name" está centrada entre los límites izquierdo y derecho. Si se desea modificar el alineamiento del gestor `FlowLayout`, es necesario insertar una de las constantes de alineamiento (`FlowLayout.LEFT`, `FlowLayout.CENTER`, `FlowLayout.RIGHT`) en la llamada al constructor `FlowLayout`. Por ejemplo, he aquí cómo especificar el alineamiento a la izquierda:

```
setLayout(new FlowLayout(FlowLayout.LEFT));
```

Esto es lo que despliega el programa Greeting cuando se usa un alineamiento:



Cambios en la distribución

Normalmente, `setLayout` se llama justo una vez en un programa: cuando el programa inicialmente establece los componentes. Pero si hay necesidad de ajustar dinámicamente el esquema de la distribución, es necesario llamar de nuevo a `setLayout`. Por ejemplo, si se quiere que el usuario pueda ajustar la alineación del texto, es necesario agregar los botones Align Left, Align Center y Align Right. Por cada botón se agrega un oyente. En cada oyente se llama a `setLayout`. Éste sería el oyente para el botón `Align Left`:

```
private class Listener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    setLayout(new FlowLayout(FlowLayout.LEFT));
```

```
        validate();
      } // end actionPerformed
    } // end class Listener
```

Observe el método *validate* en el fragmento de código de arriba. Hace que el gestor de distribución regenere la distribución del componente. Si la ventana es visible (es decir, que se llamó a `setVisible(true)`), y de alguna manera se intenta modificar su distribución, es necesario llamar a `validate` para que el cambio surta efecto. Estas llamadas a métodos intentan cambiar la distribución:

- `setLayout`: cambia el gestor de distribución de la ventana.
- `add`: agrega un componente a la ventana.
- `setSize`: cambia el tamaño de la ventana.

⚠ Si la ventana ya es visible y se llama a uno de estos métodos, no debe olvidarse llamar a `validate` a continuación. Si se tiene una serie de estas llamadas, no es necesario separar las llamadas al método `validate`. Colocar una llamada al método `validate` al final funciona bien.

17.4 Gestor `BorderLayout`

Este gestor es conocido porque es fácil de usar. Simplemente agregue componentes a su contenedor y ya está. Algunas veces todo lo que se requiere es algo sencillo. Pero tenga en mente que el gestor `FlowLayout` no proporciona mucho control sobre el sitio en que han de colocarse los componentes. Al usar un gestor `FlowLayout`, los componentes pueden colocarse horizontalmente (izquierda, centro, derecha) pero no es posible colocarlos verticalmente.

Si es necesario colocar los componentes a lo largo de ambas dimensiones (horizontal y vertical), debe usarse uno de los otros gestores de distribución. En esta sección se analiza el gestor `BorderLayout`, que permite colocar los componentes a lo largo de ambas dimensiones.

Regiones `BorderLayout`

El gestor `BorderLayout` es particularmente útil para ventanas que requieren componentes cerca de sus bordes. Es común colocar un título cerca del borde superior de una ventana. Es común colocar un menú cerca del borde izquierdo de una ventana. Es común colocar botones cerca del borde inferior de una ventana. El gestor `BorderLayout` resuelve todas estas situaciones al dividir su contenedor en cinco *regiones* o compartimientos. Cuatro de las regiones están cerca de los bordes y una está en el centro. El acceso a las cuatro regiones de los bordes es mediante nombres geográficos: norte, sur, este y oeste. Observe las posiciones de las regiones en la figura 17.3.

Suponga que está dentro de una clase contenedora. Para asignar un gestor `BorderLayout` al contenedor, el método `setLayout` se llama así:

```
setLayout(new BorderLayout(<horizontal-gap>, <vertical-gap>));
```

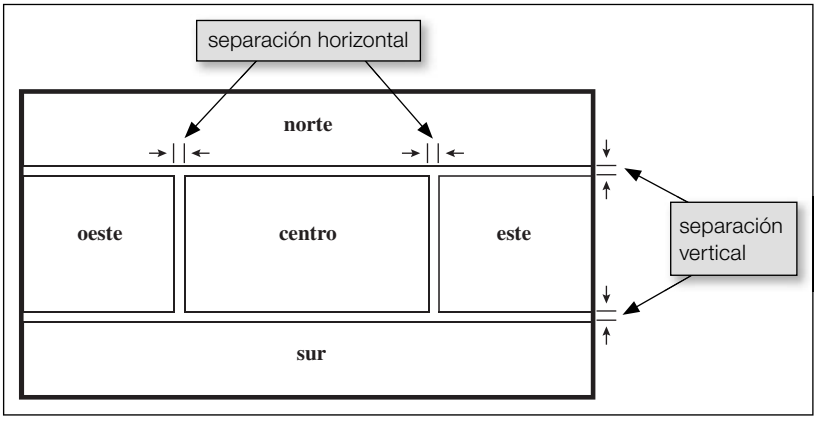


Figura 17.3 Regiones `BorderLayout`.

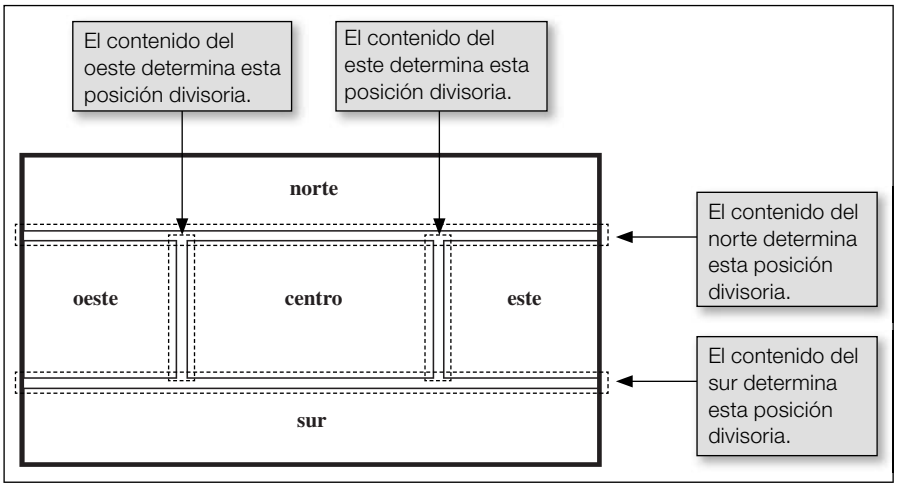
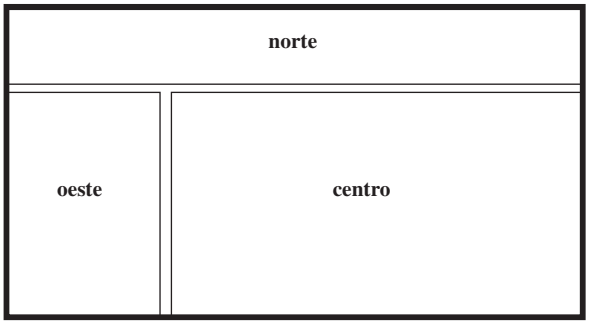


Figura 17.4 Regiones `BorderLayout`.

El argumento *separación horizontal* especifica el número de píxeles de espacio en blanco que separan las regiones oeste, centro y este. Esto se ilustra en la figura 17.3. El argumento *separación vertical* especifica el número de pixeles de espacio en blanco que separa la región norte de las otras regiones y la región sur de las otras regiones. De nuevo, esto se ilustra en la figura 17.3. Si se omiten los argumentos de las separaciones, los valores de éstas son cero por defecto. En otras palabras, si el constructor `BorderLayout` se llama sin argumentos, entonces entre las regiones no hay separación.

Los tamaños de las cinco regiones se determinan durante el tiempo de ejecución, y se basan en el contenido de cada región. Por tanto, si la región oeste contiene una etiqueta larga, el gestor de distribución intenta ampliar la región oeste. En forma semejante, si la región oeste contiene una etiqueta corta, el gestor de distribución intenta estrechar la región oeste.

Si una región exterior está vacía, se colapsa, de modo que ya no ocupa ningún espacio. Pero, ¿qué ocurre exactamente durante el colapso? Cada región exterior controla sólo una línea divisoria, de modo que por cada región colapsada sólo se mueve una línea divisoria. La figura 17.4 muestra que la línea divisoria de la región oeste es la frontera entre el oeste y el centro, que la línea divisoria de la región norte es la frontera entre el norte y abajo, etc. Entonces, si la región norte está vacía, la línea divisoria del norte se mueve hasta la frontera superior y las regiones oeste, centro y este —todas— se expanden hacia arriba. ¿Qué ocurre si ambas regiones, este y sur, están vacías? La región este vacía ocasiona que la línea divisoria del este se desplace hacia la frontera derecha. La región sur vacía ocasiona que la línea divisoria del sur se desplace hacia la frontera inferior. Ésta es la distribución resultante:



¿Qué ocurre si la región centro está vacía? La región centro no controla ninguna de las líneas divisorias, de modo que no ocurre nada.

Adición de componentes

Suponga que se tiene una clase contenedora que usa el gestor `BorderLayout`. Para agregar un componente a una de las regiones contenedoras de `BorderLayout`, el método `add` se llama así:

```
add(<component>, <region>);
```

Sustituya *<component>* por un componente (un objeto `JLabel`, un objeto `JButton`, etc.) y sustituya *<region>* por una de las constantes nombradas: `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.WEST`, `BorderLayout.EAST`, o `BorderLayout.CENTER`.

Por ejemplo, he aquí cómo se agrega un botón Tunisia a la región norte:

```
add(new JButton("Tunisia"), BorderLayout.NORTH);
```

Si el método `add` se llama sin argumento de región, por defecto se usa la región centro. Así, para agregar un botón Central African Republic a la región centro, puede usarse cualquiera de las dos declaraciones siguientes:

```
add(new JButton("Central African Republic"), BorderLayout.CENTER);
add(new JButton("Central African Republic"));
```

¿Cuál es mejor? Aquí se prefiere la primera porque facilita la comprensión del código. Más formalmente, se dice que la primera declaración es autodocumentada.

Con un contenedor `FlowLayout` se pueden agregar tantos componentes como se quiera. Con un contenedor `BorderLayout`, sólo es posible agregar cinco componentes en total, una para cada una de las cinco regiones. Si se agrega un componente a una región que ya tiene un componente, entonces el nuevo componente se sobrepone al componente anterior. Así, al ejecutar las líneas siguientes, el botón Somalia se sobrepone al botón Djibouti:

```
add(new JButton("Djibouti"), BorderLayout.EAST);
add(new JButton("Somalia ", BorderLayout.EAST));
```

Si se requiere agregar más de un componente a una región, es fácil cometer el error de llamar al método `add` dos veces para la misma región. Después de todo, no hay ningún error de tiempo de compilación para advertir al lector de su fechoría. Sin embargo, lo que realmente debe hacerse es agregar un componente `JPanel`. Este componente se analizará más adelante en este capítulo. Permite almacenar múltiples componentes en un sitio en el que sólo se permite un componente.

Programa AfricanCountries con botones

A continuación, este material sobre `BorderLayout` se pondrá en práctica al usarlo en un programa completo. En el programa AfricanCountries se agregan botones de un país africano a las cinco regiones de una ventana `BorderLayout`. Vea la ventana de salida en la parte inferior de la figura 17.5. Los cinco rectángulos representan las cinco regiones, aunque también son los cinco botones. Los botones son del mismo tamaño que las regiones porque, con el gestor `BorderLayout`, los componentes se expanden automáticamente para llenar toda la región. Observe cómo los tamaños de las cuatro regiones exteriores se ajustan bastante bien a su contenido. En otras palabras, la región oeste es suficientemente ancha para mostrar "Western Sahara". La región sur es suficientemente alta para mostrar "South Africa", etc. En contraste, observe cómo la región centro es incapaz de desplegar en todo su contenido "Central African Republic". Esto se debe a que las regiones exteriores controlan las líneas divisorias. La región centro obtiene el espacio que queda libre.

Eche una ojeada al listado del programa AfricanCountries en la figura 17.5. La mayor parte del código es directa. Sin embargo, esta declaración es bastante peculiar:

```
add(new JButton("<html>South<br>Africa</html>"), BorderLayout.SOUTH);
```

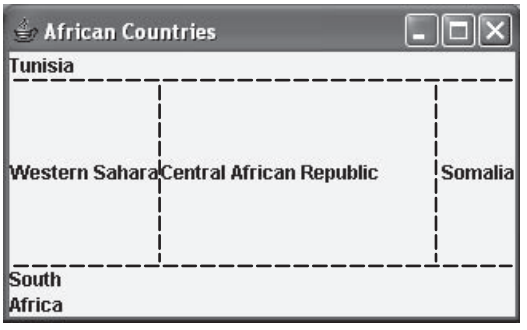
A continuación se revisará brevemente que estos comandos entre paréntesis angulares son lo que se ve: <html>,
 y </html>. Como quizá recuerde del programa HTMLGenerator en el capítulo 15, los elementos entre paréntesis angulares se denominan etiquetas. La etiqueta `<html>` indica el inicio de un archivo HTML, la etiqueta `<br>` indica el rompimiento de una línea (es decir, una nueva línea), y la etiqueta

`</html>` indica el final de un archivo HTML. Normalmente, las etiquetas se insertan en un archivo HTML. Pero aquí se están insertando en un componente de texto a fin de producir una nueva línea. Cuando `JLabel` y `JButton` se usan en texto, las etiquetas `<html>` y `</html>` indican al compilador de Java que el texto entre las etiquetas `<html>` y `</html>` debe interpretarse como texto HTML. Y la etiqueta `<br>` indica al compilador de Java que debe insertar un carácter de nueva línea en el texto.[1]

Es conveniente mencionar una cuestión adicional en el programa AfricanCountries. Es posible omitir el método `setLayout`. Como ya se dijo, el gestor de distribución por defecto para ventanas `JFrame` es `BorderLayout`. En consecuencia, si se omite la llamada al método `setLayout`, el programa sigue funcionando bien. No obstante, aquí se prefiere mantener la llamada al método `setLayout` porque facilita la comprensión del programa.

Programa AfricanCountries con etiquetas

Quizás el lector observó las líneas divisorias en la ventana de salida en la figura 17.5. Las líneas provienen de los límites de los botones, no del gestor `BorderLayout`. Si se usan componentes de etiqueta en lugar de componentes de botón, las líneas divisorias no se ven. En forma semejante, en la figura 17.5, los márgenes alrededor de las palabras provienen de los componentes de botón. Si se usan componentes etiqueta en lugar de componentes botón, no se ven los márgenes alrededor de las palabras. A continuación se muestra lo que despliega el programa AfricanCountries cuando los componentes botón se sustituyen por componentes etiqueta. Tenga en cuenta que las líneas discontinuas no aparecen en la ventana real. Se han trazado para mostrar los límites de las regiones.



Las regiones se parecen bastante a las anteriores, excepto en que las regiones oeste y este son más estrechas. Esto es porque alrededor de las palabras no hay márgenes. Regiones oeste y este más estrechas significan que hay más espacio para la región centro. Así, la región centro despliega todo el texto "Central African Republic".

Observe que las etiquetas de los países africanos están alineadas a la izquierda. Ésta es la situación por defecto para una etiqueta en una región `BorderLayout`. Si se desea tener una alineación distinta a la que se proporciona por defecto, la etiqueta debe instanciarse con una alineación de la siguiente forma:

```
new JLabel(<label's-text>, <alignment-constant>)
```

<alignment-constant> debe sustituirse por una de las constantes nombradas: `SwingConstants.LEFT`, `SwingConstants.CENTER` o `SwingConstants.RIGHT`. A continuación se presenta un ejemplo que agrega una etiqueta alineada a la izquierda a una región norte `BorderLayout`:

```
add(new JLabel("Tunisia", SwingConstants.CENTER), BorderLayout.NORTH);
```

Si esa línea de código se aplica al programa AfricanCountries y un código semejante de alineación a la izquierda se aplica a las regiones centro y sur, el programa despliega lo siguiente:

[1] Quizá el lector haya insertado el carácter de línea nueva, `\n`, en el texto del componente. Desafortunadamente, eso no funciona con los componentes `JButton` y `JLabel`. Sin embargo, funciona para el componente `JTextArea`. Más adelante, en este capítulo, se describe el componente `JTextArea`.

```
/*************************************************************
* AfricanCountries
* Dean & Dean
*
* Este programa muestra la distribución de componentes
* para el gestor BorderLayout.
*************************************************************/

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

public class AfricanCountries extends JFrame
{
  private static final int WIDTH = 325;
  private static final int HEIGHT = 200;

  public AfricanCountries()
  {
    setTitle("African Countries");
    setSize(WIDTH, HEIGHT);
    setDefaultCloseOperation(EXIT _ ON _ CLOSE);
    setLayout(new BorderLayout());
    add(new JButton("Tunisia"), BorderLayout.NORTH);
    add(new JButton("<html>South<br>Africa</html>"),
      BorderLayout.SOUTH);
    add(new JButton("Western Sahara"), BorderLayout.WEST);
    add(new JButton("Central African Republic"),
      BorderLayout.CENTER);
    add(new JButton("Somalia"), BorderLayout.EAST);
    setVisible(true);
  } // end AfricanCountries constructor

  //*************************************************************

  public static void main(String[] args)
  {
    new AfricanCountries();
  } // end main
} // end class AfricanCountries
```

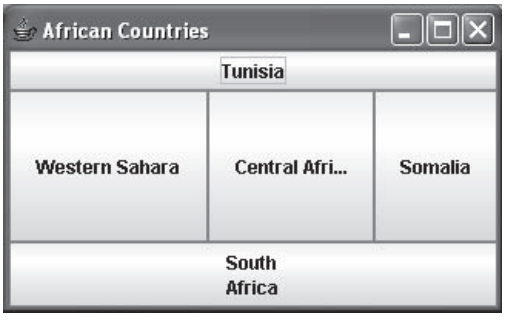
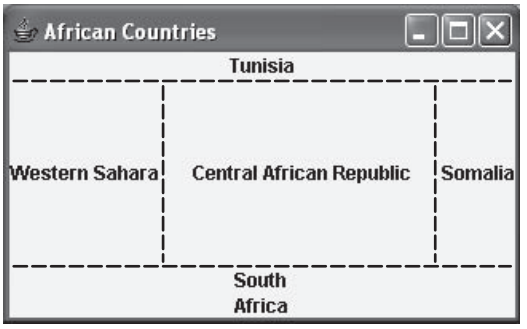


Figura 17.5 Programa AfricanCountries y su salida.



De nuevo, las líneas discontinuas no aparecen en la ventana real. Se han trazado para mostrar los límites de las regiones. No tiene caso aplicar una alineación al centro a las etiquetas oeste y este. Para estas etiquetas, la alineación es irrelevante porque las etiquetas oeste y este no tienen a dónde moverse. Como indican las líneas discontinuas, ya están alineadas con sus límites izquierdo y derecho.

Ahora, se regresará a las constantes de alineamiento (`SwingConstants.LEFT`, `SwingConstants.CENTER`, `SwingConstants.RIGHT`). Podría pensarse que `SwingConstants` constituye una clase, puesto que su primera letra es mayúscula. Si fuese una clase, describiría un objeto. Pero no describe ningún objeto y no es una clase. En realidad, `SwingConstants` es una interfaz definida en el paquete `java.swing`. Sun proporciona la interfaz `SwingConstants` como un arsenal para varias constantes nombradas relacionadas con GUI. Para acceder a una constante nombrada en `SwingConstants`, el nombre de la constante debe anteponerse con el nombre de la interfaz. Por ejemplo, para acceder a la constante de alineamiento `LEFT`, se usa `SwingConstants.LEFT`. Si el lector desea conocer detalles adicionales sobre interfaces, debe consultar la sección 13.9 del capítulo 13.

Es fácil confundirse entre el alineamiento de etiquetas para un contenedor `BorderLayout` y el alineamiento de etiquetas para un contenedor `FlowLayout`. Con un contenedor `BorderLayout`, si se quiere especificar un alineamiento de etiquetas, es necesario especificar un valor `SwingConstants` como parte de la instanciación de la etiqueta. Si esto se hace con un contenedor `FlowLayout`, el código compila, pero no tiene ningún impacto en el alineamiento de etiquetas. Con un contenedor `FlowLayout`, el alineamiento de etiquetas individuales es irrelevante: lo que importa es el alineamiento del contenedor. Si el contenedor usa alineamiento izquierdo, entonces todos sus componentes están alineados a la izquierda; si el contenedor usa alineamiento al centro, entonces todos sus componentes están alineados al centro, etcétera. Para establecer el alineamiento del contenedor, es necesario insertar una de las constantes de alineamiento (`FlowLayout.LEFT`, `FlowLayout.CENTER`, `FlowLayout.RIGHT`) en la llamada al constructor `FlowLayout`. A continuación se muestra cómo especificar alineamiento izquierdo para todos los componentes en el contenedor `FlowLayout`:

```
setLayout(new FlowLayout(FlowLayout.LEFT));
```

17.5 Gestor `GridLayout`

El esquema de partición del gestor `BorderLayout` (norte, sur, este, oeste, centro) funciona bien para muchas situaciones, pero no para todas. A menudo es necesario desplegar información usando un formato tabular; es decir, se requiere mostrar información que esté organizada en renglones y columnas. El gestor `BorderLayout` no es adecuado para formatos tabulares, pero el gestor `GridLayout` ¡funciona muy bien!

Celdas `GridLayout`

El gestor `GridLayout` dispone los componentes de un contenedor en un arreglo reticular. La rejilla cuenta con celdas del mismo tamaño y cada celda puede contener un solo componente.

Suponga que se esté dentro de una clase contenedora. Para asignar un gestor `GridLayout` al contenedor, el método `setLayout` se llama así:

```
setLayout(new GridLayout(<number-of-rows>, <number-of-columns>,
 <horizontal-gap>, <vertical-gap>));
```

Los argumentos <*number-of-rows*> y <*number-of-columns*> especifican el número de renglones y el número de columnas, respectivamente, en la rejilla rectangular. El argumento <*horizontal-gap*> especifica el número de pixeles de espacio en blanco que aparecen entre cada columna en la rejilla. El argumento <*vertical-gap*> especifica el número de pixeles de espacio en blanco que aparecen entre cada renglón en la rejilla. Si se omiten los argumentos de separación, entonces los valores de separación son cero por defecto. En otras palabras, si el constructor `GridLayout` se llama con sólo dos argumentos, entonces no hay separación entre las celdas.

Adición de componentes

Suponga que está dentro de una clase contenedora `GridLayout`. Para agregar un componente a una de las celdas, el método `add` se llama así:

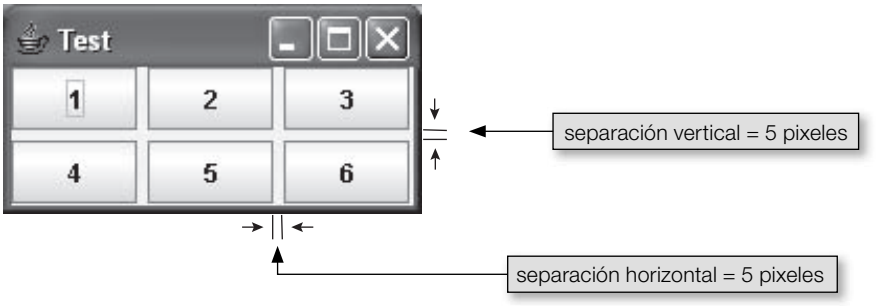
```
add(<component>);
```

Observe la sencillez de la llamada al método `add`. En particular, observe que no se menciona la celda a la que va a conectarse el componente. Entonces, ¿cómo sabe el gestor `GridLayout` a qué celda conectar el componente? El gestor `GridLayout` posiciona los componentes dentro del contenedor mediante el uso del orden izquierda-derecha, arriba-abajo. El primer componente agregado se coloca en la celda superior izquierda; el siguiente se coloca en la celda contigua a la derecha del primer componente, y así sucesivamente.

El fragmento de código que se muestra a continuación genera una tabla de dos renglones por tres columnas con seis botones. El fragmento de código especifica separaciones de cinco pixeles entre los renglones y las columnas.

```
setLayout(new GridLayout(2, 3, 5, 5));
add(new JButton("1"));
add(new JButton("2"));
add(new JButton("3"));
add(new JButton("4"));
add(new JButton("5"));
add(new JButton("6"));
```

Suponga que el fragmento de código de arriba forma parte de un programa completo que funciona. Esto es lo que despliega el programa:



Los seis rectángulos mostrados corresponden a seis celdas, pero también son seis botones. Los botones son del mismo tamaño que las celdas porque, con un gestor `GridLayout`, los componentes se expanden para ocupar sus celdas. Esto debe sonar conocido; los componentes `BorderLayout` hacen lo mismo.

Especificación del número de renglones y el número de columnas

Cuando se crea un gestor `GridLayout`, el constructor `GridLayout` se llama con un argumento número de renglones y un argumento número de columnas. Estos dos argumentos requieren de una explicación. Para ello, se considerarán tres casos diferentes:

Caso uno:

Si se conoce el número de renglones y de columnas en la tabla y la tabla está llena por completo (es decir, que no hay celdas vacías), el constructor `GridLayout` se llama con el número real de renglones y el número real de columnas.

Esto es lo que se hizo en el ejemplo previo. Se sabía que era necesaria una tabla de tres renglones por tres columnas con seis botones, por lo que para el argumento de los renglones se especificó dos y para el de las columnas, tres.

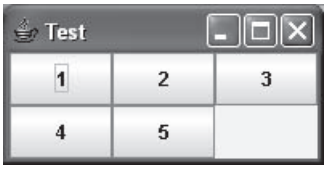
Caso dos:

Algunas veces podría requerirse un desplegado orientado a renglones. En otras palabras, se desea mostrar cierto número de renglones y el número de columnas no tiene importancia o no se sabe cuántas columnas se necesitan. Si éste es el caso, entonces el constructor `GridLayout` se llama con el número real de renglones para el argumento de los renglones y con cero para el argumento de las columnas. Cero para el argumento de las columnas indica que la decisión de determinar el número de columnas se deja al gestor `GridLayout`.

El fragmento de código que se muestra a continuación genera un `GridLayout` de dos renglones con cinco botones. Puesto que la llamada a `setLayout` no especifica valores de separación, `GridLayout` no muestra separaciones entre los botones.

```
setLayout(new GridLayout(2, 0));
add(new JButton("1"));
add(new JButton("2"));
add(new JButton("3"));
add(new JButton("4"));
add(new JButton("5"));
```

Suponga que el fragmento de código de arriba forma parte de un programa completo que funciona. Esto es lo que despliega el programa:



Caso tres:

Algunas veces podría requerirse un desplegado orientado a columnas. En otras palabras, se desea mostrar cierto número de columnas y el número de renglones no tiene importancia o no se sabe cuántos renglones se necesitan. Si éste es el caso, entonces el constructor `GridLayout` se llama con el número real de columnas para el argumento de las columnas y con cero para el argumento de los renglones. Cero para el argumento de los renglones indica que la decisión de determinar el número de renglones se deja al gestor `GridLayout`.

El fragmento de código que se muestra a continuación genera un `GridLayout` de cuatro columnas con cinco botones.

```
setLayout(new GridLayout(0, 4));
add(new JButton("1"));
add(new JButton("2"));
add(new JButton("3"));
add(new JButton("4"));
add(new JButton("5"));
```

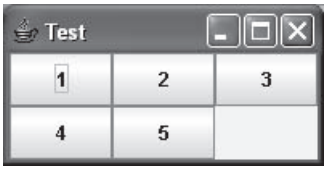
Suponga que el fragmento de código de arriba forma parte de un programa completo que funciona. Esto es lo que despliega el programa:

A continuación se mencionarán dos cuestiones de las que hay que cuidarse. Como se sabe, hay una importancia especial cuando se llama al constructor `GridLayout` con renglones = 0 o columnas = 0. Lo anterior asigna al gestor `GridLayout` la responsabilidad de escoger el número de renglones o de columnas. Pero esto sólo funciona si se tiene un argumento con valor cero, no dos. Si el constructor `GridLayout` se llama con dos argumentos con valor cero, se obtiene un error de tiempo de compilación.

¿Y qué ocurre en el caso opuesto?, es decir, cuando el constructor `GridLayout` se llama con dos argumentos con valor distinto de cero para los argumentos de renglón y de columna. No hay problema en la medida en que la tabla esté completamente llena. Si la tabla no está completamente llena, pueden esperarse resultados imprevistos. Por ejemplo, la ventana de arriba de cuatro columnas no está completamente llena. Suponga que accidentalmente se especifica un valor para el argumento de renglón:

```
setLayout(new GridLayout(2, 4));
```

Esto es lo que despliega el programa:



¡Esto sí que es raro! Hay tres columnas, aunque se especificaron cuatro. Moraleja: llame al constructor `GridLayout` con dos argumentos con valor distinto de cero sólo si la tabla está llena por completo.[2]

17.6 Ejemplo de juego de gato

En esta sección se presenta un programa simple de juego de gato (tic-tac-toe). Se eligió este juego porque la idea es ilustrar los detalles de `GridLayout` y el juego de gato, con su tablero de tres renglones por tres columnas, es la oportunidad perfecta para lograrlo.

Interfaz del usuario

El programa despliega inicialmente una rejilla de tres renglones y tres columnas de botones en blanco. Dos usuarios, el jugador X y el jugador O, hacen clic por turnos en los botones en blanco. Empieza el jugador X. Cuando el jugador X hace clic en un botón, la etiqueta del botón cambia de blanco a X. Cuando el jugador O hace clic en un botón, la etiqueta del botón cambia de blanco a O. El jugador X gana al lograr colocar tres X en un renglón, tres X en una columna o tres X en una diagonal. El jugador O gana si hace lo mismo, salvo que en lugar de X se ve O. Para tener más destreza en este juego, observe la sesión muestra en la figura 17.6.

Detalles del programa

Consulte los listados del programa juego de gato en las figuras 17.7*a* y 17.7*b*. La mayor parte del código ahora ya debe resultar conocido, pues su estructura imita la de los programas previos GUI. Se omitirá el código conocido y se pasará directamente al código más difícil.

Observe la llamada al método `setLayout` en la figura 17.7*a*. Contiene una llamada al constructor `GridLayout` que especifica tres renglones y tres columnas. La llamada al constructor no incluye ningún argumento de separación horizontal ni de separación vertical, de modo que los botones del juego de gato aparecen sin separaciones entre ellos.

[2] Éste es el lado flaco. Si el constructor `GridLayout` se llama con dos valores diferentes de cero para los argumentos de los renglones y las columnas, el argumento de las columnas es ignorado y el gestor `GridLayout` determina por sí solo el número de columnas. Para el caso en que se tienen dos valores distintos de cero y la tabla está completamente llena, el gestor `GridLayout` sigue determinando por sí solo el número de columnas. Pero el número determinado de columnas funciona a la perfección (es decir, el número determinado de columnas coincide con el número especificado de columnas).

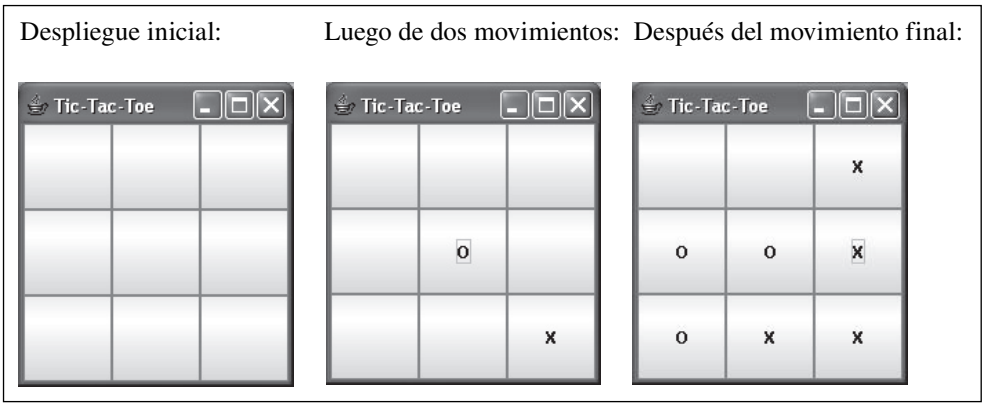


Figura 17.6 Sesión de muestra para el programa juego de gato.

Ahora se considerará la clase `Listener` en la figura 17.7*b*. En particular, observe la declaración donde se obtiene el botón en que se ha hecho clic y se guarda en una variable local:

```
JButton btn = (JButton) e.getSource();
```

⚠ El operador tipo cast (`JButton`) se usa porque si no hubiese operador tipo cast, el compilador generaría un error. ¿Por qué? Porque el compilador vería que un `Object` a la derecha se asigna a un `JButton` a la izquierda (ve un objeto a la derecha porque `getSource` se define con un tipo de retorno `Object`). En este caso, puesto que `getSource` devuelve un `JButton`, es legal transformar su valor devuelto a `JButton`, y así se satisface al compilador y se elimina el error.

Ahora se examinará la clase `Listener` de la declaración `If`:

```
if (btn.getText().isEmpty())
{
  btn.setText(xTurn ? "X" : "O");
  xTurn = !xTurn;
}
```

Primero es necesario verificar que el botón es uno en blanco. Luego se reasigna la etiqueta del botón mediante el uso de un operador condicional. Si `xTurn` es `true`, entonces X se asigna a la etiqueta del botón. En caso contrario, O se asigna a la etiqueta del botón. Luego se cambia el valor de `xTurn` al asignarle su valor negado. Más específicamente, si `xTurn` es `false`, se asigna `true` a `xTurn`. Y si `xTurn` es `true`, se asigna `false` a `xTurn`.

17.7 Resolución de problema: triunfo en el juego de gato (opcional)

Como quizá ya observó el lector, el programa juego de gato en la sección previa no verifica un movimiento ganador. Como ejercicio de solución de problemas, a continuación se analizará cómo agregar esa funcionalidad. En vez de proporcionarle una solución en Java, se le dará el difícil proceso para llegar a una solución. Este proceso se codificará usando seudocódigo. En uno de los proyectos del capítulo se solicita al lector que termine la tarea al implementar una solución de programa Java.

Repita para incrementar.

Para verificar un triunfo (es decir, para comprobar si hay tres X en un renglón, tres en una columna o tres en una diagonal), el oyente necesita acceder a múltiples botones. Como está ahora, el oyente del juego gato sólo puede acceder a un botón: el botón en que se hizo clic. Obtiene ese botón al llamar a `getSource`. ¿Cómo debe modificarse el programa de modo que el oyente pueda acceder a múltiples botones?

Para acceder a múltiples botones, es necesario declarar múltiples botones. Aunque es posible declarar nueve botones distintos, una solución más elegante es declarar un arreglo bidimensional de botones de tres renglones por tres columnas. La siguiente pregunta es ¿dónde debe declararse el arreglo? ¿Se declara como una variable local dentro del oyente o como una variable de instancia en la parte superior del

```
/*************************************************************
* TicTacToe.java
* Dean & Dean
*
* Este programa implementa el juego de gato. Cuando se hace clic
* en el primer botón en blanco, su etiqueta cambia a X. Las etiquetas
* de los botones en blanco en que se hace clic cambian a O o X
* en forma alterna.
*************************************************************/

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

public class TicTacToe extends JFrame
{
  private boolean xTurn = true; // sigue la pista del turno del jugador
                                // correspondiente: X u O.

  //*********************************************************

  public TicTacToe()
  {
    setTitle("Tic-Tac-Toe");
    setSize(200, 220);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents();
    setVisible(true);
  } // end TicTacToe constructor

  //*********************************************************

  // Crea componentes y agrega una ventana.

  private void createContents()
  {
    JButton button; // reinstancia este botón y lo usa
                    // para llenar todo el tablero
    setLayout(new GridLayout(3, 3));

    for (int i=0; i<3; i++)
    {
      for (int j=0; j<3; j++)
      {
        button = new JButton();
        button.addActionListener(new Listener());
        add(button);
      } // end for j
    } // end for i
  } // end createContents
```

Figura 17.7*a* Programa juego de gato, parte A.

programa? En general, las variables locales son preferibles, aunque en este caso una variable local no funciona. Las variables locales no persisten. Es necesario poder actualizar un botón desde el interior del oyente y hacer que la actualización sea recordada la próxima vez que se llame al oyente. Así, es necesario declarar los arreglos de botones como una variable de instancia.

```
    //*************************************************************

    // Si el usuario hace clic en un botón, la etiqueta de éste cambia a "X" u "O".

    private class Listener implements ActionListener
    {
      public void actionPerformed(ActionEvent e)
      {
        JButton btn = (JButton) e.getSource();
        if (btn.getText().isEmpty())
        {
          btn.setText(xTurn ? "X" : "O");
          xTurn = !xTurn;
        }
      } // end actionPerformed
    } // end class Listener

    //*************************************************************

    public static void main(String[] args)
    {
      new TicTacToe();
    }
  } // end class TicTacToe
```

Figura 17.7*b* Programa juego de gato, parte B.

Verificar un triunfo sólo es necesario cuando el usuario hace clic en un botón. Así, es necesario añadir código para verificar un triunfo al método `actionPerformed` dentro del oyente del botón. Al agregar este código, use diseño arriba-abajo (top-down). En otras palabras, no se preocupe sobre los detalles de nivel bajo; simplemente asuma la tarea. He aquí el método `actionPerformed` actualizado. El código añadido está en seudocódigo:

```
public void actionPerformed(ActionEvent e)
{
  JButton btn = (JButton) e.getSource();
  if (btn.getText().isEmpty())
  {
    btn.setText(xTurn ? "X" : "O");
    if win()                          ┐
    {                                 │
      print winning player            ├ ← seudocódigo
      prepare for new game            │
    }                                 ┘
    else
    {
      xTurn = !xTurn;
    }
  }
} // end actionPerformed
```

El seudocódigo contiene tres tareas: verificar un triunfo, imprimir el ganador y preparar un nuevo juego. La verificación de un triunfo requiere lo más difícil, de modo que esta tarea se pospone momentáneamente. Primero se analizarán las otras dos tareas.

Imprimir el ganador debe ser directo. Simplemente se llama a `JOptionPane.showMessageDialog` con un mensaje de felicitación. El mensaje debe incluir el nombre del ganador, X u O, que puede obtenerse reusando el código del operador condicional, `xTurn ? "X" : "O"`.

Preparar un nuevo juego también debe ser directo. Simplemente la cadena vacía se asigna a las etiquetas de los botones del tablero y `true` se asigna a la variable `xTurn` (X siempre va primero).

Siéntase con confianza para implementar las tareas de impresión del ganador y de preparación de un nuevo juego como un código insertado dentro de la declaración `if` o como métodos de ayuda por separado. Cualquier forma está bien. Pero la tarea de verificación de un ganador definitivamente debe implementarse como un método de ayuda por separado. ¿Por qué? Observe cuán limpiamente se llama a `win` en el seudocódigo de arriba. Esta imagen limpia puede retenerse en el código final de Java sólo si la tarea de verificación de un ganador se implementa como un método, no como un código insertado.

Al implementar el código para el ganador es necesario comprobar que en el arreglo bidimensional de botones haya tres en un renglón, tres en una columna o tres en una diagonal. Normalmente, cuando se accede a un grupo de elementos en un arreglo, debe usarse un ciclo `for`. Quizá sea conveniente usar un ciclo `for` para acceder a los elementos en el primer renglón, otro ciclo `for` para acceder a los elementos en el segundo renglón, etc. Pero para eso se requieren ocho ciclos `for`:

```
for loop for first row
for loop for second row
...
for loop for second diagonal
```

¡Cómo! ¡Ésos son muchos ciclos! ¿Hay una mejor forma de hacer lo anterior? ¿Qué tal aplicar el enfoque opuesto y no usar ciclos `for`? Use una gran declaración `if` como ésta:

```
if (btns[0][0] = X && btns[0][1] = X && btns[0][2] = X) ||
   (btns[1][0] = X && btns[1][1] = X && btns[1][2] = X) ||
    ...
   (btns[0][2] = X && btns[1][1] = X && btns[2][0] = X)
  return true
else
  return false
end-if
```

Esto funciona bien, pero si el lector se molesta por la longitud de la condición `if` (ocho veces más larga), quizá sea conveniente intentar lo siguiente. Use un ciclo `for` para todos los renglones, un ciclo `for` para todas las columnas, y una declaración `if` para las dos diagonales:

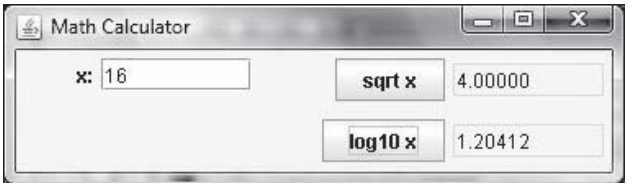
```
for (i=0; i<3; i++)
  if (btns[i][0] = X && btns[i][1] = X && btns[i][2] = X)
    return true
  end-if
end-for

for (j=0; j<3; j++)
  if (btns[0][j] = X && btns[1][j] = X && btns[2][j] = X)
    return true
  end-if
end-for

if (btns[0][0] = X && btns[1][1] = X && btns[2][2] = X) ||
   (btns[0][2] = X && btns[1][1] = X && btns[2][0] = X)
  return true
end-if

return false
```

De las tres soluciones, aquí se prefiere la última porque se tiene la sensación de que el código es más comprensible.

Para hacer que el programa juego de gato sea más del "mundo real", tal vez convenga proporcionar funcionalidad adicional. En particular, verificar un "juego de gato", que es cuando el tablero está lleno y no gana nadie. En uno de los proyectos del capítulo se le pide implementar esa funcionalidad.

17.8 Gestores de distribución insertados

Suponga que quiere implementar esta ventana calculadora matemática:



¿Qué tipo de esquema usaría? Contar con un buen esquema de distribución a menudo requiere creatividad. Se le conducirá por el proceso creativo para este ejemplo de calculadora matemática.

Prueba de los diferentes gestores de distribución

Parece que esta ventana calculadora matemática tiene dos renglones y cuatro columnas. Entonces, ¿resulta apropiado un esquema `GridLayout` de dos renglones por cuatro columnas? El gestor `GridLayout` usualmente es adecuado para posicionar componentes en una forma tabular organizada, aunque está limitado por un factor: cada una de sus celdas debe ser del mismo tamaño. Si se usa un esquema `GridLayout` de dos renglones por cuatro columnas para la ventana calculadora matemática, entonces se tendrán ocho celdas del mismo tamaño. Eso está bien para la mayor parte de las celdas, pero no para la celda superior izquierda. Esta celda contendría la etiqueta x: con una etiqueta así de pequeña, se querría una celda relativamente pequeña para la etiqueta. Pero con un esquema `GridLayout`, una "celda relativamente pequeña" no es una opción.

Puesto que el gestor `GridLayout` es menos que ideal, quizá sea conveniente pensar en el gestor `FlowLayout`. Esto podría hacer el trabajo si se usan componentes alineados por la derecha. Pero entonces es en función de que el usuario no modifique la ventana. Si el usuario hace más ancha la ventana, entonces el botón log10 x se irá hasta la línea superior, lo cual es indeseable. Así, el gestor `FlowLayout` también es menos que ideal. El gestor `BorderLayout` ni siquiera se aproxima. Entonces, ¿cuál es la solución?

Uso de un esquema de distribución insertado

Delegue. Al surgir distribuciones para ventanas más complicadas, a menudo la clave consiste en insertar gestores de distribución dentro de otros gestores de distribución. Primero se abordará el gestor de distribución externo. Para la ventana calculadora matemática, se quiere que la entrada esté a la izquierda y la salida a la derecha. Estos dos entes tienen aproximadamente el mismo ancho, de modo que tiene sentido considerar el uso de un `GridLayout` de dos columnas para aquéllos. La columna izquierda debe contener los componentes de entrada: la etiqueta x y la ventana de texto de entrada. La columna derecha debe contener los componentes de salida: el botón de raíz cuadrada y la ventana de texto de salida y el botón de logaritmo y la ventana de texto de salida. Se quiere organizar los componentes de salida de modo que los datos de raíz cuadrada estén arriba de los datos de logaritmo. Esto significa utilizar dos renglones para el `GridLayout`. Observe el esquema `GridLayout` de dos renglones por dos columnas en la figura 17.8.

Como se sabe, los gestores `GridLayout` sólo permiten un componente por celda. Pero en la figura 17.8 se muestran dos componentes en la celda superior izquierda y dos componentes en la celda superior derecha. Para implementar este esquema de organización, es necesario agrupar cada uno de los pares de componentes en sus contenedores por separado. Y para obtener la distribución idónea, es necesario aplicar gestores de distribución a cada uno de estos contenedores. El contenedor de la celda superior izquierda usa un gestor `FlowLayout` con alineación al centro. Los contenedores de las celdas derechas usan un gestor `FlowLayout` con alineación a la derecha. Ya está: gestores de distribución dentro de un gestor de distribución. Nada mal, ¿eh?

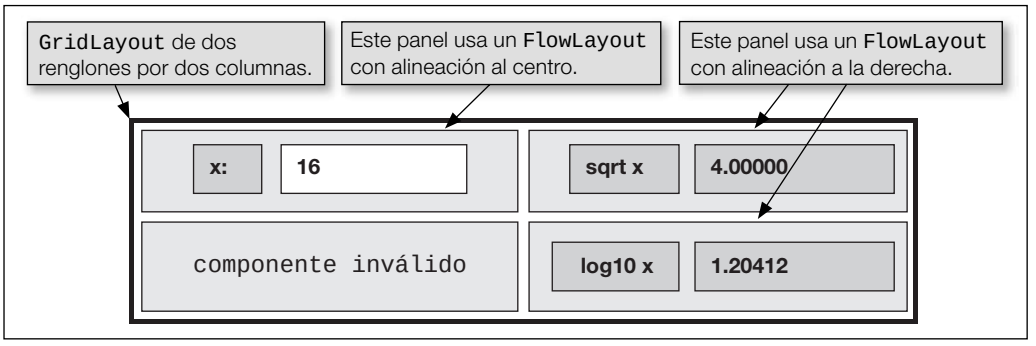


Figura 17.8 `GridLayout` con paneles `FlowLayout` insertados en tres de las celdas.

Cuando se tiene una ventana no trivial, es muy común que haya gestores de distribución insertados. Y cuando esto ocurre, puede llevarse una cantidad considerable de ajustes pequeños para que la ventana quede bien. A pesar de lo anterior, el uso de gestores de distribución insertados sigue siendo mucho más fácil que posicionar manualmente los componentes con valores de pixeles como antaño. En la siguiente sección se proporcionan detalles sobre los contenedores para los gestores de distribución insertados.

17.9 Clase `JPanel`

Antes de proseguir con la implementación del programa de la calculadora matemática, es necesario analizar la clase `JPanel`. Un objeto contenedor `JPanel` es un área de almacenamiento genérica para componentes. Si se tiene una ventana complicada con muchos componentes, tal vez convenga compartir los componentes al colocar grupos de éstos en contenedores `JPanel`. Los contenedores `JPanel` son particularmente útiles con ventanas `GridLayout` y `BorderLayout` porque cada compartimiento en estas distribuciones puede almacenar un solo componente. Si se requiere que un compartimiento almacene más de un componente, se hace que ese componente sea un contenedor `JPanel`, y coloque múltiples componentes en el contenedor `JPanel`.

Implementación

Como puede recordar, las clases GUI que empiezan con *J* provienen del paquete `javax.swing`. De modo que de ahí provienen las clases contenedor `JPanel`, por lo que es necesario importar el paquete `javax.swing` para usar `JPanel`.

Para instanciar un contenedor `JPanel`, use esta sintaxis:

`JPanel <`*`JPanel-reference`*`> = new JPanel(<`*`layout-manager`*`>);`

El argumento <*layout-manager*> es opcional. Si se omite, entonces, por defecto, el alineamiento del gestor `FlowLayout` es al centro.

Así, el gestor de distribución por defecto del contenedor `JPanel` es `FlowLayout`. Prueba rápida: ¿recuerda cuál es el gestor de distribución por defecto del contenedor `JFrame`? Es `BorderLayout`. Esto debe tener sentido cuando el lector se percata de que los contenedores `JFrame` están diseñados para manejar una ventana como un todo. Para la ventana como un todo, el esquema del `BorderLayout` por defecto funciona bien porque sus regiones orientadas por reporte (norte para un encabezado, sur para un pie, centro para el cuerpo principal) coinciden con las necesidades de muchas ventanas de programas. Por otra parte, los contenedores `JPanel` están diseñados para manejar compartimientos dentro de una ventana. Para tales compartimientos, el esquema del `FlowLayout` por defecto funciona bien porque su forma libre coincide con las necesidades de muchos compartimientos.

Adición de componentes a `JPanel`

Después de instanciar un `JPanel`, conviene agregarle componentes. La adición de componentes a un `JPanel` es lo mismo que agregar componentes a un `JFrame`. Se llama el método `add`. Como se sabe, el

método `add` funciona distinto para los diferentes gestores de disposición. Si su `JPanel` usa un gestor `FlowLayout` o un gestor `GridLayout`, el método `add` se llama así:

<*JPanel-reference*>.add(<*component*>);

Si su `JPanel` usa un gestor `BorderLayout`, es necesario añadir un segundo argumento para especificar la región del componente:

<*JPanel-reference*>.add(<*component*>, <*BorderLayout-region*>);

Adición de `JPanel` a una ventana

Luego de añadir componentes a un `JPanel`, es necesario agregar el `JPanel` a una ventana. Si su ventana usa un gestor `FlowLayout` o un gestor `GridLayout`, el método `add` se llama así:

add(<*JPanel-reference*>);

Si su ventana usa un gestor `BorderLayout`, quizá convenga agregar un segundo argumento para especificar la región del componente:

add(<*JPanel-reference*>, <*BorderLayout-region*>);

En la siguiente sección se volverá al programa de la calculadora matemática. Así se tendrá la oportunidad de ver cómo funciona `JPanel` en el contexto de un programa completo.

17.10 Programa calculadoraMatematica

Vea el listado del programa calculadoraMatematica en las figuras 17.9*a*, 17.9*b* y 17.9*c*. Usted debe leer con detenimiento todo el programa, pero aquí el centro de atención es esencialmente el código relacionado con los paneles.

Del método `createContents` del programa calculadoraMatematica, éste es el código que crea el panel de la celda superior izquierda:

```
xPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
xPanel.add(xLabel);
xPanel.add(xBox);
```

La primera declaración instancia el contenedor `JPanel`. Puesto que el constructor `JPanel` usa por defecto un `FlowLayout` con alineamiento a la derecha, la primera declaración puede escribirse como se muestra a continuación y obtener el mismo resultado:

```
xPanel = new JPanel();
```

Pero aquí se prefiere la declaración original puesto que es autodocumentado. Las declaraciones segunda y tercera agregan al panel la etiqueta `X:` y la ventana de texto de entrada.

Luego, en el método `createContents`, éste es el código que agrega los paneles a la ventana:

```
add(xPanel);
add(xSqrtPanel);
add(new JLabel()); // dummy component
add(xLogPanel);
```

Las declaraciones primera, segunda y cuarta agregan los paneles a las celdas superior izquierda, superior derecha e inferior derecha, respectivamente. La tercera declaración agrega un componente inválido (una etiqueta en blanco) a la celda inferior izquierda. El componente inválido es necesario porque sin él, el `xLogPanel` iría a la celda inferior izquierda, y no es esto lo que se quiere.

Hay una cuestión adicional en este programa que se debe comentar. Observe la llamada al método `String.format` en el método `actionPerformed` en la figura 17.9*c*. El método `String.format` funciona igual que el método `printf`, excepto que en lugar de imprimir un valor formateado, devuelve un valor formateado. En el método `actionPerformed` se llama a `String.format` para recuperar una versión formateada del valor del logaritmo calculado. Específicamente, el especificador de conversión `%7.5f` devuelve un valor de punto flotante con cinco cifras decimales y siete caracteres en total.

```
/*************************************************************
* MathCalculator.java
* Dean & Dean
*
* Este programa usa gestores de distribución insertados para desplegar la
* raíz cuadrada y el logaritmo de un número introducido por el usuario.
*************************************************************/

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

public class MathCalculator extends JFrame
{
  private static final int WIDTH = 380;
  private static final int HEIGHT = 110;

  private JTextField xBox;     // valor de entrada del usuario
  private JTextField xSqrtBox; // raíz cuadrada generada
  private JTextField xLogBox;  // logaritmo generado

  //*********************************************************

  public MathCalculator()
  {
    setTitle("Math Calculator");
    setSize(WIDTH, HEIGHT);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents();
    setVisible(true);
  } // end MathCalculator constructor

  //*********************************************************

  // Crea componentes y los agrega a una ventana.

  private void createContents()
  {
    JPanel xPanel;     // contiene la etiqueta x y su ventana de texto
    JPanel xSqrtPanel; // contiene la etiqueta "sqrt x" y su ventana de texto
    JPanel xLogPanel;  // contiene la etiqueta "log x" y su ventana de texto
    JLabel xLabel;
    JButton xSqrtButton;
    JButton xLogButton;
    Listener listener;

    setLayout(new GridLayout(2, 2));
```

Figura 17.9*a* Programa calculadoraMatematica, parte A.

17.11 Componente `JTextArea`

En el capítulo previo se presentaron algunos componentes GUI, `JLabel`, `JTextField`, `JButton` y `JOptionPane`, que proporcionan funcionalidad básica de entrada/salida. Ahora se presentarán más componentes GUI, `JTextArea`, `JCheckBox`, `JRadioButton` y `JComboBox`, que proporcionan más funcionalidad avanzada de E/S. Se empezará con el componente `JTextArea`.

```
    // Crea el panel x:
    xLabel = new JLabel("x:");
    xBox = new JTextField(8);
    xPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
    xPanel.add(xLabel);
    xPanel.add(xBox);

    // Crea el panel raíz cuadrada:
    xSqrtButton = new JButton("sqrt x");
    xSqrtBox = new JTextField(8);
    xSqrtBox.setEditable(false);
    xSqrtPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
    xSqrtPanel.add(xSqrtButton);
    xSqrtPanel.add(xSqrtBox);

    // Crea el panel logaritmo:
    xLogButton = new JButton("log10 x");
    xLogBox = new JTextField(8);
    xLogBox.setEditable(false);
    xLogPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
    xLogPanel.add(xLogButton);
    xLogPanel.add(xLogBox);

    // Agrega paneles a la ventana:
    add(xPanel);
    add(xSqrtPanel);
    add(new JLabel()); // componente inválido
    add(xLogPanel);

    listener = new Listener();
    xSqrtButton.addActionListener(listener);
    xLogButton.addActionListener(listener);
  } // end createContents

  //*********************************************************

  // Clase interna para cálculos matemáticos.

  private class Listener implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      double x;      // valor numérico para la x introducida por el usuario
      double result; // valor calculado
```

Agrega un componente inválido, de modo que la celda inferior izquierda se llena.

Figura 17.9*b* Programa calculadoraMatematica, parte B.

Interfaz del usuario

El componente `JLabel` funciona bastante bien para desplegar una sola línea de texto. Como se describió en el capítulo 16, este componente puede usarse para mostrar múltiples líneas de texto, pero obtener múltiples líneas requiere enredar el código con etiquetas `<br>` HTML de rompimiento de líneas. La técnica preferida para desplegar múltiples líneas de texto es mediante el uso del componente `JTextArea`. La gran zona en blanco en la figura 17.10 es un componente `JTextArea`. Por cierto, la pequeña región sombreada en la parte inferior de la figura 17.10 es un componente `JCheckBox`. Los componentes `JCheckBox` se describirán en la siguiente sección.

```
        try
        {
          x = Double.parseDouble(xBox.getText());
        }
        catch (NumberFormatException nfe)
        {
          x = -1;   // indica una x inválida
        }

        if (e.getActionCommand().equals("sqrt x"))
        {
          if (x < 0)
          {
            xSqrtBox.setText("undefined");
          }
          else
          {
            result = Math.sqrt(x);
            xSqrtBox.setText(String.format("%7.5f", result));
          }
        } // end if

        else // calcula el logaritmo
        {
          if (x < 0)
          {
            xLogBox.setText("undefined");
          }
          else
          {
            result = Math.log10(x);
            xLogBox.setText(String.format("%7.5f", result));
          }
        } // end else
      } // end actionPerformed
    } // end class Listener

    //*******************************************************

    public static void main(String[] args)
    {
      new MathCalculator();
    } // end main
  } // end class MathCalculator
```

Figura 17.9*c* Programa calculadoraMatematica, parte C.

Implementación

Para crear un componente `JTextArea`, el constructor `JTextArea` se llama así:

`JTextArea` <*JTextArea-reference*> `= new JTextArea(`<*display-text*>`);`

El *display-text* es el texto que aparece en el componente `JTextArea`. Si se omite el argumento del *display-text*, entonces el componente `JTextArea` no despliega nada.

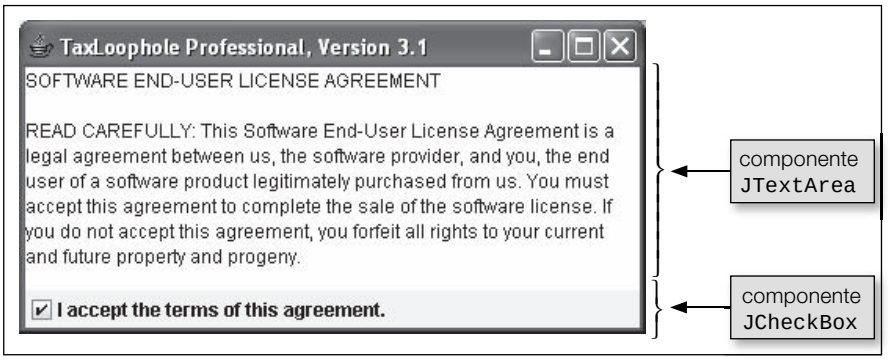


Figura 17.10 Ventana con un componente `JTextArea` y un componente `JCheckBox`.

Métodos

La clase `JTextArea`, como todas las clases de componentes GUI, posee algunos métodos. Éstos son los encabezados y las descripciones de los métodos `JTextArea` más conocidos:

`public String getText()`
Devuelve el texto en la zona de texto.

`public void setText(String text)`
Asigna el texto en la zona de texto.

`public void setEditable(boolean flag)`
Hace editable o no editable el texto.

`public void setLineWrap(boolean flag)`
Enciende o apaga el ajuste de línea.

`public void setWrapStyleWord(boolean flag)`
Especifica si para el ajuste de línea se usan límites de palabra.

Los componentes `JTextArea` son editables por defecto, lo cual significa que a los usuarios se les permite teclear en ellos. Si se desea impedir que los usuarios editen un componente `JTextArea`, es necesario llamar a `setEditable` con un valor de argumento `false`. Hacer lo anterior impide que los usuarios actualicen la zona de texto, pero no evita que los programadores actualicen dicha zona de texto. Los programadores pueden llamar al método `setText` sin importar si la zona de texto es editable o no lo es.

Por defecto, los componentes `JTextArea` tienen apagado el ajuste de línea. Normalmente, conviene encender el ajuste de línea al llamar a `setLineWrap(true)`. Así, las líneas largas automáticamente se ajustan al siguiente renglón, en lugar de desaparecer cuando llegan al límite derecho de la zona de texto.

Para los componentes `JTextArea` con ajuste de línea encendido, la situación por defecto es ejecutar un ajuste de línea en el punto en que el texto se encuentra con el límite derecho de la zona de texto, sin importar que ese punto se encuentre a la mitad de una palabra. Normalmente, conviene evitar este comportamiento draconiano[3] por defecto y hacer que el ajuste de línea ocurra sólo en los límites de palabra. Para cambiar a una política de ajuste de línea en función del límite de la palabra, es necesario llamar a `setWrapStyleWord(true)`.

Ejemplo de acuerdo de licencia

Observe de nuevo el componente `JTextArea` del acuerdo de licencia en la figura 17.10. La figura 17.11 contiene el código asociado con ese componente. A continuación se analizará el código de la figura 17.11.

[3] Una política draconiana es una política estricta o severa. El término proviene de Draco, un militar gobernante ateniense del siglo VII a.C. responsable de elaborar las leyes locales. Las leyes de Draco eran excesivamente severas. Por ejemplo, incluso los delitos menores eran castigados con la pena de muerte.

```
private void createContents()
{
  JTextArea license;
  JCheckBox confirmBox;

  setLayout(new BorderLayout());
  license = new JTextArea(
    "SOFTWARE END-USER LICENSE AGREEMENT\n\n" +
    "READ CAREFULLY: This Software End-User License Agreement" +
    " is a legal agreement between us, the software provider," +
    " and you, the end user of a software product legitimately" +
    " purchased from us. You must accept this agreement to" +
    " complete the sale of the software license. If you do not" +
    " accept this agreement, you forfeit all rights to your" +
    " current and future property and progeny.");
  license.setEditable(false);
  license.setLineWrap(true);
  license.setWrapStyleWord(true);
  confirmBox = new JCheckBox(
    "I accept the terms of this agreement.", true);

  add(license, BorderLayout.CENTER);
  add(confirmBox, BorderLayout.SOUTH);
} // end createContents
```

Figura 17.11 Código que creó el despliegue en la figura 17.10.

Observe el `\n\n` en la llamada al constructor `JTextArea`. Como puede recordar, las `\n` se ignoran dentro del texto `JLabel`. Pero funcionan dentro del texto `JTextArea`. Observe las llamadas `setEditable(false)`, `setLineWrap(true)` y `setWrapStyleWord(true)`. Estas llamadas son comunes para componentes de `JTextArea`.

Observe el color de fondo del componente `JTextArea` para el acuerdo de licencia en la figura 17.10. Es blanco, lo cual contrasta con el resto de la ventana. Si se quiere que la zona de texto destaque, entonces el fondo blanco es apropiado, pero si se desea armonizarlo, entonces es inapropiado. ¿Cómo es posible cambiar su color de fondo de modo que sea armonioso? Más específicamente, ¿cómo puede modificarse el código de modo que la ventana se vea como la figura 17.12? La solución requiere el uso de unos cuantos métodos no mencionados en la lista de arriba de métodos API. Pero en el pasado se han usado métodos para otras necesidades GUI. Trate de imaginar esto antes de seguir leyendo.

Excelente tono con otros métodos API.

Para cambiar el color de fondo de un componente, debe llamarse a `setBackground(<color>)`. Para el componente del acuerdo de licencia, se quiere que su color coincida con el color de la ventana, de

Figura 17.12 Color de fondo modificado para el componente `JTextArea` del acuerdo de licencia.

modo que es necesario llamar a `setBackground` con un valor de color igual al color de fondo de la ventana. Para obtener el color de fondo de la ventana, es necesario llamar a `getContentPane().getBackground()`. Ésta es la solución:

```
license.setBackground(getContentPane().getBackground());
```

17.12 Componente JCheckBox

Interfaz del usuario

Observe la *casilla de verificación* en la parte inferior de la figura 17.12. La casilla de verificación debe usarse si el lector desea presentar una opción. Un componente de casilla de verificación despliega un pequeño cuadrado con una etiqueta a la derecha. Cuando el cuadrado está en blanco, la ventana está sin seleccionar. Cuando la ventana contiene una marca de verificación, la ventana está seleccionada. Los usuarios hacen clic en la casilla de verificación para activar entre seleccionada y no seleccionada.

Implementación

Para crear un componente de casilla de verificación, el constructor `JCheckBox` se llama así:

```
JCheckBox <JCheckBox-reference> = new JCheckBox(<label>, <selected>);
```

El argumento *label* especifica el texto que aparece a la derecha de la casilla de verificación. Si se omite el argumento de la etiqueta, entonces a la derecha de la casilla de verificación no aparece ningún texto. El argumento *selected* especifica si la casilla de verificación se ha seleccionado inicialmente: `true` significa seleccionada y `false`, no seleccionada. Si se omite el argumento seleccionado, entonces la casilla de verificación no es seleccionada inicialmente.

He aquí cómo se creó la casilla de verificación en la ventana del acuerdo de licencia:

```
confirmBox = new JCheckBox("I accept the terms of this agreement.", true);
```

Métodos

Éstos son los encabezados y las descripciones API para los métodos `JCheckBox` más conocidos:

`public boolean isSelected()`
Devuelve `true` si la casilla de verificación es seleccionada, y `false` en caso contrario.

`public void setVisible(boolean flag)`
Hace visible o invisible la casilla de verificación.

`public void setSelected(boolean flag)`
Hace seleccionada o no seleccionada la casilla de verificación.

`public void setEnabled(boolean flag)`
Posibilita o imposibilita la casilla de verificación.

`public void addActionListener(ActionListener listener)`
Agrega un oyente a la casilla de verificación.

Los métodos `isSelected` y `setVisible` son directos, pero los otros tres métodos requieren una explicación. Se empezará con `setSelected`. ¿Por qué habría de llamarse `setSelected` y ajustar el estado de selección de una casilla de verificación? Porque tal vez se quiera que la entrada de un usuario impacte en la entrada de otro usuario. Por ejemplo, en la figura 17.13, la selección del usuario de la versión estándar contra la versión ajustable[4] debe impactar en las selecciones de la casilla de verificación. Más específicamente, si el usuario selecciona la opción Standard, las selecciones de la casilla de verificación deben ir a sus decisiones "estándar". Como puede verse en la ventana izquierda en la figura 17.13,

[4] Los círculos Standard y Custom en la parte superior de la figura 17.13 se denominan botones de radio. Los componentes `JradioButton` se describirán en la siguiente sección.

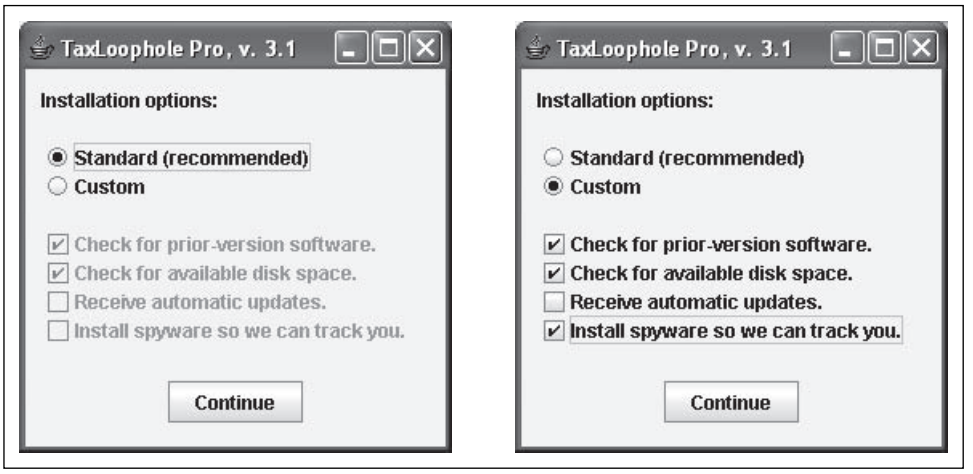


Figura 17.13 Ejemplo que ilustra los métodos `setSelected` y `setEnable` de `JCheckBox`.

los establecimientos estándar para las casillas de verificación son las dos seleccionadas en la parte superior y las dos no seleccionadas en la parte inferior. Para que su programa seleccione las dos casillas de verificación superiores, estas dos casillas de verificación deben llamar a `setSelected(false)`. Para que su programa desactive una casilla de verificación, ésta debe llamar a `setEnable(false)`.

¿Por qué razón habría de llamarse a `setEnable(false)` y desactivar una casilla de verificación? Porque tal vez quiera impedirse que el usuario modifique el valor de esa casilla de verificación. Por ejemplo, si el usuario escoge la opción Standard como se muestra en la ventana izquierda en la figura 17.13, las selecciones de la casilla de verificación deben establecerse en sus ajustes estándar (como se explicó arriba), y luego cada casilla de verificación debe llamar a `setEnabled(false)`. De esa forma, el usuario no puede hacer ningún cambio a los valores de la configuración de la casilla de verificación. En la ventana izquierda en la figura 17.13, observe que las casillas de verificación son grises. Se dice que tienen grisalla. Ésta es la forma estándar de GUI para indicar al usuario que algo está desactivado.

Oyentes Check Box

Con un componente `JButton`, casi siempre conviene contar con un oyente asociado. Pero con un componente `JCheckBox`, es posible decidir si se quiere o no un oyente asociado. Si se tiene una casilla de verificación sin oyente, entonces la casilla de verificación simplemente sirve como un ente de entrada. Si éste es el caso, entonces el valor de la casilla de verificación (verificado o no), típicamente se lee y procesa cuando el usuario hace clic en un botón. Por otra parte, si el lector quiere que algo ocurra de inmediato, justo cuando el usuario selecciona una casilla de verificación, entonces es necesario agregar un oyente al componente de la casilla de verificación. Suponga que se tiene una casilla de verificación Green Background. Si se desea que el color de fondo de la ventana cambie a verde justo cuando el usuario hace clic en la casilla de verificación, debe agregarse un oyente a la casilla de verificación. La sintaxis para agregar un oyente a un componente `JCheckBox` es la misma sintaxis que para agregar un oyente a un componente `JButton`. Se proporciona un oyente que implemente la interfaz `ActionListener` y luego se agrega el oyente a la componente `JCheckBox` al llamar a `addActionListener`.

Tenga en cuenta que Sun proporciona una interfaz oyente alterna para la componente `JCheckBox`: la interfaz `ItemListener`. Un `ActionListener` escucha que el usuario haga clic en una casilla de verificación. Un `ItemListener` escucha un *cambio de estado*; es decir, escucha que una casilla de verificación cambie de seleccionada a no seleccionada o viceversa. El cambio de estado de una casilla de verificación se activa cuando un usuario hace clic en la casilla de verificación o cuando el programa llama a `setSelected` con un valor distinto al valor actual. Puesto que la interfaz `ActionListener` es la interfaz preferida para la mayor parte de las situaciones, aquí se sigue esta preferencia al implementar los oyentes `JCheckBox`. Una vez que se obtengan los componentes `JRadioButton` y `JComboBox` en las siguientes secciones, seguirá utilizándose la interfaz `ActionListener`, no la interfaz `ItemListener`.

```
private class Listener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    if (e.getSource() == standard) // standard option chosen
    {
      prior.setEnabled(false);
      diskSpace.setEnabled(false);
      updates.setEnabled(false);
      spyware.setEnabled(false);
      prior.setSelected(true);
      diskSpace.setSelected(true);
      updates.setSelected(false);
      spyware.setSelected(false);
    }
    else                            // custom option chosen
    {
      prior.setEnabled(true);
      diskSpace.setEnabled(true);
      updates.setEnabled(true);
      spyware.setEnabled(true);
    }
  } // end actionPerformed
} // end Listener
```

Figura 17.14 Código Listener para las ventanas de opciones de instalación para la figura 17.13.

Ejemplo de opciones de instalación

Ahora ya es el momento de poner en práctica estos conceptos de casillas de verificación al mostrar algo de código. Observe de nuevo las ventanas de opciones de instalación en la figura 17.13. En la figura 17.14 se proporciona el código oyente asociado con estas ventanas. A continuación se analizará el código. En la condición de la declaración `if`, se verifica si se seleccionó la opción estándar. En caso afirmativo, las casillas de verificación se deshabilitan al llamar a `setEnabled(false)` para cada casilla de verificación. Luego, las casillas de verificación se asignan a sus posiciones estándar al llamar a `setSelected(true)` o a `setSelected(false)` para cada casilla de verificación. En el bloque `else`, se maneja la opción seleccionada. Las casillas de verificación se habilitan al hacer que cada ventana llame a `setEnabled(true)`. Esto permite que el usuario controle si la ventana está seleccionada o no.

17.13 Componente `JRadioButton`

Interfaz del usuario

Observe los círculos en la figura 17.13. Se denominan *botones de radio*. Los componentes de un `JRadioButton` despliegan un pequeño círculo con una etiqueta a su derecha. Cuando el círculo está en blanco, el botón de radio no está seleccionado. Cuando el círculo contiene un punto grande, el botón de radio está seleccionado.

Según la descripción hecha, los botones de radio se parecen bastante a las casillas de verificación. Despliegan una forma y una etiqueta, y siguen la pista de si algo está encendido o apagado. La diferencia fundamental entre los botones de radio y las casillas de verificación es que los botones de radio siempre aparecen en grupos. Y dentro de un grupo de botones de radio, sólo es posible seleccionar un botón a la vez. Si un usuario hace clic en un botón de radio no seleccionado, entonces el botón del usuario devuelve seleccionado y el botón previamente seleccionado en el grupo devuelve no seleccionado. Si un usuario hace clic en un botón de radio seleccionado, no ocurre ningún cambio (es decir, el botón en que se hizo

clic permanece sin cambio). En contraste, si un usuario hace clic en una casilla de verificación, ésta cambia su estado de seleccionada a no seleccionada.

Implementación

Para crear un componente botón de radio, el constructor `JRadioButton` se llama así:

```
JRadioButton <JRadioButton-reference> =
 new JRadioButton(<label>, <selected>);
```

El argumento *label* especifica el texto que aparece a la derecha del círculo del botón de radio. Si se omite el argumento de la etiqueta, entonces a la derecha del círculo del botón de radio no aparece ningún texto. El argumento *selected* especifica si el botón de radio estaba inicialmente seleccionado: `true` significa seleccionado, `false` significa no seleccionado. Si se omite el argumento seleccionado, entonces el botón de radio estaba inicialmente no seleccionado.

Este ejemplo muestra cómo se crearon los botones de radio `standard` y `custom` en el programa de opciones de instalación:

```
standard = new JRadioButton("Standard (recommended)", true);
custom = new JRadioButton("Custom");
```

Para habilitar la funcionalidad del único botón seleccionado a la vez de un grupo de botones de radio, se crea un objeto `ButtonGroup` y se le agregan componentes individuales del botón de radio. He aquí cómo:

```
ButtonGroup <ButtonGroup-reference> = new ButtonGroup();
<ButtonGroup-reference>.add(<first-button-in-group>);
...
<ButtonGroup-reference>.add(<last-button-in-group>);
```

El siguiente ejemplo muestra cómo se crearon los botones de radio para los botones de radio `standard` y `custom` en el programa de opciones de instalación:

```
ButtonGroup rbGroup = new ButtonGroup();
rbGroup.add(standard);
rbGroup.add(custom);
```

Después de agregar botones de radio a un grupo de botones, es necesario agregarlos a un contenedor. Los botones de radio funcionan igual que los otros componentes en términos de agregarlos a un contenedor. El método `add` del contenedor se llama así:

```
add(<first-button-in-group>);
...
add(<last-button-in-group>);
```


Lo anterior es mucho agregar. Es necesario agregar a cada botón dos veces: una a un grupo de botones de radio y una a un contenedor. Si al lector le agradan los atajos, quizás esté pensando: ¿por qué Java me hace agregar los botones de radio individuales al contenedor? ¿Por qué no se agregan automáticamente cuando se agregan al grupo de botones de radio? Agregar los botones por separado desde el grupo de botones proporciona libertad para posicionar los botones. Si se quiere hacer esto, es posible colocarlos incluso en paneles distintos.

Puesto que la clase `JRadioButton` empieza con *J*, el lector puede asumir correctamente que está definida en el paquete `javax.swing`. Pero ¿qué ocurre con la clase `ButtonGroup`? Aun cuando no empieza con *J*, también está definida en el paquete `javax.swing`.

Métodos

Éstos son los encabezados y las descripciones API para los métodos `JRadioButton` más conocidos:

`public boolean isSelected()`

Devuelve `true` si el botón de radio es seleccionado, y `false` en caso contrario.

`public void setSelected(boolean flag)`
Selecciona el botón de radio si el argumento es `true`. No hace nada si el argumento es `false`.

`public void setEnabled(boolean flag)`
Habilita o no el botón de radio. En caso de habilitarlo, responde a clics en el ratón.

`public void addActionListener(ActionListener listener)`
Agrega un oyente al botón de radio.

Estos mismos métodos se describieron en la sección de `JCheckBox`. Sólo uno requiere atención adicional: el método `setSelected`. Para comprender cómo funciona el método `setSelected`, primero es necesario comprender por completo la forma en que un usuario interactúa con un grupo de botones de radio. Para seleccionar un botón de radio, un usuario hace clic en él. Esto hace que el botón de radio se vuelva seleccionado y que todos los demás botones de radio en el grupo se vuelvan no seleccionados. Para seleccionar un botón de radio desde el programa, se cuenta con la llamada `setSelected(true)` del botón de radio. Esto hace que el botón de radio se vuelva seleccionado y que todos los demás botones de radio en el grupo se vuelvan no seleccionados. Como ya se mencionó, no hay forma de que el usuario elija como no seleccionado un botón. En forma semejante, no hay ninguna forma de que un programa elija como no seleccionado un botón. Ésta es la razón de que al llamar a `setSelected(false)` no ocurra nada. Compila y ejecuta, pero no hace nada que modifique el estado seleccionado o no seleccionado de los botones.

17.14 Componente `JComboBox`

Interfaz del usuario

Un *cuadro combo* permite al usuario seleccionar un artículo de una lista de artículos. Los cuadros combo a menudo se denominan *listas de cortina* porque si un usuario hace clic en un flecha dirigida hacia debajo de una caja combo, desde el despliegue original aparece una lista de artículos a elegir. Luego, si un usuario hace clic en una de las opciones de la lista de cortina, ésta desaparece y sólo permanece desplegado el objeto elegido. Para tener una mejor idea de lo que se está hablando, observe un cuadro combo para seleccionar un día en la figura 17.15.

Los cuadros combo y los botones de radio son semejantes en el sentido de que ambos permiten que el usuario seleccione un artículo de una lista de artículos. Pero un cuadro combo ocupa menos espacio en una ventana. Entonces, si se tiene una larga lista de artículos de los cuales escoger y se quiere ahorrar espacio, debe usarse un cuadro combo en lugar de un grupo de botones de radio.

Implementación

La creación de un cuadro combo es un proceso de dos pasos. Primero, se instancia un arreglo de opciones en una lista. Luego, el arreglo se usa como parte de una instanciación `JComboBox`. Ésta es la sintaxis para la instanciación de un `JComboBox`:

`JComboBox <`*JComboBox-reference*`> = new JComboBox(<`*array-of-list-options*`>);`

El siguiente ejemplo muestra cómo se creó el cuadro combo en la figura 17.15:

```
String[] days =
  {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"};
daysBox = new JComboBox(days);
```

La primera vez que aparece un desplegado de un cuadro combo, se selecciona el primer artículo en el arreglo. Así, en el ejemplo de arriba, Lunes se selecciona cuando el cuadro combo aparece por primera vez.

Métodos

Éstos son los encabezados y las descripciones API para los métodos `JComboBox` más conocidos:

`public void setVisible(boolean flag)`
Hace visible o invisible al cuadro combo.

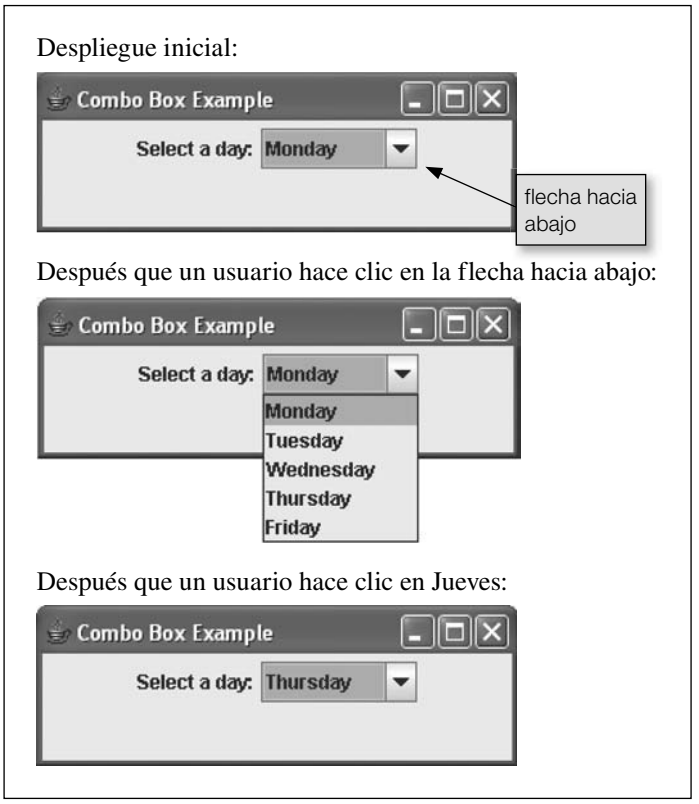


Figura 17.15 Ejemplo de cuadro combo para seleccionar un día.

`public void setEditable(boolean flag)`
Hace editable o no editable la porción superior del cuado combo.

`public Object getSelectedItem()`
Devuelve el artículo seleccionado actualmente.

`public void setSelectedItem(Object item)`
Cambia el artículo seleccionado actualmente al artículo que ha pasado.

`public int getSelectedIndex()`
Devuelve el índice del artículo seleccionado actualmente.

`public void setSelectedIndex(int index)`
Cambia el artículo seleccionado actualmente al artículo en la posición dada por el índice.

`public void addActionListener(ActionListener listener)`
Agrega un oyente al cuadro combo.

Los métodos `setVisible` y `addActionListener` ahora ya deben resultar conocidos. Los otros métodos son nuevos y requieren una explicación adicional. Se empezará con `setEditable`. Si un cuadro combo llama a `setEditable(true)`, la parte superior del cuadro combo se vuelve editable. Esto significa que un usuario puede introducir texto en el cuadro como si fuese un componente ventana de texto. Además, el usuario puede usar la porción de cortina del cuadro combo igual que siempre. Los cuadros combo se denominan "combo" por "combinación" porque son capaces de implementar una mezcla de componentes: parte de listas de cortina, parte de ventana de texto. Pero la mayoría de los programadores no se ocupan de la capacidad de ventana de texto de los cuadros combo. Suelen adherirse al comportamiento por defecto, donde la porción superior del cuadro combo no es editable.

El método `getSelectedItem` devuelve el artículo seleccionado actualmente. Por ejemplo, he aquí cómo es posible recuperar el artículo `daysBox` seleccionado actualmente y almacenarlo en una variable `favoriteDay`:

```
String favoriteDay = (String) daysBox.getSelectedItem();
```

¿De qué se trata el operador tipo cast `(String)`? El método `getSelectedItem` se define de modo que cuente con un tipo de retorno `Object`. En consecuencia, si no hay operador tipo cast, el compilador vería que un `Object` a la derecha se asigna a un `String` en la izquierda, lo cual generaría un error de compilación. Pero sí hay un operador tipo cast, de modo que el compilador ve que un `String` a la derecha se asigna a un `String` en la izquierda, lo cual hace feliz al compilador.

Si el lector desea seleccionar programáticamente una opción de un cuadro combo, llame a `setSelectedItem` y pase el artículo que desea seleccionar. Por ejemplo, para seleccionar `Friday` del componente `daysBox`, haga lo siguiente:

```
daysBox.setSelectedItem("Friday");
```

Normalmente, `setSelectedItem` se llama con un argumento que coincide con uno de los artículos del cuadro combo. Pero no siempre es así. Si se desea limpiar el cuadro combo de modo que no se seleccione ninguna opción, llame a `setSelectedItem(null)`. Si `setSelectedItem` se llama con un artículo diferente (que sea no nulo y no sea un artículo del cuadro combo), entonces no ocurre nada. Bueno, en realidad no ocurre nada si se trata de un cuadro combo estándar. Pero si es un cuadro combo editable, entonces el artículo que se ha pasado se ubica en la porción editable del cuadro combo.

Hay dos formas para acceder artículos en un cuadro combo: usar nombres del artículo o usar índices del artículo. Los métodos `getSelectedItem` y `setSelectedItem` usan nombres del artículo. Los métodos `getSelectedIndex` y `setSelectedIndex` usan índices del artículo. Por ejemplo, observe cómo este fragmento de código llama a `setSelectedIndex` con un valor de índice igual a 2:

```
String[] days =
  {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"};
daysBox = new JComboBox(days);
daysBox.setSelectedIndex(2);
```

Puesto que los cuadros combo almacenan sus artículos en arreglos, los valores de índice de los cuadros combo se basan en cero. En consecuencia, en el fragmento de código de arriba, *Monday* es 0, *Tuesday* es 1 y *Wednesday* es 2. Por tanto, `daysBox.setSelectedIndex(2)` cambia el artículo seleccionado a *Wednesday*.

Indexar ayuda a procesar los datos.

Ahora, un rompecabezas: dado el fragmento de código de arriba, ¿cómo puede cambiarse el día actualmente seleccionado al día siguiente? Se realiza algo de operaciones aritméticas, como esto:

```
daysBox.setSelectedIndex(daysBox.getSelectedIndex() + 1);
```

17.15 Ejemplo aplicación de tarea

En esta sección se pone en práctica lo aprendido en las tres secciones previas. Se presenta un programa completo que usa casillas de verificación, botones de radio y un cuadro combo. El programa implementa una forma de solicitud de empleo. Si el usuario introduce valores indicadores de un buen empleado, el programa despliega un mensaje alentador ("Gracias por presentar su solicitud. Lo buscaremos después de su información"). Estudie la sesión de muestra en la figura 17.16 para tener una mejor idea de lo que se está hablando.

Consulte los listados del programa JobApplication en las figuras 17.17*a*, 17.17*b* y 17.17*c*. El lector debe estudiar detenidamente todo el programa, particularmente el código del oyente, aunque aquí la atención se dirige sólo a la parte más importante: el diseño de la distribución.

Aquí se dedicó bastante tiempo a la distribución del programa JobApplication para que las cosas se vieran bien. Inicialmente, se pensó que funcionaría un gestor `GridLayout` de una sola columna. Se

agregó un componente por celda y se añadieron tres componentes de relleno (empty `JLabel`) para crear separaciones entre las cuatro distintas zonas de entrada. Se pensó que ese plan conduciría a la distribución mostrada en la parte izquierda de la figura 17.18. Desafortunadamente, cuando se introdujo el código, el programa real produjo la distribución que se muestra en la imagen derecha en la figura 17.18. Hay tres problemas con la distribución real: el botón Submit es demasiado ancho, faltan las dos separaciones superiores y los componentes tocan el límite izquierdo. A continuación se analizará la forma de resolver estos problemas.

Si al principio no tiene éxito, inténtelo de nuevo.

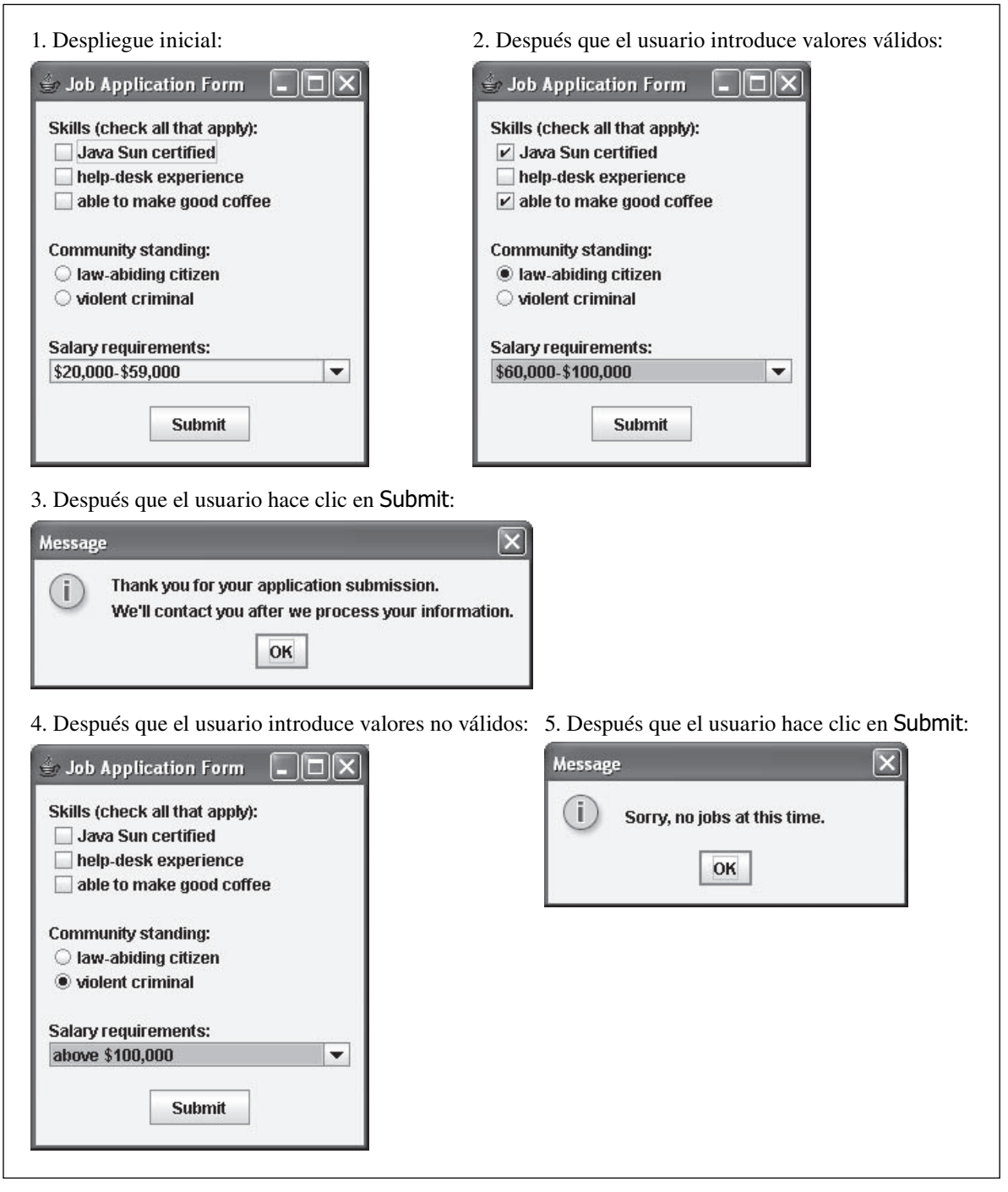


Figura 17.16 Sesión muestra para el programa JobApplication.

```
/*************************************************************
* JobApplication.java
* Dean & Dean
*
* Este programa implementa cuestiones relacionadas con una solicitud de
* empleo con casillas de verificación, botones de radio y cuadros combo.
*************************************************************/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;    // para EmptyBorder

public class JobApplication extends JFrame
{
  private static final int WIDTH = 250;
  private static final int HEIGHT = 300;

  private JCheckBox java;       // ¿Certificado por Java Sun?
  private JCheckBox helpDesk;   // ¿Experiencia en oficina?
  private JCheckBox coffee;     // ¿Hace buen café?
  private JRadioButton goodCitizen, criminal;
  private JComboBox salary;
  private String[] salaryOptions =
    {"$20,000-$59,000", "$60,000-$100,000", "above $100,000"};
  private JButton submit;       // submit the application

  //*********************************************************

  public JobApplication()
  {
    setTitle("presente la solicitud");
    setSize(WIDTH, HEIGHT);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents();
    setVisible(true);
  } // end JobApplication constructor

  //*********************************************************

  // Crea componentes y los agrega a la ventana.
```

Figura 17.17*a* Programa JobApplication, parte A.

Problema 1: El botón Submit es demasiado ancho

Inserte otro gestor.

Como recordará del capítulo previo, los botones se expanden si se agregan directamente a una celda `GridLayout`. Eso explica lo ancho del botón Submit. El problema puede resolverse insertando un panel `FlowLayout` en el área del botón Submit, y luego agregando el botón Submit al panel `FlowLayout`. Con un gestor `FlowLayout`, los botones no se expanden; conservan su tamaño natural.

Problema 2: Faltan las dos separaciones

En el primer corte del programa se usó este código para implementar los componentes de relleno:

```
JLabel filler = new JLabel();
...
add(filler);
```

```
private void createContents()
{
  ButtonGroup radioGroup;

  // Nota:
  // la implementación más directa consiste en usar un gestor
  // GridLayout para el JFrame y agregar todos los componentes
  // a sus celdas. Eso no funciona bien porque:
  // 1) No es posible aplicar un margen a JFrame.
  // 2) El panel de botones es más alto que los otros componentes.

  // Se requiere windowPanel para la separación del panel del sur y el margen exterior.
  JPanel windowPanel = new JPanel(new BorderLayout(0, 10));
  windowPanel.setBorder(new EmptyBorder(10, 10, 10, 10));

  // CenterPanel contiene todos los componentes, excepto button
  JPanel centerPanel = new JPanel(new GridLayout(11, 1));

  // Se requiere un panel para el botón, de modo que el alineamiento puede ser al centro
  JPanel southPanel = new JPanel(new FlowLayout());

  java = new JCheckBox("Java Sun certified");
  helpDesk = new JCheckBox("help-desk experience");
  coffee = new JCheckBox("able to make good coffee");
  goodCitizen = new JRadioButton("law-abiding citizen");
  criminal = new JRadioButton("violent criminal");
  radioGroup = new ButtonGroup();
  radioGroup.add(goodCitizen);
  radioGroup.add(criminal);
  salary = new JComboBox(salaryOptions);
  submit = new JButton("Submit");
  submit.addActionListener(new ButtonListener());

  centerPanel.add(new JLabel("Skills (check all that apply):"));
  centerPanel.add(java);
  centerPanel.add(helpDesk);
  centerPanel.add(coffee);
  centerPanel.add(new JLabel());  // filler
  centerPanel.add(new JLabel("Community standing:"));
  centerPanel.add(goodCitizen);
  centerPanel.add(criminal);
  centerPanel.add(new JLabel());  // filler
  centerPanel.add(new JLabel("Salary requirements:"));
  centerPanel.add(salary);

  windowPanel.add(centerPanel, BorderLayout.CENTER);
  southPanel.add(submit);
  windowPanel.add(southPanel, BorderLayout.SOUTH);
  add(windowPanel);
} // end createContents
```

Figura 17.17*b* Programa JobApplication, parte B.

```
...
add(filler);
...
add(filler);
```

Sólo se instanció una etiqueta y se usó tres veces. Al lector le gusta reusar, ¿no es cierto? Bien, al gestor de distribución, no. El gestor de distribución ve sólo un objeto y por ello sólo hace una celda. No hace

```
  //************************************************************

  // Lee valores introducidos y despliega un mensaje idóneo.

  private class ButtonListener implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      if (
        (java.isSelected() || helpDesk.isSelected()
                           || coffee.isSelected()) &&

        (goodCitizen.isSelected()) &&
        (!salary.getSelectedItem().equals("above $100,000")))
      {
        JOptionPane.showMessageDialog(null,
          "Gracias por su solicitud.\n" +
          "Lo buscaremos después de procesar su información.");
      }
      else
      {
        JOptionPane.showMessageDialog(null,
          "Lo sentimos; por el momento no hay puestos disponibles.");
      }
    } // end actionPerformed
  } // end class ButtonListener

  //************************************************************

  public static void main(String[] args)
  {
    new JobApplication();
  }
} // end class JobApplication
```

Figura 17.17*c* Programa JobApplication, parte C.

Use objetos por separado.

celdas para las dos primeras llamadas `add(filler)`; sólo hace una celda para la última llamada `add(filler)`. Este problema puede resolverse usando tres objetos anónimos `JLabel` como esto:

```
add(new JLabel());
...
add(new JLabel());
...
add(new JLabel());
```

Problema 3: Los componentes tocan el límite izquierdo

Por defecto, los contenedores no tienen márgenes. Entonces, si un contenedor posee componentes alineados a la izquierda, estos componentes tocan el límite izquierdo. Esto explica las espantosas líneas en el límite izquierdo en la parte derecha de la figura 17.18. Es posible añadir un margen al llamar a `setBorder` así:

<*container*>.setBorder(new EmptyBorder(<*top*>, <*left*>, <*bottom*>, <*right*>));

Al llamar a `setBorder`, es necesario pasar a un objeto como un argumento. Hay varios tipos distintos de clases de límites. La clase `EmptyBorder` debe usarse porque un límite vacío produce un margen,

que es lo que se desea. Al llamar al constructor `EmptyBorder`, es necesario pasar valores de pixeles en valores de 10 pixeles para todos los cuatro lados de los límites:

```
windowPanel.setBorder(new EmptyBorder(10, 10, 10, 10));
```

No olvide que la clase `EmptyBorder` está en el paquete `javax.swing.border`, de modo que si desea crear un límite vacío, es necesario importar este paquete.

Quizá piense que el método `setBorder` funciona para todos los contenedores. No es así. Funciona para el contendor `JPanel`, pero no para el contenedor `JFrame`. En consecuencia, es necesario agregar un panel contenedor `JPanel` a la ventana `JFrame` de `JobApplication` y llamar a `setBorder` desde el contenedor `JPanel`.

Use un panel.

¿Qué tipo de gestor de distribución es idóneo para el contenedor `JPanel`? Si se usa un gestor `GridLayout`, eso funciona bien, pero no mucho. Con un `GridLayout`, todos los renglones tienen la misma altura. En la figura 17.16 observe cómo el botón Submit es ligeramente más alto que los otros componentes. La altura agregada del botón Submit proporciona un elemento visual para la importancia del botón. Para dar cabida a un botón más alto que los demás componentes, use un gestor `BorderLayout`. Agregue el panel del botón a la región sur y agregue todos los demás componentes a la región centro. En realidad puesto que la región centro permite sólo un componente, es necesario agregar los componentes a un panel `GridLayout` y luego agregar el panel `GridLayout` a la región centro.

Documente códigos difíciles

El diseño de la distribución del programa `JobApplication` es más bien complicado y de alguna manera no intuitivo. Si alguna vez el lector escribe códigos complicados y no intuitivos, debe documentarlos con comentarios detallados. De no hacerlo, entonces alguien (quizás usted) puede perder tiempo tratando de entenderlos. Observe todos los comentarios para las declaraciones en los paneles en la figura 17.17*b*. Esos comentarios son útiles para clarificar el código del diseño de la distribución.

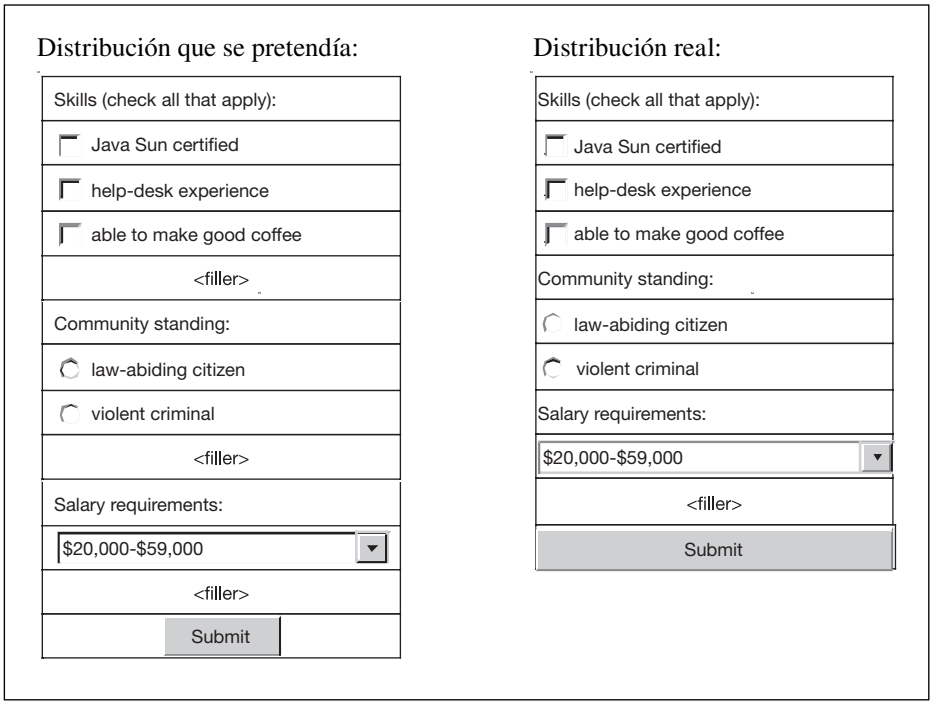


Figura 17.18 Distribuciones pretendida y real con un esquema `GridLayout` de 13 renglones por una columna.

17.16 Más componentes de Swing

En este capítulo y en el capítulo previo se aprendió bastante sobre la biblioteca Swing. Suficiente para levantarse y ejecutar las necesidades básicas de GUI. Si el lector decide que quiere aprender más, debe consultar el sitio Sun API de Java en la red. En particular, consulte esta página en el sitio Sun API de Java en la red:

http://java.sun.com/docs/books/tutorial/uiswing/components/componentlist.html

Contiene ejemplos gráficos de todos los componentes Swing estándares y vínculos con información más detallada. Al analizar detalladamente esa página ahora, el lector sabrá lo que está disponible.

Menús y paneles de desplazamiento

Como primer intento por aprender por su cuenta los componentes Swing, se recomienda que analice las clases `JMenuBar`, `JMenu` y `JMenuItem` en el sitio Sun en la red. Estas clases permiten agregar una *barra de menú* y *menús* en la parte superior de la ventana. También analice la clase `JScrollPane`. Permite crear un contenedor desplazable. Vea la figura 17.19. Ahí se muestra una ventana con una barra de menú y una barra de desplazamiento. La barra de menú contiene dos menús: uno permite al usuario ajustar el brillo del color de fondo de la ventana y uno permite que el usuario ajuste el tamaño de la fuente en

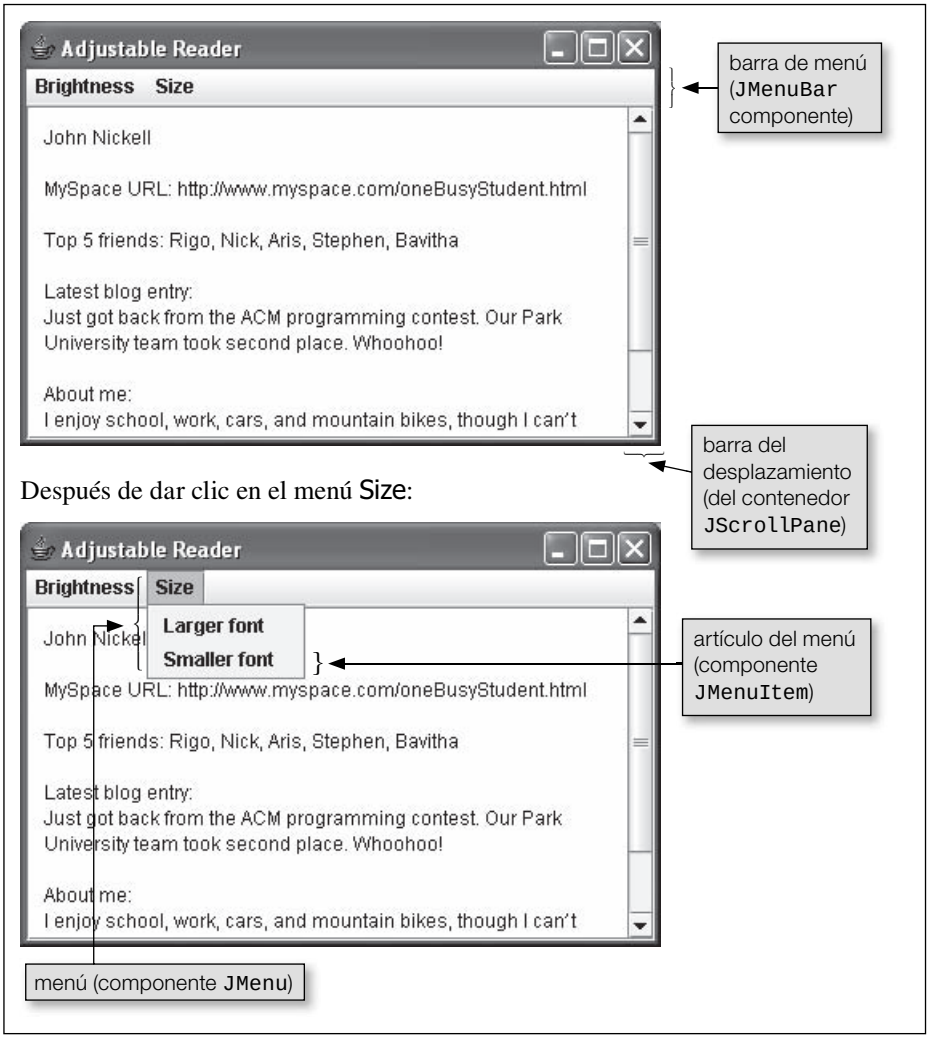


Figura 17.19 Un programa de lectura que utilice una barra de menú y una barra de desplazamiento para ajustar la pantalla.

```
private JMenuBar mBar;                // la barra de menú
private JMenu menu1, menu2;           // los dos menús
private JMenuItem mi1, mi2, mi3, mi4; // los cuatro artículos del menú
⋮
menu1 = new JMenu("Brillo");
menu2 = new JMenu("Tamaño");

mi1 = new JMenuItem("Fondo más claro");
mi2 = new JMenuItem("Fondo más oscuro");
mi3 = new JMenuItem("Fuente más grande");
mi4 = new JMenuItem("Fuente más pequeña);

mi1.addActionListener(new BrightnessListener());
mi2.addActionListener(new BrightnessListener());
mi3.addActionListener(new SizeListener());
mi4.addActionListener(new SizeListener());

menu1.add(mi1);
menu1.add(mi2);
menu2.add(mi3);
menu2.add(mi4);

mBar = new JMenuBar();
mBar.add(menu1);
mBar.add(menu2);
setJMenuBar(mBar);
```

Figura 17.20 Código que crea una barra de menú, menús y artículos de menú para el programa de la figura 17.19.

el texto de la ventana. La barra de desplazamiento forma parte de lo que se conoce como *panel de desplazamiento*. La barra de desplazamiento permite que el usuario se desplace de un lado a otro o vea el contenido de toda la ventana.

Si el lector desea ver todo el programa en la figura 17.19, debe consultar el archivo `ReaderMenu.java` en el sitio en la red de este libro. En la figura 17.20 se muestra una porción de ese programa: la porción que crea la barra de menú, los menús y artículos de menú. Y la siguiente declaración muestra la porción que crea el panel de desplazamiento. Más específicamente, la siguiente declaración crea un panel de desplazamiento para un componente zona de texto y luego agrega el panel de desplazamiento a la ventana.

```
add(new JScrollPane(textArea));
```

Controles deslizantes

En otro ejemplo para autodidactas, se recomienda estudiar la clase `JSlider` en el sitio Web de Sun. La clase `JSlider` permite agregar un componente control *deslizante* a una ventana. Este tipo de control permite que el usuario seleccione un valor de un intervalo de valores. Para seleccionar un valor, el usuario arrastra un "índice" a lo largo de una barra de valores. Vea la figura 17.21. Imita un eclipse lunar al cubrir un círculo blanco (la Luna) con un círculo gris (la sombra de la Tierra). Cuando el usuario arrastra el índice hacia la derecha, la sombra se desplaza a la derecha. Cuando el usuario arrastra el índice hacia la izquierda, la sombra se desplaza a la izquierda. El control deslizante utiliza un manipulador de eventos para ajustar la posición de la sombra.

El código siguiente del programa del eclipse lunar muestra cómo instanciar un control deslizante, establecer sus propiedades y agregar un oyente:

```
slider = new JSlider(SwingConstants.HORIZONTAL, 0, 100, 0);
slider.setMajorTickSpacing(5);
```

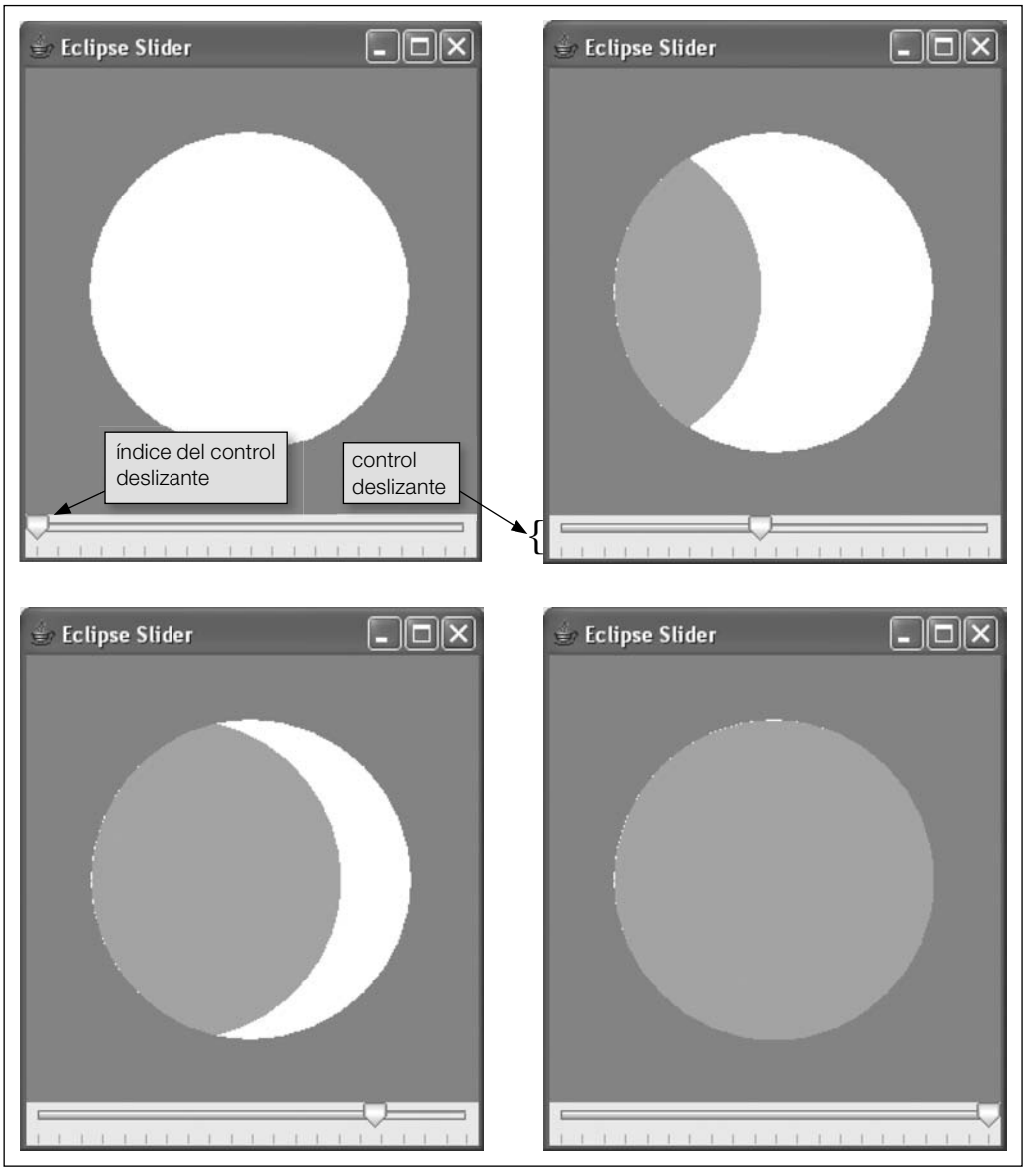


Figura 17.21 Cuatro imágenes de una ventana que usa un control deslizante para simular un eclipse.

```
slider.setPaintTicks(true);
slider.addChangeListener(new Listener());
```

Esto agrega el control deslizante al `JFrame` actual:

```
add(slider, BorderLayout.SOUTH);
```

Si el lector desea ver el programa de la figura 17.21 en su totalidad, debe consultar el archivo `EclipseSlider.Java` en el sitio de este libro en la red.

Resumen

- Los gestores de distribución automatizan el posicionamiento de los componentes dentro de contenedores.
- La clase `FlowLayout` implementa un esquema simple de distribución de un compartimiento que permite que múltiples componentes sean insertados en el compartimiento.

- El gestor `BorderLayout` proporciona cinco compartimientos, norte, sur, este, oeste y centro, donde insertar los componentes.
- La interfaz `SwingConstants` almacena un conjunto de constantes relacionadas con GUI que suelen utilizarse en varios programas GUI.
- El gestor `GridLayout` dispone los componentes de un contenedor en una rejilla rectangular cuyas celdas tienen el mismo tamaño. Cada celda puede contener sólo un componente.
- Si se tiene una ventana complicada con muchos componentes, quizá sea conveniente compartimentarlas al almacenar grupos de componentes en contenedores `JPanel`.
- Para desplegar múltiples líneas de un texto, utilice un componente `JTextArea`.
- Un componente `JCheckBox` despliega un cuadrado pequeño con una etiqueta de identificación. Los usuarios hacen clic en la casilla de verificación para activar entre seleccionado y no seleccionado.
- Un componente `JRadioButton` despliega un círculo pequeño con una etiqueta a su derecha. Si se hace clic en un botón no seleccionado, este botón se vuelve seleccionado y el botón previo en el grupo se vuelve no seleccionado.
- Un componente `JComboBox` permite que el usuario seleccione un artículo de una lista de artículos. Los componentes de `JComboBox` se denominan "cuadros combo" porque son una combinación de un cuadro de texto (normalmente, parecen cuadros de texto) y una lista (cuando se hace clic en la flecha hacia abajo, se ven como una lista).

Preguntas de revisión

§17.2 Diseño GUI y gestores de distribución

1. Los gestores de distribución se adaptan automáticamente a los cambios en el tamaño de un contenedor o en uno de sus componentes. (F/C)
2. ¿Qué paquete contiene a los gestores de distribución?

§17.3 Gestor `FlowLayout`

3. ¿Cómo dispone el gestor `FlowLayout` los componentes?
4. Escriba una simple declaración que proporcione al contenedor actual un alineamiento a la derecha.

§17.4 Gestor `BorderLayout`

5. ¿Cuáles son las cinco regiones establecidas por el gestor `BorderLayout`?
6. Los tamaños de las cinco regiones en un gestor `BorderLayout` se determinan en tiempo de ejecución con base en el contenido de las cuatro regiones exteriores. (F/C)
7. Por defecto, ¿cuántos componentes pueden colocarse en una de las regiones de una distribución de límites?
8. Escriba una simple declaración que agregue una nueva `JLabel` con el texto "Stop" a la región centro de un gestor `BorderLayout`. La etiqueta debe estar centrada en la región centro.

§17.5 Gestor `GridLayout`

9. Cuando se instancia un gestor `GridLayout`, siempre se especifica el número de renglones y el número de columnas. (F/C)
10. En un gestor `GridLayout`, todas las celdas son del mismo tamaño. (F/C)

§17.6 Ejemplo de juego de gato

11. ¿Qué ocurre a la variable `xTurn` en el programa de juego de gato si se hace clic dos veces en la misma celda?

§17.9 Clase `JPanel`

12. ¿Por qué los contenedores `JPanel` son particularmente útiles con ventanas `GridLayout` y `BorderLayout` (en oposición a las ventanas `FlowLayout`)?

§17.10 Programa `calculadoraMatematica`

13. En el método `createContents` del programa calculadoraMatematica, ¿cuál es el propósito de la declaración `add(new JLabel());`?

§17.11 Componente `JTextArea`

14. Por defecto, los componentes `JTextArea` son editables. (F/C)
15. Por defecto, los componentes `JTextArea` usan un ajuste de línea. (F/C)

§17.12 Componente JCheckBox

16. ¿Qué ocurre si se hace clic en una casilla de verificación que ya estaba seleccionada?

17. Escriba una declaración que produzca una casilla de verificación denominada `attendance`. La casilla de verificación debe ser preseleccionada y debe tener una etiqueta "I will attend".

§17.13 Componente JRadioButton

18. ¿Qué ocurre si se hace clic en un botón de radio que ya estaba seleccionado?

19. ¿Qué ocurre si se hace clic en un botón de radio que inicialmente no estaba seleccionado y que pertenece a un `RadioGroup`?

§17.14 Componente JComboBox

20. ¿En qué se parecen los cuadros combo y los grupos de botones de radio?

21. Escriba dos métodos que pueden llamarse para determinar la selección actual de un cuadro combo.

§17.15 Ejemplo aplicación de tarea

22. El programa `JobApplication` contiene el siguiente fragmento de código. ¿Qué ocurre al programa si se omite el fragmento de código?

```
radioGroup = new ButtonGroup();
radioGroup.add(goodCitizen);
radioGroup.add(criminal);
```

23. Escriba una declaración que agregue un margen en blanco de 20 pixeles a un contenedor `JPanel` denominado `panel`.

§17.16 Más componentes de Swing

24. Proporcione una llamada al constructor `JSlider` donde el valor mínimo sea cero, el valor máximo sea 50 y el valor inicial sea 10. Sugerencia: consulte la respuesta en el sitio API Sun de Java en la red.

Ejercicios

1. [Después de §17.2] ¿Cuál es el gestor de distribución para una ventana `JFrame`?

2. [Después de §17.3] Con un gestor `FlowLayout`, un componente botón se expande de modo que llena por completo el tamaño de la región en que está colocada. (F/C)

3. [Después de §17.4] Escriba un programa completo que sea una modificación del programa Greeting del capítulo 16. El programa nuevo debe usar un gestor `BorderLayout` (en lugar de un gestor `FlowLayout`), y debe generar el siguiente desplegado después que se introduce un nombre. Haga que el tamaño del marco mida 300 pixeles de ancho por 80 pixeles de alto.

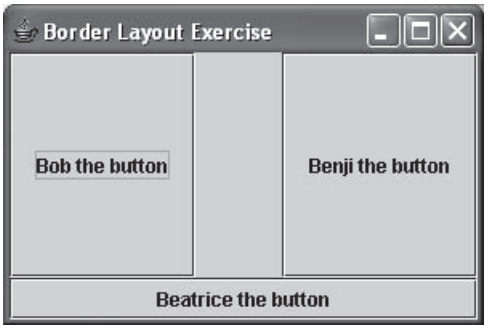
4. [Después de §17.4] Con un `BorderLayout`, ¿qué ocurre si la región este está vacía? En otras palabras, ¿qué región (o regiones) se expande(n) si la región este está vacía?

5. [Después de §17.4] Suponga que se tiene este programa:

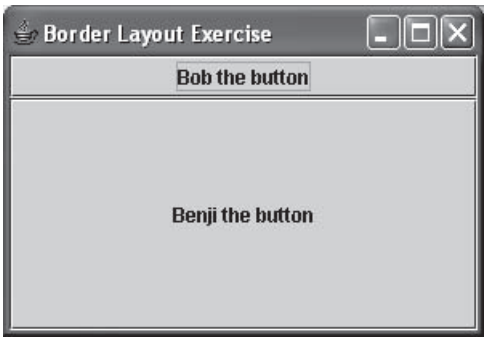
```
import javax.swing.*;
import java.awt.*;

public class BorderLayoutExercise extends JFrame
{
  public BorderLayoutExercise()
  {
    setTitle("Border Layout Exercise");
    setSize(300, 200);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new BorderLayout());
    add(new JLabel("Lisa the label"), BorderLayout.NORTH);
```

```
    add(new JLabel("LaToya the label"), BorderLayout.CENTER);
    add(new JLabel("Lemmy the label"), BorderLayout.SOUTH);
    setVisible(true);
  } // end BorderLayoutExercise constructor

  //*************************************************

  public static void main(String[] args)
  {
    new BorderLayoutExercise();
  }

} // end class BorderLayoutExercise
```

a) Especifique los cambios que haría al código de arriba para obtener esta salida:



b) Especifique los cambios que haría al código de arriba para obtener esta salida:



6. [Después de §17.5] Si un componente `JButton` se agrega directamente a una celda `GridLayout`, se expande de modo que llena por completo el tamaño de su celda. (F/C)

7. [Después de §17.5] Dado el siguiente fragmento de código, haga un dibujo que ilustre las posiciones de los botones dentro de la ventana del programa.

```
setLayout(new GridLayout(0, 3));
add(new JButton("1"));
add(new JButton("2"));
add(new JButton("3"));
add(new JButton("4"));
add(new JButton("5"));
add(new JButton("6"));
add(new JButton("7"));
```

8. [Después de §17.9] ¿Qué tipo de contenedor debe colocarse dentro de una celda individual `GridLayout` o en una región de distribución de límite para permitir que esa celda o esa región contenga más de un componente?

9. [Después de §17.11] Suponga que se tiene una ventana con dos componentes `JTextArea` identificadas por `msg1` y `msg2`, y un componente `JButton`. Cuando se hace clic en el botón, éste cambia el contenido de las dos zonas de texto. Escriba el código que efectúa la operación de intercambio. Más específicamente, proporcione el código que va dentro del siguiente método `ActionPerformed`:

```
private class Listener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    ...
  }
}
```

10. [Después de §17.12] Escriba una declaración que produzca una casilla de verificación denominada `bold`. La casilla de verificación debe no estar seleccionada, y debe tener una etiqueta de "tipo negrita".

11. [Después de §17.12] ¿Cómo puede su código determinar si una casilla de verificación está seleccionada o no?

12. [Después de §17.13] Escriba un método `createContents` para un programa que despliegue esta ventana:

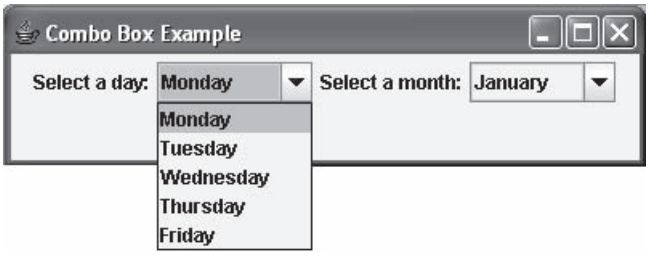
Los botones de radio, masculinos y femeninos, deben comportarse en una forma normal: cuando se selecciona uno, el otro permanece sin seleccionar. Observe que el botón masculino se selecciona cuando la ventana aparece por primera vez. Su método `createContents` debe trabajar en forma conjunta con esta estructura de programa:

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

public class MaleFemaleRadioButtons extends JFrame
{
  private JRadioButton male;
  private JRadioButton female;

  public MaleFemaleRadioButtons()
  {
    setTitle("Male-Female Radio Buttons");
    setSize(275, 100);
    setDefaultCloseOperation(EXIT _ ON _ CLOSE);
    createContents();
    setVisible(true);
  } // end MaleFemaleRadioButtons constructor

  <The createContents method goes here.>

  public static void main(String[] args)
  {
    new MaleFemaleRadioButtons();
  }
} // end class MaleFemaleRadioButtons
```

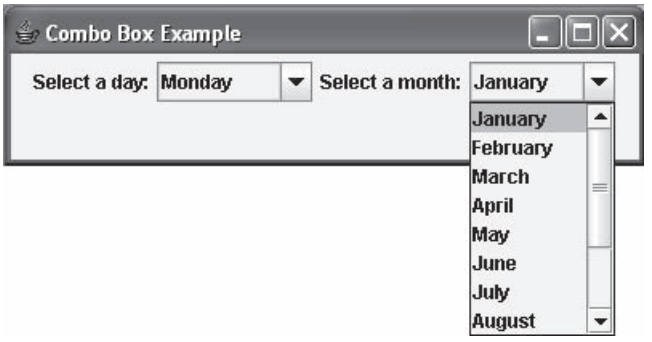
13. [Después de §17.14] ¿En qué paquete están definidos los componentes `JCheckBox`, `JRadioButton` y `JComboBox`?

14. [Después de §17.14] Proporcione un método `createContents` para un programa que inicialmente despliega esta ventana:

Cuando el usuario hace clic en el cuadro combo izquierdo, despliega esto:



Cuando el usuario hace clic en el cuadro combo derecho, despliega esto:



Su método `createContents` debe trabajar conjuntamente con esta estructura de programa:

```
import javax.swing.*;
import java.awt.*;
public class ComboBoxExample extends JFrame
{
  private JComboBox daysBox;
  private JComboBox monthsBox;
  private String[] days =
    {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"};
  private String[] months =
    {"January", "February", "March", "April", "May", "June",
     "July", "August", "September", "October", "November",
     "December"};

  public ComboBoxExample()
  {
    setTitle("Combo Box Example");
    setSize(400, 100);
    setDefaultCloseOperation(EXIT _ ON _ CLOSE);
    createContents();
    setVisible(true);
  } // end ComboBoxExample constructor

  <The createContents method goes here.>

  public static void main(String[] args)
  {
    new ComboBoxExample();
  }
} // end class ComboBoxExample
```

Soluciones a las preguntas de revisión

1. Cierto.
2. Los gestores `Layout` están en el paquete `java.awt`.

3. El gestor `FlowLayout` coloca los componentes de izquierda a derecha en un renglón hasta que ya no queda espacio, y entonces pasa al siguiente renglón y hace lo mismo, y así sucesivamente.
4. `setLayout(new FlowLayout(FlowLayout.RIGHT));`
5. Las cinco regiones de un gestor `BorderLayout` son norte en la parte superior, sur en la parte inferior, y oeste, centro y este en un renglón entre ellas.
6. Cierto.
7. Cero o una.
8.
```
add(new JLabel("Stop", SwingConstants.CENTER),
BorderLayout.CENTER);
```
o bien
```
add(new JLabel("Stop", SwingConstants.CENTER));
```
9. Falso. Ambos valores se especifican, para renglones y para columnas, sólo si se conoce el número de renglones y el número de columnas en la tabla y la tabla está completamente llena (es decir, que no hay celdas vacías). En caso contrario, se especifica una sola dimensión de la que se tenga certeza y se especifica cero para la otra dimensión.
10. Cierto.
11. Nada. No modifica el valor.
12. Los contenedores `JPanel` son particularmente útiles con ventanas `GridLayout` y `BorderLayout` porque cada compartimiento en estas distribuciones puede almacenar sólo un componente. Si se requiere que un compartimiento almacene más de un componente, se hace que ese componente sea un contenedor `JPanel`, y múltiples componentes se colocan en el contenedor `JPanel`.
13. La declaración `add(new JLabel());` agrega un componente inválido (una etiqueta en blanco) a la celda inferior izquierda. El componente inválido es necesario porque, sin él, `xLogPanel` iría a la celda inferior izquierda, lo cual es inapropiado.
14. Cierto. Los componentes `JTextArea` son editables por defecto.
15. Falso. Los componentes `JTextArea` no usan un ajuste de línea por defecto.
16. Si se hace clic en una casilla de verificación que ya estaba seleccionada, la casilla de verificación se vuelve no seleccionada.
17. El código siguiente produce una casilla de verificación denominada `attendance`. La casilla de verificación es preseleccionada y tiene una etiqueta “I will attend”.
```
JCheckBox attendance = new JCheckBox("I will attend", true);
```
18. Nada. Permanece seleccionado.
19. El botón de radio en que se hace clic se vuelve seleccionado y todos los demás botones en el grupo se vuelven no seleccionados.
20. Los cuadros combo y los grupos de botones de radio se parecen en que ambos permiten que el usuario seleccione un artículo de una lista de artículos.
21. Para determinar la selección actual de un cuadro combo, debe llamarse a `getSelectedItem` o a `getSelectedIndex`.
22. Si el código `radioGroup` se omite del programa JobApplication, el programa sigue compilando y puede ejecutarse, pero los botones de radio operan de manera independiente. En otras palabras, al hacer clic en un botón de radio no provoca que el otro se vuelva no seleccionado.
23. Esta declaración agrega un margen en blanco de 20 pixeles a un contenedor `JPanel` denominado `panel`:
```
panel.setBorder(new EmptyBorder(20, 20, 20, 20));
```
24. Llamada al constructor `JSlider`:
```
new JSlider(0, 50, 10);
```

APÉNDICE 1

Conjunto de caracteres Unicode/ASCII con códigos hexadecimales

Java asigna un valor de código numérico de dos bytes a cada carácter en consonancia con el estándar Unicode. Puesto que dos bytes contienen un total de 16 bits, esto constituye un total de $2^{16} = 65\,536$ códigos diferentes. En la figura 11.4 del capítulo 11 se describen o muestran estos caracteres cuyo valor de código está en el intervalo de 0 al decimal 127. En este intervalo particular de valores de código, los valores de código Unicode coinciden exactamente con los valores de código ASCII (ASCII = *American Standard Code for Information Interchange*).

Los números hexadecimales usan dígitos que pueden asumir uno de esos 16 valores. Los valores permitidos son del 0 al 9 y de la A a la F. Los valores de la A a la F representan los números del 10 al 15. En la figura 11.4 se usaron números decimales para representar valores de código Unicode (y ASCII) en el intervalo de 0 al decimal 127. No obstante, a veces en este intervalo numérico, y especialmente a valores numéricos superiores, resulta más conveniente usar números hexadecimales para representar valores Unicode, así como ubicaciones en la memoria. Para proporcionar una mejor idea de cómo funciona el conteo en hexadecimal, y para ayudar al lector a encontrar valores hexadecimales para caracteres en el importante conjunto de caracteres ASCII, en la figura A1.1*a* se muestran los caracteres ASCII con números hexadecimales mostrados junto con los números decimales correspondientes.

Observe que los valores de código hexadecimal para los caracteres numéricos '1' a '9' son los hexadecimales del 31 al 39. Ahora observe el resto de los caracteres ASCII en la figura A1.1*b*. Observe que el valor de código para la primera letra mayúscula, A, es el hexadecimal 41, y que el valor de código para la primera letra minúscula, *a*, es el hexadecimal 61. Para cambiar de minúscula a mayúscula o viceversa, simplemente se suma o resta el hexadecimal 20. ¡Quienes asignaron estos valores estaban "pensando" en hexadecimal!

En las columnas bajo el título "Unicode", en las figuras A1.1*a* y A1.1*b* también se muestra la secuencia de escape Unicode para cada carácter. Éstos son los que se usan cuando se desea insertar en una `String` cualquier carácter o símbolo que no puede teclearse directamente. Cada una de estas secuencias de escape Unicode coloca un prefijo `\u` en una versión hexadecimal de cuatro sitios del código del número. (En caso de ser necesario, el número hexadecimal crudo que queda se rellena de ceros para incrementar a cuatro el número total de dígitos hexadecimales). Este formato permite que la secuencia de escape Unicode de cabida hasta un total de $16^4 = 65\,535$ caracteres o símbolos distintos.

Por supuesto, hay muchos otros caracteres y símbolos. Para un análisis e información más detallados, consulte el material en la sección opcional 11.12, Apartado GUI: Unicode. En la figura 11.12 se muestra una muestra de algunos de los otros caracteres y símbolos disponibles. Ese desplegado fue generado por el programa en la figura 11.11, y ese programa puede modificarse para desplegar los caracteres para cualquier intervalo de códigos. Para conocer todos los detalles sobre el estándar Unicode, consulte:

http://www.unicode.org/

Este sitio Web contiene dos diagramas de una página que clasifican los alfabetos, símbolos y puntuación más importantes del mundo. Puede seleccionarse el alfabeto o el tipo de símbolo que se quiera y obtener imágenes y números de código para todos los caracteres en esa categoría.

valor del código			carácter
dec	hex	Unicode	
0	0	\u0000	nulo
1	1	\u0001	inicio de encabezado
2	2	\u0002	inicio de texto
3	3	\u0003	fin de texto
4	4	\u0004	fin de transmisión
5	5	\u0005	consulta
6	6	\u0006	reconocimiento
7	7	\u0007	campana audible
8	8	\u0008	tecla de retroceso
9	9	\u0009	tabulador horizontal (\t)
10	A	\U000A	avance de línea (\n)
11	B	\U000B	tabulador vertical
12	C	\U000C	avance de forma
13	D	\u000D	regreso de carro (\r)
14	E	\u000E	cambio
15	F	\u000F	cambio
16	10	\u0010	escape del enlace para transmisión de datos
17	11	\u0011	control 1 del dispositivo
18	12	\u0012	control 2 del dispositivo
19	13	\u0013	control 3 del dispositivo
20	14	\u0014	control 4 del dispositivo
21	15	\u0015	reconocimiento negativo
22	16	\u0016	vacío síncrono
23	17	\u0017	final de bloque de transmisión
24	18	\u0018	cancelar
25	19	\u0019	fin del medio
26	1A	\u001A	sustituir
27	1B	\u001B	escape
28	1C	\u001C	separador de archivo
29	1D	\u001D	separador de grupo
30	1E	\u001E	separador de registro
31	1F	\u001F	separador de unidad

valor del código			carácter
dec	hex	Unicode	
32	20	\u0020	espacio
33	21	\u0021	!
34	22	\u0022	"
35	23	\u0023	#
36	24	\u0024	$
37	25	\u0025	%
38	26	\u0026	&
39	27	\u0027	'
40	28	\u0028	(
41	29	\u0029	)
42	2A	\U002A	*
43	2B	\U002B	+
44	2C	\U002C	,
45	2D	\u002D	-
46	2E	\u002E	.
47	2F	\u002F	/
48	30	\u0030	0
49	31	\u0031	1
50	32	\u0032	2
51	33	\u0033	3
52	34	\u0034	4
53	35	\u0035	5
54	36	\u0036	6
55	37	\u0037	7
56	38	\u0038	8
57	39	\u0039	9
58	3A	\u003A	:
59	3B	\u003B	;
60	3C	\u003C	<
61	3D	\u003D	=
62	3E	\u003E	>
63	3F	\u003F	?

Figura A1.1*a* Códigos de carácter Unicode/ASCII, parte A.

Además del prefijo de secuencia de escape Unicode `\u`, hay otras dos anotaciones hexadecimales que es necesario conocer. Algunas veces conviene usar la forma hexadecimal de un número literal en una declaración o en una fórmula matemática, ya que podría ser más fácil o más autodocumentada que la forma decimal. Para hacer esto, simplemente aplique `0x` como prefijo al número hexadecimal crudo. Por ejemplo, para indicar al compilador que se está escribiendo un número hexadecimal en lugar de uno decimal, es necesario escribir `0x41` para especificar el número cuyo valor decimal es 65. Si se quiere desplegar la forma hexadecimal de una constante o una variable entera usando el método `printf`, use `%x` para el sitio que contiene al número en la cadena de formato. Si lo que se está representando es una lite-

valor del código			carácter
dec	hex	Unicode	
64	40	\u0040	@
65	41	\u0041	A
66	42	\u0042	B
67	43	\u0043	C
68	44	\u0044	D
69	45	\u0045	E
70	46	\u0046	F
71	47	\u0047	G
72	48	\u0048	H
73	49	\u0049	I
74	4A	\U004A	J
75	4B	\U004B	K
76	4C	\U004C	L
77	4D	\u004D	M
78	4E	\u004E	N
79	4F	\u004F	O
80	50	\u0050	P
81	51	\u0051	Q
82	52	\u0052	R
83	53	\u0053	S
84	54	\u0054	T
85	55	\u0055	U
86	56	\u0056	V
87	57	\u0057	W
88	58	\u0058	X
89	59	\u0059	Y
90	5A	\u005A	Z
91	5B	\u005B	[
92	5C	\u005C	\
93	5D	\u005D	]
94	5E	\u005E	^
95	5F	\u005F	_

valor del código			carácter
dec	hex	Unicode	
96	60	\u0060	`
97	61	\u0061	a
98	62	\u0062	b
99	63	\u0063	c
100	64	\u0064	d
101	65	\u0065	e
102	66	\u0066	f
103	67	\u0067	g
104	68	\u0068	h
105	69	\u0069	i
106	6A	\U006A	j
107	6B	\U006B	k
108	6C	\U006C	l
109	6D	\u006D	m
110	6E	\u006E	n
111	6F	\u006F	o
112	70	\u0070	p
113	71	\u0071	q
114	72	\u0072	r
115	73	\u0073	s
116	74	\u0074	t
117	75	\u0075	u
118	76	\u0076	v
119	77	\u0077	w
120	78	\u0078	x
121	79	\u0079	y
122	7A	\u007A	z
123	7B	\u007B	{
124	7C	\u007C	|
125	7D	\u007D	}
126	7E	\u007E	~
127	7F	\u007F	delete

Figura A1.1*b* Códigos de carácter Unicode/ASCII, parte B.

ral, su forma en la lista de datos no importa. Puede ser un decimal o un hexadecimal en el prefijo `0x`. Por ejemplo, suponga que se quiere obtener una salida como ésta:

Salida:

```
The hexadecimal value for 10 is a
```

Esa salida puede generarse con este código:

```
System.out.printf("The hexadecimal value for 10 is %x\n", 10);
```

Observe que la salida hexadecimal es una 'a' minúscula. Que sea minúscula o mayúscula no importa.

APÉNDICE 2

Precedencia de operadores

Los grupos de operadores en la parte superior de la tabla tienen mayor precedencia que los grupos de operadores en la parte inferior de la tabla. Todos los operadores dentro de un grupo con una precedencia particular tienen la misma precedencia. Si una expresión tiene dos o más operadores con la misma precedencia, entonces dentro de esa expresión, tales operadores se ejecutan de izquierda a derecha o de derecha a izquierda según se indique en el encabezado del grupo.

1. agrupamientos y accesos (de izquierda a derecha):	
(<*expression*>)	expresiones
(<*list*>)	argumentos o parámetros
[<*expression*>]	índices
<*type-or-member*>.<*type-or-member*>	acceso a miembros
2. operaciones unarias (de derecha a izquierda):	
x++	post incremento
x--	post decremento
++x	preincremento
--x	predecremento
+x	más
-x	menos
!x	inversión lógica
~	inversión de bit
new <*classname*>	instanciación de un objeto
(<*type*>) x	tipo cast
3. multiplicación y división (de izquierda a derecha):	
x * y	multiplicación
x / y	división
x % y	residuo
4. suma y resta; concatenación (de izquierda a derecha):	
x + y	suma
x - y	resta
s1 + s2	concatenación de cadenas
5. operaciones de desplazamiento de bits (de izquierda a derecha):	
x << n	desplazamiento aritmético a la izquierda ($*2^n$)
x >> n	desplazamiento aritmético a la derecha ($*2^{-n}$; mismo bit MS)
x >>> n	desplazamiento lógico a la derecha (bit MS = 0)

Figura A2.1*a* Precedencia de operadores, parte A.

5. comparaciones de intervalos (de izquierda a derecha):

`x < y`	menor que
`x <= y`	menor o igual a
`x >= y`	mayor o igual a
`x > y`	mayor que
<*object*> `instanceof` <*class*>	se ajusta a

6. comparaciones de igualdad (de izquierda a derecha):

`x == y`	igual
`x != y`	no es igual

7. `boolean` incondicional o bit AND (de izquierda a derecha):

`x & y`	ambos

8. `boolean` incondicional o bit EXCLUSIVE OR (de izquierda a derecha):

`x ^ y`	uno de los dos pero no ambos

9. `boolean` incondicional o bit OR (de izquierda a derecha):

`x | y`	uno de los dos o ambos

10. `boolean` condicional AND (de izquierda a derecha):

`x && y`	ambos (en caso que no esté resuelto)

11. `boolean` condicional OR (de izquierda a derecha):

`x || y`	uno de los dos o ambos (en caso que no esté resuelto)

12. evaluación condicional ternaria (de derecha a izquierda):

`x ? y : z`	si `x` es `true`, `y`, o bien `z`

13. asignación (de derecha a izquierda):

`y = x`	`y ← x`
`y += x`	`y ← y + x`
`y -= x`	`y ← y - x`
`y *= x`	`y ← y * x`
`y /= x`	`y ← y / x`
`y %= x`	`y ← y % x`
`y <<= n`	`y ← y << n`
`y >>= n`	`y ← y >> n`
`y >>>= n`	`y ← y >>> n`
`y &= x`	`y ← y & x`
`y ^= x`	`y ← y ^ x`
`y |= x`	`y ← y | x`

Figura A2.1*b* Precedencia de operadores, parte B.

APÉNDICE 3

Palabras reservadas en Java

Las palabras reservadas en Java no pueden usarse para la denominación de cualquier cosa que se defina porque ya tienen significados especiales. La mayor parte de estas palabras son *palabras clave*: tienen funciones particulares en un programa de Java. Un asterisco indica que la palabra en cuestión no se utiliza en el cuerpo de este texto.

`abstract`: no realizable. Éste es un modificador para clases y métodos y un modificador implicado para interfaces. Un método `abstract` no está definido. Una clase `abstract` contiene uno o más métodos `abstract`. Todos los métodos en una interfaz son abstractos. No es posible instanciar una interfaz o una clase `abstract`.

`assert`*: indica que algo es verdadero. En cualquier sitio en el programa es posible insertar declaraciones que digan `assert` <*boolean expression*>; luego, si el programa se ejecuta con la opción `enableassertions`, la JVM lanza una excepción `AssertionError` cuando encuentra una `assert` y la evalúa como `false`.

`boolean`: valor lógico. Este tipo de dato primitivo se evalúa como `true` o `false`.

`break`: saltar de. Este comando provoca la ejecución en una declaración `switch` o que el ciclo salte hacia delante a la primera declaración luego del final de esa declaración `switch` o de ese ciclo.

`byte`: ocho bits. Éste es el tipo de datos enteros primitivos más pequeño. Es el tipo que se almacena en archivos binarios.

`caso`: una alternativa particular. El valor `byte`, `char` o `int` inmediatamente después de una palabra clave `case` identifica una de las alternativas `switch`.

`catch`: captura, atrapar. Un bloque `catch` contiene código que se ejecuta cuando el código en un bloque `try` precedente lanza (`throws`) una excepción, que es un objeto especial que describe un error.

`char`: un carácter. Éste es un tipo de datos primitivos que contiene un número de código numérico para un carácter de texto o cualquier otro símbolo definido en el estándar Unicode.

`clase`: tipo complejo. Este bloque de código Java define los atributos y el comportamiento de un tipo particular de objeto. Así, define un tipo de datos que es más complejo que un tipo de datos primitivos.

`const`*: una constante. Este término arcaico es sustituido por `final`.

`continue`*: omitir hasta el final. Con este comando se omiten las declaraciones restantes en el código del ciclo y se va directamente a la condición de continuación del ciclo.

Figura A3.1*a* Palabras reservadas, parte A.

`default:` en caso contrario. Suele ser el último caso en una declaración `switch`. Representa todos los otros casos (casos no identificados en los bloques `case` previos).

`do:` ejecutar. Ésta es la primera palabra clave en un ciclo do-while. La condición de continuación aparece entre paréntesis después de la palabra clave `while` al final del ciclo.

`double:` dos veces la cantidad de algo. Este tipo de datos de punto flotante primitivo requiere el doble de almacenamiento, ocho bytes, que el tipo de datos anterior de punto flotante, `float`, que sólo requiere cuatro bytes.

`else:` en caso contrario. Esta palabra clave puede usarse en una declaración `if` compuesta como encabezado (o parte del encabezado) de un bloque de código que se ejecute en caso de que no se satisfaga la condición previa `if`.

`enum*:` enumeración. Este tipo especial de clase define un conjunto de constantes nombradas, que implícitamente con `static` y `final`.

`extends:` se deriva de. Esta extensión del encabezado de una clase especifica que la clase que se está definiendo heredará todos los miembros de la clase nombrada después de la palabra clave `extends`.

`false:` no. éste es uno de los dos valores `boolean` posibles.

`final:` forma o valor último. Este modificador evita la redefinición de las clases y los métodos, e indica que un valor nombrado es una constante.

`finally:` última operación. Puede usarse después de bloques `try` y `catch` para especificar operaciones que es necesario efectuar después que un bloque `catch` procesa una excepción.

`float:` punto flotante. Éste es un antiguo tipo de datos de punto flotante. Requiere cuatro bytes.

`for:` el tipo de ciclo más versátil. Esta palabra clave introduce un ciclo cuyo encabezado especifica y controla el intervalo de iteración.

`goto*:` ir a. Este comando depreciado especifica una rama incondicional. No lo use.

`if:` ejecución condicional. Esta palabra clave inicia la ejecución de un bloque si se satisface una condición asociada.

`implements:` define. Esta extensión de encabezado de clase especifica que la clase que se está definiendo definirá todos los métodos declarados por la `interface` nombrada después de la palabra clave `implements`.

`import:` traer. Esto indica al compilador que ponga a disposición clases identificadas ulteriores para su uso en el programa actual.

Figura A3.1*b* Palabras reservadas, parte B.

`inner`: interno. Cuando va seguida por la palabra clave `class`, esto especifica que la clase definida en el bloque de código subsiguiente esté anidada en la clase actual.

`instanceof`: se ajusta a. Este operador `boolean` prueba si el objeto a la izquierda es una instancia de la clase a la derecha o un ancestro de esa clase.

`int`: entero. Éste es el tipo de datos enteros estándar. Requiere cuatro bytes.

`interface`: lo que observa alguien de afuera. Una interfaz de Java declara un conjunto de métodos pero no los define. Una clase que implementa una `interface` debe definir a todos los métodos declarados en esa `interface`. Una `interface` también puede definir constantes `static`. Otro tipo de interfaz simplemente transporta un mensaje particular al compilador.>>

`long`: entero largo. Éste es el tipo de datos enteros más largo. Requiere seis bytes.

`native`: indígena. El código nativo es el código que se ha compilado en el lenguaje (de bajo nivel) del procesador local. Algunas veces se denomina código de máquina.

`new`: instancia nueva de. Este comando Java llama a un constructor de clase para crear un nuevo objeto en tiempo de ejecución.

`null`: nada. Éste es el valor en una variable de referencia que no hace referencia a nada.

`package`: un grupo asociado. En Java, éste es un contenedor para un grupo de clases relacionadas que un programador puede importar.

`private`: controlado localmente. Este modificador de métodos y variables los hace accesibles sólo desde dentro de la clase en que están declarados.

`protected`: oculto a la exposición pública. Éste es un modificador de métodos y variables que los hace accesibles sólo desde dentro de la clase en que están declarados, desde los descendientes de esa clase o desde otras clases en el mismo paquete.

`public`: accesible para todos. Éste es un modificador de clases, métodos y variables, los hace accesibles desde cualquier parte. Una interfaz de Java es implícitamente `public`.

`return`: ir y quizá regresar algo. Este comando ocasiona que el control del programa abandone el método actual y regrese al punto que está inmediatamente a continuación del punto desde el que se llamó al método. También es posible devolver un valor o una referencia.

`short`: entero pequeño. Este tipo de datos enteros requieren sólo dos bytes.

`static`: siempre presente. Este modificador de métodos y variables les otorga alcance de clase y existencia continua.

Figura A3.1*c* Palabras reservadas, parte C

`strictfp`*: punto flotante estricto. Este modificador de una clase o de un método restringe la precisión de punto flotante a la especificación de Java y evita que en los cálculos se usen bits de precisión extra que puede proporcionar el procesador local.

`super`: padre o progenitor. Ésta es una referencia a un constructor o método que puede ser heredado por la clase del objeto si sobre ella no está sobrepuesta una nueva definición de esa clase.

`switch`: seleccionar una alternativa. Esto hace que el control del programa salte hacia delante hacia el código que sigue al `case` que coincide con la condición proporcionada inmediatamente a continuación de la palabra clave `switch`.

`synchronized`*: Este modificador de métodos impide la ejecución simultánea de un método particular por diferentes *threads*.

`this`: el objeto actual. La referencia punto `this` distingue una variable de instancia de una variable o parámetro local, o indica que el objeto que llama a otro método es el mismo que el objeto que llamó al método en que reside el código que llama, o produce la iniciación del objeto construcción a otro constructor (sobrecargado) en la misma clase.

`throw`*: genera una excepción. Este comando, seguido por el nombre de una excepción, hace que se lance una excepción. Permite que un programa lance explícitamente una excepción.

`throws`: puede lanzar una excepción. Esta palabra clave, seguida por el nombre de un tipo particular de excepción puede agregarse al encabezado de un método para transferir la responsabilidad `catch` al método que llamó al método actual.

`transient`*: puede abandonarse. Este modificador de variable indica al software de serialización de Java que el valor en la variable modificada no debe guardarse en un objeto archivo.

`true`: sí. Éste es uno de los dos valores `boolean`.

`try`: intentar. Un bloque `try` contiene código que podría lanzar una excepción más código que es posible omitir si se lanza una excepción.

`void`: nada. Esto describe el tipo de un método que no regresa nada.

`volatile`*: errático. Esta palabra clave evita que el compilador trate de optimizar una variable que puede modificarse asincrónicamente.

`while`: en tanto que. Esta palabra clave más una condición `boolean` encabeza un ciclo `while`, o termina un ciclo *do-while*.

Figura A3.1*d* Palabras reservadas, parte D.

APÉNDICE 4

Paquetes

Como quizá recuerda el lector, un paquete es un grupo de clases relacionadas. En este apéndice se describen los paquetes Sun creados para organizar su biblioteca de clases API de Java. Luego se mostrará cómo crear sus propios paquetes para clases definidas por el programador. Finalmente, se presentan algunas opciones avanzadas ingeniosas.

Paquetes API de Java

Cuando se descarga una versión de Java desde Sun, se obtienen las jerarquías del paquete API del *Software Development Kit* (SDK) de Java 2. La instalación de este "kit" automáticamente hace que los paquetes Java formen parte del entorno de Java.

Las clases API de Java están organizadas en jerarquías de paquetes. En la figura A4.1 se muestra parte de estas jerarquías de paquetes API de Java. Observe que se muestran dos jerarquías, una con el paquete `java` como raíz y otra con el paquete `javax` como raíz. Esta imagen incluye todos los paquetes API que se importan en algún punto en este libro, pero los paquetes que se muestran en la figura A4.1 son sólo una pequeña fracción de todos los paquetes en las jerarquías de paquetes API.

Esta organización en jerarquías ayuda a que las personas encuentren clases particulares que requieren utilizar. Está bien que varias tengan el mismo nombre de clase si están en paquetes distintos. Así, encapsular grupos pequeños de clases en paquetes individuales permite reusar el nombre de una clase

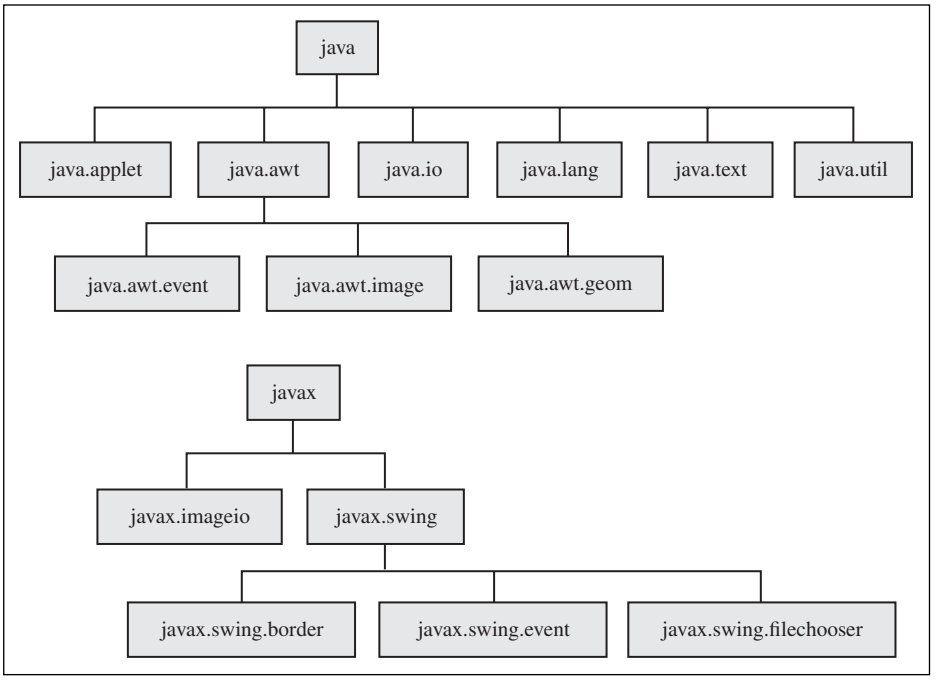


Figura A4.1 Jerarquías de paquete Java API abreviadas.

dada en diversos contextos. Asimismo, cada paquete protege de acceso desde el exterior a los miembros `protected` de las clases que incluye ese paquete.

Para hacer que las clases en un paquete particular estén disponibles para el programa que se está escribiendo, es necesario importar ese paquete, como en estas declaraciones, que importan los paquetes `java.util` y `java.awt`:

```
import java.util.*;
import java.awt.*;
```

En la figura A4.1 se incluyen algunos paquetes en un tercer nivel desde la parte superior. Considere, por ejemplo, el paquete `java.awt.event` bajo `java.awt` en el árbol `java`. Cuando el paquete `java.awt` se importa en la declaración de arriba, así se obtiene acceso a todas las clases en el paquete `java.awt` en sí, pero no se obtiene acceso a los paquetes bajo él. En otras palabras, tampoco importa el paquete `java.awt.event` Si también se requiere acceder a las clases en el paquete `java.awt.event`, también es necesario importar de manera explícita ese paquete al agregar esta tercera declaración importante:

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

Paquetes personalizados

El lenguaje Java permite crear sus propios paquetes para organizar clases definidas por el programador en jerarquías de paquetes. Eso implica los pasos siguientes:

Primero, diseñar una estructura de paquete que tenga sentido para el programa que se está creando. Luego se crea una estructura de directorio que corresponda exactamente a esa estructura de paquete. (Más adelante se mostrará cómo crear esta estructura de directorio automáticamente, aunque el proceso manual que se está describiendo ahora es más fácil de comprender.) En la figura A4.2*a* se muestra parte de una estructura de paquete que puede usarse para los ejemplos de este libro. En la figura A4.2*b* se muestra la estructura de directorio correspondiente. Observe "IPCJ" en la parte superior de ambas figuras. IPCJ es el acrónimo de *Introducción a la programación con Java*.

Siempre que se compile una clase que se quiera colocar en un paquete, esta línea debe insertarse en la parte superior de la clase, por arriba de todas las declaraciones, inclusive arriba de las declaraciones `import`:

```
package <package-path>
```

package-path es una ruta de subdirectorio, excepto que usa un punto (.) en vez de un slash (/ o \). El primer nombre en la ruta del paquete debe ser el nombre de la raíz de la jerarquía del paquete y el nombre del directorio superior en la parte de la estructura de directorio que le corresponde. El último nombre en la ruta del paquete debe ser el nombre del directorio que contiene la clase que se está definiendo. Así, por ejemplo, si se está definiendo una clase `Car` y se quiere que el archivo del código de byte de `Car.class` esté en el paquete `IPWJ.chapter13.things` que se muestra en la figura A4.2*a*, entonces la primera declaración en el archivo debe ser:

```
package IPWJ.chapter13.things;
```

Cuando se compila el código fuente `Car.java`, por defecto el código de byte de `Car.class` generado va hacia el directorio actual, y la declaración `package` de arriba en sí no modifica lo anterior. Por tanto,

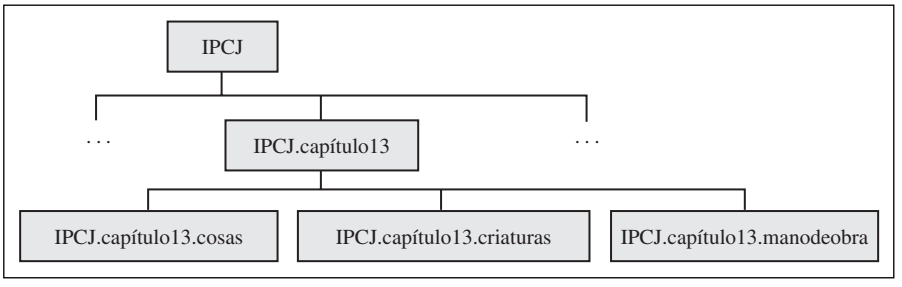


Figura A4.2*a* Estructura de paquete típica definida por el programador.

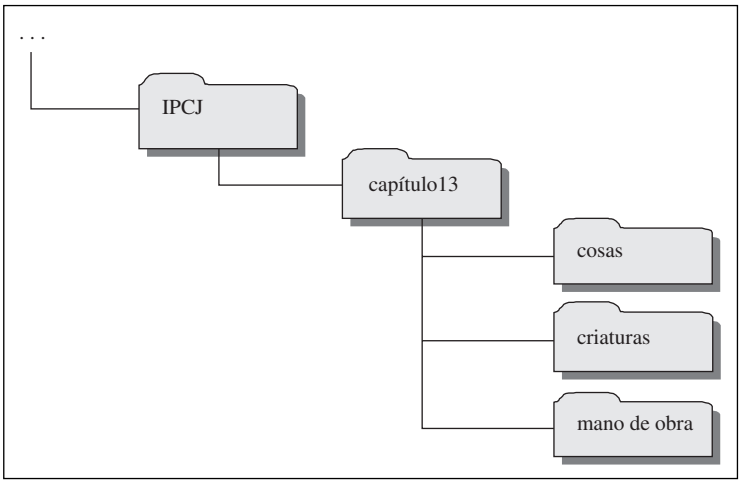


Figura A4.2*b* Estructura de directorio que corresponde a la estructura de paquete en la figura A4.2*b*.

si el lector elige escribir su código fuente en el directorio . . . `IPWJ/chapter13/things`, el archivo `Car.class` también pasa de inmediato a este directorio. Si se quiere que este directorio incluya tanto el código fuente como el código de byte, ya se ha terminado.

Si se quiere que el código fuente y el código de byte estén en directorios separados, quizá sea conveniente que decida escribir su código fuente en un directorio de código fuente por separado y luego mover el código de byte generado al directorio que coincide con su paquete especificado. (Más adelante se mostrará cómo puede solicitarse al compilador que haga lo anterior.)

Para que el compilador de Java importe una clase que está en un paquete por separado, esa clase debe ser accesible a través de una *ruta de clase* establecida previamente en el entorno de su sistema operativo. Puede haber más de una ruta de clase. En una máquina Windows, es posible ver toda la `CLASSPATH` registrada al abrir una ventana de desplegado e introducir el comando `set`. (En UNIX, el commando es `env.`). Después de `CLASSPATH,` usted verá varias especificaciones de clases de ruta, con punto y coma entre ellas. (En los separadores son comas.) Típicamente, la primera ruta de clase en la lista es un simple punto. Esto significa "directorio actual". Suponga que `myJava` es un directorio padre en la unidad `C`, y suponga que el directorio IPCJ mostrado en la figura A4.2*b* está en el directorio `myJava`. Para que las clases en la jerarquía de paquete IPCJ sean accesibles al compilador de Java, la `CLASSPATH` de la computadora del lector debe incluir la siguiente ruta:

```
C:/myJava.
```

Así, el nombre de ruta completo del archivo `Car.class` en el directorio cosas mostrado en la figura A4.2*b* sería:

```
C:/myJava/IPWJ/chapter13/things/Car.class
```

En un entorno Windows, la forma correcta de agregar una ruta de clase al entorno del sistema operativo es ir al icono Control Panel y hacer clic en System. Luego, bajo el tabulador Advanced, hacer clic en Environment Variables. . . . Luego, seleccionar CLASSPATH y hacer clic en Edit. . . Al final de la lista se escribe un punto y coma, y luego se introduce la nueva ruta de clase deseada; por ejemplo, `C:/myJava`. En UNIX, use el comando `setenv` como sigue:

```
setenv classpath .:/myJava
```

Algunas opciones avanzadas

Opcionalmente, se puede solicitar al compilador de Java que coloque en forma automática el archivo compilado `.class` en el destino que usted quiera. Para hacer lo anterior, debe invocar al compilador desde un comando de despliegue con la opción `-d`, como sigue:

```
javac -d <class-path> <source-code>
```

Para el ejemplo `Car`, lo anterior sería:

```
javac -d E:/myjava Car.java
```

El nombre de ruta completo del directorio que obtiene el código compilado es *<class-path>/<package-path>*. Si el directorio destino ya existe, entonces el archivo generado `.class` va a ese directorio. Si el directorio destino no existe, el compilador crea en forma automática el directorio requerido y luego inserta ahí el archivo generado `.class`. Por tanto, si el lector planea usar esta opción, no necesita crear explícitamente la estructura de directorio en la figura A4.2. Puede dejar que el compilador lo haga. IDE típicas también proporcionan formas para hacer lo anterior.

Si el lector está desarrollando una aplicación de Java para que otros la usen, quizá sea conveniente organizar las clases de su aplicación en una estructura de paquete y almacenarla en los archivos `.class` en una estructura de directorio correspondiente, como ya se describió. Una vez que termine el desarrollo de su aplicación, es muy sencillo usar un programa como WinZip para comprimir todos los archivos de su aplicación en un solo archivo, denominado tal vez `IPCJ.zip`. Después de descargar este archivo `.zip`, su cliente puede insertar este archivo `.zip` en su estructura de directorio y luego establecer una ruta de clase hacia ese archivo `.zip` que incluya el nombre del archivo `.zip` en sí como el elemento final en la ruta de clase. Por ejemplo, si el archivo `IPCJ.zip` está en su directorio `C:/myJava` del cliente, entonces la `CLASSPATH` de su cliente debe incluir esta ruta de clase:

```
C:/myJava/IPWJ.zip
```

Esto permite que el compilador Java de su cliente acceda a todas las clases de paquete mientras éstas se encuentran en forma comprimida. Observe que la ruta de clase a un archivo `.zip` debe incluir el archivo `.zip` en sí, pero si el lector desea descomprimir el archivo, la ruta de clase debe ir sólo al directorio contenedor.

Por supuesto, esto también funciona en la otra dirección. Si el lector desea adquirir una aplicación de Java desarrollada por otra persona, quizá tenga sus clases preempacadas y comprimidas en un archivo `.zip` (o algún otro tipo de *archivo*). En ese caso, quizás el lector pueda colocar ese archivo `.zip` comprimido donde quiera en la estructura de directorio de su computadora y luego simplemente agregar una ruta de clase hacia ese archivo `.zip`, con el nombre de archivo `.zip` como el elemento final en esa ruta de clase.

APÉNDICE 5

Convenciones de estilo de codificación en Java

En este apéndice se describen las convenciones de estilo de codificación en Java. La mayor parte de estas directrices son bastante aceptadas. No obstante, en ciertas áreas existen directrices alternas. Las convenciones de estilo de codificación presentadas en este documento son, en su mayor parte, un subconjunto simplificado de las convenciones de estilo de codificación presentadas en esta página:

http://java.sun.com/docs/codeconv/

Si el lector tiene alguna pregunta sobre el estilo que no se haya abordado en este documento, consulte la página mencionada.

Cuando lea las siguientes secciones, consulte el programa ejemplo en la última sección. Es posible imitar el estilo de ese ejemplo.

Prólogo

1. Coloque esta sección del prólogo en la parte superior del archivo:

```
/**********************************************
* <nombre del archivo>
* <nombre del programador>
*
* <descripción del archivo>
**********************************************/
```

2. Incluya una línea en blanco abajo de la sección del prólogo.

Delimitación de sección

1. Después de las definiciones de las variables de estado, y entre las definiciones del constructor y del método, introduzca una línea de asteriscos, como ésta:

```
//*********************************************************
```

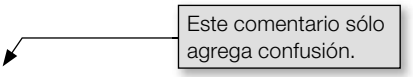
   Deje una línea en blanco arriba y abajo de esta línea de asteriscos.

2. Dentro de un gran constructor o método, inserte líneas en blanco entre las secciones lógicas del código. Por ejemplo, a menos que los ciclos sean pequeños y estén estrechamente relacionados, introduzca una línea en blanco entre el fin de un ciclo y el inicio de otro.

Comentarios insertados

1. Proporcione comentarios para código que pudieran confundir a alguien que lea por primera vez su programa. Suponga que el lector comprende la sintaxis de Java.

2. No comente código que es evidente. Por ejemplo, este comentario es innecesario y por tanto exhibe un estilo deficiente:




```
for (int i=0; i<10; i++)   // para el encabezado del ciclo
```

3. Escriba sus programas en código autodocumentado claro para reducir la necesidad de comentarios. Por ejemplo, use nombres descriptivos de identificación mnemónica (descriptivos).

4. Siempre incluya un espacio entre // y el texto de comentario.

5. La longitud del comentario determina el formato.

- Si el comentario ocupa más de una línea, use líneas completas, como éstas:

```
// Éste es un bloque de comentario. Úselo para comentarios
// que ocupan más de una línea. Observe el alineamiento
// para /'s y palabras.
```

- Si un comentario debe estar en una sola línea, colóquelo arriba de la línea de código que describe. Sangre las // igual que la línea de código descrita. Incluya una línea en blanco arriba de la línea del comentario. Éste es un ejemplo:

```
// Error de desplegado si el nombre del archivo es inválido.
if (fileName == null || filename.getName().equals(""))
{
  JOptionPane.showMessageDialog(Este comentario sólo agrega confusión);
}
```

- Muchos comentarios son suficientemente pequeños para caber en el espacio a la derecha del código que describen. Siempre que sea posible, todos estos comentarios deben empezar en la misma columna, en la medida de lo posible. El siguiente ejemplo muestra el posicionamiento idóneo para comentarios breves.

```
float testScores = new float[80]; // el índice es el número del
estudiante;

...
while (testScores[student] >= 0)  // se sale del índice negativo.
{
  testScores[student] = score;
  ...
```

6. Proporcione un comentario final para cada llave que sea un número importante de líneas (¿cinco o más?) debajo de su paréntesis de llave de apertura coincidente. Por ejemplo, observe el siguiente comentario `// end for row` y `// end getSum`:

```
public double getSum(float table[][], int rows, int cols)
{
  double sum = 0.0;

  for (int row=0; row<rows; row++)
  {
    for (int col=0; col<cols; col++)
    {
      sum += table[row][col];
    } // end for col
```

```
    } // end for row

    return sum;
  } // end getSum
```

Declaraciones de variables

1. Normalmente, sólo se debe declarar una variable por línea. Por ejemplo:

```
float avgScore;      // promedio puntaje en el examen
int numOfStudents;   // número de estudiantes en el curso
```

   Excepción:
   Si varias variables están estrechamente relacionadas, es aceptable declararlas juntas en una línea. Por ejemplo:

```
int x, y, z;         // coordenadas de un punto
```

2. Normalmente, es necesario incluir un comentario para cada línea de declaración de variable.

   Excepción:
   No incluya ningún comentario para nombres que son evidentes (por ejemplo, `studentId`) o estándares (por ejemplo, `i` para una variable de índice de ciclo `for`, `ch` para una variable de carácter).

Paréntesis de llaves que rodean una declaración

1. Para los constructos `if`, `else`, `for`, `while` y `do` que ejecutan una sola declaración, es una buena práctica tratar esa sola declaración como si fuese una declaración compuesta y escribirla entre paréntesis de llaves, como esto:

```
for (int i=0; i<scores.length; i++)
{
  sumOfSquares += scores[i] * scores[i];
}
```

2. Excepción:
   Si fuese ilógico agregar *a posteriori* otra declaración al constructo, las llaves pueden omitirse cuando la omisión mejore la legibilidad. Por ejemplo, esto es aceptable para un programador experimentado:

```
for (; num>=2; num--)
  factorial *= num;
```

Colocación de paréntesis de llaves

1. Coloque paréntesis de llaves de apertura y cierre por sí mismas de modo que estén alineadas con la línea arriba de las llaves de apertura. Para ciclos `do`, coloque la condición `while` en la misma línea que las llaves de cierre.

2. Ejemplos:

```
public class Counter
{
  <field-and-method-declarations>
}


if (...)
{
  <statements>
```

```
}
else if (...)
{
  <statements>
}
else
{
  <statements>
}

for/while (...)
{
  <statements>
}

do
{
  <statements>
} while (...);

switch (...)
{
  case ... :
    <statements>
    break;
  case ... :
    <statements>
    break;
    ...
  default:
    <statements>
}

int doIt()
{
  <statements>
}
```

3. El alineamiento de los paréntesis de llaves es una cuestión polémica. Las convenciones de código Sun de Java en la red recomiendan colocar la llave de apertura al final de la línea previa. Éste es un sitio en el que estas convenciones de documentación no coinciden con las convenciones de Sun. Recomendamos colocar la llave de apertura en su propia línea porque eso ayuda a que las declaraciones compuestas destaquen.
4. Para constructores con cuerpo vacío, coloque las llaves de apertura y de cierre en la misma línea y sepárelas con un espacio en blanco, como esto:

```
public Counter()
{ }
```

El constructo `else if`

1. Si el cuerpo de un `else` está arriba de otro `if`, forme un constructo `if` (coloque el `else` y el `if` en la misma línea). Vea la sección previa sobre colocación de llaves para consultar un ejemplo de un constructo `else if`.

Alineamiento y sangría

1. Todo el código debe alinearse de modo que esté lógicamente al mismo nivel. Vea la sección previa sobre colocación de llaves para consultar ejemplos de alineamiento idóneo.
2. Al código debe darse una sangría de modo que esté lógicamente dentro de otro código. Es decir, para lógica anidada, use sangría anidada. Por ejemplo:

```
for (...)
{
  while (...)
  {
    <statements>
  }
}
```

3. El ancho de la sangría que puede usarse es de dos o cinco espacios. Una vez que se escoge un ancho para la sangría, debe mantenerse. Use el mismo ancho para la sangría en todo su programa.
4. Cuando una declaración es demasiado larga y no cabe en una línea, escríbala en múltiples líneas, de modo que las líneas de continuación tengan una sangría idónea. Si la declaración larga es seguida por una declaración sencilla que esté lógicamente dentro de la declaración larga, use llaves para abarcar la declaración sencilla. Use cualquiera de las técnicas siguientes para sangrar las líneas de continuación:
   - Sangre una posición de columna de modo que entradas semejantes estén alineadas. En el siguiente ejemplo, los entes que están alineados son las tres llamadas a un método:

```
while (bucklingTest(expectedLoad, testWidth, height) &&
       stressTest(expectedLoad, testWidth) &&
       slendernessTest(testWidth, height))
{
  numOfSafeColumns++;
}
```

   - Sangre el mismo número de espacios que todas las demás sangrías. Por ejemplo:

```
while (bucklingTest(expectedLoad, testWidth, height) &&
  stressTest(expectedLoad, testWidth) &&
  slendernessTest(testWidth, height))
{
  numOfSafeColumns++;
}
```

Múltiples declaraciones en una línea

1. Normalmente, cada declaración debe escribirse en una línea por separado:

   Excepción:
   Si las declaraciones están estrechamente relacionadas y son muy breves, es aceptable (aunque no es un requisito) colocarlas juntas en una línea. Por ejemplo,

```
a++; b++; c++;
```

2. Para declaraciones de asignación que están estrechamente relacionadas y usan el mismo valor asignado, es aceptable (pero no necesario) combinarlas en una declaración de asignación. Por ejemplo:

```
x = y = z = 0;
```

Espacios en una línea de código

1. Nunca ponga un espacio a la izquierda de un punto y coma.

2. Paréntesis:

   - Nunca introduzca un espacio dentro de los paréntesis.
   - Si el ente a la izquierda de un paréntesis izquierdo es un operador o una palabra clave constructo (`if`, `switch`, etc.), entonces preceda el paréntesis con un espacio.
   - Si el ente a la izquierda de un paréntesis izquierdo es el nombre de un método, entonces no preceda el paréntesis con un espacio.

   Por ejemplo: preceda el paréntesis con un espacio.

```
if ((a == 10) && (b == 10))
{
  printIt(x);
}
```

3. Operadores:

   - Normalmente, un operador debe estar rodeado por espacios. Por ejemplo:

```
if (response == "avg")
{
  y = (a + b) / 2;
}
```

   - Casos especiales:
     - Expresiones complejas:
       - Dentro de un componente interno de una expresión compleja, no rodee los operadores de las componentes con espacios.
       - Dos apariciones comunes de expresiones complejas son expresiones condicionales y encabezados de ciclos `for`. Vea los siguientes ejemplos.
     - Operador punto: no hay espacios a su izquierda o derecha.
     - Operador coma: no hay espacio a su izquierda.
     - Operadores unarios: no hay espacio entre un operador unario y su operando asociado.

   Por ejemplo:

```
if (zeroMinimum)
{
  x = (x<0 ? 0 : x);
}

while (list1.row != list2.row)
{
  <statements>
}

for (int i=0,j=0; i<=bigI; i++,j++)
{
  <statements>
}
```

Operadores shortcut

1. Use operadores incremento y decremento en lugar de sus formas más largas equivalentes. Por ejemplo:

No use	Use esto
`x = x + 1`	`x++` or `++x` (dependiendo del contexto)
`x = x - 1`	`x--` or `--x` (dependiendo del contexto)

2. Use asignaciones compuestas en lugar de sus formas más largas equivalentes. Por ejemplo:

No use	Use esto
`x = x + 5`	`x += 5`
`x = x * (3 + y)`	`x *= 3 + y`

Convenciones de nombrado

1. Use nombres con sentido para sus identificadores.
2. Para constantes nombradas, siempre use mayúsculas. Si hay varias palabras, use guiones bajos para separarlas. Por ejemplo:

```
public static final int SECONDS_IN_DAY = 86400;
private final int ARRAY_SIZE;
```

3. Para nombres de clases (y sus constructores asociados), use mayúsculas para la inicial y minúsculas para las demás letras. Si en el nombre de clase hay varias palabras, use mayúsculas para la inicial de todas las palabras. Por ejemplo:

```
public class InnerCircle
{
  public InnerCircle(radius)
  {
     <constructor-body>
  }
```

4. Para todos los identificadores que no sean constantes ni constructores, siempre use minúsculas. Si en el identificador hay varias palabras, use mayúsculas en la primera letra de todas las palabras a continuación de la primera palabra. Por ejemplo:

```
float avgScore;      // promedio puntaje en el examen
int numOfStudents;   // número de estudiantes en el curso
```

Organización de los métodos y del constructor

1. Normalmente, la definición de cada método debe ir precedida por una sección de prólogo. El prólogo del método contiene:

   - una línea en blanco
   - una línea de asteriscos
   - una línea en blanco
   - una descripción del propósito del método
   - una línea en blanco
   - descripciones de parámetros (para los parámetros no evidentes)
   - una línea en blanco

   Idealmente, todos los parámetros de métodos deben usar nombres suficientemente descriptivos, de modo que el propósito de cada parámetro sea intrínsecamente evidente. No obstante, si éste no es el caso, entonces es necesario incluir una lista de parámetros y sus descripciones en un prólogo de método arriba del encabezado del método. Por ejemplo, en un programa de juego de gato, un método que maneja un movimiento de un jugador normalmente suele ser complicado, por lo que tal vez requiera un prólogo de método como éste:

```
//*****************************************************

// Este método solicita al usuario que introduzca un movimiento, valida
// la entrada y refleja ese movimiento en el tablero. También verifica
// si el movimiento es un movimiento ganador.
//
```

```
// Parameters: tablero: tablero del juego de gato/arreglo
//             jugador: contiene al jugador actual ('X' u 'O')

public void handleMove(char[][] board, char player)
{
```

En el supuesto de que el lector describa las variables de instancia y las variables de clase al declararlas, no debe proporcionar prólogos para accesores, controladores y constructores "triviales" que simplemente leen o escriben desde o hacia variables de instancia y de clase. Por otra parte, si un controlador realiza una validación sobre un parámetro antes de asignarlo a su variable de instancia asociada, entonces no es trivial, por lo que es necesario incluir un prólogo en él. El mismo razonamiento es válido para un constructor. Un constructor con una sola asignación no debe tener un prólogo. Un constructor de validación debe tener un prólogo.

2. En aras de agrupar cosas semejantes entre sí, es posible omitir las líneas de asteriscos entre constructores triviales, y también es necesario omitir líneas de asteriscos entre métodos controladores y de acceso.

Suponga que una clase contiene dos constructores triviales, varios métodos controladores y de acceso, y otros dos métodos simples. Éste es el marco para una clase así:

```
<class-heading>
{
   <instance-variable-declarations>

   //*********************************************************

   <trivial-constructor-definition>

   <trivial-constructor-definition>

   //*********************************************************

   <mutator-definition>

   <mutator-definition>

   <accessor-definition>

   <accessor-definition>

   //*********************************************************

   <simple-method-definition>

   //*********************************************************

   <simple-method-definition>
}
```

En el marco anterior, observe que no hay descripciones para constructores, reguladores o para métodos simples. Observe también que arriba del primer regulador hay una línea de asteriscos, pero no arriba de los reguladores y accesores subsiguientes. Estas omisiones ayudan a que el programa sea más legible al agrupar cosas semejantes. Observe también que arriba de los dos métodos simples en la parte inferior de la clase no hay comentarios, aunque sí hay líneas de asteriscos.

3. Coloque las declaraciones de variables inmediatamente abajo del encabezado del método. No coloque declaraciones de variables locales dentro del código ejecutable.

Excepción: Declare una variable de índice de ciclo `for` dentro de su encabezado de ciclo `for`.

Organización de clase

1. Cada una de sus clases puede contener los siguientes artículos (en el orden mostrado):
   a) sección de prólogo de la sección
   b) declaraciones `import`
   c) variables de clase constantes
   d) variables de clase no constantes
   e) variables de instancia
   f) métodos abstractos
   g) constructores
   h) métodos de instancia
   i) métodos de clase
2. Normalmente, el lector debe colocar un método `main` y cualquiera de sus métodos de ayuda en su propia clase controladora. Pero algunas veces es correcto incluir un breve método `main` dentro de la clase que controla como herramienta de prueba insertada. Este método debe colocarse al final de la definición de la clase.

Programa Java de muestra

```
/*****************************************************************
* Student.java
* Dean & Dean
*
* Esta clase maneja el procesamiento del nombre de un estudiante.
*****************************************************************/

import java.util.Scanner;

public class Student
{
  private String first = ""; // nombre del estudiante
  private String last = "";  // apellido del estudiante

  //*************************************************************

  public Student()
  { }

  // Este constructor verifica que cada nombre que se ha pasado
  // empiece con mayúscula y siga con minúsculas.

  public Student(String first, String last)
  {
    setFirst(first);
    setLast(last);
  }
```

Figura A5.1*a* Clase `Student`, usada para ilustrar las convenciones de codificación, parte A.

```
  //*************************************************************

  // Este método verifica que cada nombre que se ha pasado empiece
  // con mayúscula y siga con minúsculas.

  public void setFirst(String first)
  {
    // [A-Z][a-z]* is a regular expression. See API Pattern class.
    if (first.matches("[A-Z][a-z]*"))
    {
      this.first = first;
    }
    else
    {
      System.out.println(first + " es un nombre inválido.\n" +
        "Los nombres deben empezar con una mayúscula y tener" +
        " minúsculas a partir de ahí.");
    }
  } // end setFirst

  //*************************************************************

  // Este método verifica que el apellido empiece con mayúscula
  // y a partir de ahí sean minúsculas.

  public void setLast(String last)
  {
    // [A-Z][a-z]* es una expresión normal. Consulte la clase Pattern de API.
    if (last.matches("[A-Z][a-z]*"))
    {
      this.last = last;
    }
    else
    {
      System.out.println(last + " es un nombre inválido.\n" +
        "Los nombres deben empezar con una mayúscula y tener" +
        " minúsculas a partir de ahí.");
    }
  } // end setLast

  //*************************************************************

  // Imprime el nombre y el apellido del estudiante.

  public void printFullName()
  {
    System.out.println(first + " " + last);
  } // end printFullName
} // end class Student
```

Figura A5.1*b* Clase `Student`, usada para ilustrar las convenciones de codificación, parte B.

```
/************************************************
* StudentDriver.java
* Dean & Dean
*
* Esta clase funciona como un driver para la clase Student.
************************************************/

public class StudentDriver
{
  public static void main(String[] args)
  {
    Student s1;  // first student
    Student s2;  // second student

    s1 = new Student();
    s1.setFirst("Adeeb");
    s1.setLast("Jarrah");
    s2 = new Student("Heejoo", "Chun");
    s2.printFullName();
  } // end main
} // end class StudentDriver
```

Figura A5.2 Clase `StudentDriver`, usada con la clase `Student` en las figuras A5.1*a* y A5.1*b*.

APÉNDICE 6

Javadoc

En el apéndice 5 se describe un estilo de programación que se ha optimizado para la presentación de código en un texto introductorio y para que los estudiantes escriban programas relativamente simples. La mayor parte de las sugerencias se trasladan a una práctica de programación profesional. Pero hay algunas excepciones notables. En programación profesional es necesario proporcionar información de interfaces sobre clases ya compiladas como la documentación que Sun proporciona para sus clases API de Java.

El `javadoc` ejecutable viene en el mismo directorio que los ejecutables `javac` y `java`, de modo que si se ejecutan `javac` y `java`, también puede ejecutarse `javadoc`. Para ejecutar `javadoc`, en un comando de desplegado, debe introducirse este comando

```
javadoc -d <output-directory> <source-files>
```

La opción `-d` <*output-directory*> ("d" significa "destino") hace que la salida vaya a otro directorio. Si se omite la opción `-d`, por defecto la salida se dirige al directorio actual, pero no es una buena idea, ya que `javadoc` crea muchos archivos que pueden enredar el directorio actual. Es posible colocar documentación para más de una clase en el mismo directorio. Use espacios para separar múltiples nombres de archivos fuente con espacios.[1]

Suponga que se quiere generar documentación de interfaces sobre la clase `Student` cuyo código fuente se presenta en la figura A5.1 del apéndice 5. En el supuesto de que actualmente se está en el directorio que contiene el código fuente, y suponiendo que se desea que la salida de `javadoc` se dirija a un subdirectorio denominado `doclase`, el comando se vería como esto:

```
javadoc -d docs Student.java
```

Para ver la salida, abra un buscador de Red como Windows Explorer, navegue al archivo `doclase` y haga clic en `index.html`. La figura A6.1 muestra la parte superior del documento de interfaz que crea `javadoc`: la información "Summary". Este documento de interfaz contiene una cantidad impresionante de información, aunque no toda es necesaria. Por ejemplo, no incluye el comentario en la última línea del prólogo que describe la clase en general; no incluye el comentario prólogo que describe el constructor de dos parámetros, y no incluye los comentarios que describen los tres métodos.

Para que `javadoc` pueda extraer esta otra información del código fuente, es necesario que toda esa información de interfaz esté ubicada inmediatamente arriba del encabezado de lo que esté describiendo. También se requiere que esta información esté encerrada en un comentario `javadoc` *block-comment* que empiece con un simple *slash* seguido por dos asteriscos y que termine con un simple asterisco seguido por un simple *slash* hacia delante, como esto:

```
/** <extractable-information> */
```

Puesto que la figura A5.1 contiene una declaración `import` entre el prólogo general y el encabezado de clase, es necesario mover el comentario general fuera del prólogo general y ponerlo en un comentario de bloque `javadoc` ubicado justo arriba del encabezado de clase. En forma semejante, el constructor individual y la información de interfaz del método en los comentarios del bloque `javadoc` situados

[1] Para ver otras opiniones y otros posibles argumentos, entre usted mismo a `javadoc`.

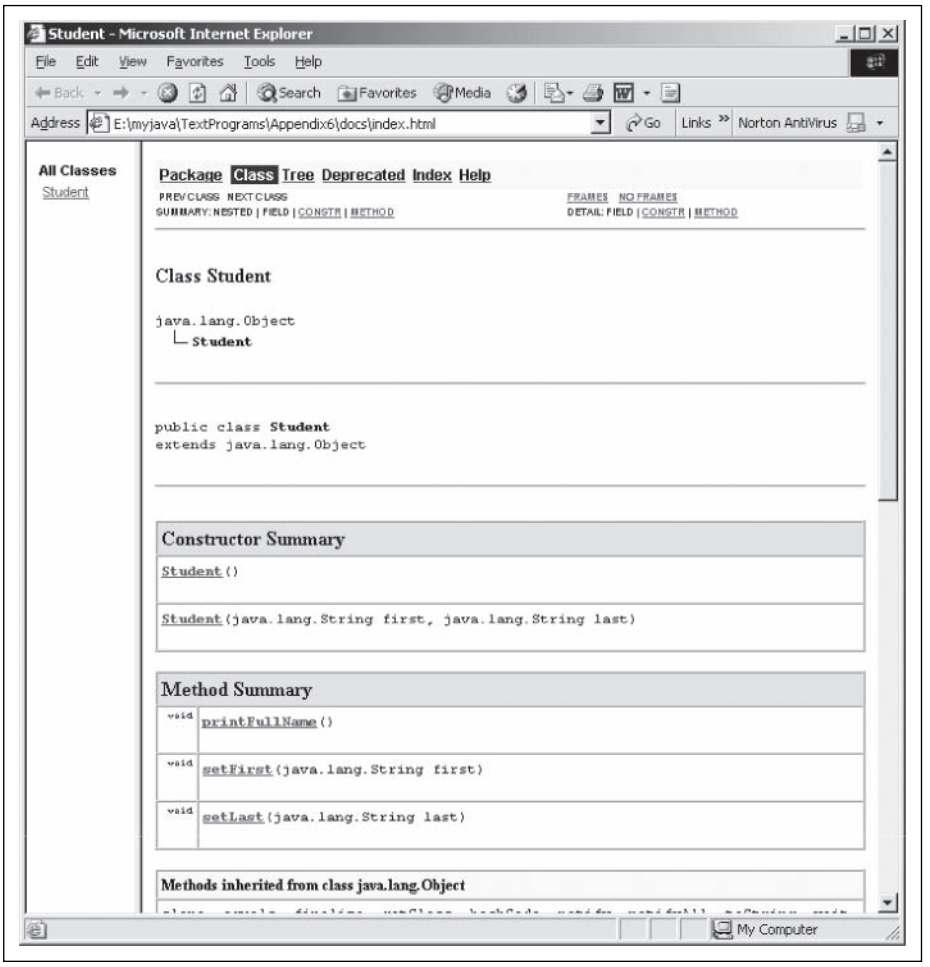


Figura A6.1 Parte superior de `javadoc` de la salida para la clase `Student` definido en la figura A5.1

justo arriba de sus encabezados respectivos. Hay algo de flexibilidad. La información que puede extraerse en uno de estos comentarios del bloque `javadoc` no necesariamente debe estar en una sola línea. También, si se desea, el /** de apertura y el /* de cierre pueden colocarse en líneas arriba y abajo del texto, como se muestra en la figura A6.2.

Con estos cambios implementados en el código `Student.java`, la figura A6.3 muestra la parte superior de lo que genera `javadoc`. Si esto se compara con la figura A6.1, se verá que la figura A6.3 incluye el comentario general para toda la clase y el comentario especial para el constructor de dos parámetros. También se cambió el resto del código de modo que `Student_jd` también tenga comentarios del bloque `javadoc` /** . . . */ arriba de los encabezados del método. En consecuencia, la salida `javadoc` también incluye comentarios especiales para cada método. Los comentarios del constructor y del método también aparecen en las partes "Detail" del desplegado de salida, que está abajo del que se observa en las figuras A6.1 y A6.3.

Dentro de un bloque de comentario `javadoc` bloque /** . . . */, `javadoc` también reconoce varias *etiquetas* especiales, que le permiten extraer otros tipos de información. Para una descripción completa, consulte

http://java.sun.com/j2se/javadoc/

En la figura A6.4 se presenta una lista abreviada de etiquetas `javadoc`.

Las etiquetas más importantes son las *@param* y *@return*. En la figura A6.5 se muestra una clase originalmente definida en la figura 13.11, pero con comentarios modificados por `javadoc`. La funcio-

```
/*****************************************************************
* Student_jd.java
* Dean & Dean
*****************************************************************/

import java.util.Scanner;

/** Esta clase maneja el procesamiento del nombre de un estudiante. */

public class Student_jd
{
  private String first = ""; // nombre del estudiante
  private String last = "";  // apellido del estudiante

  //***************************************************************

  public Student_jd()
  { }

  /**
  Este constructor verifica que cada nombre que se ha pasado empiece
  con mayúscula y luego continúe con minúsculas.
  */

  public Student_jd(String first, String last)
  {
    setFirst(first);
    setLast(last);
  }
```

comentario `javadoc` en una sola línea

comentario `javadoc` en varias líneas

Figura A6.2 Parte superior de la clase `Student` en la figura A5.1, modificada para dar cabida a `javadoc`.

nalidad de esta clase es exactamente la misma que la definida en la figura 13.11. Pero esta versión permite varias características `javadoc`. Observe cómo la descripción general de la clase ha sido movida del prólogo a un bloque de comentario `javadoc` por separado inmediatamente arriba del encabezado de clase. En el bloque de comentario `javadoc` arriba del constructor hay dos descripciones etiquetadas de parámetro. En el bloque de comentario `javadoc` arriba del método hay una descripción etiquetada del valor `return`.

Suponga que el directorio actual contiene un código fuente para la clase `Employee` copiado de la figura 13.11, y que también contiene el código fuente para la clase `Salaried_jd` mostrado en la figura A6.5. Luego, suponga que se abre una ventana *Command Prompt* y se introduce el siguiente comando:

```
javadoc -d docs Employee.java Salaried_jd.java
```

Esto crea información de interfaz para ambas clases y produce esta documentación combinada al subdirectorio `doclase`. En la figura A6.6*a* se presenta lo que se vería si se abre un buscador de Web, se navega al directorio `doclase`, se hace clic en `index.html` y se selecciona `Salaried_jd` en el panel izquierdo bajo "All Classes".

En el panel derecho, cerca de la parte superior, puede verse la documentación de la herencia de `Salaried_jd` de `Employee`. En la documentación `Salaried_jd`, `Employee` se ha coloreado y subrayado en varios sitios. Éstas son ligas, y si se hace clic en cualquiera de ellas, la disposición cambia de inmediato a la documentación de la clase `Employee`. En la figura A6.5, el comentario general tiene dos oraciones, y ambas oraciones aparecen en el comentario general en la figura A6.6*a*. Observe que los blo-

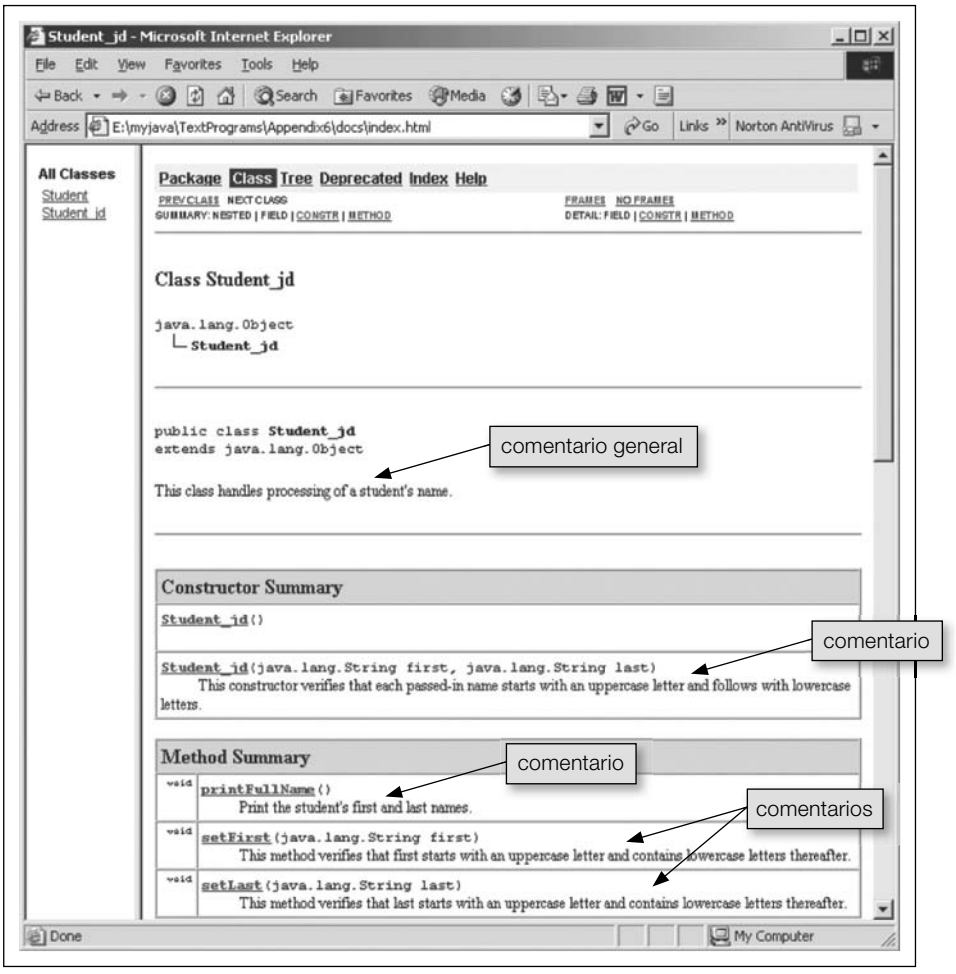


Figura A6.3 Parte superior de una salida `javadoc` para la clase `Student` modificada.

Descripción del parámetro de un constructor o de un método:
`@param` <*parameter-name*> <*explanation*>

Descripción de un valor de retorno:
`@return` <*explanation*>

Descripción de una excepción que pudiera ser lanzada:
`@throws` <*exception_type*> <*explanation*>

Referencia de Hyperlink a otro artículo documentado:
`@see` <*package-name*>.<*class-name*>
`@see` <*package-name*>.<*class-name*>#<*method-name*>(<*type1*>,...)
`@see` <*package-name*>.<*class-name*>#<*variable-name*>

Figura A6.4 Lista abreviada de etiquetas `javadoc`.

```
/*************************************************************
* Salaried_jd.java
* Dean & Dean
*************************************************************/

/**
Esta clase implementa un empleado asalariado.
Tiene la misma funcionalidad que la clase Salaried en el capítulo 13.
*/

public class Salaried_jd extends Employee
{
  private double salary;

  //*********************************************************

  /**
  @param name    nombre de la persona
  @param salary  salario anual en dólares
  */

  public Salaried_jd(String name, double salary)
  {
    super(name);
    this.salary = salary;
  } // end constructor

  //*********************************************************

  /** @return    salario quincenal en dólares  */

  public double getPay()
  {
    return this.salary / 24;
  } // end getPay
} // end class Salaried_jd
```

movido desde el prólogo

comentarios etiquetados

comentario etiquetado

Figura A6.5 Clase `Salaried` de la figura 13.11 modificada para habilitar a `javadoc`.

ques summary del constructor y del método no contienen ningún comentario. Las etiquetas `@param` y `@return` no producen ninguna salida de bloque summary. Si se hubiese incluido texto en el bloque de comentario `javadoc` arriba del encabezado del constructor o del método en la figura A6.5, sólo la primera oración de ese texto (la oración "summary"), hubiera aparecido en el bloque summary correspondiente en la figura A6.6*a*.

Ahora suponga que se usa la barra de desplazamiento a la derecha. Esto muestra lo que se ve en la figura A6.6*b*. Observe que los bloques "Detail" muestran el parámetro etiquetado y devuelven información suministrada en los bloques de comentario `javadoc` arriba de los encabezados del constructor y del método en la figura A6.5. Si se hubiera incluido texto en un bloque de comentario `javadoc` antes del encabezado del constructor o del método en la figura A6.5, todo ese texto hubiera aparecido en el bloque "Detail" correspondiente en la figura A6.6*b*. Por último, observe que `javadoc` también indica que el método `getPay` definido en `Salaried_jd` se sobrepone al método `getPay` definido en `Employee`.

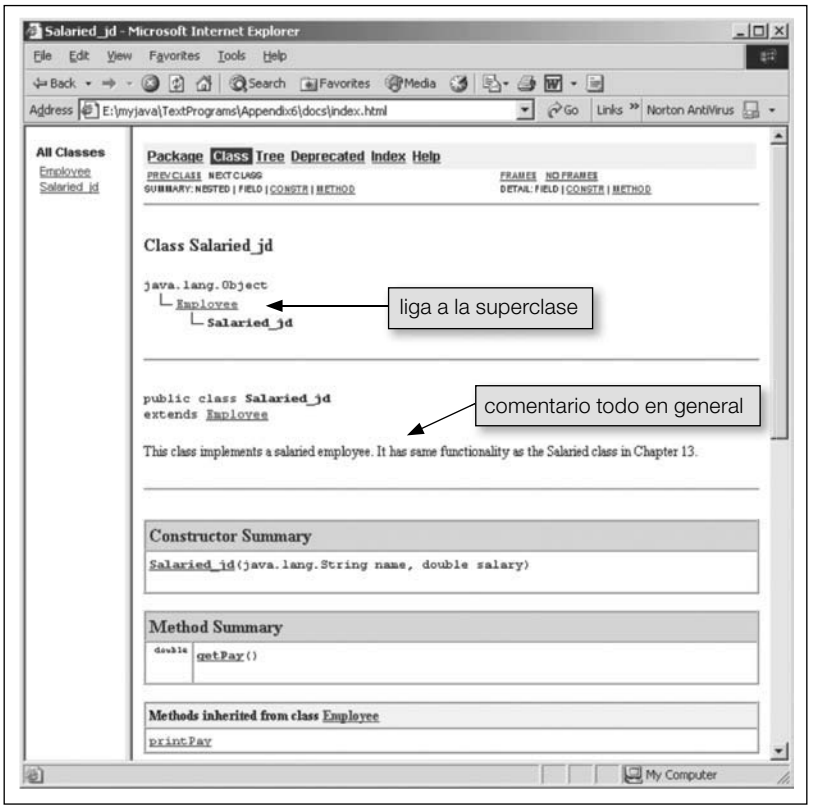


Figura A6.6*a* Salida de `javadoc` para la clase `Salaried` comentada por `javadoc`, parte A.

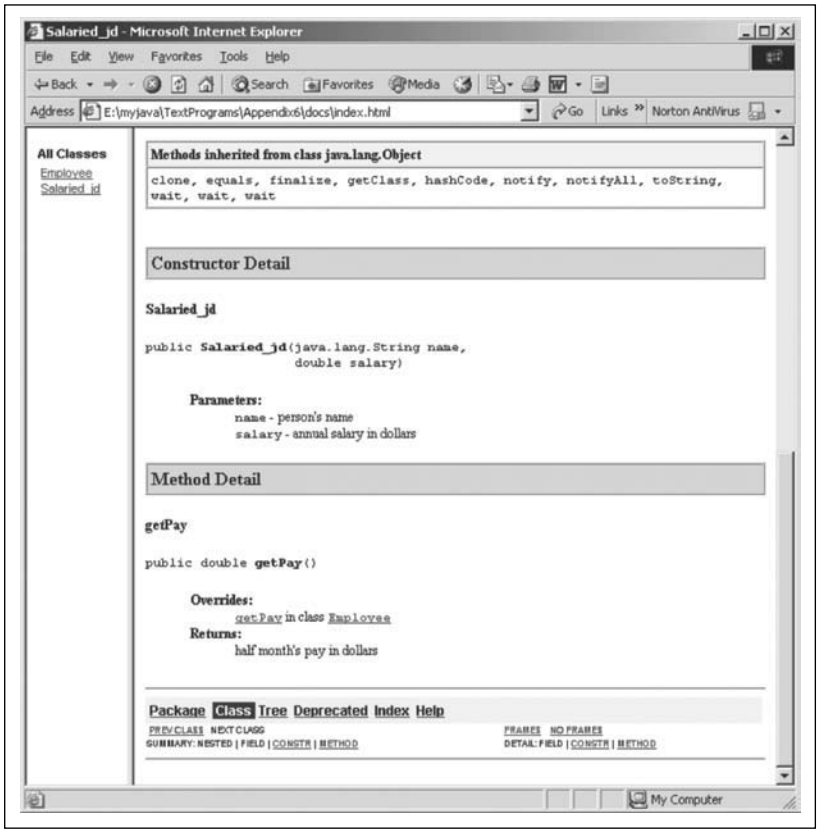


Figura A6.6*b* Salida de `javadoc` para la clase `Salaried` comentada por `javadoc`, parte B.

APÉNDICE 7

Diagramas UML

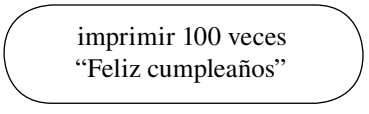
El *Unified Modeling Language* (UML) —Lenguaje Unificado de Modelado— es un lenguaje descriptivo que ayuda a los diseñadores de programas a organizar el tema de la materia de un programa prospectivo orientado a objetos, y proporciona documentación de alto nivel de la estructura y el comportamiento. Es independiente de cualquier lenguaje de programación particular y no se compila en un programa ejecutable. Es simplemente una herramienta organizativa. Fue desarrollada por los "tres amigos": Grady Booch, James Rumbaugh e Ivar Jacobson, en la Rational Software Corp, que ahora forma parte de IBM. Actualmente es mantenido por el consorcio altruista Object Management Group (OMG).

UML especifica muchos tipos de diagramas de visualización.[1] En este apéndice, la atención se centra en sólo dos de éstos: los *diagramas de actividad* (que representan el comportamiento) y los *diagramas de clase* (que representan la estructura). Cuando UML describe el comportamiento, flechas apuntan a lo que ocurre a continuación. Cuando UML describe la estructura, flechas apuntan a lo que suministra soporte, lo cual es opuesto a la dirección del "flujo de información". Así, en el análisis siguiente, el lector debe estar preparado para cambiar la dirección de las flechas a medida que se pase de diagramas de actividad a diagramas de clase.

Diagramas de actividad UML

Los diagramas de actividad son la versión UML de los diagramas de flujo que se presentaron en el capítulo 2. Ilustran el flujo de control de un algoritmo. En la figura A7.1 se muestra un ejemplo de diagrama de actividad UML para el algoritmo Feliz Cumpleaños presentado como diagrama de flujo en la figura 2.9. El círculo negro completo es un *estado inicial*, y el punto blanco en círculo blanco es un *estado final*. Los óvalos representan *estados de acción* o *actividades*. Contienen descripciones informales de acciones coherentes. Las flechas son transiciones. Las etiquetas entre corchetes cerca de algunas de las transiciones son condiciones `boolean` denominadas *guardias*: una transición particular ocurre si y sólo si el valor del guardia adyacente es `true`. Las acciones o actividades mostradas en la figura A7.1 representan operaciones primitivas o de bajo nivel.

A un nivel superior de escala, la actividad mostrada en un solo óvalo puede representar todo un conjunto de acciones. Por ejemplo, es posible usar un solo símbolo de actividad para representar toda la operación de ciclo mostrada en la figura A7.1 como esto:



O bien, es posible usar un solo símbolo de actividad para representar todas las acciones realizadas por un método completo. Se supone que un símbolo de actividad no representa el código en sí. Se supone que representa la "actividad" del código. Así, es apropiado repetir un símbolo de actividad que represente un método completo cuando se llama al método más de una vez.

Cuando hay más de una clase y tal vez varios objetos, UML sugiere que las actividades se organicen en columnas, de modo que todas las actividades para una clase u objeto estén en una columna dedicada a

[1] La especificación completa UML contiene mil páginas. Para consultar una breve introducción de 20 páginas a la especificación UML, consulte: ftp://ftp.software.ibm.com/software/rational/web/whitepapers/2003/intro_rdn.pdf. Para una descripción más completa, consulte: Sinan Si Alhir, *UML In a Nutshell*, O'Reilly, 1998. También consulte: Ivar Jacobson, Grady Booch y James Rumbaugh, *The Unified Software Development Process*, Addison.Wesley, 2005.

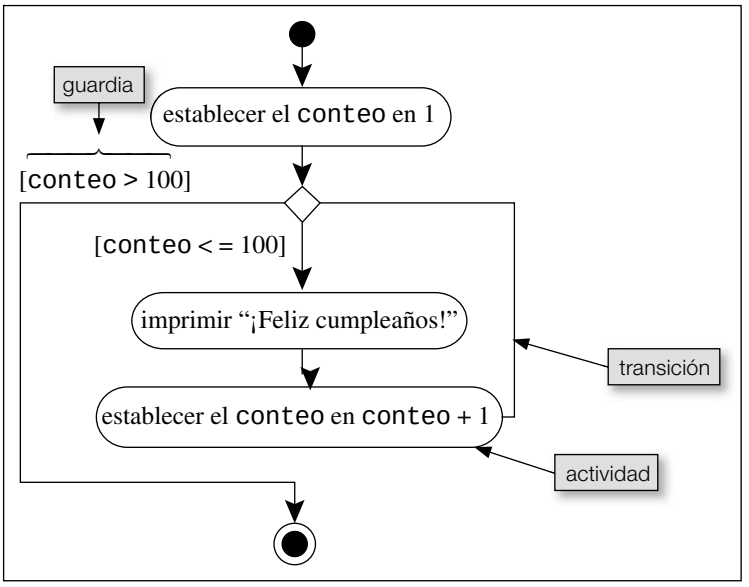


Figura A7.1 Diagrama de actividad UML para el algoritmo Feliz cumpleaños en la figura 2.9.

esa clase u objeto. UML denomina *carriles* a estas columnas. Líneas verticales discontinuas separan carriles adyacentes. En la parte superior del diagrama sobre carril(es) idóneo(s), escriba el nombre de clase para el o los carriles de abajo. Cada nombre de clase debe anteponerse con dos puntos y debe colocarse en un recuadro rectangular por separado. Cuando quiera indicar la instanciación de un objeto, el nombre del objeto debe escribirse seguido por dos puntos y su nombre de clase. Es necesario subrayarlo y colocarlo en un recuadro rectangular situado justo después de la actividad que lo crea.

En la figura A7.2 se muestra el diagrama de actividad UML para el programa Mouse2 definido en las figuras 6.14 y 6.15. Observe cómo cada actividad (óvalo) está alineada bajo su propia clase y (si procede) su propio objeto. Las actividades para los objetos de niveles inferiores típicamente representan métodos completos. Las actividades para los objetos de niveles superiores típicamente representan fragmentos de código. Las flechas negras sólidas representan flujo de control. Siempre van de una actividad a otra actividad. Observe cómo el flujo de control se mueve en forma continua hacia abajo.

Las flechas negras discontinuas representan flujo de datos asociado con cada actividad. Van de una actividad a un objeto o de un objeto a una actividad, pero nunca de una actividad a otra actividad. Estas líneas discontinuas suelen omitirse para no enredar más las cosas, pero puede verse cómo ayudan a mostrar lo que hacen las actividades. Por ejemplo, observe cómo la línea discontinua del objeto "Mickey: Mouse2" a la actividad "imprimir los atributos de mickey" ayuda a explicar lo que ocurre y permite suprimir las dos llamadas al método "get" insertadas en la declaración de impresión:

```
System.out.printf("Age = %d, weight = %.3f\n",
  mickey.getAge(), mickey.getWeight());
```

La introducción de los constructores en el capítulo 7 posibilita la inclusión de la actividad "establecer la tasa de crecimiento" dentro de la actividad "crear ratón". Esto debe sustituir las dos transiciones superiores de cruce de carriles por una sola transición a partir de la actividad "crear ratón" a la actividad "introducir los días" en el mismo carril de la izquierda. Minimizar el cruce de carriles es un buen objetivo de diseño.

Diagramas de clase y de objeto UML

Desde el inicio del capítulo 6, gradualmente se introdujeron varias características de los diagramas de clase UML. Los diagramas de objeto UML son semejantes, excepto porque el título (nombre del objeto seguido por dos puntos seguido por el nombre de la clase) está subrayado, como en el diagrama de actividad UML en la figura A7.2. Un bloque de objetos no incluye un compartimiento de métodos, y sólo las variables de interés actual deben enumerarse en el compartimiento de atributos. Los diagramas de objeto son instantáneas dependientes del contexto, cuyos valores de atributo son valores actuales, más que valo-

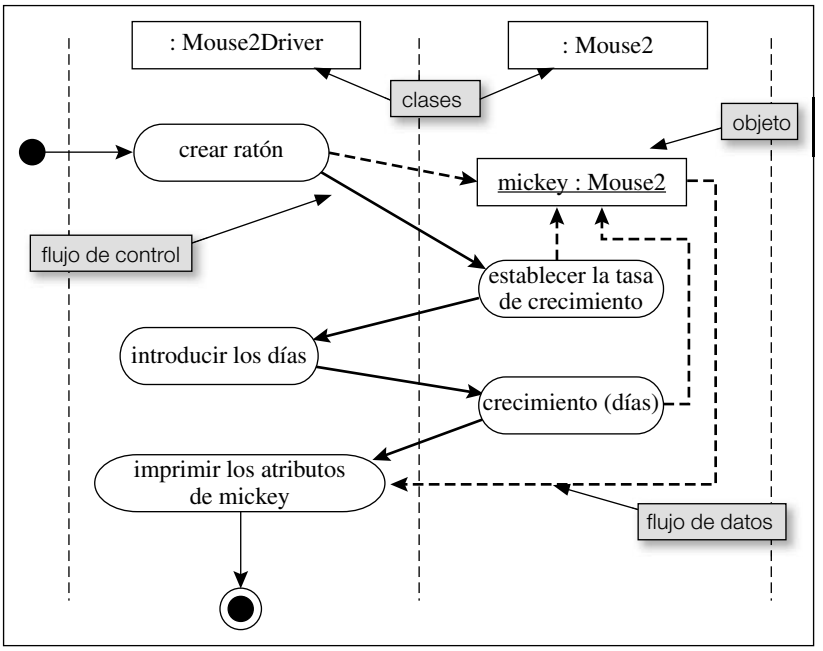


Figura A7.2 Diagrama de actividad UML para el algoritmo Mouse2 en las figuras 6.14 y 6.15. Los óvalos son actividades. Los rectángulos son clases u objetos: los objetos están subrayados. Las líneas verticales discontinuas grises separan a *carriles* adyacentes, con un carril para cada clase u objeto. Las flechas negras sólidas representan flujo de control. Las flechas negras discontinuas representan flujo de datos.

res iniciales. Los diagramas de clase tienen aplicaciones más generales, y a partir de este momento la atención se centra en ellos.

Se usará un ejemplo exhaustivo para resumir la mayor parte de las características de los diagramas de clase UML presentados a lo largo de la mayor parte del libro. El ejemplo que se usará es el programa Payroll3 descrito en la sección 13.9. En la figura A7.3 se muestra un diagrama del primer corte donde cada clase está representada por un solo compartimiento rectangular que sólo contiene el nombre de la clase. Las líneas continuas trazadas entre clases relacionadas son simples líneas de asociación. Una línea de asociación (sin ornamentos) implica conocimiento bidireccional: la clase en cada extremo conoce la clase en el otro extremo. Por tanto, una simple línea indica que las dependencias son mutuas, pero no dice nada sobre la naturaleza de la relación entre clases conectadas.

A medida que se avanza en el desarrollo del diseño, se definen las descripciones de clase, quizá tomando la decisión de hacer abstractas algunas de las clases o de convertirlas en interfaces. Además, muchas de las líneas de asociación se modificarán al añadir símbolos especiales que describen tipos especiales de relación. Además, quizá convenga agregar puntas de flecha con lengüetas a fin de convertir asociaciones de bidireccionales a unidireccionales y hacer que las dependencias vayan en una sola dirección. Las dependencias unidireccionales son preferibles a las dependencias bidireccionales porque simplifican la gestión del software: es menos probable que los cambios de software a una clase requieran cambios a otras clases.

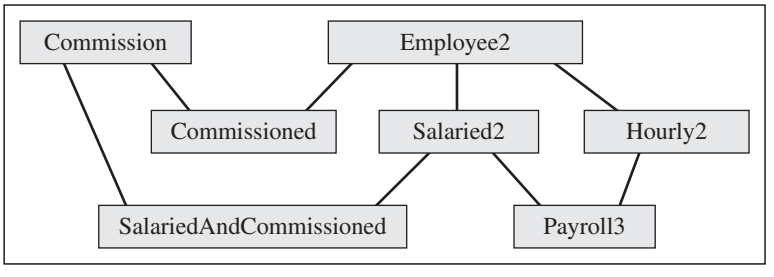


Figura A7.3 Diagrama de primer corte para Payroll3 de la sección 13.9.

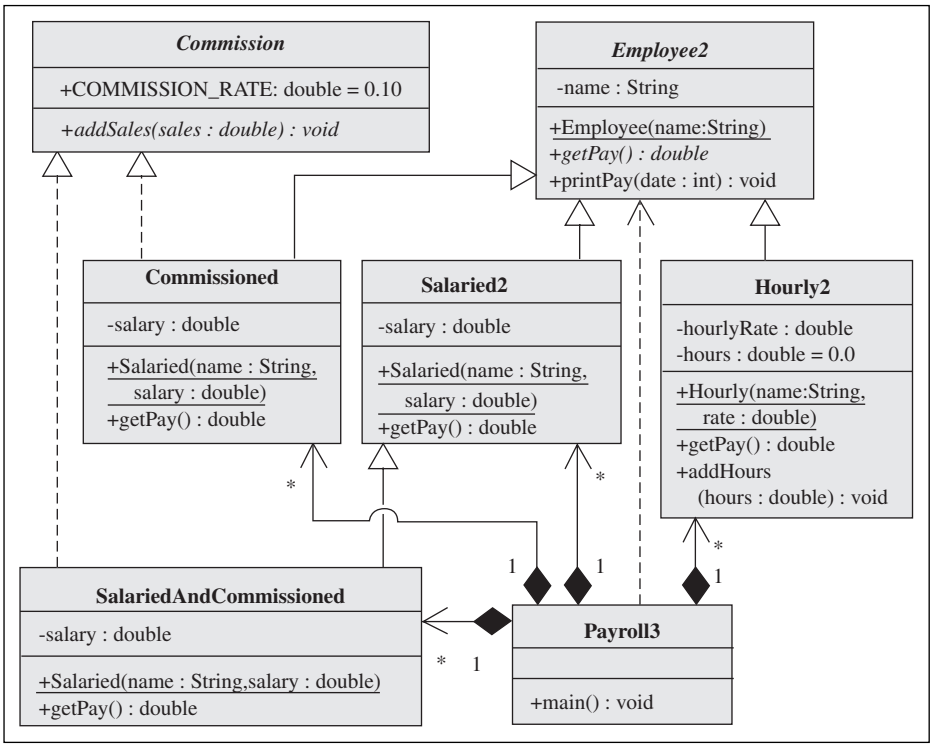


Figura A7.4 Diagrama de clase UML para el programa Payroll3.
Esto muestra herencia de clases e implementación de una interfaz. También muestra composición. Puesto que muchas líneas de asociación en esta figura tienen algún tipo de punta de flecha, todas estas asociaciones son unidireccionales. La línea de asociación discontinua entre `Payroll3` y `Employee2` es una simple dependencia. Eso significa que el tipo `Employee2` aparece en la sentencia de un parámetro o en algún lugar de la variable local en el código `Payroll3`.

La figura A7.4 contiene una versión definida y modificada del diagrama de clase UML de primer corte en la figura A7.3. Observe las cursivas en los nombres de clase `Commission` y `Employees2`. Esto significa que tienen por lo menos un método `abstract` y no pueden ser instanciados. También se escriben con cursivas todos los métodos `abstract` que contienen. Luego, observe las puntas de flecha vacías, que indican herencia. Las puntas de flecha de herencia indican extensión de una clase. Las puntas de flecha de herencia sobre líneas discontinuas indican implementación de una interfaz. Las puntas de flecha apuntan en la dirección de generalización: hacia el ente más general. Los entes más específicos saben sobre los entes más generales y dependen de ellos. Debido a esta dependencia, cambios a clases ancestro o cambios a interfaces pueden obligar a cambios a clases descendientes o de implementación. Por otra parte, puesto que una clase ancestro o una interfaz no saben nada acerca de sus descendientes, los cambios en descendientes o implementaciones nunca obligan a cambios en ancestros o interfaces. La herencia es automáticamente una asociación unidireccional.

Ahora considere los indicadores de composición.[2] Aquí se eligió mostrarlos como composiciones (diamante sólido) en lugar de como agregaciones (diamante hueco) porque la clase que instancia las componentes (`Payroll3`) insertan componentes anónimas en su arreglo contenedor. Todas las líneas de composición tienen multiplicidades. Esto indica que siempre hay exactamente una nómina y que puede haber cualquier número de empleados de cualquiera de los cuatro tipos. Puesto que `Hourly2`, `Salaried2`, `Commissioned` y `SalariedAndCOmmissioned` descienden de la clase `Employee2`, es posible colocar las instancias de todas estas cuatro clases en un arreglo común `Employees2`, como se hace en la definición de clase `Payroll3` en la figura 13.17.

[2] Observe cómo la línea de asociación entre `Payroll3` y `Commissioned` hace un arco sobre la línea de asociación entre `SalariedAndCommissioned` y `Salaried2`. Este detalle UML ayuda a distinguir un cruce de un empalme.

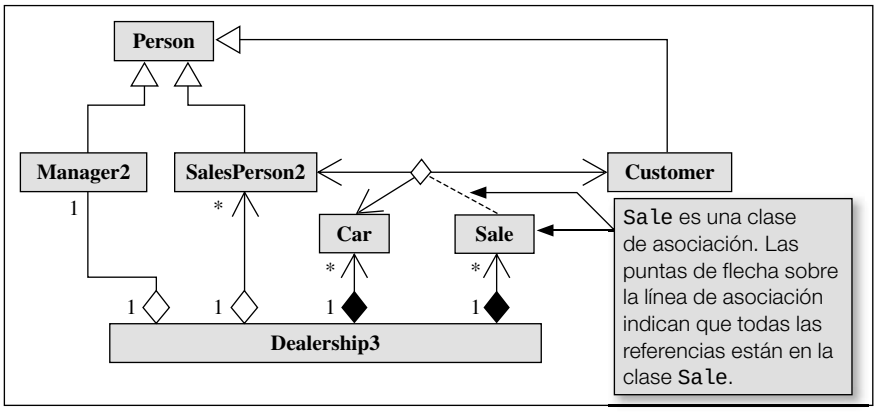


Figura A7.5 Versión mejorada del diagrama de clase de la figura 12.22.
Una punta de flecha sobre una línea de asociación significa que la clase adyacente no tiene referencias para las otras clases en esa asociación.

Por último, observe las puntas de flecha con lengüetas que se han agregado a las líneas de asociación de composición. Como ya se dijo, todas las líneas de asociación son bidireccionales por defecto, y uno de los objetivos del diseño es convertir las asociaciones bidireccionales en asociaciones unidireccionales. Las puntas de flecha con lengüetas en los extremos sin diamante de las cuatro líneas de composición hacen lo anterior. Dicen a los componentes de la composición que no saben nada sobre su contenedor. Eso está bien en este caso, ya que este contenedor es justamente un controlador, y muchos controladores son efímeros: hoy están y mañana ya no.

En la figura A7.4 también se incluye una línea de asociación discontinua con una punta de flecha con lengüeta que apunta a la clase `abstract`, `Employee2`. Esto reconoce que la variable local denominada `employees` "depende de" la clase `Employee2` porque el tipo de sus elementos es `Employees2`. La punta de flecha con lengüeta en el extremo `Employees2` de esta línea de asociación discontinua indica que la asociación es unidireccional. `Payroll3` sabe sobre `Employees2`, pero `Employees2` no sabe nada sobre `Payroll3`. Entonces, cambios a `Employees2` podrían requerir cambios a `Payroll3`, pero cambios a `Payroll3` jamás podrían requerir cambios a `Employees2`. UML utiliza líneas de asociación discontinuas para indicar dependencias de variables de instancia y de clase.

Como se describió en la sección opcional al final del capítulo 12, UML también usa líneas de asociación discontinuas para unir una clase de asociación con una asociación entre dos o varias clases. En la figura 12.22 se muestra una línea de asociación que une las tres clases, `SalesPerson2`, `Customer` y `Car`. Aunque este detalle no se analizó en el capítulo 12, el hecho de que esta línea de asociación sea continua y no tenga puntas de flecha con lengüeta en sus extremos sugiere que cada una de estas tres clases tiene variables de instancia que se refieren a instancias particulares de las otras dos clases.

La clase de asociación denominada `Sales` hace innecesarias estas referencias adicionales, porque la clase `Sales` contiene a todas estas referencias en sí en un solo sitio. Por tanto, esta clase de asociación extra reduce el número de variables de referencia. Lo que es más importante, elimina la necesidad de modificar la definición de las clases `SalesPerson2` y `Car` cuando al programa se agregan la clase `Customer` y una asociación `Sale`. Para reflejar el hecho de que las clases `SalesPerson2`, `Car` y `Customer` no necesitan ninguna referencia a las instancias de otras clases en la asociación común, se colocaron puntas de flecha con lengüeta en los tres extremos de la línea de asociación que los une. Esto cambia la figura 12.22 a lo que se ve en la figura A7.5.[3] Observe que la figura A7.5 también incluye una asociación de composición entre `Dealership3` y `Sale`. Las puntas de flecha con lengüeta en los extremos `Sale` y `Car` de sus respectivas líneas de composición y en el extremo `SalesPerson2` de su línea de agregación indican que `Dealership` depende de estas otras clases. En otras palabras, `Dealership3` tiene referencias a instancias de las clases `Sale`, `Car` y `SalesPerson2`, pero no viceversa. En contraste, la asociación de agregación entre `Dealership3` y `Manager2` no tiene ninguna punta de flecha. Esto indica que cada una tiene una referencia a la otra.

[3] Observe el diamante pequeño en la intersección de las líneas de asociación de `Sale`. Este detalle UML ayuda a distinguir un empalme de un cruce.

APÉNDICE 8

Recursión

Para comprender este apéndice, es necesario que el lector conozca la programación y los arreglos orientados a objetos. Por tanto, quizá debe leer hasta el capítulo 10.

Recursión es cuando un método se llama a sí mismo. Lo que sigue es un enfoque general para resolver una tarea con una implementación recursiva.

Primero, identifique una forma de hacer que el problema sea más simple en forma progresiva, e identifique una condición, denominada "condición de detención", que esté asociada con la versión más simple del problema. Use una declaración `if` para verificar la condición de detención. El cuerpo `if` debe contener la solución de la versión más simple del problema. El cuerpo `if` debe contener una o varias llamadas al mismo método con uno o más valores de argumento que lo hagan más simple en forma progresiva. Una vez que se llama al método, éste continúa llamándose a sí mismo en forma automática con condiciones más simples en forma progresiva, hasta que se satisface la condición de detención.

Una vez que se satisface la condición de detención, el método resuelve la versión más simple del problema. Luego regresa la solución más simple del problema a la ejecución previa del método al siguiente nivel superior de dificultad. Esa ejecución del método genera la solución de su versión del problema. Este proceso de regresar soluciones más simples a la ejecución previa del método continúa hasta la ejecución del método original, que genera la solución del problema original.

La recursión no agrega funcionalidad única: todos los algoritmos deben convertirse en programas de ciclo que no usan recursión. Entonces, ¿por qué usar recursión? Porque con algunos problemas, una solución recursiva es más directa que una solución de ciclo. Por ejemplo, algunos conceptos matemáticos, como el factorial de un número y la sucesión de Fibonacci, se definen recursivamente, y se prestan bastante bien a soluciones programáticas que usan recursión. Y algunos juegos, como las torres de Hanoi y recorridos en laberintos, pueden resolverse mejor con razonamiento recursivo, y también se prestan a soluciones programáticas que usan recursión. Tenga en mente que la recursión tiene un lado débil. Los programas recursivos tienden a ser lentos porque generan muchas llamadas a funciones y éstas pueden tener mucha *sobrecarga*. La sobrecarga es trabajo que la computadora tiene que realizar. Para cada llamada a una función, la computadora debe 1) guardar las variables locales del módulo que llama, 2) encontrar el método, 3) hacer copias de los argumentos que llaman por valor, 4) pasar los argumentos, 5) ejecutar el método, 6) encontrar al módulo que llama, y 7) restituir las variables locales del módulo que llama. ¡Uf! Todo este trabajo requiere tiempo. Ésta es la razón por la cual algunas implementaciones recursivas pueden ser prohibitivamente lentas. Para estos casos, es necesario considerar volver a escribir la solución con una implementación de ciclo.

Determinación del factorial de un número

El factorial de un número n es n! = n * (n-1) * (n-2) * . . . * 2 * 1. Este problema es sencillo, y fue posible resolverlo por completo dentro del encabezado de un ciclo vacío `for` en la figura 11.9. Así, realmente no es un buen candidato para la recursión en la práctica. Pero debido a que es fácil, constituye una ilustración introductoria a la medida de cómo funciona la recursión.

La forma de hacer más simple este problema es reduciendo el valor de n, y la condición de detención es cuando n = 1. En ese punto, n! = 1. Para cualquier otro valor de n, se tiene la relación:

```
n! = n * (n-1)!
```

Esta fórmula es la *relación recursiva* en el cálculo de un factorial.

```
 1  /*************************************************
 2  * Factorial.java
 3  * Dean & Dean
 4  *
 5  * Este programa calcula el factorial de un entero.
 6  *************************************************/
 7
 8  public class Factorial
 9  {
10    public static void main(String[] args)
11    {
12      System.out.println(factorial(5));
13    } // end main
14
15    //***********************************************
16
17    private static int factorial(int n)
18    {
19      int nF; // n factorial
20
21      if (n == 1)  // condición de detención
22      {
23        nF = 1;
24      }
25      else
26      {
27        nF = n * factorial(n-1);
28      }
29      return nF;
30    } // end factorial
40  } // end Factorial class
```

llamada recursiva al método

trabaja en el valor devuelto

Salida:

```
120
```

Figura A8.1 Uso de recursión para calcular el factorial de un entero.

La figura A8.1 contiene una implementación de Java del cálculo recursivo de un factorial. El método recursivo, `factorial`, es una declaración "if-else". La condición `if` es la condición de detención, y el cuerpo `else` incluye una llamada a un método recursivo, donde el método se llama a sí mismo. Cuando `factorial` se llama por primera vez desde `main`, su valor de parámetro es `5`, de modo que la condición de detención no se satisface, y el método se llama a sí mismo desde el cuerpo de `else` con un valor de argumento igual a 5 menos 1, lo que es igual a 4. Esta llamada recursiva al método continúa hasta que el parámetro del método es igual a 1. En ese momento, el método devuelve 1 a la ejecución previa del método, donde el valor del parámetro era 2. Esa ejecución del método devuelve 2*1 = 2 a la ejecución previa del método, donde el valor del parámetro era 3, etc., hasta que regresa a la ejecución del método original, que devuelve 5*24 = 120 como valor del argumento en el método `println` llamado en `main`.

La figura A8.2 muestra una traza de la ejecución del programa en la figura A8.1. La primera llamada al método `factorial` está en la línea 12 en el método `main` con un argumento de 5. Las cuatro siguientes llamadas al método factorial es desde el interior del método `factorial` en la línea 27 con argumentos de 4, 3, 2, y 1. Cuando el parámetro es igual a 1, la condición de detención se satisface, y a `nF` se asigna el valor de 1 en la línea 23. Luego, comienza el proceso de regreso. El 1 de la quinta llamada al método `factorial` regresa a la cuarta llamada a `factorial`, que en la línea 27 lo multiplica por 2 y devuelve 2. Este valor regresa a la tercera llamada a `factorial`, que en la línea 27 lo multiplica por 3 y devuelve 6. Este valor regresa a la segunda llamada a `factorial`, que en la línea 27 lo multiplica por

Factorial											
	factorial		factorial		factorial		factorial		factorial		
línea#	**n**	**nF**	**n**	**nF**	**n**	**nF**	**n**	**nF**	**n**	**nF**	**salida**
12	5	?									
27			4	?							
27					3	?					
27							2	?			
27									1	?	
23										1	
27								2			
27						6					
27				24							
27		120									
12											120

Figura A8.2 Traza del programa Factorial en la figura A8.1.

4 y devuelve 24. Este valor regresa a la primera llamada a `factorial`, que en la línea 27 lo multiplica por 5 y devuelve 120. Este valor regresa al argumento de la declaración `println` en el método `main` en la línea 12, y esta declaración imprime el valor calculado. En este problema, todo el trabajo útil es realizado en la secuencia de regreso, después que se alcanza la condición de detención.

La variable local `nF` se incluyó en el programa en la figura A8.1 simplemente para darle algo de sustancia a este rastreo. Con un poco de suerte, lo anterior ayuda a visualizar la llamada recursiva que llega hasta el caso más simple y la subsiguiente acumulación que resulta a medida que regresan los métodos anidados. En la práctica, sin embargo, los programadores experimentados no incluirían la variable local `nF`. En lugar de ello, tal vez el lector escribiría el método `factorial` como el que se muestra en la figura A8.3.

Observe que la implementación en la figura A8.3 extiende la condición de detención a `n == 0` a fin de incluir el caso de 0!, que también es igual a la unidad.

Búsqueda binaria de un arreglo ordenado

A continuación se considerará otro ejemplo que es ligeramente más difícil de implementar con ciclos y constituye un mejor caso para recursión. En este caso, el lector verá que todo el trabajo útil se realiza mientras el algoritmo está llegando a la condición de detención y lo que se ha regresado apenas pasa la respuesta de regreso.

```
private static int factorial(int n)
{
  if (n == 0)  // condición de detención
  {
    return 1;
  }
  else
  {
    return n * factorial(n-1);
  }
} // terminar factorial
```

Figura A8.3 Implementación más limpia del método `factorial`.

Éste es el problema: suponga que se desea encontrar la ubicación de un valor particular en el arreglo. Ésta es una operación común de base de datos. Si el arreglo no está ordenado, lo mejor que puede hacerse es una búsqueda secuencial y ver cada artículo, uno por uno. Si el arreglo es muy corto, la mejor forma de búsqueda es una búsqueda secuencial, porque ésta es muy sencilla. No obstante, si el arreglo es largo, y es relativamente estable, a menudo es mas rápido ordenar el arreglo y luego usar búsqueda *binaria*. (En el capítulo 10 se describen algoritmos de ordenamiento.)

¿Por qué una búsqueda binaria es más rápida que una búsqueda secuencial? El número de pasos necesarios para una búsqueda secuencial es igual a <*array*>>.`length`, pero el número de pasos necesarios en una búsqueda binaria es igual a sólo $\log_2$(<*array*>>.`length`). Por ejemplo, si en el arreglo hay un millón de artículos, eso significa un millón de pasos para una búsqueda secuencial, pero sólo alrededor de 20 pasos para una búsqueda binaria. Inclusive si un paso de una búsqueda binaria es más complicado que un paso de una búsqueda secuencial, la búsqueda binaria es significativamente más rápida cuando el arreglo es muy largo.

Para implementar una búsqueda binaria es correcto usar recursión. La forma de hacer más sencillo el problema consiste en dividir el arreglo en dos arreglos casi del mismo tamaño y continuar dividiéndolos hasta que cada mitad no contenga más que un elemento, que es la condición de detención. En la figura A8.4 se muestra nuestra implementación de este algoritmo. Se han incluido declaraciones de impresión sombreadas en sitios apropiados del código para mostrar lo que el código hace mientras se ejecuta. Después de depurar el programa, conviene eliminar todas estas declaraciones de impresión sombreadas.

En la figura A8.5 se observa un controlador que demuestra el algoritmo de búsqueda binaria implementado en la figura A8.4. En la sección de salida, las áreas sombreadas son salidas generadas por las declaraciones de impresión sombreadas en la figura A8.4. En la recursión, el verdadero trabajo se realiza mientras el proceso intenta llegar a la condición de detención. Observe cómo los valores `first` y `last` convergen en la coincidencia o en el lugar en que estaría la coincidencia si estuviese ahí. La respuesta es generada en el punto en que se alcanza la condición de detención. El anidamiento regresa justo recién pasada esta respuesta de regreso. Cuando se eliminan las declaraciones de impresión sombreadas de la figura A8.4, la única salida que se ve son las partes no sombreadas de la salida.

Problema de las torres de Hanoi

Ahora es el momento de presentar un problema que sería muy difícil de resolver con ciclos, y donde la recursión es incuestionablemente la mejor forma de proceder. Imagine que usted es un señor medieval que observa a un grupo de monjes aburridos. Para mantenerlos ocupados en temporada de lluvia, les pide resolver el problema de las torres de Hanoi. Hay tres ubicaciones, “A”, “B” y “C”. Inicialmente, en la ubicación “A” hay una pila de discos. EL disco más pequeño está en la parte superior, y el diámetro de los discos aumenta a medida que se desciende hacia la base de la “torre”. Las ubicaciones, “B” y “C” están vacías inicialmente. En la figura A8.6 se muestra la situación en el momento en que los cuatro discos de la torre más alta están en la ubicación “A”:

Los monjes deben mover los discos de las torres de la ubicación A a la ubicación C, siempre siguiendo estas reglas:

- Mover un disco a la vez.
- Nunca colocar un disco encima de un disco más pequeño.

La tarea consiste en presentar el algoritmo más sencillo para resolver este problema. Si esto tuviera que hacerse usando ciclos, sería un revoltijo. Pero si se usa recursión, es razonablemente simple. Siempre que se quiera hacer un movimiento, se tiene una ubicación fuente, `s`, una ubicación destino, `d`, y una ubicación temporal, `t`. Para el objetivo global, `s` es `A`, `d` es `C` y `t` es `B`. A medida que se avanza hacia la solución final, es necesario subordinar metas, con ubicaciones distintas para `s`, `d` y `t`. Éste es el algoritmo general. Es válido para cualquier subconjunto de discos desde el disco n hasta el disco 1, donde `n` es cualquier número desde el número máximo de discos hasta 1:

- Mover el grupo de discos arriba del disco `n` desde `s` hasta t.
- Mover el disco `n` a `d`.
- Mover el grupo de discos previamente arriba del disco n desde `t` hasta `d`.

```
/*************************************************************
* BinarySearch.java
* Dean & Dean
*
* Esto usa recursión para encontrar el índice de un valor objetivo en un arreglo
* ordenado en forma ascendente. Si no se encuentra, el resultado es -1.
*************************************************************/

public class BinarySearch
{
  public static int binarySearch(
    int[] arr, int first, int last, int target)
  {
    int mid;
    int index;

    System.out.printf("first=%d, last=%d\n", first, last);
    if (first == last)             // condición de detención
    {
      if (arr[first] == target)
      {
        index = first;
        System.out.println("found");
      }
      else
      {
        index = -1;
        System.out.println("not found");
      }
    }
    else                           // continuar la recursión
    {
      mid = (last + first) / 2;
      if (target > arr[mid])
      {
        first = mid + 1;
      }
      else
      {
        last = mid;
      }
      index = binarySearch(arr, first, last, target);
      System.out.println("returnedValue=" + index);
    }
    return index;
  } // end binarySearch
} // end BinarySearch class
```

Hace el mismo trabajo.

Luego sigue buscando.

Figura A8.4 Implementación del algoritmo de búsqueda binaria.

En la figura A8.7 se muestra un ejemplo de este algoritmo en acción. (Ocurre que este ejemplo en particular tiene el menor número de pasos en la solución final.) La configuración a la izquierda es una condición que existe poco antes de alcanzar la meta. La configuración a la derecha es la condición final. Las flechas discontinuas indican cada una de las tres operaciones descritas arriba para el caso no trivial más sencillo en que arriba del disco sólo hay un disco. El caso trivial es la condición de detención. Ocurre cuando se mueve el disco que está en la parte superior, el disco 1, desde una ubicación fuente a una ubi-

```
/*************************************************************
* BinarySearchDriver.java
* Dean & Dean
*
* Esto controla la clase BinarySearch.
*************************************************************/

public class BinarySearchDriver
{
  public static void main(String[] args)
  {
    int[] array = new int[] {-7, 3, 5, 8, 12, 16, 23, 33, 55};

    System.out.println(BinarySearch.binarySearch(
      array, 0, (array.length - 1), 23));
    System.out.println(BinarySearch.binarySearch(
      array, 0, (array.length - 1), 4));
  } // end main
} // end BinarySearchDriver class
```

Salida:

```
first=0, last=8
first=5, last=8
first=5, last=6
first=6, last=6
found
returnedValue=6
returnedValue=6
returnedValue=6
6
first=0, last=8
first=0, last=4
first=0, last=2
first=2, last=2
not found
returnedValue=-1
returnedValue=-1
returnedValue=-1
-1
```

Reduce el intervalo y sigue buscando.

Simplemente regresa la respuesta.

Figura A8.5 Controlador de la clase `BinarySearch` en la figura A8.4.

cación destino. Un ejemplo de lo anterior aparece en el último paso en la figura A8.7. Ocurre que ésta es la condición de detención final, pero como se verá en breve, un programa que resuelve el problema de las Torres de Hanoi llega a la condición de detención y automáticamente vuelve a recomenzar muchas veces durante su ejecución.

Suponga que el proceso de evaluación está actualmente en el marco izquierdo de la figura A8.7. La siguiente operación llama al siguiente método `move` con argumentos (2, 'A', 'C', 'B'). Dentro de este método, la llamada al primer método subordinado *move* de la cláusula `else` usa los argumentos (1, 'A', 'B', 'C') para implementar "Último paso – 2" en la figura A8.7. La declaración ulterior `printf` implementa "Último paso – 1" en la figura A8.7. La llamada al segundo método subordinado `move` de la cláusula `else` usa los argumentos (1, 'B', 'C', 'A') para implementar "Último paso" en la figura A8.7.

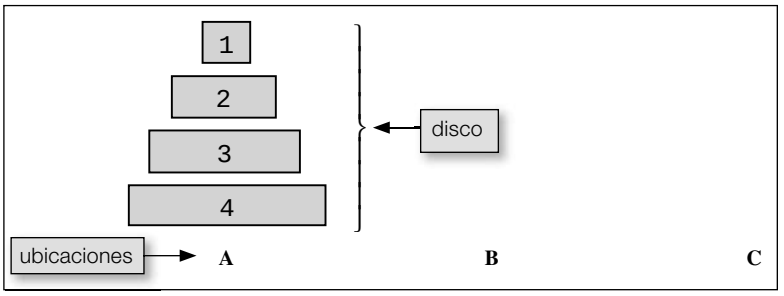


Figura A8.6 Disposición para el problema de las Torres de Hanoi.

```
private static void move(int n, char s, char d, char t)
{
  if (n == 1)           // condición de detención recursiva
  {
    System.out.printf("move %d from %s to %s\n", n, s, d);
  }
  else
  {
    move(n-1, s, t, d);  // fuente a temporal
    System.out.printf("move %d from %s to %s\n", n, s, d);
    move(n-1, t, d, s);  // temporal a destino
  }
}
```

La llamada inicial al método recursivo debe establecer el objetivo global, que es mover toda la torre de la ubicación `A` a la ubicación `C`. Para que el disco más grande esté primero en la nueva ubicación, debe empezarse con la `n` máxima posible. El algoritmo indica que esto puede hacerse al mover el conjunto subordinado de discos 1, 2 y 3 de la ubicación fuente, `A`, a la ubicación temporal, `B`. Luego, el disco 4 se mueve de la ubicación fuente, `A`, a la ubicación destino, `C`, colocándolos así en la parte superior del disco más grande, 4. El problema con esto es que las reglas no permiten mover más de un disco a la vez. Así, para mover el conjunto subordinado de discos 1, 2 y 3 es necesario llamar recursivamente al mismo método a fin de mover los discos 1 y 2. Para hacer esto, nuevamente debe llamar recursivamente al mismo método para mover sólo el disco 1.

Por supuesto, el primer disco que se mueve es el disco 1, aunque es difícil saber dónde colocarlo. ¿Debe ponerse en la ubicación `B` o en la ubicación `C`? El objetivo del programa es decir exactamente cómo proceder. El programa se muestra en la figura A8.8. Las declaraciones de impresión sombreadas no forman parte de la solución, por lo que es necesario omitirlas del producto terminado. Se insertaron sólo como ayuda para que el lector rastree la tortuosa actividad recursiva, en caso de así desearlo. Para cada invocación al método, imprimen justo después de la llamada al método y justo antes que regrese a fin de mostrar los detalles de lo que está ocurriendo.

En la figura A8.9 se muestra la salida. Las líneas sombreadas son líneas impresas por las declaraciones de impresión sombreadas en la figura A8.8. Como se dijo, sólo están ahí para mostrar lo que ha ocu-

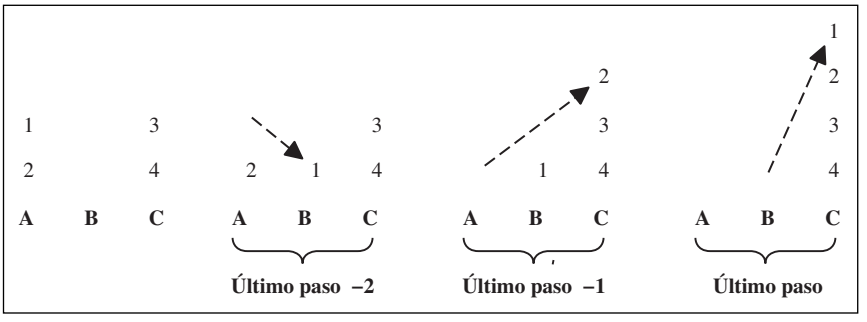


Figura A8.7 Ilustración del algoritmo de las Torres de Hanoi en acción.

```
/*************************************************************
* Towers.java
* Dean & Dean
*
* Esto usa un algoritmo recursivo para el problema de las Torres de Hanoi.
*************************************************************/

public class Towers
{
  public static void main(String[] args)
  {
    move(4, 'A', 'C', 'B');
  }

  // Mueve n discos de la fuente s al destino d usando la ubicación temporal t.
  private static void move(int n, char s, char d, char t)
  {
    System.out.printf(
      "call n=%d, s=%s, d=%s, t=%s\n", n, s, d, t);
    if (n == 1)            // condición de detención recursiva
    {
      System.out.printf("move %d from %s to %s\n", n, s, d);
    }
    else
    {
      move(n-1, s, t, d);  // fuente a temporal
      System.out.printf("move %d from %s to %s\n", n, s, d);
      move(n-1, t, d, s);  // temporal a destino
    }
    System.out.println("return n=" + n);
  }
} // end class Towers
```

dos puntos de regreso

Figura A8.8 Solución al problema de las Torres de Hanoi.
Las declaraciones sombreadas se usan para rastreo diagnóstico. Es necesario quitarlas para la implementación final.

rrido y no son parte de la solución. La solución está dada por las salidas sin sombrear. La parte más difícil al rastrear un algoritmo recursivo como éste consiste en seguir la pista del sitio en que se hizo una llamada y en consecuencia dónde se resume la ejecución después de un regreso. Como se indica en la nota en la figura A8.8, algunas veces la llamada se hace desde la declaración "fuente a temporal", y algunas veces la llamada se hace desde la declaración "temporal a destino". Por fortuna, si un algoritmo recursivo se define correctamente, es posible ignorar los detalles de cuál es su rol durante la ejecución del programa.

Recomendamos que el lector corte cuatro discos de papel de diferentes tamaños, los numere del 1 al 4 de menor a mayor y construya una torre en la ubicación "A" a su izquierda. Luego, mueva los discos uno por uno como se indica en las salidas sombreadas en la figura A8.9. Verá que la torre de hecho se mueve de la ubicación "A" a la ubicación "C" de conformidad precisa con las reglas especificadas. Esto funciona porque los movimientos eventuales cuentan con información del objetivo proporcionada por las llamadas anteriores al método.

Salida:

```
call n=4, s=A, d=C, t=B
call n=3, s=A, d=B, t=C
call n=2, s=A, d=C, t=B
call n=1, s=A, d=B, t=C
move 1 from A to B
return n=1
move 2 from A to C
call n=1, s=B, d=C, t=A
move 1 from B to C
return n=1
return n=2
move 3 from A to B
call n=2, s=C, d=B, t=A
call n=1, s=C, d=A, t=B
move 1 from C to A
return n=1
move 2 from C to B
call n=1, s=A, d=B, t=C
move 1 from A to B
return n=1
return n=2
return n=3
move 4 from A to C
call n=3, s=B, d=C, t=A
call n=2, s=B, d=A, t=C
call n=1, s=B, d=C, t=A
move 1 from B to C
return n=1
move 2 from B to A
call n=1, s=C, d=A, t=B
move 1 from C to A
return n=1
return n=2
move 3 from B to C
call n=2, s=A, d=C, t=B
call n=1, s=A, d=B, t=C
move 1 from A to B
return n=1
move 2 from A to C
call n=1, s=B, d=C, t=A
move 1 from B to C
return n=1
return n=2
return n=3
return n=4
```

Figura A8.9 Salida del programa en la figura A8.8.
Las declaraciones no sombreadas son la solución. Las declaraciones sombreadas sólo proporcionan un rastreo.

APÉNDICE 9

Multithreads

Para comprender este apéndice, el lector debe estar familiarizado con programación orientada a objetos, herencia, manejo de excepciones y archivos. Por tanto, debe haber leído hasta el capítulo 15.

En este capítulo se presenta una característica, *multithreads*, que ayuda a que los programas en Java aprovechen las capacidades de procesamiento paralelo contenidas en muchas computadoras modernas. Al aprovechar las capacidades de procesamiento paralelo, *multithreading* puede conducir a programas más rápidos. Y a su vez, lo anterior puede producir programas más amistosos para el usuario.

Threads

Un *thread* es un "proceso de peso ligero".[1] Considérelo como un fragmento de código coherente. Idealmente, una vez que tiene una cierta cantidad mínima de información inicial, un thread puede ejecutarse de principio a fin sin ninguna información adicional del mundo exterior. Idealmente, threads diferentes son independientes.

Un thread es un objeto derivado de una clase que extiende (`extends`) la clase predefinida `Thread`, y es necesario sobreponer el método `run` de la clase `Thread` al especificar qué desea que haga su thread. El método `run` del objeto sólo puede llamarse una vez, y es necesario hacerlo en forma indirecta al llamar al método `public void start()`, que su clase hereda de la clase `Thread`. El método `start` pide a la JVM que llame al método `run` del lector.[2]

Esta operación hace que el thread recientemente iniciado se ejecute en paralelo con el software que lo llamó. Después de haber iniciado un thread, el software puede empezar otro thread, y entonces se tendrían tres trozos de código ejecutándose en paralelo. Puede continuarse así para obtener tantas operaciones paralelas como se quiera. La computadora aprovecha en forma automática estos trozos de código relativamente independientes para mantener ocupadas sus diversas componentes paralelas de hardware tanto como sea posible. El controlador en la figura A9.1 muestra lo fácil que es lanzar nuevos threads.

La clase en la figura A9.1 es el nivel superior de un programa que describe la interacción entre un predador como un zorro y una presa como un grupo de ratones de campo. En este ejemplo, la presa obtiene su comida en forma continua de la vegetación perenne, mientras el predador obtiene su comida en forma intermitente al devorar a la presa cuando ocurre que el predador y la presa se encuentran. La presa es un thread. El predador es otro thread. Observe cómo este controlador inicia los dos threads: el `prey` y el `predator`. Estos threads representan las vidas paralelas de estas criaturas en su ecosistema.

---

[1] Un "proceso de peso ligero" tiene su propio "contador de programa" y su propia "pila", pero aparte de esto tiene acceso normal al resto del programa en que existe. El contador de programa de un thread sigue la pista del sitio en que ocurre la ejecución, y la pila recuerda cómo regresar de llamadas a funciones. Siempre que un thread se detiene temporalmente, la computadora toma una instantánea de ese contador de programa del thread y de la pila, lo cual permite que la ejecución reinicie exactamente donde se quedó cuando el thread empieza a ejecutarse de nuevo.

[2] Si la clase del lector ya implementa alguna otra clase, puede hacer que trabaje como un thread al implementar también la interfaz `Runnable`. Para empezar el método `run` de una clase que implemente `Runnable` pero que no se `extiend` `Thread`, la clase del lector también debe incluir un método `start` que haga lo siguiente:

```
public void start()
{
  new Thread(this).start();
}
```

Para no complicar las cosas, este análisis se restringe a clases que implementan la clase `Thread`.

```
/*************************************************************
* Ecosystem.java
* Dean & Dean
*
* Controlador para un sistema simple predador/presa (consumidor/
* productor). Los objetos predador y presa son threads separados,
* y encounter es un objeto que describe su relación.
*************************************************************/

public class Ecosystem
{
  public static void main(String[] args)
  {
    Prey prey = new Prey();                  // thread del productor
    Predator predator = new Predator();      // thread del consumidor
    Encounter encounter = new Encounter(prey, predator);

    // start threads
    prey.start();
    predator.start();
  } // end main
} // end Ecosystem class
```

Figura A9.1 Nivel superior de un programa que simula un ecosistema simple. Esta clase controla las clases en las figuras A9.2, A9.3 y (A9.4 o A9.5*a* y A9.5*b*).

Los threads `prey` y `predator` son objetos. También hay otro objeto, denominado `encounter`. Este objeto representa una relación intermitente que tiene lugar entre el predador y la presa. En esta relación, algunas presas se ponen al alcance del predador y éste se las come. Tal vez el predador sólo come una porción de la presa en cada encuentro particular, y entretanto la presa continúa abasteciéndose al reproducirse y comer vegetación. En la relación de encuentro, la presa proporciona comida y el predador consume comida. Así, personas dedicadas a la computación, como nosotros, dirían que el thread de la presa es el thread de un *productor* y que el thread del predador es el thread de un *consumidor*. Por supuesto, la presa también "consume" vegetación, de modo que si este modelo incluyese una relación entre los ratones de campo y la vegetación que consumen, en ese contexto el thread de la `presa` podría llamarse thread del "consumidor". Así, los términos "productor" y "consumidor" no deben estar asociados en absoluto a ningún otro thread.

Cualquier relación entre threads viola el carácter ideal de la "dependencia de threads". Complica la vida de las criaturas reales y complica el programa que las simula.

En la figura A9.2 se muestra la clase que describe los threads `prey`. Observe que no extiende la clase `Threads`. Sólo hay una variable de instancia, una referencia a la relación `encounter`: los ratones de campo indudablemente están al tanto de su desagradable relación con un zorro. El constructor de parámetro cero asigna un nombre por defecto a todos los objetos de la clase. Eso es suficiente para nuestro ejemplo, ya que el controlador crea sólo uno de estos objetos, aunque el lector también puede proporcionar un constructor de un parámetro para asignar nombres distintos a instancias de thread diferentes. El método `public setEncounter` permite al mundo exterior establecer la referencia `encounter` en cualquier momento después de la instanciación del thread `Prey`. El método `run` es el núcleo de la definición de un thread. En este caso, es bastante simple. Lo que la presa quiere es "permanecer separada", de modo que el método `run` llame al método `beApart` de la relación.

En la figura A9.3 se muestra la clase que describe los threads `Predator`. También extiende la clase `Threads`. También tiene una variable de instancia que refiere a la relación `encounter`: un zorro ciertamente está consciente de la agradable relación con los ratones de campo. También tiene un constructor de parámetro cero, así como un método `setEncounter`.

Observe que `Predator` también declara un arreglo de retrasos temporales. Con referencias cruzadas idóneas, esta información de retraso temporal hubiera podido colocarse en cualquiera de las clases.

```
/*************************************************************
* Prey.java
* Dean & Dean
*
* Esto modela al predador (productores), que evita los encuentros.
*************************************************************/

public class Prey extends Thread
{
  private Encounter encounter;

  //*********************************************************

  public Prey()
  {
    super ("prey");
  } // end constructor

  //*********************************************************

  public void setEncounter(Encounter encounter)
  {
    this.encounter = encounter;
  } // end setEncounter

  //*********************************************************

  public void run()
  {
    int number;

    do
    {
      number = encounter.beApart();
    } while (number < encounter.EVENTS - 1);
    System.out.println(getName() + " run finished. ");
  } // end run
} // end Prey class
```

Figura A9.2 Clase que describe a las presas (productores) que quieren escapar de los predadores. Esto es controlado por la clase en la figura A9.1.

Pero debido a que el predador constituye la "causa" primordial de los encuentros, se decidió colocarla en la definición de `Predator` e implementarla en el método `run` de `Predator`. Esta implementación de retraso temporal hace que el método `run` de `Predator` sea más complicado que el método `run` de `Prey`. Cada retraso se implementa al hacer pasar un número entero al método `sleep` previamente escrito, que está en la clase `Thread` en el siempre disponible paquete `java.lang`:

```
public static void sleep(long millis)
   throws InterruptedException
```

¿Qué hace el método `sleep`? Hace que el thread que se está ejecutando en ese momento cese su ejecución durante una cantidad de milisegundos igual al valor del parámetro. En nuestro ejemplo, el primer elemento en el arreglo `DELAY` es 2347, de modo que cuando se ejecuta el programa, se experimenta una pausa de 2.347 segundos entre las salidas primera y segunda en la pantalla.

Observe que el método `sleep` puede lanzar una `InterruptedException`. Si se busca `InterruptedException`, se encontrará que se deriva directamente de la clase `Exception`, de modo que

```
/*************************************************************
* Predator.java
* Dean & Dean
*
* Esto modela a los predadores (consumidores), que desean encuentros.
*************************************************************/

public class Predator extends Thread
{
  // delay times in milliseconds
  public final long[] DELAY = {2347, 1325, 1266, 3534};
  private Encounter encounter;

  //*********************************************************

  public Predator ()
  {
    super ("predator");
  } // end constructor

  //*********************************************************

  public void setEncounter(Encounter encounter)
  {
    this.encounter = encounter;
  } // end setEncounter

  //*********************************************************

  public void run()
  {
    int i;

    for (i=0; i<DELAY.length; i++)
    {
      try
      {
        Thread.sleep(DELAY[i]);          // se queda y caza
      }
      catch (Exception e) { }
      encounter.beTogether();            // devora a la presa
    }
    System.out.println(getName() + " ejecución terminada.");
  } // end run
} // end Predator class
```

Figura A9.3 Clase que describe a un predador (consumidor) que busca a la presa. Esto es controlado por la clase en la figura A9.1.

es una excepción comprobada. En consecuencia, la llamada al método que podría lanzar esta excepción debe estar en un bloque `try`, que es donde se coloca. El programa jamás hace algo que pudiera provocar el lanzamiento de una excepción,[3] de modo que se usa un bloque `catch` *vacío*. Para mejor retroalimentación de depuración, en el bloque `catch` puede colocarse algo como `e.printStackTrace()`.

[3] Una `InterruptedException` se lanza cuando el thread actual está durmiendo y otro thread lo despierta prematuramente al llamar a su método `interrupt`, pero en nuestro programa nunca se usa el método `interrupt`.

```
/*************************************************************
* Encounter.java
* Dean & Dean
*
* Esto describe la interacción predador/presa (consumidor/productor).
*************************************************************/

public class Encounter
{
  public final int EVENTS;
  private int number = -1;
  private Prey prey;
  private Predator predator;

  //*********************************************************

  public Encounter(Prey prey, Predator predator)
  {
    this.prey = prey;
    this.predator = predator;
    prey.setEncounter(this);
    predator.setEncounter(this);
    EVENTS = predator.DELAY.length;
  } // end constructor

  //*********************************************************

  public int beApart()
  {
    // prey has access, so go apart
    number++;
    System.out.println(Thread.currentThread().getName() +
      " start beApart " + number);
    return number;
  } // end beApart

  //*********************************************************

  public int beTogether()
  {
    // el predador tiene acceso, de modo que vienen juntos
    System.out.println(Thread.currentThread().getName() +
      " terminar beTogether " + número);
    return number;
  } // end beTogether
} // end Encounter class
```

¡CUIDADO!
¡Esta implementación no funciona!

Figura A9.4 Implementación inadecuada de la clase `Encounter`.
Puesto que los threads `prey` y `predator` se ejecutan en paralelo, sólo el thread `predator` contiene retrasos, no se intercalan apropiadamente. El thread `prey` termina rápido, mientras inclusive la primera salida del método predator se retrasa hasta mucho después.

A continuación se abordará un primer intento rudimentario para implementar la clase `Encounter`, que aparece en la figura A9.4. En las vidas de un predador y un grupo de sus presas (los threads elegidos), los encuentros ocurren varias veces. La clase `Encounter` hubiera podido escribirse de modo que cada objeto `encounter` representase un *evento discreto*, aunque es más fácil seguir la pista de las relaciones en tiempo y espacio si se agrupan eventos relacionados entre sí. Entonces, uno de los objetos `en-`

`counter` representa una secuencia completa de encuentros entre el thread `predator` y el thread `prey`. En programación de simulación, este tipo de relaciones suele denominarse *proceso*.

Las variables de instancia en el objeto `encounter` sigue la pista del número total de eventos, el número de secuencia del evento actual, y referencias a los threads `prey` y `predator`. Si vuelve a ver la figura A9.1, observará que el constructor `Encounter` se llama después de llamar a los constructores `Prey` y `Predator`. Esta secuencia de llamadas permite pasar las referencias `predator` y `prey` al objeto `encounter` una vez que se instancie éste. Entonces, en el constructor `Encounter` se corresponde al enviar una referencia `encounter` a los objetos `predator` y `prey`.

Ahora considere los métodos `beApart` y `beTogether`. Estos métodos representan las dos fases de la relación encounter que está ocurriendo. El método `beApart` describe un largo periodo pasivo en que el predador se queda y caza. Esto se denomina clase `Prey` en la figura A9.2. El método `beTogether` describe un breve periodo violento en que el predador encuentra la presa, ataca y devora parte de la presa. Esto se denomina clase `Predator` en la figura A9.3.

Según aparecen en la figura A9.4, estos métodos no hacen mucho. El método `beApart` actualiza el número de ciclo. Luego, imprime el nombre del thread que lo llamó y el valor actual del número de ciclo. El método `beTogether` imprime el nombre del thread que lo llamó y el valor actual del número de ciclo. En un modelo más completo, el método `beTogether` también llamaría a los métodos `Predator` y `Prey` para cambiar las masas de esos objetos durante el tiempo que permanecieron separados. Luego, calcularía el cambio en los pesos violento juntos en el periodo.

Si se ejecuta el programa en las figuras A9.1, A9.2, A9.3 y A9.4, se obtiene lo siguiente:

Salida:

```
prey start beApart 0
prey start beApart 1
prey start beApart 2
prey start beApart 3
prey run finished.
predator finish beTogether 3
predator finish beTogether 3
predator finish beTogether 3
predator finish beTogether 3
predator run finished.
```

¿Esto es lo que se desea? ¡No! Porque los threads `prey` y `predator` se ejecutan en paralelo, aunque sólo el thread `predator` contiene retrasos, de modo que no se intercalan apropiadamente. El thread `prey` termina rápido, mientras inclusive la primera salida del método `predator` no ocurre sino hasta mucho después.

Sincronización

Cuando threads diferentes acceden a objetos comunes, deben estar *sincronizados*. Se sincronizan con respecto al objeto común al incluir el modificador `synchronized` en el encabezado de cualquier método de objeto común que pueda ser llamado por un thread que deba estar sincronizado. Además, se usa un *semáforo* para otorgar acceso sólo a un thread a la vez, y se *bloquean* (detención temporal) todos los otros threads.

La figura A9.5*a* contiene una versión corregida de la primera parte de la clase `Encounter` en la figura A9.4. En esta parte corregida de la definición de clase, lo único distinto es la adición de otra variable de instancia, un semáforo `boolean` que indica cuál thread tiene acceso actualmente al objeto común. Esta declaración adicional aparece en negritas.

La figura A9.5*b* contiene una versión corregida de la segunda parte de la clase `Encounter` en la figura A9.4. Todo el código nuevo aparece en negritas. Eso muestra lo que se hace al sincronizar varios threads. Primero, debe incluirse el modificador `synchronized` en el encabezado de los métodos que se desea sincronizar. Luego, al inicio de cada uno de estos métodos, es necesario colocar un ciclo `while` con un bloque `try` que contenga la declaración simple:

```
wait();
```

```
/*************************************************************
* Encounter.java
* Dean & Dean
*
* Esto describe la interacción predador/presa (consumidor/productor).
*************************************************************/

public class Encounter
{
  public final int EVENTS;
  private int number = -1;
  private Prey prey;
  private Predator predator;
  private boolean predatorHasAccess = false; // access semaphore

  //*********************************************************

  public Encounter(Prey prey, Predator predator)
  {
    this.prey = prey;
    this.predator = predator;
    prey.setEncounter(this);
    predator.setEncounter(this);
    EVENTS = predator.DELAY.length;
  } // end constructor
```

Figura A9.5*a* Versión corregida de la clase `Encounter`, parte A.

Esta declaración bloquea el acceso al resto de ese método, de modo que cuando se quiera bloquear el acceso, es necesario hacer `true` la condición `while`. Por último, inserte un par de declaraciones especiales justo antes del regreso. En la primera de estas declaraciones especiales, la fase del semáforo debe establecerse de modo que haga `true` la condición `while` precedente. La segunda de de estas declaraciones debe ser:

```
notifyAll();
```

En ambos casos, la condición del ciclo `while` es la fase del semáforo que bloquea al thread que llama. Si esta condición es `true` cuando un thread externo llama al método, el flujo se dirige de inmediato a la declaración `wait`,[4] lo cual bloquea la ejecución del thread que llama en ese punto de la ejecución. Ese thread permanece en este estado bloqueado hasta que recibe una llamada de "despertador" del sistema iniciada por otra ejecución del thread del método `notifyAll`, en cuyo momento reanuda su ejecución a partir del momento en que fue bloqueado. Si el programa está escrito correctamente, la condición en el ciclo `while` de exactamente uno de los métodos sincronizados es `false`. Cuando se llama este método particular, el flujo salta del ciclo `while` al siguiente código ejecutable.

Por ejemplo, cuando en la figura A9.2 el thread `prey` llama por primera vez a `encounter.beApart`, el semáforo `predatorHasAccess` es `false`. Así, en el método `beApart` en la figura A9.5*b*, la ejecución del thread `prey` se salta al ciclo `while` e imprime lo siguiente:

```
prey start beApart 0
```

[4] El método `wait` es heredado por todos los objetos de la clase `Object`, y `lanza` una `InterruptedException` (justo como el método `sleep`) si el thread que está esperando es interrumpido durante la espera. La llamada `wait` debe colocarse en un bloque `try` porque (como ya se indicó), la excepción que pudiera lanzar es una excepción comprobada, aun cuando nunca se creó la condición que lanza esa excepción.

```
  //*************************************************************

  public synchronized int beApart()
  {
    while (predatorHasAccess)
    {
      try
      {
        wait();      // El thread prey espera aquí hasta que es notificado.
      }
      catch (Exception e) { }
    }
    // prey has access, so go apart
    number++;
    System.out.println(Thread.currentThread().getName() +
      " start beApart " + number);
    predatorHasAccess = true;
    notifyAll();
    return number;
  } // end beApart

  //*************************************************************

  public synchronized int beTogether()
  {
    while (!predatorHasAccess)
    {
      try
      {
        wait();   // El thread predator espera aquí hasta que es notificado.
      }
      catch (Exception e) { }
    }
    // el predador tiene acceso, de modo que vienen juntos
    System.out.println(Thread.currentThread().getName() +
      " finish beTogether " + number);
    predatorHasAccess = false;
    notifyAll();
    return number;
  } // end beTogether
} // end Encounter class
```

Figura A9.5*b* Versión corregida de la clase `Encounter`, parte B.

Luego, la ejecución cambia el semáforo `predatorHasAccess` a `true`, llama a `notifyAll` y regresa. La próxima vez que el thread `prey` llama a `encounter.beApart`, el valor `true` de la condición `while` provoca la ejecución de la declaración `wait`, lo cual bloquea al thread `prey` en ese punto.

Mientras tanto (tan pronto como empieza), el thread `predator` comienza a ejecutarse en paralelo con el thread `prey`. Cuando en la figura A9.3 el thread `predator` llama por primera vez a su método `sleep`, detiene su ejecución durante 2.347 segundos. Durante la mayor parte del tiempo de ese retraso, ambos thread permanecen bloqueados y no se ejecuta ninguno. Cuando transcurren los 2.347 segundos, el thread `predator` despierta automáticamente, salta a catch y llama a `encounter.beTogether`. Mucho antes de este tiempo, el thread `prey` ya cambió a `true` el semáforo de acceso `predator-`

`HasAccess` en la figura A9.5*a*, de modo que en la figura A9.5*b*, la ejecución del thread `predator` salta el ciclo `while` de `beTogether` e imprime la salida:

```
predator finish beTogether 0
```

Luego, la ejecución cambia a `false` el semáforo de acceso `predatorHasAccess`, llama a `notifyAll` y regresa. De vuelta en el método `run` de la figura A9.3, el thread `predator` entra a la segunda iteración del ciclo `for` y de nuevo queda dormido en su segundo retraso temporal.

Mientras tanto, la llamada previa a `notifyAll` del thread `predator` reactiva al thread `prey` que está en espera y le permite continuar con el ciclo `while` en el método `beApart` en la figura A9.5*b*. Esta vez, cuando la ejecución regresa a la condición `while`, encuentra que el valor del semáforo `predatorHasAccess` es `false`. Esto le permite escapar del ciclo `while` e imprimir la salida:

```
prey start beApart 1
```

Luego, como antes, cambia el semáforo `predatorHasAccess` a `true`, llama a `notifyAll` y regresa.

Esta alternancia entre estar separados y estar juntos continúa hasta que (en el método `run` de la figura A9.2) `number == encounter.EVENTS - 1`, lo cual termina el thread `prey`. Poco después (en el método `run` de la figura A9.3) `i == DELAY.length`, lo cual termina el thread `predator`. Al usar la versión corregida de la clase `Encounter` que aparece en las figuras A9.5*a* y A9.5*b*, la salida del programa se parece a esto:

Salida:

```
prey start beApart 0
predator finish beTogether 0
prey start beApart 1
predator finish beTogether 1
prey start beApart 2
predator finish beTogether 2
prey start beApart 3
prey run finished.
predator finish beTogether 3
predator run finished.
```

Alentamos al lector para que ejecute este programa para que adquiera una sensación física de la interacción entre las operaciones de retraso temporal y de espera.

Índice

D

R