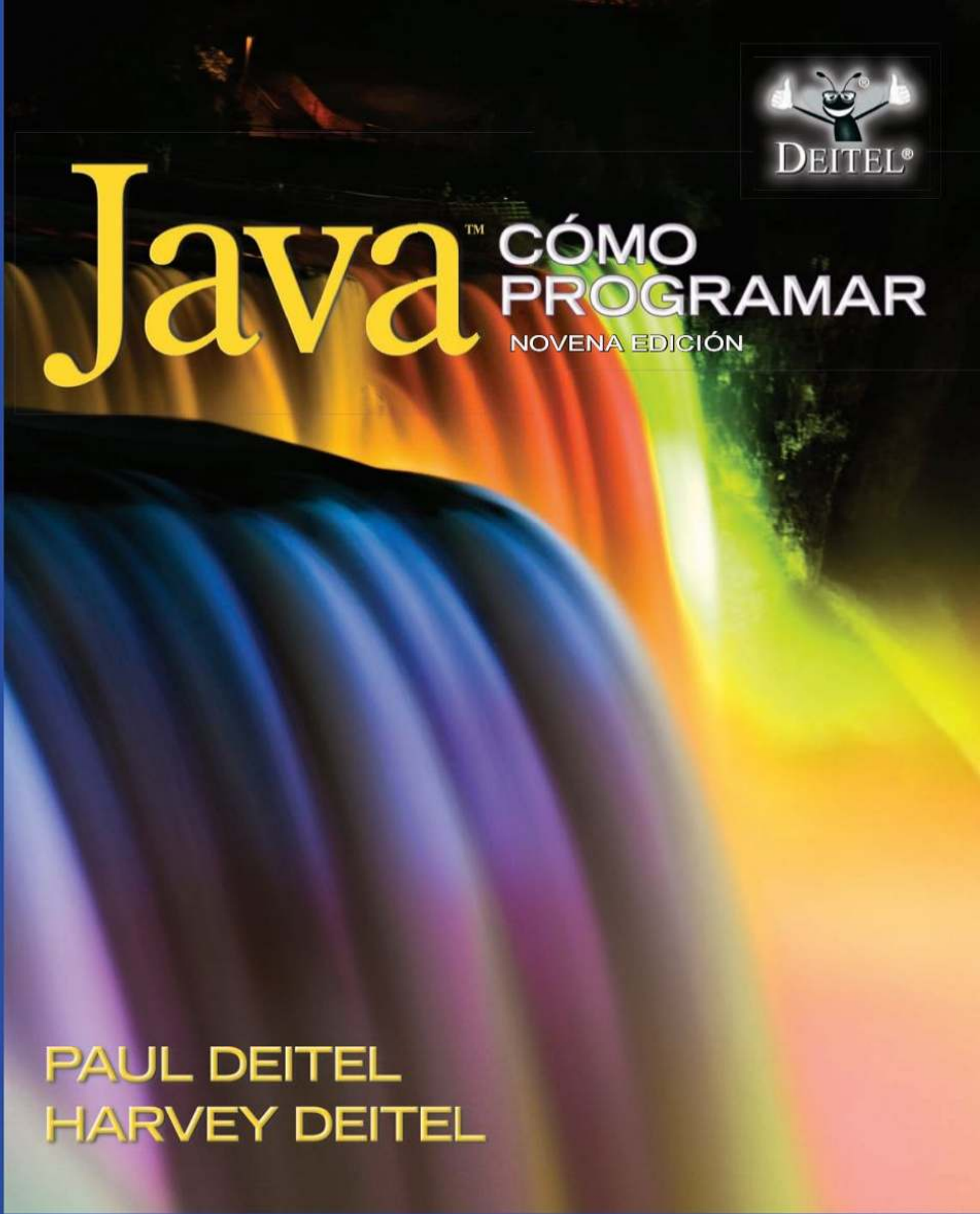


ALWAYS LEARNING
PEARSON

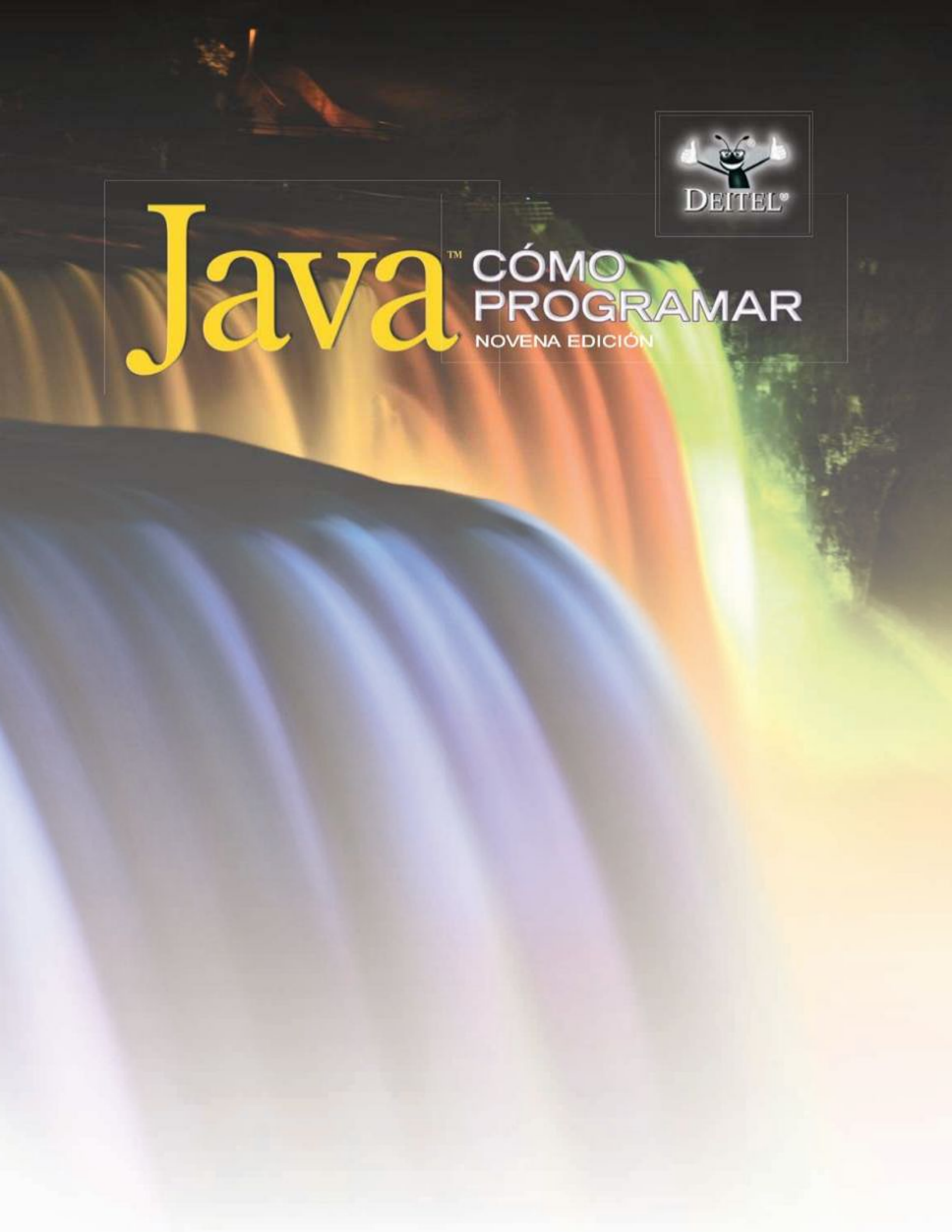
ACCESO A LOS CAPÍTULOS ADICIONALES DEL LIBRO

Para acceder a los capítulos 12 a 19 (en español), 20 a 31 y Apéndices M a Q (en inglés) mencionados en el texto, visite el sitio Web de este libro:

www.pearsonenespañol.com/deitel

Utilice una moneda para descubrir el código de acceso.
(No use objetos filosos porque podría dañarlo).

IMPORTANTE:
¡Este código de acceso tiene vigencia de 2 días!
Asegúrese que el código no aparezca dañado ya que sólo puede usarse una vez y no será reemplazado en ningún caso.


DEITEL®
Java™ CÓMO PROGRAMAR
NOVENA EDICIÓN

Java™ CÓMO PROGRAMAR

NOVENA EDICIÓN

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

Traductor
Alfonso Vidal Romero Elizondo
Ingeniero en Sistemas Electrónicos
ITESM, Campus Monterrey

Revisión técnica
Roberto Martínez Román
Departamento de Tecnología de Información y Computación
ITESM, Campus Estado de México

Domingo Acosta Infante
Departamento de Ingeniería en Informática
Instituto Tecnológico de Morelia

PEARSON

Datos de catalogación bibliográfica

DEITEL, PAUL y HARVEY DEITEL

Cómo programar en Java
Novena edición

PEARSON EDUCACIÓN, México, 2012

ISBN: 978-607-32-1150-5
Área: Computación

Formato: 20 × 25.5 cm Páginas: 616

Edición en español

Dirección Educación Superior:	Mario Contreras
Editor Sponsor:	Luis Miguel Cruz Castillo e-mail: luis.cruz@pearson.com
Editor de Desarrollo:	Bernardino Gutiérrez Hernández
Supervisor de Producción:	José D. Hernández Garduño
Gerente Editorial Educación Superior Latinoamérica:	Marisa de Anta

NOVENA EDICIÓN, 2012

ISBN VERSIÓN IMPRESA: 978-607-32-1150-5
ISBN VERSIÓN E-BOOK: 978-607-32-1151-2
ISBN E-CHAPTER: 978-607-32-1152-9

Impreso en México. *Printed in Mexico.*
1 2 3 4 5 6 7 8 9 0 - 15 14 13 12

PEARSON

En memoria del sargento Shriver,
El primer director del Cuerpo de Paz y fundador
de numerosas organizaciones sociales:

Por una vida de marcar la diferencia.

Paul y Harvey Deitel

Contenido

Los capítulos 12 a 19 se encuentran en español en el sitio Web del libro

Los capítulos 20 a 31 se encuentran en inglés en el sitio Web del libro

Los apéndices M a Q se encuentran en inglés en el sitio Web del libro

Prefacio

No vivas más en fragmentos, conéctate.
—Edgar Morgan Foster

Bienvenido a *Cómo programar en Java*, novena edición. Este libro presenta las tecnologías de vanguardia para estudiantes, profesores y desarrolladores de software.

El nuevo capítulo 1 atrae la atención de los estudiantes con hechos y cifras fascinantes, para que encuentren más emocionante el hecho de estudiar sobre las computadoras y la programación. Ofrece los siguiente: una tabla sobre algunos proyectos de investigación que se hacen posibles gracias a las computadoras; un análisis sobre el hardware y las tendencias tecnológicas actuales; jerarquía de datos; una tabla de plataformas de aplicaciones móviles y de Internet; una nueva sección sobre redes sociales; una introducción a Android; una tabla de los servicios Web más populares; una tabla de las publicaciones tecnológicas y de negocios, además de los sitios Web que le ayudarán a estar al día con las noticias y tendencias más recientes sobre tecnología; y ejercicios actualizados.

El libro es apropiado para secuencias de cursos introductorios apoyados en las recomendaciones curriculares de ACM/IEEE y sirve como preparación para el examen de Colocación avanzada (AP) de ciencias computacionales.

Nos enfocamos en las mejores prácticas de ingeniería de software. La base del libro es nuestro reconocido "método de código activo": los conceptos se presentan en el contexto de programas funcionales completos, en lugar de hacerlo a través de fragmentos separados de código. Cada ejemplo de código completo viene acompañado de ejemplos de ejecuciones actuales. Todo el código fuente está disponible en `www.deitel.com/books/jhtp9/` (en inglés) y en el sitio Web de este libro `www.pearsonenespañol.com/deitel` (en español).

Si surge alguna duda o pregunta mientras lee este libro, envíe un correo electrónico a `deitel@deitel.com`; le responderemos a la brevedad. Para obtener actualizaciones sobre este libro, visite `www.deitel.com/books/jhtp9/`, síganos en Facebook (`www deitel.com/deitelfan`) y Twitter (`@deitel`). También puede suscribirse al boletín de correo electrónico *Deitel® Buzz Online* (`www.deitel.com/newsletter/subscribe.html`).

Características nuevas y mejoradas

He aquí las actualizaciones que realizamos a la 9a edición:

Java Standard Edition (SE) 7

- ***Fácil de usar como libro para Java SE 6 y Java SE 7.*** Hay unas cuantas características de Java Standard Edition (SE) 7 que afectan a los cursos de ciencias computacionales CS 1 y CS 2. Cubrimos esas características en secciones modulares opcionales que se pueden incluir u omitir con facilidad. He aquí una parte de la nueva funcionalidad: objetos `String` en instrucciones `switch`, la instrucción `try` con recursos (`try`-with-resources) para administrar objetos `AutoClosable`, multi-`catch` para definir un solo manejador de excepciones en sustitución de varios que realizan la misma tarea, las API del sistema de archivos NIO y la inferencia de tipos de objetos genéricos a partir de la variable a la que están asignados, mediante el uso de la notación `<>`. También veremos las generalidades sobre las nuevas características de la API concurrente.

- ***Nuevas API del sistema de archivos de Java SE 7.*** Ofrecemos una versión en línea *alternativa* (en inglés) del capítulo 17, Archivos, flujos y serialización de objetos, que se volvió a implementar con las nuevas API del sistema de archivos de Java SE 7.
- ***Versiones `AutoClosable` de `Connection`, `Statement` y `ResultSet` de Java SE 7.*** Con el código fuente para el capítulo 28 (en inglés), proporcionamos una versión del primer ejemplo del capítulo que se implementó mediante el uso de las versiones `AutoClosable` de `Connection`, `Statement` y `ResultSet`. Los objetos `AutoClosable` reducen la probabilidad de fugas de recursos cuando se utilizan con la instrucción `try` con recursos (try-with-resources) de Java SE 7, la cual cierra de manera automática los objetos `AutoClosable` que se asignan en los paréntesis después de la palabra clave `try`.

Características pedagógicas

- ***Mejoramos el conjunto de ejercicios Marcar la diferencia.*** Le alentamos a utilizar las computadoras e Internet para investigar y resolver problemas sociales relevantes. Estos ejercicios están diseñados para aumentar la conciencia y el análisis en torno a los problemas importantes a los que se enfrenta el mundo. Esperamos que usted los aborde con sus propios valores, políticas y creencias. Dé un vistazo a nuestro nuevo Centro de recursos para marcar una diferencia (en inglés) en `www.deitel.com/MakingADifference`, en donde obtendrá ideas adicionales que tal vez desee investigar más a fondo.
- ***Números de página para los términos clave en los resúmenes de cada capítulo.*** En la lista de términos clave que aparece en el resumen de cada capítulo incluimos el número de página donde se define el término.
- ***Comentarios en video.*** En el sitio Web de este libro encontrará comentarios en video (VideoNotes), en inglés, en las que el coautor Paul Deitel explica con detalle la mayoría de los programas de los capítulos básicos. Los profesores nos han dicho que estos comentarios constituyen un recurso valioso para sus estudiantes.

Tecnología de objetos

- ***Programación y diseño orientados a objetos.*** En el capítulo 1 presentamos la terminología y los conceptos básicos de la tecnología de objetos. En el capítulo 3 los estudiantes desarrollan sus primeras clases y objetos personalizados. Al presentar los objetos y las clases en los primeros capítulos hacemos que los estudiantes de inmediato "piensen en objetos" y dominen estos conceptos [en los cursos que requieren una metodología en la que se presenten los objetos en capítulos posteriores, le recomendamos el libro *Java How to Program, Late Objects Version, 8ª edition* (en inglés), el cual presenta en los primeros seis capítulos los fundamentos de la programación (incluyendo dos sobre instrucciones de control) y continúa con varios capítulos que introducen los conceptos de programación orientada a objetos en forma gradual].
- ***Manejo de excepciones.*** Integramos el manejo básico de excepciones en los primeros capítulos del libro; además los profesores pueden extraer con facilidad más material del capítulo 11, Manejo de excepciones: un análisis más detallado, para mostrarlo con anticipación.
- ***Las clases `Arrays` y `ArrayList`.*** El capítulo 7 cubre la clase `Arrays` —que contiene métodos para realizar manipulaciones comunes de arreglos— y la clase `ArrayList` —que implementa una estructura de datos tipo arreglo, cuyo tamaño se puede ajustar en forma dinámica. Esto va de acuerdo con nuestra filosofía de obtener mucha práctica al *utilizar las clases existentes, al tiempo que el estudiante aprende a definir sus propias clases.*
- ***Casos de estudio orientados a objetos (OO).*** La presentación de las clases y los objetos en los primeros capítulos del libro aportan casos de estudio de `Tiempo`, `Empleado` y `LibroCalificaciones`, que se entretejen a través varias secciones y capítulos, e introducen conceptos de OO cada vez más profundos.

- ***Ejemplo práctico opcional: uso de UML para desarrollar un diseño orientado a objetos y una implementación en Java de un cajero automático (ATM).*** El UML™ (Lenguaje Unificado de Modelado™) es el lenguaje gráfico estándar en la industria para modelar sistemas orientados a objetos. Los capítulos 12 y 13 (en el sitio Web) contienen un Ejemplo práctico *opcional* sobre diseño orientado a objetos mediante el uso de UML. Diseñamos e implementamos el software para un cajero automático (ATM) simple. Analizamos un documento de requerimientos típico, el cual especifica el sistema que se va a construir. Determinamos las clases necesarias para implementar ese sistema, los atributos que deben tener esas clases, los comportamientos que necesitan exhibir, y especificamos cómo deben interactuar las clases entre sí para cumplir con los requerimientos del sistema. A partir del diseño creamos una implementación *completa* en Java. A menudo los estudiantes informan que pasan por un "momento de revelación": el Ejemplo práctico les ayuda a "atar cabos" y comprender en verdad la orientación a objetos.
- ***Se reordenó la presentación de estructuras de datos.*** Empezamos con la clase genérica `ArrayList` en el capítulo 7. Como los *estudiantes comprenderán los conceptos básicos sobre los genéricos en los primeros capítulos del libro*, nuestros análisis posteriores sobre las estructuras de datos ofrecen un tratamiento más detallado de las colecciones de genéricos, puesto que enseñan a utilizar las colecciones integradas de la API de Java. Luego mostramos cómo implementar los métodos y las clases genéricas. Por último, mostraremos cómo crear estructuras de datos genéricas personalizadas.

Desarrollo Web y de bases de datos (material en inglés en el sitio Web del libro)

- ***JDBC 4.*** El capítulo 28, trata sobre JDBC 4; aquí se utilizan los sistemas de administración de bases de datos Java DB/Apache Derby y MySQL. El capítulo contiene un Ejemplo práctico de OO sobre cómo desarrollar una libreta de direcciones controlada por una base de datos; en este ejemplo se demuestran las instrucciones preparadas y el descubrimiento automático de controladores de JDBC 4.
- ***Java Server Faces (JSF) 2.0.*** Los capítulos 29 y 30 se actualizaron para introducir la tecnología JavaServer Faces (JSF) 2.0, que simplifica en gran medida la creación de aplicaciones Web con JSF. El capítulo 29 presenta ejemplos sobre la creación de interfaces GUI de aplicaciones Web, la validación de formularios y el rastreo de sesiones. El capítulo 30 habla sobre las aplicaciones JSF controladas por datos y habilitadas para Ajax. Este capítulo cuenta con una libreta de direcciones Web multinivel controlada por una base de datos, la cual permite a los usuarios agregar contactos y buscarlos. Esta aplicación habilitada para Ajax proporciona al lector una sensación real del desarrollo de software Web 2.0.
- ***Servicios Web.*** El capítulo 31, Web Services, demuestra cómo crear y consumir servicios Web basados en SOAP *y* REST. Los Ejemplos prácticos presentan el desarrollo de los servicios Web del juego de blackjack y un sistema de reservaciones de una aerolínea.
- ***Java Web Start y el Protocolo de lanzamiento de red de Java (JNLP).*** Presentamos Java Web Start y JNLP, que permiten lanzar applets *y* aplicaciones a través de un navegador Web. Los usuarios pueden instalar estos applets y aplicaciones en forma local para ejecutarlos después. Los programas también pueden solicitar permiso al usuario para acceder a los recursos locales del sistema y a los archivos: con lo cual usted podrá desarrollar applets y aplicaciones más robustas que se ejecuten en forma segura mediante el modelo de seguridad de caja de arena (sandbox) de Java, el cual se aplica al código descargado.

Multihilos (en inglés en el sitio Web)

- ***Multihilos.*** Rediseñamos por completo el capítulo 26, Multithreading [con agradecimiento especial a la orientación de Brian Goetz y Joseph Bowbeer, dos de los coautores de *Java Concurrency in Practice*, Addison-Wesley, 2006].
- ***La clase `SwingWorker`.*** Utilizamos la clase `SwingWorker` para crear *interfaces de usuario multihilos.*

GUI y gráficos

- ***Presentación escalable de GUI y gráficos.*** Los profesores que impartan cursos introductorios tienen una amplia gama de dónde elegir en cuanto a la cantidad de GUI y gráficos por cubrir: desde cero hasta una secuencia introductoria de 10 secciones breves, las cuales se entrelazan con los primeros capítulos hasta llegar a un análisis detallado en los capítulos 14, 15 y 25 y en el apéndice I (este último y el apéndice, en inglés en el sitio Web).
- ***Administrador de esquemas `GroupLayout`.*** Analizamos el administrador de esquemas `GroupLayout` dentro del contexto de la herramienta de diseño de GUI en el entorno de desarrollo integrado (IDE) NetBeans.
- ***Herramientas de ordenamiento y filtrado de `JTable`.*** El capítulo 28 (en inglés en el sitio Web) utiliza estas herramientas para reordenar los datos en un objeto JTable y filtrarlos mediante expresiones regulares.

Otras características

- ***Android.*** Debido al enorme interés en los teléfonos inteligentes y tabletas basadas en Android, hemos integrado una introducción de tres capítulos para el desarrollo de aplicaciones de Android (los encontrará en inglés en el sitio Web del libro). Estos capítulos son de nuestro nuevo libro *Android for Programmers: An App-Driven Approach de la serie Deitel Developer*. Una vez que aprenda Java, descubrirá que es bastante simple desarrollar y ejecutar aplicaciones Android en el emulador gratuito que puede descargar de `developer.android.com`.
- ***Conceptos comunes de ingeniería de software.*** Analizamos el desarrollo ágil de software, la refactorización, los patrones de diseño, LAMP, SaaS (Software as a Service), PaaS (Platform as a Service), la computación en la nube, el software de código abierto y muchos conceptos más.

Gráfico de dependencias

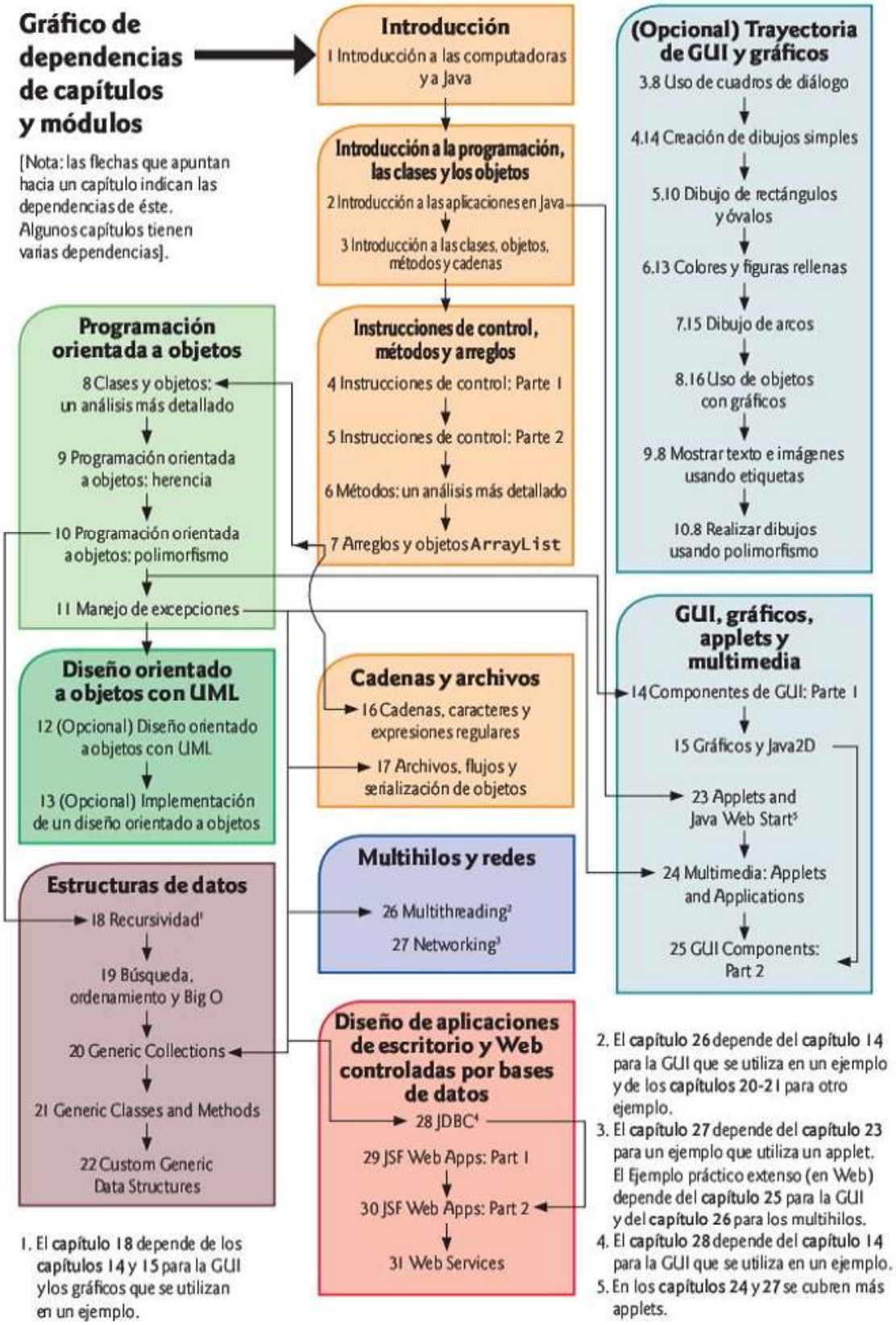
El gráfico de la siguiente página muestra las dependencias entre los capítulos para ayudar a los profesores a planear su programa de estudios. *Cómo programar en Java 9ª edición* es un libro extenso, apropiado para una gran variedad de cursos de programación en distintos niveles, en especial CS 1 y CS 2, además de las secuencias de cursos de introducción en disciplinas relacionadas. El libro tiene una organización modular, claramente delineada. Los capítulos 1 a 11 y 14 a 17 forman una secuencia de programación elemental accesible, con una sólida introducción a la programación orientada a objetos. Los capítulos *opcionales* 12 y 13 constituyen una introducción accesible al diseño orientado a objetos con UML. Tanto la trayectoria de GUI y gráficos como los capítulos 14, 15, 23, 24 y 25 forman una secuencia sustancial de GUI, gráficos y multimedia. Los capítulos 18 a 22 forman una excelente cadena de estructuras de datos. Los capítulos 26 y 27 constituyen una sólida introducción a los conceptos de multihilos y el trabajo en red a través de Internet. Los capítulos 28 a 31 forman una secuencia enriquecida de desarrollo Web con uso intensivo de bases de datos.

Métodos de enseñanza

Cómo programar en Java 9a edición contiene cientos de ejemplos funcionales completos. Hacemos hincapié en la claridad de los programas y nos concentramos en crear software bien diseñado.

Gráfico de dependencias de capítulos y módulos

[Nota: las flechas que apuntan hacia un capítulo indican las dependencias de éste. Algunos capítulos tienen varias dependencias].



1. El **capítulo 18** depende de los **capítulos 14 y 15** para la GUI y los gráficos que se utilizan en un ejemplo.
2. El **capítulo 26** depende del **capítulo 14** para la GUI que se utiliza en un ejemplo y de los **capítulos 20-21** para otro ejemplo.
3. El **capítulo 27** depende del **capítulo 23** para un ejemplo que utiliza un applet. El Ejemplo práctico extenso (en Web) depende del **capítulo 25** para la GUI y del **capítulo 26** para los multihilos.
4. El **capítulo 28** depende del **capítulo 14** para la GUI que se utiliza en un ejemplo.
5. En los **capítulos 24 y 27** se cubren más applets.

Resaltado de código. Colocamos rectángulos de color gris alrededor de los segmentos de código clave en cada programa.

Uso de fuentes para dar énfasis. Resaltamos en **negritas**, dentro del texto, y en el índice, los términos clave en los lugares donde se define. Enfatizamos los componentes en pantalla en la fuente **Helvetica en negritas** (por ejemplo, el menú **Archivo**) y enfatizamos el texto del programa en la fuente `Lucida` (por ejemplo, `int x = 5;`).

Acceso Web. Todo el código fuente utilizado en este libro se puede descargar de:

Capítulos 2 a 19 (en español): `www.pearsonenespañol.com/deitel`
Capítulos 2 a 31 (en inglés): `www.deitel.com/books/jhtp9`

Objetivos. Las citas de apertura van seguidas de una lista de objetivos del capítulo.

Ilustraciones/figuras. Integramos una gran cantidad de tablas, dibujos lineales, diagramas UML, programas y salidas de programa.

Tips de programación. Incluimos tips de programación para ayudarle a enfocarse en los aspectos importantes del desarrollo de programas. Estos tips y prácticas representan lo mejor que hemos podido recabar a lo largo de siete décadas combinadas de experiencia en la programación y la enseñanza.

Buenas prácticas de programación

Las Buenas prácticas de programación *son técnicas que le ayudarán a producir programas más claros, comprensibles y fáciles de mantener.*

Errores comunes de programación

Al poner atención en estos Errores comunes de programación *se reduce la probabilidad de que usted pueda caer en ellos.*

Tips para prevenir errores

Estos tips contienen sugerencias para exponer los errores o gusanos informáticos y eliminarlos de sus programas; muchos de ellos describen aspectos de Java que evitan que entren siquiera a sus programas.

Tips de rendimiento

Estos recuadros resaltan las oportunidades para hacer que sus programas se ejecuten más rápido o para minimizar la cantidad de memoria que ocupan.

Tips de portabilidad

Los Tips de portabilidad *le ayudan a escribir código que pueda ejecutarse en varias plataformas.*

Observaciones de ingeniería de software

Las Observaciones de ingeniería de software *resaltan temas de arquitectura y diseño, lo cual afecta la construcción de los sistemas de software, especialmente los de gran escala.*

Observaciones de apariencia visual

Las Observaciones de apariencia visual *resaltan las convenciones de la interfaz gráfica de usuario. Además, le ayudan a diseñar interfaces gráficas de usuario atractivas y amigables en conformidad con las normas de la industria.*

Viñetas de resumen. Presentamos un resumen detallado del capítulo, estilo lista con viñetas, sección por sección. Para facilitar la referencia, incluimos dentro del texto el número de página donde aparecen los términos clave.

Ejercicios de autoevaluación y respuestas. Se proveen diversos ejercicios de autoevaluación con sus respuestas para que los estudiantes practiquen por su cuenta. Todos los ejercicios en el Ejemplo práctico opcional sobre el ATM están resueltos en su totalidad.

Ejercicios. Los ejercicios de los capítulos abarcan:

- Recordatorio simple de la terminología y los conceptos importantes.
- ¿Cuál es el error en este código?
- ¿Qué hace este código?
- Escritura de instrucciones individuales y pequeñas porciones de métodos y clases.
- Escritura de métodos, clases y programas completos.
- Proyectos importantes.
- En muchos capítulos, ejercicios del tipo Hacer la diferencia.

Índice. Incluimos un índice extenso. Donde se definen los términos clave se resaltan con un número de página **en negritas**.

Software utilizado en *Cómo programar en Java 9ª edición*

Podrá descargar todo el software necesario para este libro sin costo a través de Web. En la sección "Antes de empezar", después de este Prefacio, encontrará vínculos para cada descarga.

Para escribir la mayoría de los ejemplos de este libro utilizamos el kit de desarrollo gratuito Java Standard Edition Development Kit (JDK) 6. Para los módulos opcionales de Java SE 7 utilizamos la versión JDK 7 de acceso anticipado de OpenJDK. En los capítulos 29 a 31 también utilizamos el IDE Netbeans; en el capítulo 28 usamos MySQL y MySQL Connector/J. Encontrará recursos y descargas de software adicionales en nuestros Centros de recursos de Java, ubicados en:

```
www.deitel.com/ResourceCenters.html
```

Suplementos para el profesor (en inglés)

Los siguientes suplementos están disponibles sólo para profesores a través del Centro de recursos para el profesor de Pearson (`www.pearsonenespañol.com/deitel`):

- ***Diapositivas de PowerPoint®*** con todo el código y las figuras del texto, además de elementos en viñetas que sintetizan los puntos clave.
- ***Test Item File (Archivo de pruebas)*** con preguntas de opción múltiple (aproximadamente dos por cada sección del libro).
- ***Manual de soluciones*** con soluciones para la gran mayoría de los ejercicios de final de capítulo.

El acceso a estos recursos está limitado estrictamente a profesores universitarios que impartan clases con base en el libro. Sólo ellos pueden obtener acceso a través de los representantes de Pearson. *No* se proveen soluciones para los ejercicios de "proyectos". Revise nuestro Centro de recursos de proyectos de programación (`www.deitel.com/ProgrammingProjects/`), en donde encontrará muchos ejercicios adicionales y proyectos nuevos.

Si no es un miembro docente registrado, póngase en contacto con su representante de Pearson.

Reconocimientos

Queremos agradecer a Abbey Deitel y Barbara Deitel por las extensas horas que dedicaron a este proyecto. Somos afortunados al haber trabajado en este proyecto con el dedicado equipo de editores profesionales de Pearson. Apreciamos la orientación, inteligencia y energía de Michael Hirsch, editor en jefe de Ciencias computacionales. Carole Snyder reclutó a los revisores del libro y se hizo cargo del proceso de revisión. Bob Engelhardt se hizo cargo de la producción del libro.

Revisores

Queremos agradecer los esfuerzos de los revisores de la octava y novena ediciones, quienes revisaron exhaustivamente el texto y los programas, y proporcionaron innumerables sugerencias para mejorar la presentación: Lance Andersen (Oracle), Soundararajan Angusamy (Sun Microsystems), Joseph Bowbeer (Consultor), William E. Duncan (Louisiana State University), Diana Franklin (University of California, Santa Barbara), Edward F. Gehringer (North Carolina State University), Huiwei Guan (Northshore Community College), Ric Heishman (George Mason University), Dr. Heinz Kabutz (JavaSpecialists.eu), Patty Kraft (San Diego State University), Lawrence Premkumar (Sun Microsystems), Tim Margush (University of Akron), Sue McFarland Metzger (Villanova University), Shyamal Mitra (The University of Texas at Austin), Peter Pilgrim (Consultor), Manjeet Rege, Ph.D. (Rochester Institute of Technology), Manfred Riem (Java Champion, Consultor, Robert Half), Simon Ritter (Oracle), Susan Rodger (Duke University), Amr Sabry (Indiana University), José Antonio González Seco (Parlamento de Andalucía), Sang Shin (Sun Microsystems), S. Sivakumar (Astra Infotech Private Limited), Raghavan "Rags" Srinivas (Intuit), Monica Sweat (Georgia Tech), Vinod Varma (Astra Infotech Private Limited) y Alexander Zuev (Sun Microsystems).

Bueno, ¡ahí lo tiene! A medida que lea el libro, apreciaremos con sinceridad sus comentarios, críticas, correcciones y sugerencias para mejorarlo. Dirija toda su correspondencia a:

```
deitel@deitel.com
```

Le responderemos oportunamente. Esperamos que disfrute el trabajo con este libro. ¡Buena suerte!

Paul y Harvey Deitel

Acerca de los autores

Paul J. Deitel, CEO y Director Técnico de Deitel & Associates, Inc., es egresado del Sloan School of Management del MIT, en donde estudió Tecnología de la Información. A través de Deitel & Associates, Inc., ha impartido cursos de Java, C, C++, C#, Visual Basic y programación en Internet a clientes de la industria, como: Cisco, IBM, Siemens, Sun Microsystems, Dell, Lucent Technologies, Fidelity, NASA en el Centro Espacial Kennedy, el National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, SunGard Higher Education, Stratus, Cambridge Technology Partners, One Wave, Hyperion Software, Adra Systems, Entergy, CableData Systems, Nortel Networks, Puma, iRobot, Invensys y muchos más. Él y su coautor, el Dr. Harvey M. Deitel, son autores de los libros de programación más vendidos en el mundo.

Dr. Harvey M. Deitel, Presidente y Consejero de Estrategia de Deitel & Associates, Inc., tiene 50 años de experiencia en el campo de la computación. El Dr. Deitel obtuvo una licenciatura y una maestría por el MIT y un doctorado de la Universidad de Boston. Tiene muchos años de experiencia como profesor universitario, la cual incluye un puesto vitalicio y el haber sido presidente del Departamento de Ciencias de la Computación en Boston College antes de fundar, con su hijo Paul J. Deitel, Deitel & Associates, Inc. Él y Paul son coautores de varias docenas de libros y paquetes multimedia LiveLessons, y piensan escribir muchos más. Los textos de los Deitel se han ganado el reconocimiento internacional y han sido traducidos al japonés, alemán, ruso, chino, español, coreano, francés, polaco, italiano, portugués, griego, urdú y turco. El Dr. Deitel ha impartido cientos de seminarios profesionales para grandes empresas, instituciones académicas, organizaciones gubernamentales y diversos sectores del ejército.

Capacitación corporativa de Deitel & Associates, Inc.

Deitel & Associates, Inc., es una empresa reconocida a nivel mundial, dedicada al entrenamiento corporativo y la creación de contenido. La empresa proporciona cursos impartidos por profesores en las instalaciones de sus clientes en todo el mundo, sobre la mayoría de los lenguajes y plataformas de programación, como Java™, C++, Visual C++®, C, Visual C#®, Visual Basic®, XML®, Python®, tecnología de objetos, programación en Internet y World Wide Web, desarrollo de aplicaciones para Android™ e iPhone®, y una lista cada vez mayor de cursos adicionales de programación y desarrollo de software. Los fundadores de Deitel & Associates, Inc. son Paul J. Deitel y el Dr. Harvey M. Deitel. Entre sus clientes están muchas de las empresas más grandes del mundo, agencias gubernamentales, sectores del ejército e instituciones académicas. A lo largo de su sociedad editorial de 35 años con Prentice Hall/Pearson, Deitel & Associates, Inc. ha publicado libros de texto de vanguardia sobre programación, libros profesionales, y cursos de video *LiveLessons* con base en DVD y Web. Puede contactarse con Deitel & Associates, Inc. y con los autores por medio de correo electrónico:

```
deitel@deitel.com
```

Para conocer más acerca de Deitel & Associates, Inc., sus publicaciones y su currículum mundial de la Serie de Capacitación Corporativa *Dive Into®*, visite:

```
www.deitel.com/training/
```

y suscríbase al boletín gratuito de correo electrónico, *Deitel® Buzz Online*, en:

```
www.deitel.com/newsletter/subscribe.html
```

Además puede seguir a los autores en Facebook (`www.deitel.com/deitelfan`) y Twitter (`@deitel`).

Antes de empezar

Esta sección contiene información que debería revisar antes de usar este libro, además de las instrucciones para asegurar que su computadora esté configurada de manera apropiada para utilizarla con el libro. Publicaremos las actualizaciones a la sección "Antes de empezar" (en caso de necesitarse) en el siguiente sitio Web:

```
www.deitel.com/books/jhtp9/
```

Convenciones de fuentes y nomenclatura

Utilizamos fuentes para diferenciar los componentes en la pantalla (como los nombres de menús y sus elementos) y el código o los comandos en Java. Nuestra convención hace hincapié en los componentes en pantalla en una fuente **Helvetica** sans-serif en negritas (por ejemplo, el menú **Archivo**) y enfatiza el código y los comandos en Java en una fuente `Lucida` sans-serif (por ejemplo, `System.out.println()`).

Software a utilizar con el libro

Podrá descargar todo el software necesario para este libro sin costo a través de Web.

Kit de desarrollo de software Java SE (JDK) 6 y 7

Para escribir la mayoría de los ejemplos de este libro manejamos el kit de desarrollo gratuito Java Standard Edition Development Kit (JDK) 6, que está disponible en:

```
www.oracle.com/technetwork/java/javase/downloads/index.html
```

Para los módulos opcionales de Java SE 7, empleamos la versión JDK 7 de acceso anticipado de OpenJDK, que está disponible en:

```
dlc.sun.com.edgesuite.net/jdk7/binaries-/index.html
```

Java DB, MySQL y MySQL Connector/J

En el capítulo 28 (en inglés en el sitio Web del libro) utilizamos los sistemas de administración de bases de datos Java DB y MySQL Community Edition. Java DB es parte de la instalación del JDK. Al momento de escribir este libro el instalador de 64 bits del JDK no instalaba Java DB en forma apropiada. Si usted utiliza la versión de 64 bits de Java, tal vez necesite instalar Java DB por separado. Puede descargar Java DB de:

```
www.oracle.com/technetwork/java/javadb/downloads/index.html
```

Al momento de escribir este libro, la versión más reciente de MySQL Community Edition era la 5.5.8. Para instalar MySQL Community Edition en Windows, Linux o Mac OS X, consulte las generalidades sobre la instalación para su plataforma en:

- Windows: `dev.mysql.com/doc/refman/5.5/en/windows-installation.html`
- Linux: `dev.mysql.com/doc/refman/5.5/en/linux-installation-rpm.html`
- Mac OS X: `dev.mysql.com/doc/refman/5.5/en/macosx-installation.html`

Siga con cuidado las instrucciones para descargar e instalar el software en su plataforma, el cual está disponible en:

```
dev.mysql.com/downloads/mysql/
```

También necesita instalar el **MySQL Connector/J** (la J se refiere a Java), que permite a los programas usar JDBC para interactuar con MySQL. Puede descargar el MySQL Connector/J de

```
dev.mysql.com/downloads/connector/j/
```

Al momento de escribir este libro, la versión disponible de MySQL Connector/J era la 5.1.14. La documentación de Connector/J se encuentra en

```
dev.mysql.com/doc/refman/5.5/en/connector-j.html
```

Para instalar MySQL Connector/J, siga con cuidado las instrucciones de instalación en:

```
dev.mysql.com/doc/refman/5.5/en/connector-j-installing.html
```

No le recomendamos modificar la variable de entorno `CLASSPATH` de su sistema, según se indica en las instrucciones de instalación. En cambio le enseñaremos a usar MySQL Connector/J especificándolo como una opción en la línea de comandos al ejecutar sus aplicaciones.

Cómo obtener los ejemplos de código

Los códigos de los ejemplos de este libro están disponibles para descargarlos en

```
www.pearsonenespañol.com/deitel
```

Si no está registrado aún en nuestro sitio Web (de los autores), vaya a `www.deitel.com` y haga clic en el vínculo **Register** debajo de nuestro logotipo en la esquina superior izquierda de la página. Escriba su información. El registro no tiene ningún costo y no compartiremos su información con nadie. Sólo le enviaremos correos electrónicos relacionados con la administración de su cuenta. También puede registrarse de manera independiente para recibir nuestro boletín gratuito de correo electrónico *Deitel® Buzz Online* en `www.deitel.com/newsletter/subscribe.html`. Una vez que se registre en el sitio, recibirá un correo electrónico de confirmación con su código de verificación. *Haga clic en el vínculo del correo electrónico de confirmación para completar su registro.* Configure su cliente de correo electrónico de modo que permita los correos electrónicos provenientes de `deitel.com`, para asegurar que el correo electrónico de confirmación no se filtre como correo basura.

Después vaya a `www.deitel.com` e inicie sesión usando el vínculo Login debajo de nuestro logotipo en la esquina superior izquierda de la página. Vaya a `www.deitel.com/books/jhtp9`. Encontrará el vínculo para descargar los ejemplos bajo el encabezado **Download Code Examples and Other Premium Content for Registered Users** (Descargar ejemplos de código y contenido especial adicional para usuarios registrados). Anote la ubicación en donde eligió guardar el archivo ZIP en su computadora.

En este libro suponemos que los ejemplos se encuentran en el directorio `C:\Ejemplos` de su computadora. Extraiga el contenido del archivo `Ejemplos.zip`; utilice una herramienta como WinZip (`www.winzip.com`) o las herramientas integradas de su sistema operativo.

Cómo establecer la variable de entorno PATH

La variable de entorno `PATH` en su computadora designa los directorios en los que la computadora debe buscar aplicaciones, como las que le permiten compilar y ejecutar sus aplicaciones de Java (las cuales son `javac` y `java`, respectivamente). *Siga con cuidado las instrucciones de instalación para Java en su plataforma, de modo que se asegure de establecer de manera correcta la variable de entorno* `PATH`.

Si no establece la variable PATH de manera correcta, cuando utilice las herramientas del JDK recibirá un mensaje como éste:

```
'java' no se reconoce como un comando interno o externo, programa o
archivo por lotes ejecutable.
```

En este caso, regrese a las instrucciones de instalación para establecer la variable PATH y vuelva a comprobar sus pasos. Si descargó una versión más reciente del JDK, tal vez tenga que cambiar el nombre del directorio de instalación del JDK en la variable PATH.

Cómo establecer la variable de entorno CLASSPATH

Si intenta ejecutar un programa de Java y recibe un mensaje como éste:

```
Exception in thread "main" java.lang.NoClassDefFoundError: SuClase
```

entonces su sistema tiene una variable de entorno CLASSPATH que debe modificar. Para corregir este error, siga los pasos para establecer la variable de entorno PATH, localice la variable CLASSPATH y modifique su valor para que incluya el directorio local: que por lo general se representa como un punto (.). En Windows agregue

```
.;
```

al principio del valor de CLASSPATH (sin espacios antes o después de esos caracteres). En otras plataformas, sustituya el punto y coma con los caracteres separadores de ruta apropiados: por lo general, el signo de dos puntos (:).

Apariencia visual Nimbus de Java

Java tiene una apariencia visual multiplataforma elegante, conocida como Nimbus. En los programas con interfaz gráfica de usuario, hemos configurado nuestros sistemas para usar Nimbus como la apariencia visual predeterminada.

Para establecer Nimbus como la opción predeterminada para todas las aplicaciones de Java, debe crear un archivo de texto llamado `swing.properties` en la carpeta `lib` de las carpetas de instalación del JDK y del JRE. Coloque la siguiente línea de código en el archivo:

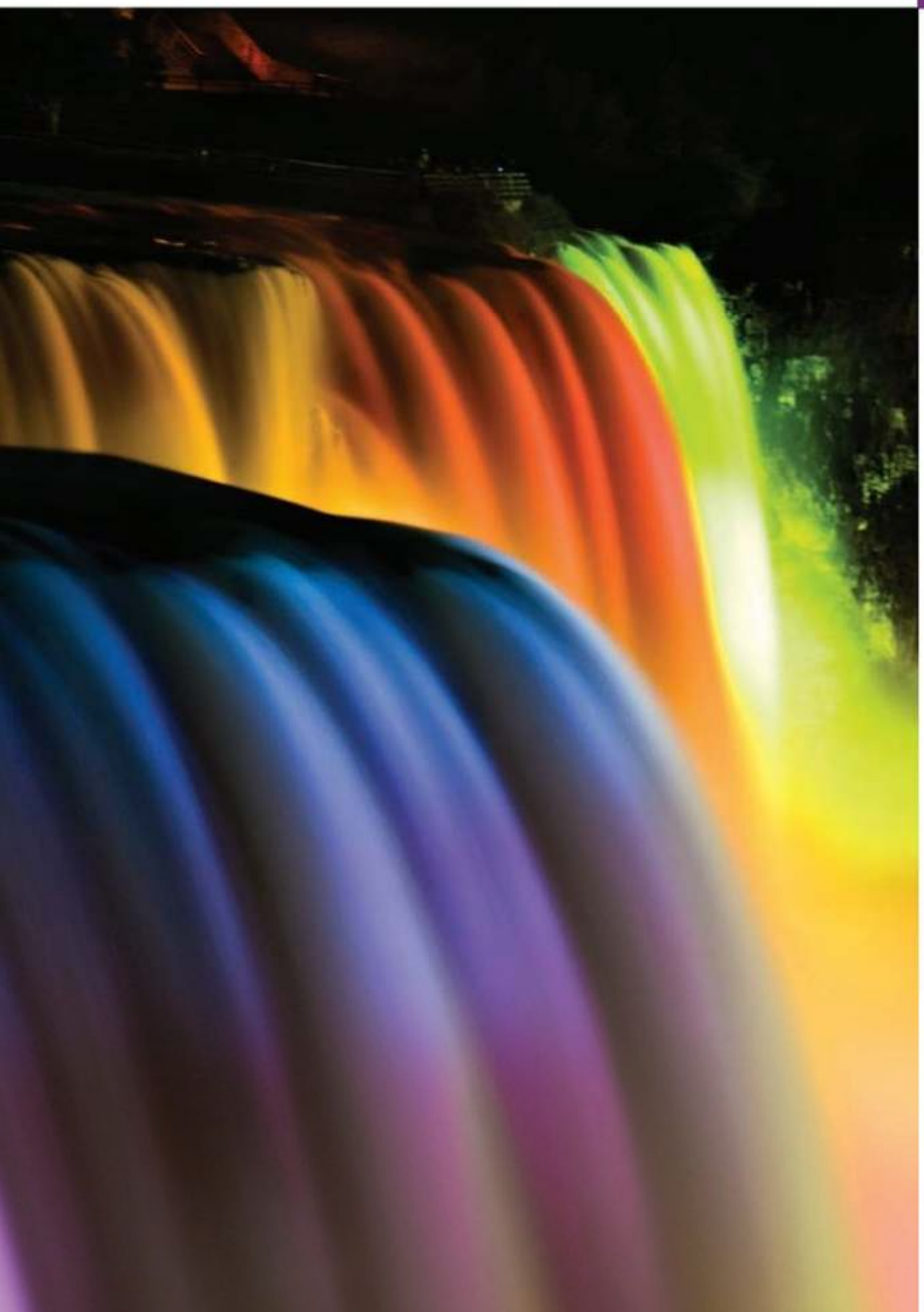
```
swing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

Para obtener más información sobre cómo localizar estas carpetas de instalación, visite `java.sun.com/javase/6/webnotes/install/index.html`. [*Nota:* además del JRE individual hay un JRE anidado en su carpeta de instalación del JDK. Si utiliza un IDE que depende del JDK (como NetBeans), tal vez también tenga que colocar el archivo `swing.properties` en la carpeta `lib` de la carpeta `jre` anidada].

Ahora está listo para empezar sus estudios de Java con el libro *Cómo programar en Java, 9ª edición.* ¡Esperamos que disfrute el libro!

Introducción a las computadoras y a Java

1



El hombre sigue siendo la computadora más extraordinaria de todas.
—John F. Kennedy

Un buen diseño es un buen negocio.
—Thomas J. Watson, fundador de IBM

Qué maravilloso es que nadie necesite esperar un solo momento para empezar a mejorar el mundo.
—Anne Frank

Objetivos

En este capítulo aprenderá sobre:

- Los emocionantes y recientes acontecimientos en el campo de las computadoras.
- Los conceptos básicos de hardware, software y redes.
- La jerarquía de datos.
- Los distintos tipos de lenguajes de programación.
- Los conceptos básicos de la tecnología de objetos.
- La importancia de Internet y de la Web.
- Un típico entorno de desarrollo de programas en Java.
- Cómo probar una aplicación en Java.
- Algunas de las recientes tecnologías clave de software.
- La forma en que las computadoras le pueden ayudar a hacer una diferencia.

Plan general

1.1 Introducción

Bienvenido a Java: el lenguaje de programación de computadoras más utilizado en el mundo. Usted ya está familiarizado con las poderosas tareas que realizan las computadoras. Mediante este libro de texto, usted escribirá instrucciones para ordenar a las computadoras que realicen esos tipos de tareas. El *software* (las instrucciones que usted escribe) controla el *hardware* (las computadoras).

Aprenderá sobre la *programación orientada a objetos*: la metodología de programación clave de la actualidad. En este texto creará y trabajará con muchos *objetos de software*.

Java es el lenguaje preferido para satisfacer las necesidades de programación empresariales de muchas organizaciones. También se ha convertido en el lenguaje de elección para implementar aplicaciones basadas en Internet y software para dispositivos que se comunican a través de una red.

Hoy en día hay en uso más de mil millones de computadoras de propósito general, además de miles de millones de teléfonos celulares, teléfonos inteligentes (smartphones) y dispositivos portátiles (como las computadoras tipo tableta) habilitados para Java. De acuerdo con un estudio realizado por eMarketer, el número de usuarios móviles de Internet llegará a cerca de 134 millones para 2013.[1] Otros estudios han proyectado ventas de teléfonos inteligentes que sobrepasa a las ventas de computadoras personales en 2011[2] y ventas de tabletas que representarán cerca del 20% de todas las ventas de computadoras personales para 2015.[3] Para 2014, se espera que el mercado de las aplicaciones de teléfonos inteligentes exceda los $40 mil millones,[4] lo cual generará oportunidades importantes para la programación de aplicaciones móviles.

Ediciones de Java: SE, EE y ME

Java ha evolucionado con tanta rapidez que esta novena edición de *Cómo programar en Java* —basada en **Java Standard Edition 6 (Java SE 6)** con módulos opcionales sobre las nuevas características de **Java SE 7** —se publicó sólo 15 años después de la primera edición. Java se utiliza en un espectro tan amplio de aplicaciones, que cuenta con otras dos ediciones. **Java Enterprise Edition (Java EE)** está orientada hacia el desarrollo de aplicaciones de red distribuidas, de gran escala, y aplicaciones basadada en Web.

1 www.circleid.com/posts/mobile_internet_users_to_reach_134_million by_2013/.
2 www.pcworld.com/article/171380/more smartphones_than_desktop_pcs_by_2011.html.
3 www.forrester.com/ER/Press/Release/0,1769,1340,00.html.
4 *Inc.* diciembre de 2010/enero de 2011, páginas 116-123.

En el pasado, la mayoría de las aplicaciones de computadora se ejecutaban en computadoras "independientes" (que no estaban conectadas en red). En la actualidad, las aplicaciones se pueden escribir con miras a comunicarse entre computadoras en todo el mundo por medio de Internet y Web. Más adelante en este libro hablaremos sobre cómo crear dichas aplicaciones basadas en Web con Java.

Java Micro Edition (Java ME) está orientada hacia el desarrollo de aplicaciones para pequeños dispositivos con memoria restringida, como los teléfonos inteligentes BlackBerry. El sistema operativo Android de Google —que se utiliza en muchos teléfonos inteligentes, tabletas (pequeñas computadoras ligeras y móviles con pantallas táctiles), lectores electrónicos y otros dispositivos— utiliza una versión personalizada de Java que no se basa en Java ME.

La computación en la industria y la investigación

Éstos son tiempos emocionantes en el campo de la computación. Muchas de las empresas más influyentes y exitosas de las últimas dos décadas son compañías de tecnología, como Apple, IBM, Hewlett Packard, Dell, Intel, Motorola, Cisco, Microsoft, Google, Amazon, Facebook, Twitter, Groupon, Foursquare, Yahoo!, eBay y muchas más; que son fuentes de empleo importantes para las personas que estudian ciencias computacionales, sistemas de información o disciplinas relacionadas. Al momento de escribir este libro, Apple era la segunda compañía más valiosa del mundo, *con* la tecnología más preciada.[5] Las computadoras también se utilizan mucho en la investigación académica e industrial. La figura 1.1 provee unos cuantos ejemplos de las increíbles formas en que se utilizan las computadoras, tanto en la investigación como en la industria.

Nombre	Descripción
Internet	Internet —una red global de computadoras— se hizo posible gracias a la *convergencia de la computación y las comunicaciones*. Tiene sus raíces en la década de 1960; su patrocinio estuvo a cargo del Departamento de Defensa de Estados Unidos. Diseñada en un principio para conectar los sistemas de cómputo principales de alrededor de una docena de universidades y organizaciones de investigación, en la actualidad son miles de millones de computadoras y dispositivos controlados por computadora en todo el mundo los que utilizan Internet. Las computadoras descomponen las extensas transmisiones en paquetes en el extremo emisor, envían los paquetes a los receptores destinados y aseguran que se reciban en secuencia y sin errores en el extremo receptor. De acuerdo con un estudio de Forrester Research, el consumidor estadounidense promedio invierte en la actualidad la misma cantidad de tiempo en línea que el que pasa en la televisión (`forrester.com/rb/Research/understanding_changing_needs_of_us_online_consumer/g/id/57861/t/2`).
Proyecto Genoma Humano	El Proyecto Genoma Humano se fundó para identificar y analizar los más de 20,000 genes en el ADN humano. El proyecto utilizó programas de computadora para analizar datos genéticos complejos, determinar las secuencias de los miles de millones de pares de bases químicas que componen el ADN humano y almacenar la información en bases de datos que se han puesto a disposición de los investigadores en muchos campos. Esta investigación ha ocasionado una tremenda innovación y crecimiento en la industria de la biotecnología.

Fig. 1.1 | Unos cuantos usos para las computadoras (parte 1 de 3).

5 `www.zdnet.com/blog/apple/apple-becomes-worlds-second-most-valuable-company/9047`.

Nombre	Descripción
Red de la Comunidad Mundial	La Red de la Comunidad Mundial (`www.worldcommunitygrid.org`) es una red de computación sin fines de lucro. Las personas de todo el mundo donan el poder de procesamiento de cómputo que no utilicen, mediante la instalación de un programa de software seguro gratuito que permite a la Red de la Comunidad Mundial aprovechar ese poder sobrante cuando las computadoras están inactivas. El poder de cómputo se utiliza en lugar de las supercomputadoras para realizar proyectos de investigación científicos que están haciendo la diferencia, entre ellos: el desarrollo de energía solar a un precio asequible, el suministro de agua potable al mundo en desarrollo, la lucha contra el cáncer, la cura de la distrofia muscular, el hallazgo de medicamentos antivirales contra la influenza, el cultivo de arroz más nutritivo para las regiones que combaten la hambruna y otros más.
Imágenes para diagnóstico médico	Las exploraciones por tomografía computarizada (CT) con rayos X, también conocidas como CAT (tomografía axial computarizada), toman rayos X del cuerpo desde cientos de ángulos distintos. Se utilizan computadoras para ajustar la intensidad del rayo X, con lo cual se optimiza la exploración para cada tipo de tejido, para después combinar toda la información y crear una imagen tridimensional (3D).
GPS	Los dispositivos con Sistema de posicionamiento global (GPS) utilizan una red de satélites para obtener información con base en la ubicación. Varios satélites envían señales con etiquetas de tiempo al dispositivo GPS, el cual calcula la distancia hacia cada satélite con base en la hora en que la señal salió del satélite y se recibió. La ubicación de cada satélite y la distancia hacia cada uno de ellos se utilizan para determinar la ubicación exacta del dispositivo. Según la ubicación en la que usted se encuentre, los dispositivos GPS pueden proveer indicaciones paso a paso, ayudarle a encontrar con facilidad negocios cercanos (restaurantes, gasolineras, etcétera) y puntos de interés, o ayudarle a encontrar a sus amigos.
SYNC® de Microsoft	Ahora muchos autos Ford cuentan con la tecnología SYNC de Microsoft, la cual provee capacidades de reconocimiento y síntesis de voz (para leer mensajes de texto a los pasajeros) que le permiten usar comandos de voz para explorar música, solicitar alertas de tráfico y otras cosas más.
AMBER™ Alert	El Sistema de alerta AMBER (Desaparecidos en América: Sistema de Transmisión de Respuesta a Emergencias) se utiliza para buscar niños secuestrados. Las autoridades notifican tanto a las difusoras de TV y radio como a los funcionarios de carreteras estatales, quienes a su vez transmiten alertas en TV, radio, señales computarizadas en las carreteras, Internet y los dispositivos inalámbricos. AMBER Alert se asoció recientemente con Facebook. Cuyos usuarios pueden hacer clic en "Like" ("Me gusta") en las páginas de AMBER Alert según la ubicación, para recibir alertas en sus transmisiones de noticias.
Robots	Los robots son máquinas computarizadas que pueden realizar tareas (como; trabajos físicos), responder a los estímulos y otras cosas más. Se pueden utilizar para tareas diarias (por ejemplo, la aspiradora Roomba de iRobot), de entretenimiento (como las mascotas robóticas), combate militar, exploración espacial y en la profundidad del océano, manufactura y otras más. En 2004, el trotamundos marciano de la NASA a control remoto –que utilizaba tecnología Java—exploró la superficie para aprender sobre la historia del agua en el planeta.

Fig. 1.1 | Unos cuantos usos para las computadoras (parte 2 de 3).

Nombre	Descripción
Una laptop por niño (OLPC)	Una Laptop Por Niño (OLPC) ofrece laptops económicas, habilitadas para Internet y de bajo consumo de energía para los niños pobres en todo el mundo; gracias a ello fomentan el aprendizaje y reducen la separación digital (`one.laptop.org`). Al proveer estos recursos educativos, OLPC aumenta las oportunidades de que los niños pobres aprendan y hagan la diferencia en sus comunidades.
Programación de juegos	El negocio de los juegos de computadora es más grande que el de las películas de estreno. El desarrollo de los videojuegos más sofisticados puede costar hasta $100 millones. El juego *Call of Duty 2: Modern Warfare* de Activision, lanzado al público en noviembre de 2009, obtuvo $310 millones en sólo un día en Norteamérica y el Reino Unido (`news.cnet.com/8301-13772_3-10396593-52.html?tag=mncol;txt`)! Los *juegos sociales* en línea, que permiten a usuarios de todo el mundo competir entre sí, están creciendo con rapidez. Zynga —creador de juegos en línea populares, como *Farmville* y *Mafia Wars*— se fundó en 2007 y ya cuenta con más de 215 millones de usuarios mensuales. Para dar cabida al aumento en el tráfico, ¡Zynga agrega casi 1,000 servidores por semana (`techcrunch.com/2010/09/22/zynga-moves-1-petabyte-of-data-daily-adds-1000-servers-a-week/`)! Las consolas de videojuegos también se están volviendo cada vez más sofisticadas. El control remoto del Wii utiliza un *acelerómetro* (para detectar la inclinación y la aceleración) junto con un sensor que determina hacia dónde apunta el dispositivo, lo cual le permite responder al movimiento. Al hacer ademanes con el control remoto del Wii en la mano, usted puede controlar el videojuego en la pantalla. Con Kinect para el Xbox 360 de Microsoft, usted —el jugador— se convierte en el controlador. Kinect utiliza una cámara, un sensor de profundidad y software sofisticado para seguir el movimiento de su cuerpo, lo cual le permite controlar el juego (`en.wikipedia.org/wiki/Kinect`). Con los juegos de Kinect puede bailar, hacer ejercicio, jugar deportes, entrenar animales virtuales y varias actividades más.
TV por Internet	Los receptores de TV por Internet (como Apple TV y Google TV) le dan acceso a diversos tipos de contenido —como juegos, noticias, películas, programas de televisión y más—, con lo cual usted puede acceder a una gran cantidad de contenido bajo demanda; ya no necesita depender de los proveedores de televisión por cable o vía satélite para recibir contenido.

Fig. 1.1 | Unos cuantos usos para las computadoras (parte 3 de 3).

1.2 Computadoras: hardware y software

Una computadora es un dispositivo capaz de realizar cálculos y tomar decisiones lógicas con una rapidez increíblemente mayor que los humanos. Muchas de las computadoras personales contemporáneas pueden realizar miles de millones de cálculos en un segundo —más de lo que un humano podría realizar en toda su vida. ¡*Las supercomputadoras* ya pueden realizar *miles de billones* de instrucciones por segundo! Dicho de otra forma, una computadora de mil billones de instrucciones por segundo puede realizar en un segundo más de 100,000 cálculos *¡para cada uno de los habitantes del planeta!* ¡Y estos "límites superiores" están aumentando con rapidez!

Las computadoras procesan datos bajo el control de conjuntos de instrucciones conocidas como **programas de computadora**. Los cuales guían a la computadora a través de conjuntos ordenados de acciones especificadas por gente conocida como **programadores** de computadoras. A los programas que se ejecutan en una computadora se les denomina **software**. En este libro aprenderá la metodología de programación clave de la actualidad que mejora la productividad del programador, con lo cual se reducen los costos de desarrollo del software: *programación orientada a objetos*.

Una computadora consiste en varios dispositivos conocidos como **hardware** (teclado, pantalla, ratón, discos duros, memoria, unidades de DVD y unidades de procesamiento). Los costos de las computadoras *han disminuido en forma espectacular*, debido a los rápidos desarrollos en las tecnologías de hardware y software. Las computadoras que ocupaban grandes habitaciones y que costaban millones de dólares hace algunas décadas, ahora pueden colocarse en las superficies de chips de silicio más pequeños que una uña, y con un costo de quizá unos cuantos dólares cada uno. Aunque suene irónico, el silicio es uno de los materiales más abundantes en el planeta (es uno de los ingredientes de la arena común). La tecnología de los chips de silicio ha vuelto tan económica a la tecnología de la computación que hay más de mil millones de computadoras de uso general funcionando a nivel mundial, y se espera que esta cifra se *duplique* en los próximos años.

Los chips de computadora (*microprocesadores*) controlan innumerables dispositivos. Entre estos **sistemas incrustados** están: frenos antibloqueo en los autos, sistemas de navegación, electrodomésticos inteligentes, sistemas de seguridad en el hogar, teléfonos celulares y teléfonos inteligentes, robots, intersecciones de tráfico inteligentes (collision avoidance systems), controles de videojuegos y más. La gran mayoría de los microprocesadores que se producen cada año están incrustados en dispositivos que no son computadoras de propósito general.[6]

Ley de Moore

Es probable que cada año, espere pagar por lo menos un poco más por la mayoría de los productos y servicios. En el caso de los campos de las computadoras y las comunicaciones se ha dado lo opuesto, en especial con relación a los costos del hardware que da soporte a estas tecnologías. Los costos del hardware han disminuido con rapidez durante varias décadas. Cada uno o dos años, las capacidades de las computadoras se *duplican* aproximadamente sin que el precio se incremente. Esta notable observación se conoce en el ámbito común como la **Ley de Moore**, y debe su nombre a la persona que identificó esta tendencia: Gordon Moore, cofundador de Intel, uno de los principales fabricantes de procesadores para las computadoras y los sistemas incrustados de la actualidad. La Ley de Moore y las observaciones relacionadas son especialmente ciertas en cuanto a la cantidad de memoria que tienen las computadoras para los programas, la cantidad de almacenamiento secundario (como el almacenamiento en disco) que tienen para guardar los programas y datos durante periodos extendidos de tiempo, y las velocidades de sus procesadores: las velocidades con que las computadoras ejecutan sus programas (realizan su trabajo). Se ha producido un crecimiento similar en el campo de las comunicaciones, en donde los costos se han desplomado a medida que la enorme demanda por el ancho de banda de las comunicaciones (la capacidad de transmisión de información) atrae una competencia intensa. No conocemos otros cambios en los que la tecnología mejore con tanta rapidez y los costos disminuyan de una manera tan drástica. Dicha mejora fenomenal está fomentando sin duda la *Revolución de la información*.

1.3 Jerarquía de datos

Los elementos de datos que procesan las computadoras forman una **jerarquía de datos** que se vuelve cada vez más grande y compleja en estructura, a medida que progresamos primero a bits, luego a caracteres, después a campos y así en lo sucesivo. La figura 1.2 ilustra una porción de la jerarquía de datos. La figura 1.3 sintetiza los niveles de la jerarquía de datos.

---

6 www.eetimes.com/electronics-blogs/industrial-control-designline-blog/4027479/Real-men-program-in-C?pageNumber=1.

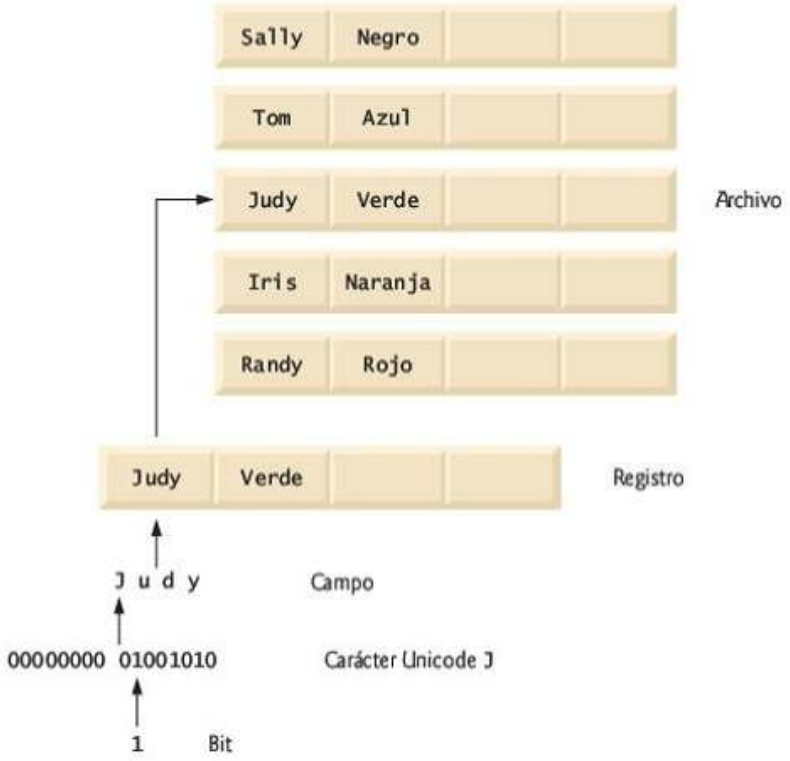


Fig. 1.2 | Jerarquía de datos.

Nivel	Descripción
Bits	El elemento de datos más pequeño en una computadora puede asumir el valor 0 o el valor 1. A dicho elemento de datos se le denomina **bit** (abreviación de "dígito binario": un dígito que puede asumir uno de dos valores). Es notable que las impresionantes funciones que realizan las computadoras sólo impliquen las manipulaciones más simples de 0s y 1s: *examinar el valor de un bit, establecer el valor de un bit e invertir el valor de un bit* (de 1 a 0 o de 0 a 1).
Caracteres	Es tedioso para las personas trabajar con datos en el formato de bajo nivel de los bits. En cambio, prefieren trabajar con *dígitos decimales* (0-9), *letras* (A-Z y a-z) y *símbolos especiales* (por ejemplo, $, @, %, &, *, (,), -, +, ", ? y /). Los dígitos, letras y símbolos especiales se conocen como **caracteres**. El **conjunto de caracteres** de la computadora es el conjunto de todos los que se utilizan para escribir programas y representar elementos de datos. Las computadoras sólo procesan los 1 y los 0, por lo que el conjunto de caracteres de una computadora representa a cada uno como un patrón de los 1 y los 0. Java usa caracteres **Unicode®** que están compuestos de dos **bytes**, cada uno de los cuales está formado a su vez de ocho bits. Unicode contiene caracteres para muchos de los idiomas en el mundo. En el apéndice L obtendrá más información sobre Unicode. En el apéndice B conocerá más información sobre el conjunto de caracteres **ASCII (Código estándar estadounidense para el intercambio de información)**: el popular subconjunto de Unicode que representa las letras mayúsculas y minúsculas, los dígitos y algunos caracteres especiales comunes.

Fig. 1.3 | Niveles de la jerarquía de datos (parte 1 de 2).

Nivel	Descripción
Campos	Así como los caracteres están compuestos de bits, los **campos** lo están por caracteres o bytes. Un campo es un grupo de caracteres o bytes que transmiten un significado. Por ejemplo, un campo compuesto de letras mayúsculas y minúsculas se puede usar para representar el nombre de una persona, y uno compuesto de dígitos decimales podría representar su edad.
Registros	Se pueden usar varios campos relacionados para componer un **registro** (el cual se implementa como una `clase` en Java). Por ejemplo, en un sistema de nómina, el registro de un empleado podría consistir en los siguientes campos (los posibles tipos para éstos se muestran entre paréntesis): • Número de identificación del empleado (un número entero) • Nombre (una cadena de caracteres) • Dirección (una cadena de caracteres) • Salario por horas (un número con punto decimal) • Ingresos del año a la fecha (un número con punto decimal) • Monto de impuestos retenidos (un número con punto decimal) Así, un registro es un grupo de campos relacionados. En el ejemplo anterior, todos los campos pertenecen al mismo empleado. Una compañía podría tener muchos empleados y un registro de nómina para cada uno.
Archivos	Un **archivo** es un grupo de registros relacionados. [*Nota:* Dicho en forma más general, un archivo contiene datos arbitrarios en formatos arbitrarios. En algunos sistemas operativos, un archivo se ve tan sólo como una *secuencia de bytes*: cualquier organización de esos bytes en un archivo, como cuando se organizan los datos en registros, es una vista creada por el programador de la aplicación]. Es muy común que una organización tenga muchos archivos, algunos de los cuales pueden contener miles de millones, o incluso billones de caracteres de información.

Fig. 1.3 | Niveles de la jerarquía de datos (parte 2 de 2).

1.4 Organización de una computadora

Sin importar las diferencias en la apariencia física, es posible percibir a las computadoras como si estuvieran divididas en varias **unidades lógicas** o secciones (figura 1.4).

Unidad lógica	Descripción
Unidad de entrada	Esta sección "receptora" obtiene información (datos y programas de cómputo) de los **dispositivos de entrada** y la pone a disposición de las otras unidades para que pueda procesarse. La mayor parte de la información se introduce a través de los teclados, pantallas táctiles y ratones. La información también puede introducirse de muchas otras formas, como hablar con su computadora, digitalizar imágenes y códigos de barras, leer dispositivos de almacenamiento secundario (como discos duros, unidades de DVD, Blu-ray Disc™ y Flash USB —también conocidas como "unidades de pulgar" o "memory sticks"), recibir video de una cámara Web e información en su computadora a través de Internet (como cuando descarga videos de YouTube™ o libros electrónicos de Amazon). Las formas más recientes de entrada son: leer los datos de la posición a través de un dispositivo GPS, y la información sobre el movimiento y la orientación mediante un acelerómetro en un teléfono inteligente o un controlador de juegos.

Fig. 1.4 | Unidades lógicas de una computadora (parte 1 de 2).

Unidad lógica	Descripción
Unidad de salida	Esta sección de "embarque" toma información que ya ha sido procesada por la computadora y la coloca en los diferentes **dispositivos de salida**, para que esté disponible fuera de la computadora. En la actualidad, la mayor parte de la información de salida de las computadoras se despliega en pantallas, se imprime en papel, se reproduce como audio o video en reproductores de medios portátiles (como los populares iPod de Apple) y pantallas gigantes en estadios deportivos, se transmite a través de Internet o se utiliza para controlar otros dispositivos, como robots y aparatos "inteligentes".
Unidad de memoria	Esta sección de "almacén" de acceso rápido, pero con relativa baja capacidad, retiene la información que se introduce a través de la unidad de entrada, para que esté disponible de manera inmediata y se pueda procesar cuando sea necesario. La unidad de memoria también retiene la información procesada hasta que la unidad de salida pueda colocarla en los dispositivos de salida. La información en la unidad de memoria es *volátil*: por lo general se pierde cuando se apaga la computadora. Con frecuencia, a la unidad de memoria se le conoce como **memoria** o **memoria principal**. Las típicas memorias principales en las computadoras de escritorio y portátiles contienen entre 1 GB y 8 GB (GB se refiere a gigabytes; un gigabyte equivale aproximadamente a mil millones de bytes).
Unidad aritmética y lógica (ALU)	Esta sección de "manufactura" realiza *cálculos* como suma, resta, multiplicación y división. También contiene los mecanismos de *decisión* que permiten a la computadora hacer cosas como, por ejemplo, comparar dos elementos de la unidad de memoria para determinar si son iguales o no. En los sistemas actuales, la ALU se implementa por lo general como parte de la siguiente unidad lógica, la CPU.
Unidad central de procesamiento (CPU)	Esta sección "administrativa" coordina y supervisa la operación de las demás. La CPU le indica a la unidad de entrada cuándo debe grabarse la información dentro de la unidad de memoria, a la ALU cuándo debe utilizarse la información de la unidad de memoria para los cálculos, y a la unidad de salida cuándo enviar la información desde la unidad de memoria hasta ciertos dispositivos de salida. Muchas de las computadoras actuales contienen múltiples CPU y, por lo tanto, pueden realizar muchas operaciones de manera simultánea. Un **procesador multinúcleo** implementa varios procesadores en un solo chip de circuitos integrados; un *procesador de doble núcleo (dual-core)* tiene dos CPU y un *procesador de cuádruple núcleo (quad-core)* tiene cuatro CPU. Las computadoras de escritorio de la actualidad tienen procesadores que pueden ejecutar miles de millones de instrucciones por segundo.
Unidad de almacenamiento secundario	Ésta es la sección de "almacén" de alta capacidad y de larga duración. Los programas o datos que no utilizan las demás unidades con frecuencia se colocan en dispositivos de almacenamiento secundario (por ejemplo, el *disco duro*) hasta que se requieran de nuevo, lo cual puede ser cuestión de horas, días, meses o incluso años después. La información en los dispositivos de almacenamiento secundario es *persistente*: se conserva aún y cuando se apaga la computadora. El tiempo para acceder a la información en almacenamiento secundario es mucho mayor que el necesario para acceder a la de la memoria principal, pero el costo por unidad de memoria secundaria es mucho menor que el correspondiente a la unidad de memoria principal. Las unidades de CD, DVD y Flash USB son ejemplos de dispositivos de almacenamiento secundario, los cuales pueden contener hasta 128 GB. Los discos duros típicos en las computadoras de escritorio y portátiles pueden contener hasta 2 TB (TB se refiere a terabytes; un terabyte equivale aproximadamente a un billón de bytes).

Fig. 1.4 | Unidades lógicas de una computadora (parte 2 de 2).

1.5 Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel

Los programadores escriben instrucciones en diversos lenguajes de programación, algunos de los cuales los comprende directamente la computadora, mientras que otros requieren pasos intermedios de *traducción*. En la actualidad se utilizan cientos de lenguajes de computación. Éstos se dividen en tres tipos generales:

1. Lenguajes máquina
2. Lenguajes ensambladores
3. Lenguajes de alto nivel

Cualquier computadora puede entender de manera directa sólo su propio **lenguaje máquina**, el cual se define según su diseño de hardware. Por lo general, los lenguajes máquina consisten en cadenas de números (que finalmente se reducen a los 1 y 0) que instruyen a las computadoras para realizar sus operaciones más elementales, una a la vez. Los lenguajes máquina son *dependientes de la máquina* (es decir, un lenguaje máquina en particular puede usarse sólo en un tipo de computadora). Dichos lenguajes son difíciles de comprender para los humanos. Por ejemplo, he aquí la sección de uno de los primeros programas en lenguaje máquina, el cual suma el pago de las horas extras al sueldo base y almacena el resultado en el sueldo bruto:

```
+1300042774
+1400593419
+1200274027
```

La programación en lenguaje máquina era demasiado lenta y tediosa para la mayoría de los programadores. En vez de utilizar las cadenas de números que las computadoras podían entender de manera directa, los programadores empezaron a utilizar abreviaturas del inglés para representar las operaciones elementales. Estas abreviaturas formaron la base de los **lenguajes ensambladores**. Se desarrollaron *programas traductores* conocidos como **ensambladores** para convertir los primeros programas en lenguaje ensamblador a lenguaje máquina, a la velocidad de la computadora. La siguiente sección de un programa en lenguaje ensamblador también suma el pago de las horas extras al sueldo base y almacena el resultado en el sueldo bruto:

```
load    sueldobase
add     sueldoextra
store   sueldobruto
```

Aunque este código es más claro para los humanos, las computadoras no lo pueden entender sino hasta que se traduce en lenguaje máquina.

El uso de las computadoras se incrementó rápidamente con la llegada de los lenguajes ensambladores, pero los programadores aún requerían de muchas instrucciones para llevar a cabo incluso hasta las tareas más simples. Para agilizar el proceso de programación se desarrollaron los **lenguajes de alto nivel**, en donde podían escribirse instrucciones individuales para realizar tareas importantes. Los programas traductores, denominados **compiladores**, convierten programas en lenguaje de alto nivel a lenguaje máquina. Los lenguajes de alto nivel permiten a los programadores escribir instrucciones que son muy similares al inglés común, y contienen la notación matemática común. Un programa de nómina escrito en un lenguaje de alto nivel podría contener *una* instrucción como la siguiente:

```
sueldoBruto = sueldoBase + sueldoExtra
```

Desde el punto de vista del programador, los lenguajes de alto nivel son mucho más recomendables que los lenguajes máquina o ensamblador. Java es, por mucho, el lenguaje de alto nivel más utilizado.

El proceso de compilación de un programa escrito en lenguaje de alto nivel a un lenguaje máquina puede tardar un tiempo considerable en la computadora. Los programas *intérpretes* se desarrollaron para ejecutar programas en lenguaje de alto nivel de manera directa (sin el retraso de la compilación), aunque con más lentitud de la que se ejecutan en los programas compilados. Hablaremos más sobre la forma en que trabajan los intérpretes en la sección 1.9, en donde aprenderá que Java utiliza una astuta mezcla de compilación e interpretación, optimizada con base en el rendimiento, para ejecutar los programas. Los ejercicios 7.35-7.37 (en la Sección especial: Cree su propia computadora) le guiarán a través del proceso de creación de un programa intérprete.

1.6 Introducción a la tecnología de los objetos

Crear software en forma rápida, correcta y económica sigue siendo un objetivo difícil de alcanzar en una época en que la demanda de software nuevo y más poderoso va en aumento. Los *objetos*, o dicho en forma más precisa —como veremos en el capítulo 3— las *clases* de las que provienen los objetos, son en esencia componentes de software *reutilizables*. Existen objetos de fecha, objetos de hora, objetos de audio, objetos de video, objetos de automóviles, objetos de personas, etcétera. Casi *cualquier* sustantivo se puede representar de manera razonable como un objeto de software en términos de sus *atributos* (como el nombre, color y tamaño) y *comportamientos* (por ejemplo, calcular, moverse y comunicarse). Los desarrolladores de software han descubierto que al usar una metodología de diseño e implementación orientada a objetos y modular, pueden crea grupos de desarrollo de software más productivos de lo que era posible con las técnicas populares anteriores, como la "programación estructurada"; por lo general los programas orientados a objetos son más fáciles de comprender, corregir y modificar.

El automóvil como un objeto

Para ayudarle a comprender los objetos y su contenido, empecemos con una analogía simple. Suponga que desea *conducir un auto y hacer que vaya más rápido al oprimir el pedal del acelerador*. ¿Qué debe ocurrir para que usted pueda hacer esto? Bueno, antes de que pueda conducir un auto, alguien tiene que *diseñarlo*. Por lo general, un auto empieza en forma de dibujos de ingeniería, similares a los *planos de construcción* que describen el diseño de una casa. Estos dibujos de ingeniería incluyen el diseño del pedal del acelerador. El pedal *oculta* los complejos mecanismos que se encargan de que el auto aumente su velocidad, de igual forma que el pedal del freno *oculta* los mecanismos que disminuyen la velocidad del auto y el volante "oculta" los mecanismos que hacen que el auto de vuelta. Esto permite que las personas con poco o nada de conocimiento acerca de cómo funcionan los motores, los frenos y los mecanismos de la dirección puedan conducir un auto con facilidad.

Por desgracia, así como no es posible cocinar en la cocina de un plano de construcción, tampoco es posible conducir los dibujos de ingeniería de un auto. Antes de poder conducir un auto, éste debe *construirse* a partir de los dibujos de ingeniería que lo describen. Un auto completo tendrá un pedal acelerador *verdadero* para hacer que aumente su velocidad, pero aún así no es suficiente; el auto no acelerará por su propia cuenta (¡esperemos que así sea!), así que el conductor debe *oprimir* el pedal del acelerador para aumentar la velocidad del auto.

Métodos y clases

Ahora vamos a utilizar nuestro ejemplo del auto para introducir algunos conceptos clave de la programación orientada a objetos. Para realizar una tarea en una aplicación se requiere un **método**, el cual aloja las instrucciones del programa que se encargan de realizar sus tareas. El método oculta al usuario estas tareas, de la misma forma que el pedal del acelerador de un auto oculta al conductor los mecanismos para hacer que el auto vaya más rápido. En Java, creamos una unidad de programa llamada **clase** para alojar el conjunto de métodos que realizan las tareas de esa clase. Por ejemplo, una

clase que representa a una cuenta bancaria podría contener un método para *depositar* dinero en una cuenta, otro para *retirar* y un tercero para *solicitar* el saldo actual. Una clase es similar en concepto a los dibujos de ingeniería de un auto, que contienen el diseño de un pedal acelerador, volante de dirección, etcétera.

Instanciamiento

Así como alguien tiene que *construir un auto* a partir de sus dibujos de ingeniería para que otra persona lo pueda conducir después, también es necesario *crear un objeto* de una clase para que un programa pueda realizar las tareas definidas por los métodos de esa clase. Al proceso de hacer esto se le denomina *instanciamiento*. Entonces, un objeto viene siendo una **instancia** de su clase.

Reutilización

Así como los dibujos de ingeniería de un auto se pueden *reutilizar* muchas veces para construir muchos autos, también es posible *reutilizar* una clase muchas veces para crear muchos objetos. Al reutilizar las clases existentes para crear nuevas clases y programas, ahorramos tiempo y esfuerzo. La reutilización también nos ayuda a crear sistemas más confiables y efectivos, debido a que con frecuencia las clases y los componentes existentes pasan por un extenso proceso de *prueba*, *depuración* y optimización del *desempeño*. De la misma manera en que la noción de *piezas intercambiables* fue crucial para la Revolución Industrial, las clases reutilizables son cruciales para la revolución de software incitada por la tecnología de objetos.

Observación de ingeniería de software 1.1

Use un método de construcción en bloques para crear programas. Evite reinventar la rueda: use piezas existentes siempre que sea posible. Esta reutilización de software es un beneficio clave de la programación orientada a objetos.

Mensajes y llamadas a métodos

Cuando conduce un auto, al oprimir el pedal del acelerador envía un *mensaje* al auto para que realice una tarea: aumentar la velocidad. De manera similar, es posible *enviar mensajes a un objeto*. Cada mensaje se implementa como **llamada a método**, para indicar a un método del objeto que realice su tarea. Por ejemplo, un programa podría llamar al método *depositar* de un objeto cuenta de banco específico para aumentar el saldo de esa cuenta.

Atributos y variables de instancia

Además de tener capacidades para realizar tareas, un auto también tiene *atributos*: color, número de puertas, cantidad de gasolina en el tanque, velocidad actual y registro del total de kilómetros recorridos (es decir, la lectura de su velocímetro). Al igual que sus capacidades, los atributos del auto se representan como parte de su diseño en sus diagramas de ingeniería (que, por ejemplo, agregan un velocímetro y un indicador de combustible). Al conducir un auto real, estos atributos van incluidos. Cada auto conserva sus *propios* atributos. Por ejemplo, cada uno sabe cuánta gasolina hay en su tanque, pero *no* cuánta hay en los tanques de *otros* autos.

De manera similar, un objeto tiene atributos que lleva consigo a medida que se utiliza en un programa. Estos atributos se especifican como parte de la clase del objeto. Por ejemplo, un objeto cuenta bancaria tiene un *atributo saldo* que representa la cantidad de dinero en la cuenta. Cada objeto cuenta bancaria conoce el saldo de la cuenta que representa, pero *no* los saldos de las *otras* cuentas en el banco. Los atributos se especifican mediante las **variables de instancia** de la clase.

Encapsulamiento

Las clases **encapsulan** (envuelven) los atributos y métodos en objetos; los atributos y métodos de un objeto están muy relacionados entre sí. Los objetos se pueden comunicar entre sí, pero por lo general no se les permite saber cómo están implementados otros objetos; los detalles de implementación están *ocultos* dentro de los mismos objetos. Este **ocultamiento de información**, como veremos más adelante, es crucial para la buena ingeniería de software.

Herencia

Es posible crear una nueva clase de objetos con rapidez y de manera conveniente mediante la **herencia**: la nueva clase absorbe las características de una clase existente, con la posibilidad de personalizarlas y agregar características únicas propias. En nuestra analogía del auto, sin duda un objeto de la clase "convertible" *es un* objeto de la clase más *general* llamada "automóvil" pero, de manera más *específica*, el techo puede ponerse o quitarse.

Análisis y diseño orientado a objetos (A/DOO)

Pronto escribirá programas en Java. ¿Cómo creará el **código** (es decir, las instrucciones) para sus programas? Tal vez, al igual que muchos programadores, sólo encenderá su computadora y empezará a escribir. Quizás este método funcione para pequeños programas (como los que presentamos en los primeros capítulos del libro), pero ¿qué tal si le pidieran crear un sistema de software para controlar miles de cajeros automáticos para un banco importante? O ¿si le piden que trabaje con un equipo de 1,000 desarrolladores de software para crear el nuevo sistema de control de tráfico aéreo en Estados Unidos? Para proyectos tan grandes y complejos, no es conveniente tan sólo sentarse y empezar a escribir programas.

Para crear las mejores soluciones, debe seguir un proceso de **análisis** detallado para determinar los **requerimientos** de su proyecto (definir *qué* se supone que debe hacer el sistema) y desarrollar un **diseño** que los satisfaga (decidir *cómo* debe hacerlo el sistema). Lo ideal sería pasar por este proceso y revisar el diseño con cuidado (además de pedir a otros profesionales de software que lo revisen) antes de escribir cualquier código. Si este proceso implica analizar y diseñar su sistema desde un punto de vista orientado a objetos, se denomina proceso de **análisis y diseño orientado a objetos (A/DOO)**. Los lenguajes como Java son orientados a objetos. La programación en un lenguaje de este tipo, conocida como **programación orientada a objetos (POO)**, le permite implementar un diseño orientado a objetos como un sistema funcional.

El UML (Lenguaje unificado de modelado)

Aunque existen muchos procesos de A/DOO distintos, hay un solo lenguaje gráfico para comunicar los resultados de *cualquier* proceso de A/DOO que se utiliza en la mayoría de los casos. Este lenguaje, conocido como Lenguaje unificado de modelado (UML), es en la actualidad el esquema gráfico más utilizado para modelar sistemas orientados a objetos. Presentamos nuestros primeros diagramas de UML en los capítulos 3 y 4; después los utilizamos en nuestro análisis más detallado de la programación orientada a objetos en el capítulo 11. En nuestro ejemplo práctico *opcional* de ingeniería de software del ATM en los capítulos 12 y 13 presentamos un subconjunto simple de las características del UML, mientras lo guiamos por una experiencia de diseño orientada a objetos.

1.7 Sistemas operativos

Los **sistemas operativos** son sistemas de software que se encargan de hacer más conveniente el uso de las computadoras para los usuarios, desarrolladores de aplicaciones y administradores de sistemas. Los sistemas operativos proveen servicios que permiten a cada aplicación ejecutarse en forma segura, eficien-

te y *concurrente* (es decir, en paralelo) con otras aplicaciones. El software que contiene los componentes básicos del sistema operativo se denomina **kernel**. Los sistemas operativos de escritorio populares son: Linux, Windows 7 y Mac OS X. Los sistemas operativos móviles populares que se utilizan en teléfonos inteligentes y tabletas son: Android de Google, BlackBerry OS y Apple iOS (para sus dispositivos iPhone, iPad e iPod Touch).

Windows: un sistema operativo propietario

A mediados de la década de 1980 Microsoft desarrolló el **sistema operativo Windows**, el cual consiste en una interfaz gráfica de usuario creada sobre DOS: un sistema operativo de computadora personal muy popular en la época en que, para interactuar con él, los usuarios tecleaban comandos. Windows tomó prestados muchos conceptos (como los iconos, menús y ventanas) que se hicieron populares gracias a los primeros sistemas operativos Apple Macintosh, desarrollados en un principio por Xerox PARC. Windows 7 es el sistema operativo más reciente de Microsoft; algunas de sus características son; mejoras en la interfaz de usuario, un arranque más veloz, un mayor grado de refinamiento en cuanto a las características de seguridad, soporte para pantalla táctil y multitáctil, y otras más. Windows es un sistema operativo *propietario*; está bajo el control exclusivo de una compañía. Windows es por mucho el sistema operativo más utilizado en el mundo.

Linux: un sistema operativo de código fuente abierto

El sistema operativo Linux es tal vez el más grande éxito del movimiento de *código fuente abierto*. El **código fuente abierto** es un estilo de desarrollo de software que se desvía del desarrollo *propietario*, el cual predominó durante los primeros años del software. Con el desarrollo de código fuente abierto, individuos y compañías unen sus esfuerzos para desarrollar, mantener y evolucionar el software a cambio del derecho de usarlo para sus propios fines, por lo general sin costo. Por lo general el código fuente abierto es escudriñado por una audiencia mucho mayor que la del software propietario, de modo que casi siempre los errores se eliminan con más rapidez. El código fuente abierto también fomenta una mayor innovación. Sun abrió el código de su implementación del Kit de desarrollo de Java y de muchas de sus tecnologías de Java relacionadas.

Algunas organizaciones en la comunidad de código fuente abierto son: la fundación Eclipse (el Entorno integrado de desarrollo Eclipse ayuda a los programadores de Java a desarrollar software de manera conveniente), la fundación Mozilla (creadores del navegador Web Firefox), la fundación de software Apache (creadores del servidor Web Apache que se utiliza para desarrollar aplicaciones basadas en Web) y SourceForge (quien proporciona las herramientas para administrar proyectos de código fuente abierto; tiene más de 260,000 de estos proyectos en desarrollo). Las rápidas mejoras en la computación y las comunicaciones, la reducción en costos y el software de código fuente abierto han logrado que sea mucho más fácil y económico crear un negocio basado en software en la actualidad de lo que era hace unas cuantas décadas. Facebook es un gran ejemplo de ello; este sitio se inició desde un dormitorio universitario y se creó con software de código fuente abierto.[7]

El kernel de **Linux** es el núcleo del sistema operativo de código fuente abierto más popular y lleno de funcionalidades, que se distribuye en forma gratuita. Es desarrollado por un equipo de voluntarios organizados de manera informal; es popular en servidores, computadoras personales y sistemas incrustados. A diferencia de los sistemas operativos propietarios como Windows de Microsoft y Mac OS X de Apple, el código fuente de Linux (el código del programa) está disponible al público para que lo examinen y modifiquen; además se puede descargar e instalar sin costo. Como resultado, los usuarios del sistema operativo se benefician; de una comunidad de desarrolladores que depuran y mejoran el kernel de

7 `developers.facebook.com/opensource/`.

manera continua, de la ausencia de cuotas y restricciones de licencias, y de la habilidad de poder personalizar por completo el sistema operativo para cumplir necesidades específicas.

En 1991, Linus Torvalds, un estudiante de 21 años en la Universidad de Helsinki en Finlandia, empezó a desarrollar el kernel de Linux como un pasatiempo (El nombre Linux se deriva de "Linus" y "UNIX": un sistema operativo desarrollado por los Laboratorios Bell en 1969). Torvalds quería mejorar el diseño de Minix, un sistema operativo académico creado por el profesor Andrew Tanenbaum de la Vrije Universiteit en Amsterdam. El código fuente de Minix estaba disponible al público para que los profesores pudieran demostrar los conceptos básicos de la implementación de sistemas operativos a sus estudiantes.

Torvalds liberó la primera versión de Linux en 1991. La respuesta favorable condujo a la creación de una comunidad que ha continuado con el desarrollo y soporte de Linux. Los desarrolladores descargaron, probaron y modificaron el código de Linux; después enviaron correcciones de errores y retroalimentación a Torvalds, quien revisó esa información y aplicó las mejoras al código.

La liberación de Linux en 1994 integró muchas características que se encontraban por lo general en un sistema operativo maduro, con lo cual Linux se convirtió en una alternativa viable con respecto a UNIX. Las compañías de sistemas empresariales como IBM y Oracle se interesaron cada vez más en Linux, a medida que éste se volvía más estable y se extendía a nuevas plataformas.

Son varias cuestiones —el poder de mercado de Microsoft, el pequeño número de aplicaciones Linux amigables para los usuarios y la diversidad de distribuciones de Linux, tales como Red Hat Linux, Ubuntu Linux y muchas más— las que han impedido que se popularice el uso de Linux en las computadoras de escritorio. Sin embargo, este sistema operativo se ha vuelto muy popular en servidores y sistemas incrustados, como los teléfonos inteligentes basados en Android de Google.

Android

Android —el sistema operativo para dispositivos móviles y teléfonos inteligentes, cuyo crecimiento ha sido el más rápido hasta ahora— está basado en el kernel de Linux y en Java. Los programadores experimentados de Java no tienen problemas para entrar y participar en el desarrollo de aplicaciones para Android. Un beneficio de desarrollar este tipo de aplicaciones es el grado de apertura de la plataforma. El sistema operativo es gratuito y de código fuente abierto.

El sistema operativo Android fue desarrollado por Android, Inc., compañía que adquirió Google en 2005. En 2007 se formó la Alianza para los dispositivos móviles abiertos™ (OHA) —un consorcio de 34 compañías en un principio, y de 79 para el año 2010—, para continuar con el desarrollo de Android. Al mes de diciembre de 2010, ¡se activaban más de 300,000 teléfonos inteligentes con Android a diario![8] Ahora los teléfonos Android se venden más que los iPhone.[9] El sistema operativo Android se utiliza en varios teléfonos inteligentes (Motorola Droid, HTC EVO™ 4G, Samsung Vibrant™ y muchos más), dispositivos lectores electrónicos (como el Noble Nook™ de Barnes and Noble), computadoras tipo tableta (Dell Streak, Samsung Galaxy Tab y otras más), quioscos con pantallas táctiles dentro de las tiendas, autos, robots y reproductores multimedia.

Los teléfonos inteligentes Android tienen la funcionalidad de un teléfono móvil, cliente de Internet (para navegar en Web y comunicarse a través de Internet), reproductor de MP3, consola de juegos, cámara digital y demás, todo envuelto en dispositivos portátiles con *pantallas multitáctiles* a todo color —éstas pantallas le permiten controlar el dispositivo con *ademanes* en los que se requieren uno o varios toques simultáneos. Puede descargar aplicaciones de manera directa a su dispositivo Android, a través del Android Market y de otros mercados de aplicaciones. Al mes de diciembre de 2010 había cerca de 200,000 aplicaciones en el Android Market de Google.

---

8 www.pcmag.com/article2/0,2817,2374076,00.asp.

9 mashable.com/2010/08/02/android-outselling-iphone-2/.

Capítulos de desarrollo de aplicaciones Android en el sitio Web complementario

Debido al enorme interés en los dispositivos y aplicaciones basadas en Android, hemos integrado en el sitio Web complementario del libro una introducción de tres capítulos al desarrollo de aplicaciones Android, los cuales pertenecen a nuestro nuevo libro, *Android for Programmers: An App-Driven Approach*. Después de que aprenda Java, descubrirá que no es tan complicado empezar a desarrollar y ejecutar aplicaciones Android. Puede colocar sus aplicaciones en el Android Market en línea (`www.market.android.com`) y, si se vuelven populares, tal vez hasta pueda iniciar su propio negocio. Sólo recuerde: Facebook, Microsoft y Dell se iniciaron desde un dormitorio.

1.8 Lenguajes de programación

En esta sección veremos unos cuantos comentarios breves sobre varios lenguajes de programación populares (figura 1.5). En la siguiente sección veremos una introducción a Java.

Lenguaje de programación	Descripción
Fortran	Fortran (FORmula TRANslator, Traductor de fórmulas) fue desarrollado por IBM Corporation a mediados de la década de 1950 para utilizarse en aplicaciones científicas y de ingeniería que requerían cálculos matemáticos complejos. Aún se utiliza mucho y sus versiones más recientes son orientadas a objetos.
COBOL	COBOL (COmmon Business Oriented Language, Lenguaje común orientado a negocios) fue desarrollado a finales de la década de 1950 por fabricantes de computadoras, el gobierno estadounidense y usuarios de computadoras de la industria, con base en un lenguaje desarrollado por Grace Hopper, un oficial de la Marina de Estados Unidos y científico informático. COBOL aún se utiliza mucho en aplicaciones comerciales que requieren de una manipulación precisa y eficiente de grandes volúmenes de datos. Su versión más reciente soporta la programación orientada a objetos.
Pascal	Las actividades de investigación en la década de 1960 dieron como resultado la *programación estructurada*: un método disciplinado para escribir programas que sean más claros y fáciles de probar, depurar, y de modificar que los programas extensos producidos con técnicas anteriores. Uno de los resultados más tangibles de esta investigación fue el desarrollo del lenguaje de programación Pascal por el profesor Niklaus Wirth en 1971. Se diseñó para la enseñanza de la programación estructurada y fue popular en los cursos universitarios durante varias décadas.
Ada	Ada, un lenguaje basado en Pascal, se desarrolló bajo el patrocinio del Departamento de Defensa (DOD) de los Estados Unidos durante la década de 1970 y a principios de la década de 1980. El DOD quería un solo lenguaje que pudiera satisfacer la mayoría de sus necesidades. El nombre de este lenguaje basado en Pascal es en honor de Lady Ada Lovelace, hija del poeta Lord Byron. A ella se le atribuye el haber escrito el primer programa para computadoras en el mundo, a principios de la década de 1800 (para la Máquina Analítica, un dispositivo de cómputo mecánico diseñado por Charles Babbage). Su versión más reciente soporta la programación orientada a objetos.
Basic	Basic se desarrolló en la década de 1960 en el Dartmouth College, para introducir a los principiantes a la programación. Muchas de sus versiones más recientes son orientadas a objetos.
C	C fue implementado en 1972 por Dennis Ritchie en los Laboratorios Bell. En un principio se hizo muy popular como el lenguaje de desarrollo del sistema operativo UNIX. En la actualidad, la mayoría del código para los sistemas operativos de propósito general se escribe en C o C++.

Fig. 1.5 | Otros lenguajes de programación (parte 1 de 2).

Lenguaje de programación	Descripción
C++	C++, una extensión de C, fue desarrollado por Bjarne Stroustrup a principios de la década de 1980 en los Laboratorios Bell. C++ proporciona varias características que "pulen" al lenguaje C, pero lo más importante es que proporciona la capacidades de una programación orientada a objetos.
Objective-C	Objective-C es un lenguaje orientado a objetos basado en C. Se desarrolló a principios de la década de 1980 y después fue adquirido por la empresa Next, que a su vez fue comprada por Apple. Se ha convertido en el lenguaje de programación clave para el sistema operativo Mac OS X y todos los dispositivos operados por el iOS (como los dispositivos iPod, iPhone e iPad).
Visual Basic	El lenguaje Visual Basic de Microsoft se introdujo a principios de la década de 1990 para simplificar el desarrollo de aplicaciones para Microsoft Windows. Sus versiones más recientes soportan la programación orientada a objetos.
Visual C#	Los tres principales lenguajes de programación de Microsoft son Visual Basic, Visual C++ (basado en C++) y C# (basado en C++ y Java; desarrollado para integrar Internet y Web en las aplicaciones de computadora).
PHP	PHP es un lenguaje orientado a objetos de "secuencias de comandos" y "código fuente abierto" (vea la sección 1.7), el cual recibe soporte por medio de una comunidad de usuarios y desarrolladores; se utiliza en numerosos sitios Web, entre ellos Wikipedia y Facebook. PHP es independiente de la plataforma: existen implementaciones para todos los principales sistemas operativos UNIX, Linux, Mac y Windows. PHP también soporta muchas bases de datos, como MySQL.
Python	Python, otro lenguaje orientado a objetos de secuencias de comandos, se liberó al público en 1991. Fue desarrollado por Guido van Rossum del Instituto Nacional de Investigación para las Matemáticas y Ciencias Computacionales en Amsterdam (CWI); la mayor parte de Python se basa en Modula-3: un lenguaje de programación de sistemas. Python es "extensible": puede extenderse a través de clases e interfaces de programación.
JavaScript	JavaScript es el lenguaje de secuencias de comandos más utilizado en el mundo. Su principal uso es para agregar capacidad de programación a las páginas Web; por ejemplo, animaciones e interactividad con el usuario. Los principales navegadores Web cuentan con él.
Ruby on Rails	Ruby fue creado a mediados de la década de 1990 por Yukihiro Matsumoto; es un lenguaje de programación orientado a objetos de código fuente abierto, con una sintaxis simple que es similar a Python. Ruby on Rails combina el lenguaje de secuencias de comandos Ruby con el marco de trabajo de aplicaciones Web Rails, desarrollado por 37Signals. Su libro, *Getting Real* (`gettingreal.37signals.com/toc.php`), es una lectura obligatoria para los desarrolladores Web. Muchos desarrolladores de Ruby on Rails han reportado ganancias de productividad superiores a las de otros lenguajes, al utilizar aplicaciones Web que trabajan de manera intensiva con bases de datos. Ruby on Rails se utilizó para crear la interfaz de usuario de Twitter.
Scala	Scala (`www.scala-lang.org/node/273`), abreviación en inglés de "lenguaje escalable", fue diseñado por Martin Odersky, un profesor en la École Polytechnique Fédérale de Lausanne (EPFL) en Suiza. Se lanzó al público en 2003; utiliza los paradigmas de orientación a objetos y de programación funcional, y está diseñado para integrarse con Java. Si programa en Scala, podrá reducir de manera considerable la cantidad de código en sus aplicaciones. Twitter y Foursquare usan Scala.

Fig. 1.5 | Otros lenguajes de programación (parte 2 de 2).

1.9 Java y un típico entorno de desarrollo en Java

La contribución más importante a la fecha de la revolución del microprocesador es que hizo posible el desarrollo de las computadoras personales. Los microprocesadores están teniendo un profundo impacto en los dispositivos electrónicos inteligentes para uso doméstico. Al reconocer esto, Sun Microsystems patrocinó en 1991 un proyecto interno de investigación corporativa dirigido por James Gosling, que resultó en un lenguaje de programación orientado a objetos y basado en C++, al que Sun llamó Java.

Un objetivo clave de Java es poder escribir programas que se ejecuten en una gran variedad de sistemas computacionales y dispositivos controlados por computadora. A esto se le conoce algunas veces como "escribir una vez, ejecutar en cualquier parte".

La popularidad del servicio Web se intensificó en 1993; en ese entonces Sun vio el potencial de usar Java para agregar *contenido dinámico*, como interactividad y animaciones, a las páginas Web. Java generó la atención de la comunidad de negocios debido al fenomenal interés en Web. En la actualidad, Java se utiliza para desarrollar aplicaciones empresariales a gran escala, para mejorar la funcionalidad de los servidores Web (las computadoras que proporcionan el contenido que vemos en nuestros exploradores Web), para proporcionar aplicaciones para los dispositivos de uso doméstico (como teléfonos celulares, teléfonos inteligentes, receptores de televisión por Internet y mucho más) y para muchos otros propósitos. En 2009, Oracle adquirió Sun Microsystems. En la conferencia JavaOne 2010, Oracle anunció que el 97% de todas las computadoras de escritorio, tres mil millones de dispositivos portátiles y 80 millones de dispositivos de televisión ejecutan Java. En la actualidad hay cerca de 9 millones de desarrolladores de Java, en comparación con los 4.5 millones en 2005.[10] Ahora Java es el lenguaje de desarrollo de software más utilizado en todo el mundo.

Bibliotecas de clases de Java

Usted puede crear cada clase y método que necesite para formar sus programas de Java. Sin embargo, la mayoría de los programadores en Java aprovechan las ricas colecciones de clases existentes en las **bibliotecas de clases de Java**, que también se conocen como **API (Interfaces de programación de aplicaciones)** de Java.

Tip de rendimiento 1.1

Si utiliza las clases y métodos de las API de Java en vez de escribir sus propias versiones puede mejorar el rendimiento de sus programas, ya que estas clases y métodos están escritos de manera cuidadosa para funcionar con eficacia. Esta técnica también reduce el tiempo de desarrollo de los programas.

Tip de portabilidad 1.1

Aunque es más fácil escribir programas portables (programas que se puedan ejecutar en muchos tipos distintos de computadoras) en Java que en la mayoría de los otros lenguajes de programación, las diferencias entre los compiladores, las JVM y las computadoras pueden ocasionar que sea difícil lograr la portabilidad. El simple hecho de escribir programas en Java no garantiza la portabilidad.

Ahora explicaremos los pasos típicos utilizados para crear y ejecutar una aplicación en Java, mediante el uso de un entorno de desarrollo (el cual se ilustra en las figuras 1.6-1.10). Por lo general, los programas en Java pasan a través de cinco fases: edición, compilación, carga, verificación y ejecución. Hablaremos sobre estos conceptos en el contexto del Kit de desarrollo de Java SE (JDK). Puede descargar el JDK más actualizado y su documentación en `www.oracle.com/technetwork/java/javase/`

10 `jaxenter.com/how-many-java-developers-are-there-10462.html`.

`downloads/index.html`. *Lea la sección Antes de empezar este libro para asegurarse de configurar su computadora en forma apropiada para compilar y ejecutar programas en Java.* Tal vez también desee visitar el centro para principiantes de Java (New to Java Center) de Oracle en:

```
www.oracle.com/technetwork/topics/newtojava/overview/index.html
```

[*Nota*: este sitio Web proporciona las instrucciones de instalación para Windows, Linux y Mac OS X. Si no utiliza uno de estos sistemas operativos, consulte la documentación del entorno de Java de su sistema o pregunte a su instructor cómo puede realizar estas tareas con base en el sistema operativo de su computadora. Si encuentra un problema con éste o cualquier otro vínculo mencionado en este libro, visite el sitio `www.deitel.com/books/jhtp9/` para consultar la fe de erratas y notifíquenos su problema al correo electrónico `deitel@deitel.com`].

Fase 1: Creación de un programa

La fase 1 consiste en editar un archivo con un *programa de edición*, conocido comúnmente como *editor* (figura 1.6). Usted escribe un programa en Java (conocido por lo general como **código fuente**) por medio del editor, realiza las correcciones necesarias y guarda el programa en un dispositivo de almacenamiento secundario, como su disco duro. Un nombre de archivo que termina con la **extensión** `.java` indica que éste contiene código fuente en Java.

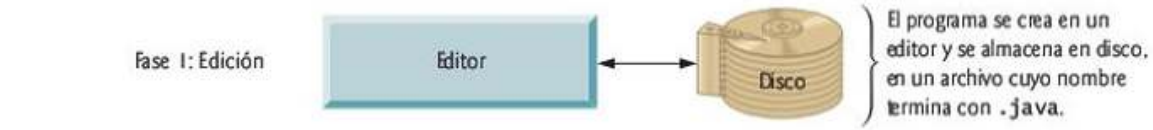


Fig. 1.6 | Entorno de desarrollo típico de Java: fase de edición.

Dos de los editores muy utilizados en sistemas Linux son `vi` y `emacs`. En Windows, basta con usar el Bloc de Notas. También hay muchos editores de freeware y shareware disponibles en línea, como EditPlus (`www.editplus.com`), TextPad (`www.textpad.com`) y jEdit (`www.jedit.org`).

Para las organizaciones que desarrollan sistemas de información extensos, hay **entornos de desarrollo integrados (IDE)** disponibles de la mayoría de los proveedores de software. Los IDE proporcionan herramientas que dan soporte al proceso de desarrollo del software, incluyendo editores para escribir y editar programas, y depuradores para localizar **errores lógicos**: errores que provocan que los programas se ejecuten en forma incorrecta. Los IDE populares son Eclipse (`www.eclipse.org`) y NetBeans (`www.netbeans.org`).

Fase 2: Compilación de un programa en Java para convertirlo en códigos de bytes

En la fase 2, el programador utiliza el comando `javac` (el **compilador de Java**) para **compilar** un programa (figura 1.7). Por ejemplo, para compilar un programa llamado `Bienvenido.java`, escriba

```
javac Bienvenido.java
```

en la ventana de comandos de su sistema (es decir, el **Símbolo del sistema** en Windows, el *indicador de shell* en Linux o la aplicación Terminal en Mac OS X). Si el programa se compila, el compilador produce un archivo **.class** llamado `Bienvenido.class` que contiene la versión compilada del programa.

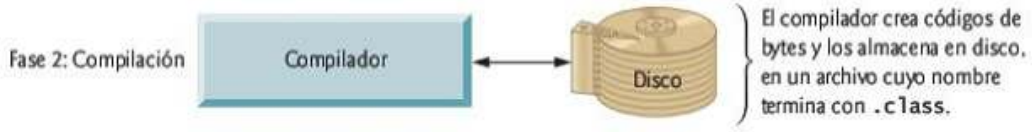


Fig. 1.7 | Entorno de desarrollo típico de Java: fase de compilación.

El compilador de Java traduce el código fuente de Java en **códigos de bytes** que representan las tareas a ejecutar en la fase de ejecución (fase 5). La **máquina virtual de Java (JVM)**, que forma parte del JDK y es la base de la plataforma Java, ejecuta los códigos de bytes. Una **máquina virtual (VM)** es una aplicación de software que simula a una computadora, pero oculta el sistema operativo y el hardware subyacentes de los programas que interactúan con ésta. Si se implementa la misma VM en muchas plataformas computacionales, las aplicaciones que ejecute se podrán utilizar en todas esas plataformas. La JVM es una de las máquinas virtuales más utilizadas en la actualidad. La plataforma NET de Microsoft utiliza una arquitectura de máquina virtual similar.

A diferencia del lenguaje máquina, que depende del hardware de una computadora específica, los códigos de bytes son instrucciones independientes de la plataforma; no dependen de una plataforma de hardware en especial. Entonces, los códigos de bytes de Java son **portables**: es decir, se pueden ejecutar los mismos códigos de bytes en cualquier plataforma que contenga una JVM que comprenda la versión de Java en la que se compilaron los códigos de bytes sin necesidad de volver a compilar el código fuente. La JVM se invoca mediante el comando `java`. Por ejemplo, para ejecutar una aplicación en Java llamada `Bienvenido`, debe escribir el comando

```
java Bienvenido
```

en una ventana de comandos para invocar la JVM, que a su vez inicia los pasos necesarios para ejecutar la aplicación. Esto comienza la fase 3.

Fase 3: Cargar un programa en memoria

En la fase 3, la JVM coloca el programa en memoria para ejecutarlo; a esto se le conoce como **cargar** (figura 1.8). El **cargador de clases** toma los archivos `.class` que contienen los códigos de bytes del programa y los transfiere a la memoria principal. El cargador de clases también carga cualquiera de los archivos `.class` que su programa utilice, y que sean proporcionados por Java: Puede cargar los archivos `.class` desde un disco en su sistema o a través de una red (como la de su universidad local o la red de la empresa, o incluso desde Internet).

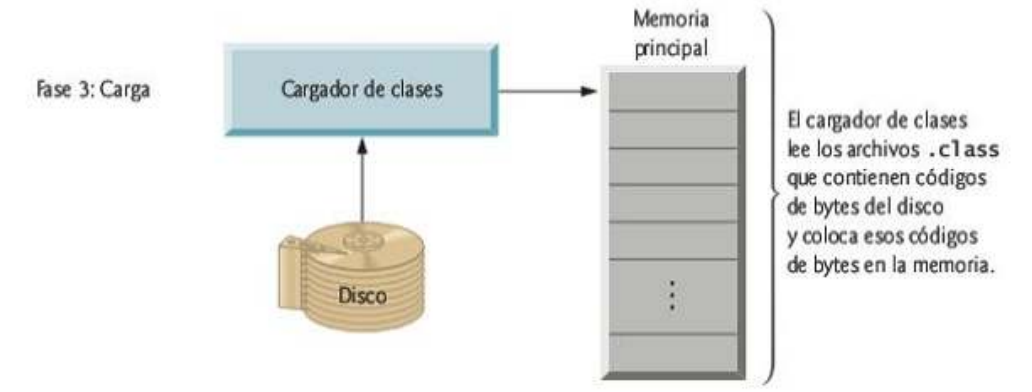


Fig. 1.8 | Entorno de desarrollo típico de Java: fase de carga.

Fase 4: Verificación del código de bytes

En la fase 4, a medida que se cargan las clases, el **verificador de códigos de bytes** examina sus códigos de bytes para asegurar que sean válidos y que no violen las restricciones de seguridad de Java (figura 1.9). Java implementa una estrecha seguridad para asegurar que los programas en Java que llegan a través de la red no dañen sus archivos o su sistema (como podrían hacerlo los virus de computadora y los gusanos).

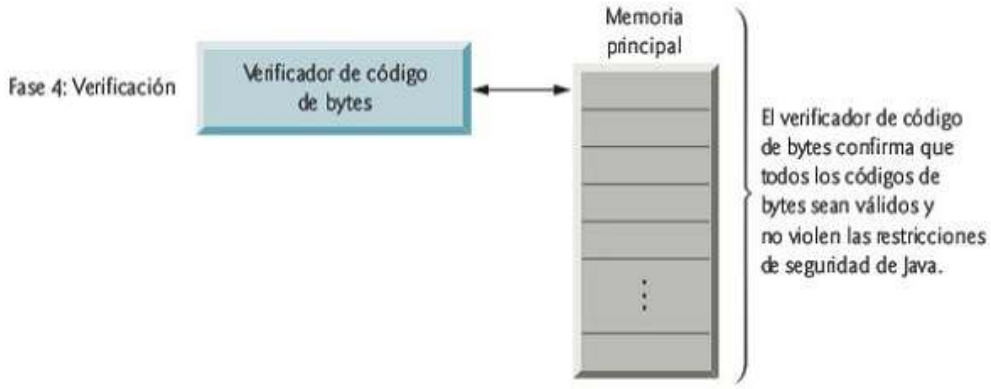


Fig. 1.9 | Entorno de desarrollo típico de Java: fase de verificación.

Fase 5: Ejecución

En la fase 5, la JVM **ejecuta** los códigos de bytes del programa, realizando así las acciones especificadas por el mismo (figura 1.10). En las primeras versiones de Java, la JVM era tan sólo un intérprete de códigos de bytes de Java. Esto hacía que la mayoría de los programas se ejecutaran con lentitud, ya que la JVM tenía que interpretar y ejecutar un código de byte a la vez. Algunas arquitecturas de computadoras modernas pueden ejecutar varias instrucciones en paralelo. Por lo general, las JVM actuales ejecutan códigos de bytes mediante una combinación de la interpretación y la denominada **compilación justo a tiempo (JIT)**. En este proceso, la JVM analiza los códigos de bytes a medida que se interpretan, en busca de *puntos activos*: partes de los códigos de bytes que se ejecutan con frecuencia. Para estas partes, un **compilador justo a tiempo (JIT)** (conocido como **compilador HotSpot de Java**) traduce los códigos de bytes al lenguaje máquina correspondiente a la computadora. Cuando la JVM encuentra estas partes compila-

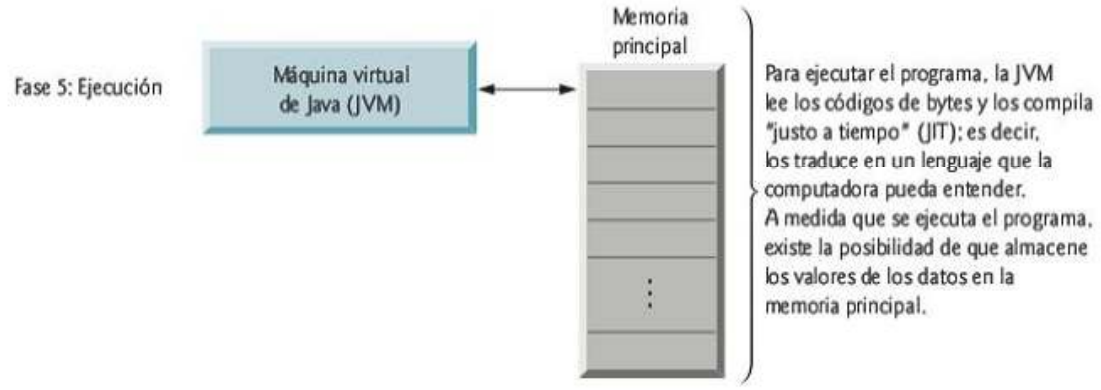


Fig. 1.10 | Entorno de desarrollo típico de Java: fase de ejecución.

das de nuevo, se ejecuta el código en lenguaje máquina, que es más rápido. Por ende, los programas en Java en realidad pasan por *dos* fases de compilación: una en la cual el código fuente se traduce a código de bytes (para tener portabilidad a través de las JVM en distintas plataformas computacionales) y otra en la que, durante la ejecución, los códigos de bytes se traducen en lenguaje máquina para la computadora actual en la que se ejecuta el programa.

Problemas que pueden ocurrir en tiempo de ejecución

Es probable que los programas no funcionen la primera vez. Cada una de las fases anteriores puede fallar, debido a diversos errores que describiremos en este texto. Por ejemplo, un programa en ejecución podría intentar una división entre cero (una operación ilegal para la aritmética con números enteros en Java). Esto haría que el programa de Java imprimiera un mensaje de error. Si esto ocurre, tendría que regresar a la fase de edición, hacer las correcciones necesarias y proseguir con las fases restantes de nuevo, para determinar que las correcciones hayan resuelto el(los) problema(s) [*Nota:* la mayoría de los programas en Java reciben o producen datos. Cuando decimos que un programa muestra un mensaje, por lo general queremos decir que aparece en la pantalla de su computadora. Los mensajes y otros datos pueden enviarse a otros dispositivos, como los discos y las impresoras, o incluso a una red para transmitirlos a otras computadoras].

Error común de programación 1.1

Los errores, como la división entre cero, ocurren a medida que se ejecuta un programa, de manera que a estos errores se les llama ***errores en tiempo de ejecución****. Los* ***errores fatales en tiempo de ejecución*** *hacen que los programas terminen de inmediato, sin haber realizado bien su trabajo. Los* ***errores no fatales en tiempo de ejecución*** *permiten a los programas ejecutarse hasta terminar su trabajo, lo que a menudo produce resultados incorrectos.*

1.10 Prueba de una aplicación en Java

En esta sección, ejecutará su primera aplicación en Java e interactuará con ella. Para empezar, ejecutará una aplicación de ATM, la cual simula las transacciones que se llevan a cabo al utilizar una máquina de cajero automático, o ATM (por ejemplo, retirar dinero, realizar depósitos y verificar los saldos de las cuentas). Aprenderá a crear esta aplicación en el ejemplo práctico *opcional* orientado a objetos que se incluye en los capítulos 12 y 13. Para los fines de esta sección vamos a suponer que está utilizando Microsoft Windows.[11]

En los siguientes pasos, ejecutará la aplicación y realizará varias transacciones. Los elementos y la funcionalidad que podemos ver en esta aplicación son típicos de lo que aprenderá a programar en este libro [*Nota:* utilizamos fuentes para diferenciar las características que se ven en una pantalla (por ejemplo, el **Símbolo del sistema**) y los elementos que no se relacionan de manera directa con una pantalla. Nuestra convención es enfatizar las características de la pantalla como los títulos y menús (por ejemplo, el menú **Archivo**) en una fuente **Helvetica sans-serif** en semi-negritas, y enfatizar los elementos que no son de la pantalla, como los nombres de archivo o los datos de entrada (como `NombrePrograma.java`) en una fuente `Lucida sans-serif`. Como tal vez ya se haya dado cuenta, la ocurrencia de definición de cada término en el texto se establece en **negritas**. En las figuras en esta sección, resaltamos en una pantalla gris claro la entrada del usuario requerida por cada paso y señalamos las partes importantes de la

11 En `www.deitel.com/books/jhtp9/`, ofrecemos una versión en Linux de esta prueba. También ofrecemos vínculos a videos que le ayudarán a empezar a trabajar con varios entornos de desarrollo integrados populares (IDE), como el Kit de desarrollo de Java SE 6 para Windows, el SKD de Eclipse para Windows, NetBeans, jGRASP, DrJava, BlueJ y el editor de texto TestPad para Windows.

aplicación. Para aumentar la visibilidad de estas características, modificamos el color de fondo de las ventanas del **Símbolo del sistema** a blanco y el color de las letras a negro]. Ésta es una versión simple que consiste de texto solamente. Más adelante en el libro, aprenderá las técnicas para rediseñar este ejemplo mediante el uso de las técnicas de GUI (interfaz gráfica de usuario).

1. ***Revise su configuración.*** Lea la sección Antes de empezar este libro para confirmar que haya instalado Java de manera apropiada en su computadora, y copiado los ejemplos del libro en su disco duro.

2. ***Localice la aplicación completa.*** Abra una ventana **Símbolo del sistema**. Para ello, puede seleccionar **Inicio | Todos los programas | Accesorios | Símbolo del sistema**. Para cambiar al directorio de la aplicación del ATM, escriba `cd C:\ejemplos\cap01\ATM` y después oprima *Intro* (figura 1.11). El comando `cd` se utiliza para cambiar de directorio.

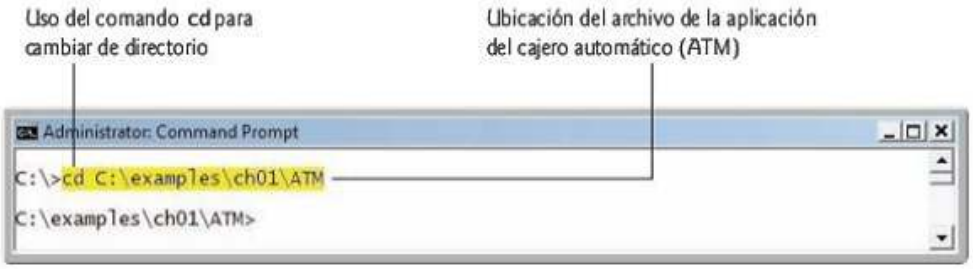


Fig. 1.11 | Abrir una ventana **Símbolo del sistema** en Windows XP y cambiar de directorio.

3. ***Ejecute la aplicación del ATM.*** Escriba el comando `java EjemploPracticoATM` y oprima *Intro* (figura 1.12). Recuerde que el comando `java`, seguido del nombre del archivo `.class` de la aplicación (en este caso, `EjemploPracticoATM`), ejecuta la aplicación. Si especificamos la extensión `.class` al usar el comando `java` se produce un error [*Nota:* los comandos en Java son sensibles a mayúsculas/minúsculas. Es importante escribir el nombre de esta aplicación con las letras A, T y M mayúsculas en "ATM", una letra E mayúscula en "Ejemplo" y una letra P mayúscula en "Practico". De lo contrario, la aplicación no se ejecutará.] Si recibe el mensaje de error "`Exception in thread "main" java.lang.NoClassDefFoundError:EjemploPracticoATM`", entonces su sistema tiene un problema con `CLASSPATH`. Consulte la sección Antes de empezar este libro para obtener instrucciones acerca de cómo corregir este problema.

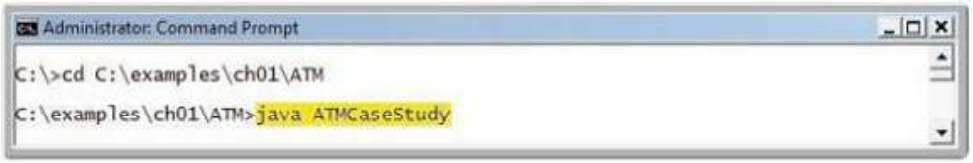


Fig. 1.12 | Uso del comando `java` para ejecutar la aplicación del ATM.

4. ***Escriba un número de cuenta.*** Cuando la aplicación se ejecuta por primera vez, muestra el mensaje "`¡Bienvenido!`" y le pide un número de cuenta. Escriba `12345` en el indicador "`Escriba su numero de cuenta:`" (figura 1.13) y oprima *Intro*.

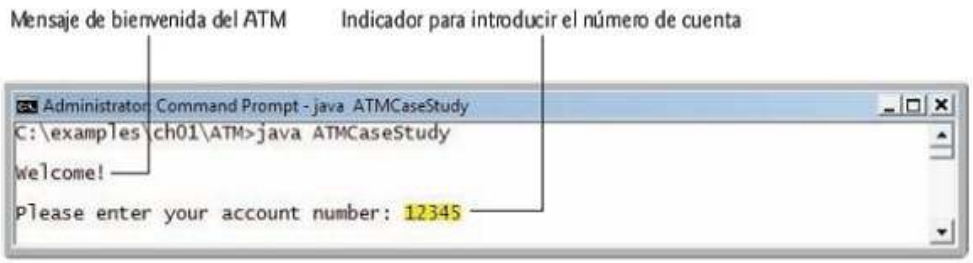


Fig. 1.13 | La aplicación pide al usuario un número de cuenta.

5. ***Escriba un NIP.*** Una vez que introduzca un número de cuenta válido, la aplicación mostrará el indicador "`Escriba su NIP:`". Escriba "`54321`" como su NIP (Número de Identificación Personal) válido y oprima *Intro*. A continuación aparecerá el menú principal del ATM, que contiene una lista de opciones (figura 1.14). En el capítulo 14 le mostraremos cómo puede introducir un NIP en forma privada mediante el uso de un objeto `JPasswordField`.

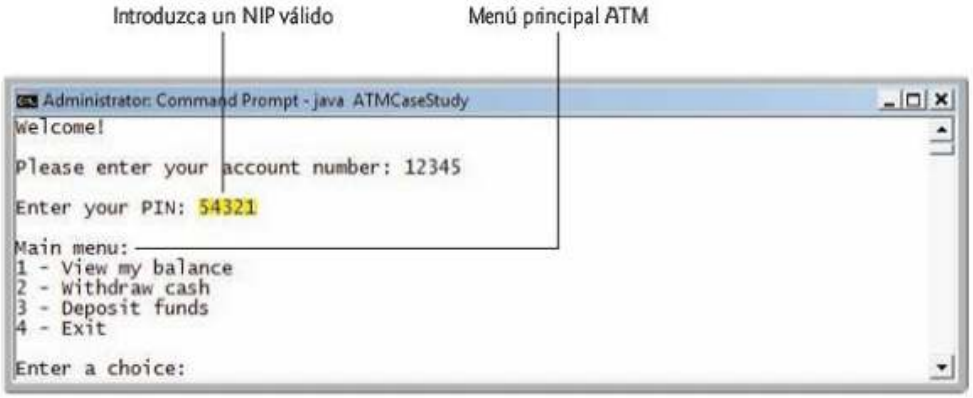


Fig. 1.14 | El usuario escribe un número NIP válido y aparece el menú principal de la aplicación del ATM.

6. ***Revise el saldo de la cuenta.*** Seleccione la opción 1, "`Ver mi saldo`" del menú del ATM (figura 1.15). A continuación la aplicación mostrará dos números: `Saldo disponible` (`$1,000.00`) y `Saldo total` (`$1,200.00`). El saldo disponible es la máxima cantidad de dinero en su cuenta, disponible para retirarla en un momento dado. En algunos casos, ciertos fondos como los depósitos recientes, no están disponibles de inmediato para que el usuario pueda retirarlos, por lo que el saldo disponible puede ser menor que el saldo total, como en este caso. Después de mostrar la información de los saldos de la cuenta, se vuelve a mostrar el menú principal de la aplicación.

7. ***Retire dinero de la cuenta.*** Seleccione la opción 2, "`Retirar efectivo`", del menú de la aplicación. A continuación aparecerá (figura 1.16) una lista de montos en dólares (por ejemplo: 20, 40, 60, 100 y 200). También tendrá la oportunidad de cancelar la transacción y regresar al menú principal. Retire $100 seleccionando la opción 4. La aplicación mostrará el mensaje "`Tome su efectivo ahora`" y regresará al menú principal. [*Nota:* por desgracia, esta aplicación sólo *simula* el comportamiento de un verdadero ATM, por lo cual no dispensa efectivo en realidad].

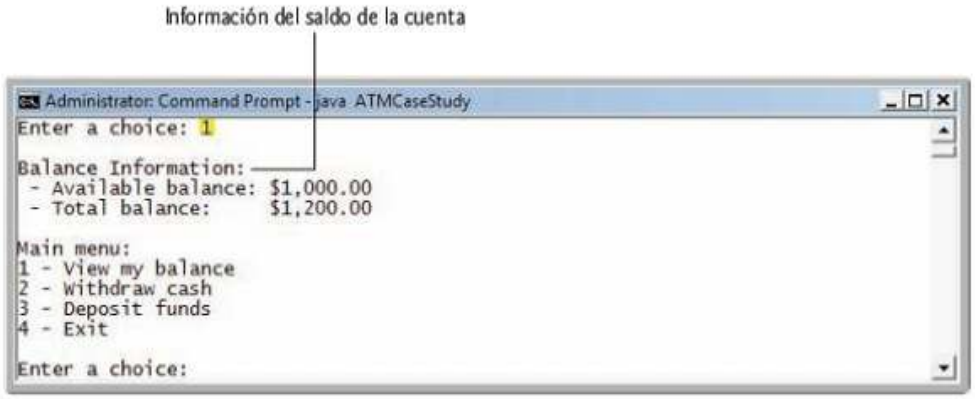


Fig. 1.15 | La aplicación del ATM muestra la información del saldo de la cuenta del usuario.

Fig. 1.16 | Se retira el dinero de la cuenta y la aplicación regresa al menú principal.

8. ***Confirme que la información de la cuenta se haya actualizado.*** En el menú principal, seleccione la opción 1 de nuevo para ver el saldo actual de su cuenta (figura 1.17). Observe que tanto el saldo disponible como el saldo total se han actualizado para reflejar su transacción de retiro.

9. ***Finalice la transacción.*** Para finalizar su sesión actual en el ATM, seleccione la opción 4, "Salir" del menú principal (figura 1.18.) El ATM saldrá del sistema y mostrará un mensaje de despedida al usuario. A continuación, la aplicación regresará a su indicador original, pidiendo el número de cuenta del siguiente usuario.

10. ***Salga de la aplicación del ATM y cierre la ventana*** **Símbolo del sistema.** La mayoría de las aplicaciones cuentan con una opción para salir y regresar al directorio del **Símbolo del sistema** desde el cual se ejecutó la aplicación. Un ATM real no proporciona al usuario la opción de apagar la máquina ATM. En vez de ello, cuando el usuario ha completado todas las transacciones deseadas y elige la opción del menú para salir, el ATM se reinicia y muestra un indicador para el número de cuenta del siguiente usuario. Como se muestra en la figura 1.18, la

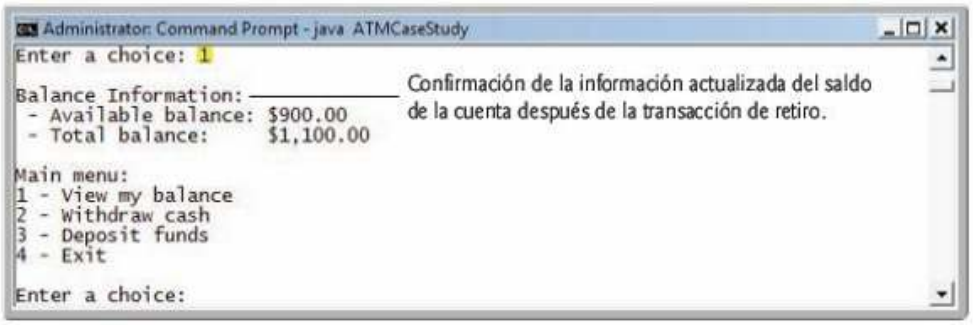


Fig. 1.17 | Verificación del nuevo saldo.

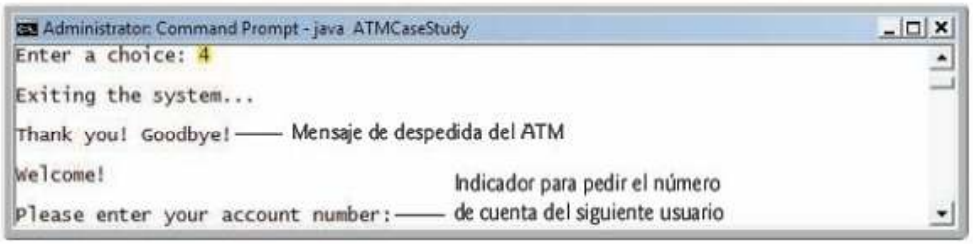


Fig. 1.18 | Finalización de una sesión de transacciones con el ATM.

aplicación del ATM se comporta de manera similar. Al elegir la opción del menú para salir sólo se termina la sesión del usuario actual con el ATM, no toda la aplicación completa. Para salir realmente de la aplicación del ATM, haga clic en el botón de cerrar (x) en la esquina superior derecha de la ventana **Símbolo del sistema**. Al cerrar la ventana, la aplicación termina su ejecución.

1.11 Web 2.0: Las redes sociales

Literalmente, la Web cobró fuerza a mediados de la década de 1990, pero surgieron tiempos difíciles a principios del año 2000, debido al desplome económico de "punto com". Al resurgimiento que empezó alrededor de 2004, se le conoce como **Web 2.0**. A Google se le considera en muchas partes como la compañía característica de Web 2.0. Algunas otras compañías con "características de Web 2.0" son: YouTube (sitio para compartir videos), FaceBook (red social), Twitter (microblogs), Groupon (comercio social), Foursquare (reportes o "check-ins" móviles), Salesforce (software de negocios que se ofrece como servicios en línea), Craigslist (listados gratuitos de anuncios clasificados), Flickr (sitio para compartir fotos), Second Life (un mundo virtual), Skype (telefonía por Internet) y Wikipedia (una enciclopedia en línea gratuita).

Google

En 1996, los candidatos a un doctorado en ciencias computacionales de Stanford, Larry Page y Sergey Brin, empezaron a colaborar en un nuevo motor de búsqueda. En 1997 le cambiaron el nombre a Google con base en el término matemático *gúgol* (en inglés, googol), una cantidad representada por el número "uno" seguido de 100 "ceros" (o 10^{100}): un número de un tamaño asombroso. La habilidad de Google para devolver resultados de búsquedas con extrema precisión le ayudó a convertirse con rapidez en el motor de búsqueda más utilizado, además de ser uno de los sitios Web más populares en el mundo.

Google continúa siendo un innovador en las tecnologías de búsqueda. Por ejemplo, Google Goggles es una fascinante aplicación móvil (disponible en Android e iPhone) que permite al usuario realizar una búsqueda, con la novedad de que utiliza una fotografía en vez de texto. Usted sólo tiene que tomar fotografías de puntos de referencia, libros (cubiertas o códigos de barras), logotipos, arte o etiquetas de botellas de vino, y Google Goggles escanea la fotografía para devolver los resultados de la búsqueda. También puede tomar una fotografía de texto (por ejemplo, el menú de un restaurante o un anuncio) y Google Goggles lo traducirá por usted.

Servicios Web y mashups

En este libro incluimos un tratamiento detallado sobre los servicios Web (capítulo 31) y presentamos la nueva metodología de desarrollo de aplicaciones conocida como *mashups*, en la que puede desarrollar con rapidez aplicaciones poderosas e intrigantes, al combinar servicios Web complementarios (a menudo gratuitos) y otras formas de fuentes de información (figura 1.19). Uno de los primeros mashups fue `www.housingmaps.com`, que combina al instante los listados de bienes raíces proporcionados por `www.craigslist.org` con las capacidades de generación de mapas de *Google Maps* para ofrecer mapas que muestren las ubicaciones de los apartamentos en renta dentro de cierta área.

Fuente de servicios Web	Cómo se utiizan
Google Maps	Servicios de mapas
Facebook	Redes sociales
Foursquare	Reportes (check-ins) móviles
LinkedIn	Redes sociales para negocios
YouTube	Búsquedas de video
Twitter	Microblogs
Groupon	Comercio social
Netflix	Renta de películas
eBay	Subastas en Internet
Wikipedia	Enciclopedia colaborativa
PayPal	Pagos
Last.fm	Radio por Internet
Amazon eCommerce	Compra de libros y otros artículos
Salesforce.com	Administración de las relaciones con el cliente (CRM)
Skype	Telefonía por Internet
Microsoft Bing	Búsqueda
Flickr	Compartir fotografías
Zillow	Precios de bienes raíces
Yahoo Search	Búsqueda
WeatherBug	Clima

Fig. 1.19 | Algunos servicios Web populares (`www.programmableweb.com/apis/directory/1?sort=mashups`).

Ajax

Ajax es una de las tecnologías de software más importantes de Web 2.0, ya que ayuda a las aplicaciones basadas en Internet a funcionar como las aplicaciones de escritorio; una tarea difícil, dado que dichas

aplicaciones sufren de retrasos en la transmisión, a medida que los datos se intercambian entre su computadora y las computadoras servidores en Internet. Mediante el uso de Ajax, las aplicaciones como Google Maps han logrado un desempeño excelente, además de que su apariencia visual se asemeja a las aplicaciones de escritorio. Aunque no hablaremos sobre la programación "pura" con Ajax en este libro (que es bastante compleja), en el capítulo 30 le mostraremos cómo crear aplicaciones habilitadas para Ajax mediante el uso de los componentes de JavaServer Faces (JSF) habilitados para Ajax.

Aplicaciones sociales

Durante los últimos años se ha producido un aumento considerable en el número de aplicaciones sociales en Web. Aún y cuando la industria de la computación ya alcanzó la madurez, estos sitios fueron capaces de tener un éxito fenomenal en un periodo de tiempo relativamente corto. La figura 1.20 analiza unas cuantas de las aplicaciones sociales que están generando un impacto.

Compañía	Descripción
Facebook	Facebook inició desde un dormitorio en Harvard en el año 2004, gracias a los alumnos Mark Zuckerberg, Chris Hughes, Dustin Moskovitz y Eduardo Saverin, y ahora tiene un valor estimado de 70 mil millones de dólares. Para enero de 2011, Facebook era el sitio más activo en Internet con más de 600 millones de usuarios —casi 9% de la población mundial—, quienes invierten 700 mil millones de minutos en Facebook al mes. Según su tasa de crecimiento actual (cerca del 5% mensual), en 2012 Facebook llegará a mil millones de usuarios ¡de los dos mil millones de Internet! La actividad en este sitio lo hace muy atractivo para los desarrolladores de aplicaciones. Cada día, los usuarios de Facebook instalan más de 20 millones de aplicaciones (`http://www.facebook.com/press/info.php?statistics`).
Twitter	Jack Dorsey, Evan Williams e Isaac "Biz" Stone fundaron Twitter en 2006: todo desde la compañía de podcasts, Odeo. Twitter revolucionó los *microblogs*. Los usuarios publican "tweets": mensajes de hasta 140 caracteres de longitud. Se publican cerca de 95 millones de tweets a diario (`twitter.com/about`). Usted puede seguir los tweets de amigos, artistas, negocios, representantes del gobierno (incluso el presidente de Estados Unidos, quien tiene 6.3 millones de seguidores), etcétera, o seguir tweets del tema para dar seguimiento a noticias, tendencias y mucho más. Al momento de escribir este libro, Lady Gaga tenía el mayor número de seguidores (más de 7.7 millones). Twitter se convirtió en el punto de origen para muchas noticias de última hora en todo el mundo.
Groupon	Groupon, un *sitio de comercio social*, fue lanzado por Andrew Mason in 2008. Para enero de 2011 la compañía estaba valuada alrededor de los $15 mil millones ¡con lo cual se convirtió en la compañía con más rápido crecimiento hasta esa fecha! Ahora está disponible en cientos de mercados en todo el mundo. Groupon muestra una oferta diaria en cada mercado para restaurantes, vendedores al detalle, servicios, atracciones y demás. Las ofertas se activan sólo hasta que se inscribe el mínimo número de personas requeridas para comprar el producto o servicio. Si usted se inscribe en una oferta y todavía no cumple con el mínimo, tal vez se vea tentado a dar aviso a otras personas sobre esa oferta por correo electrónico, Facebook, Twitter, etcétera. Si la oferta no cumple con el mínimo de ventas, se cancela. Una de las ofertas de Groupon más exitosas a nivel nacional a la fecha fue un certificado de $50 dólares en mercancía de una importante compañía de ropa a sólo $25. Se vendieron más de 440,000 cupones en un solo día.

Fig. 1.20 | Aplicaciones sociales (parte 1 de 2).

Compañía	Descripción
Foursquare	Foursquare —creada en 2009 por Dennis Crowley y Naveen Selvadurai— es una aplicación para realizar reportes (*check-ins*) móviles, la cual le permite notificar a sus amigos los lugares que visita. Puede descargar la aplicación en su teléfono inteligente y vincularla con sus cuentas de Facebook y Twitter, de modo que sus amigos puedan seguirlo desde varias plataformas. Si no tiene un teléfono inteligente, puede reportarse mediante un mensaje de texto. Foursquare utiliza el servicio GPS para determinar su ubicación exacta. Las empresas usan Foursquare para enviar ofertas a los usuarios que se encuentren cerca. Foursquare inició sus operaciones en marzo de 2009 y ya cuenta con más de 5 millones de usuarios en todo el mundo.
Skype	Skype es un producto de software que le permite realizar llamadas de voz y de video (la mayoría son gratuitas) a través de Internet, mediante el uso de una tecnología llamada *VoIP* (*Voz sobre IP*; IP se refiere a "Protocolo de Internet"). Niklas Zennström y Dane Janus Friiis fundaron Skype en 2003. Dos años después, vendieron la compañía a eBay por $2.6 mil millones.
YouTube	YouTube es un sitio para compartir videos que se fundó en 2005. Antes de que transcurriera un año, Google compró la compañía por $1.65 mil millones. En la actualidad, YouTube es responsable del 10% del tráfico total en Internet (`www.webpronews.com/topnews/2010/04/16/facebook-and-youtube-get-the-most-business-internet-traffic`). Menos de un año después de la liberación del iPhone 3GS de Apple —el primer modelo del iPhone en ofrecer video— las transferencias desde dispositivos móviles a YouTube aumentaron un 400% (`www.hypebot.com/hypebot/2009/06/youtube-reports-1700-jump-in-mobile-video.html`).

Fig. 1.20 | Aplicaciones sociales (parte 2 de 2).

1.12 Tecnologías de software

La figura 1.21 muestra una lista de palabras de moda que escuchará en la comunidad de desarrollo de software. Creamos Centros de Recursos sobre la mayoría de estos temas, y hay muchos por venir.

Tecnología	Descripción
Software ágil	El **desarrollo ágil de software** es un conjunto de metodologías que tratan de implementar software con más rapidez y menos recursos que las metodologías anteriores. Visite los sitios de Agile Alliance (`www.agilealliance.org`) y Agile Manifesto (`www.agilemanifesto.org`). También puede visitar el sitio en español `www.agile-spain.com`.
Refactorización	La **refactorización** implica reformular el código para hacerlo más claro y fácil de mantener, al tiempo que se preserva su funcionalidad. Es muy utilizado en las metodologías de desarrollo ágil. Muchos IDE contienen *herramientas de refactorización* integradas para realizar la mayor parte del proceso de refactorización de manera automática.
Patrones de diseño	Los **patrones de diseño** son arquitecturas probadas para construir software orientado a objetos flexible y que pueda mantenerse. El campo de los patrones de diseño trata de enumerar a los patrones recurrentes, y de alentar a los diseñadores de software para que los reutilicen y puedan desarrollar un software de mejor calidad con menos tiempo, dinero y esfuerzo. En el apéndice Q analizaremos los patrones de diseño de Java.

Fig. 1.21 | Tecnologías de software (parte 1 de 2).

Tecnología	Descripción
LAMP	MySQL es un sistema de administración de bases de datos de código fuente abierto. PHP es el lenguaje de "secuencias de comandos" del lado servidor de código fuente abierto más popular para el desarrollo de aplicaciones Web. **LAMP** es un acrónimo para el conjunto de tecnologías de código fuente abierto que usan muchos desarrolladores en la creación de aplicaciones Web: se refiere a Linux, Apache, MySQL y PHP (o Perl, o Python; otros dos lenguajes de secuencias de comandos).
Software como un servicio (SaaS)	Por lo general, el software siempre se ha visto como un producto; la mayoría aún se ofrece de esta forma. Para ejecutar una aplicación, hay que comprarla a un distribuidor de software. Después la instalamos en la computadora y la ejecutamos cuando sea necesario. A medida que aparecen nuevas versiones, actualizamos el software, lo cual genera con frecuencia un gasto considerable. Este proceso puede ser incómodo para las organizaciones con decenas de miles de sistemas, a los que se debe dar mantenimiento en una diversa selección de equipo de cómputo. En el **Software como un servicio (SaaS)**, éste ejecuta en servidores ubicados en cualquier parte de Internet. Que al ser actualizados, los clientes en todo el mundo ven las nuevas capacidades sin necesidad de una instalación local. Podemos acceder al servicio a través de un navegador. Los navegadores son bastante portables, por lo que podemos ver las mismas aplicaciones en una amplia variedad de computadoras desde cualquier parte del mundo. Salesforce.com, Google, Microsoft Office Live y Windows Live ofrecen SaaS.
Plataforma como un servicio (SaaS)	La **Plataforma como un servicio (PaaS)** provee una plataforma de cómputo para desarrollar y ejecutar aplicaciones como un servicio a través de Web, en vez de instalar las herramientas en su computadora. Los proveedores de PaaS más importantes son: Google App Engine, Amazon EC2, Bungee Labs, entre otros.
Computación en la nube	SaaS y PaaS son ejemplos de **computación en la nube** en donde el software, las plataformas y la infraestructura (por ejemplo, el poder de procesamiento y el almacenamiento) se alojan según la demanda a través de Internet. Esto ofrece a los usuarios flexibilidad, escalabilidad y un ahorro en los costos. Por ejemplo, considere las necesidades de almacenamiento de datos de una compañía, que pueden fluctuar de manera considerable en el transcurso de un año. En vez de invertir en hardware de almacenamiento de gran escala —cuyo costo de compra, mantenimiento y aseguramiento puede ser considerable, además de que no siempre es posible aprovechar su capacidad total—, la compañía podría comprar servicios basados en la nube (como Amazon S3, Google Storage, Microsoft Windows Azure™, Nirvanix™ y otros) según los fuera requiriendo.
Kit de desarrollo de software (SDK)	Los **Kits de desarrollo de software (SDK)** incluyen tanto las herramientas como la documentación que utilizan los desarrolladores para programar aplicaciones. Por ejemplo, usted usará el Kit de desarrollo de Java (JDK) para crear y ejecutar aplicaciones de Java.

Fig. 1.21 | Tecnologías de software (parte 2 de 2).

La figura 1.22 describe las categorías de liberación de versiones de los productos de software.

Versión	Descripción
Alfa	El software *alfa* es la primera versión de un producto de software cuyo desarrollo aún se encuentra activo. Por lo general las versiones alfa tienen muchos errores, son incompletas y estables; además se liberan a un pequeño número de desarrolladores para que evalúen las nuevas características, para obtener retroalimentación lo más pronto posible, etcétera.

Fig. 1.22 | Terminología de liberación de versiones de productos de software (parte 1 de 2).

Versión	Descripción
Beta	Las versiones *beta* se liberan a un número mayor de desarrolladores en una etapa posterior del proceso de desarrollo, una vez que se ha corregido la mayoría de los errores importantes y las nuevas características están casi completas. El software beta es más estable, pero todavía puede sufrir muchos cambios.
Candidatos para liberación (Release Candidates)	En general, los *candidatos para liberación* tienen todas sus *características completas*, están (supuestamente) libres de errores y listos para que la comunidad los utilice, con lo cual se logra un entorno de prueba diverso: el software se utiliza en distintos sistemas, con restricciones variables y para muchos fines diferentes. Cualquier error que aparezca se corrige y, en un momento dado, el producto final se libera al público en general. A menudo, las compañías de software distribuyen actualizaciones incrementales a través de Internet.
Beta permanente	El software que se desarrolla mediante este método por lo general no tiene números de versión (por ejemplo, la búsqueda de Google o Gmail). Este software, que se aloja en la nube (no se instala en su computadora), evoluciona de manera constante de modo que los usuarios siempre dispongan de la versión más reciente.

Fig. 1.22 | Terminología de liberación de versiones de productos de software (parte 2 de 2).

1.13 Cómo estar al día con las tecnologías de información

La figura 1.23 muestra una lista de las publicaciones técnicas y comerciales que le ayudarán a permanecer actualizado con la tecnología, las noticias y las tendencias más recientes. También encontrará una lista cada vez más grande de Centros de recursos relacionados con Internet y Web en `http://www.deitel.com/ResourceCenters.html`.

Publicación	URL
Bloomberg BusinessWeek	`www.businessweek.com`
CNET	`news.cnet.com`
Computer World	`www.computerworld.com`
Engadget	`www.engadget.com`
eWeek	`www.eweek.com`
Fast Company	`www.fastcompany.com/`
Fortune	`money.cnn.com/magazines/fortune/`
InfoWorld	`www.infoworld.com`
Mashable	`mashable.com`
PCWorld	`www.pcworld.com`
SD Times	`www.sdtimes.com`
Slashdot	`slashdot.org/`
Smarter Technology	`www.smartertechnology.com`
Technology Review	`technologyreview.com`
Techcrunch	`techcrunch.com`
Wired	`www.wired.com`

Fig. 1.23 | Publicaciones técnicas y comerciales.

1.14 Conclusión

En este capítulo analizamos el hardware y software de computadora, los lenguajes de programación y los sistemas operativos. Vimos las generalidades de un entorno típico de desarrollo de programas de Java y probamos una aplicación de Java. Introdujimos los fundamentos de la tecnología de objetos. Aprendió acerca de algunos de los emocionantes y nuevos acontecimientos en el campo de las computadoras. También analizamos cierta terminología clave del desarrollo de software.

En el capítulo 2 creará sus primeras aplicaciones de Java. Podrá ver cómo es que los programas muestran mensajes en la pantalla y obtienen información del usuario mediante el teclado para procesarla. Utilizará los tipos de datos primitivos y los operadores aritméticos de Java en cálculos que emplean los operadores de igualdad y relacionales de Java para escribir instrucciones simples de toma de decisiones.

Ejercicios de autoevaluación

1.1 Complete las siguientes oraciones:

a) La compañía que popularizó la computación personal fue _____ .
b) La computadora que legitimó la computación personal en los negocios y la industria fue _____ .
c) Las computadoras procesan datos bajo el control de conjuntos de instrucciones conocidas como _____ .
d) Las unidades lógicas clave de la computadora son _____, _____, _____, _____, _____ y _____ .
e) Los tres tipos de lenguajes descritos en este capítulo son _____, _____ y _____ .
f) Los programas que traducen programas en lenguaje de alto nivel a lenguaje máquina se denominan _____ .
g) _____ es un sistema operativo de teléfonos inteligentes, basado en el kernel de Linux y en Java.
h) En general, el software _____ tiene todas sus *características completas*, está (supuestamente) libre de errores y listo para que la comunidad lo utilice.
i) Al igual que muchos teléfonos inteligentes, el control remoto del Wii utiliza un _____ que permite al dispositivo responder al movimiento.

1.2 Complete las siguientes oraciones sobre el entorno de Java:

a) El comando _____ del JDK ejecuta una aplicación de Java.
b) El comando _____ del JDK compila un programa de Java.
c) Un archivo de programa de Java debe terminar con la extensión de archivo _____ .
d) Cuando se compila un programa en Java, el archivo producido por el compilador termina con la extensión _____ .

1.3 Complete las siguientes oraciones (con base en la sección 1.6):

a) Los objetos tienen una propiedad que se conoce como _____ ; aunque éstos pueden saber cómo comunicarse con los demás objetos a través de interfaces bien definidas, por lo general no se les permite saber cómo están implementados los otros objetos.
b) Los programadores de Java se concentran en crear _____ , que contienen campos y el conjunto de métodos que manipulan a esos campos y proporcionan servicios a los clientes.
c) El proceso de analizar y diseñar un sistema desde un punto de vista orientado a objetos se denomina _____ .
d) Mediante la _____ , se derivan nuevas clases de objetos al absorber las características de las clases existentes y luego agregar características únicas propias.

e) _____ es un lenguaje gráfico que permite a las personas que diseñan sistemas de software utilizar una notación estándar en la industria para representarlos.
f) El tamaño, forma, color y peso de un objeto se consideran _____ de su clase.

Respuestas a los ejercicios de autoevaluación

1.1 a) Apple. b) Computadora personal (PC) de IBM. c) programas. d) unidad de entrada, unidad de salida, unidad de memoria, unidad central de procesamiento, unidad aritmética y lógica, unidad de almacenamiento secundario. e) lenguajes máquina, lenguajes ensambladores, lenguajes de alto nivel. f) compiladores. g) Android. h) Candidato de liberación. i) acelerómetro.

1.2 a) `java`. b) `javac`. c) `.java`. d) `.class`. e) códigos de bytes.

1.3 a) ocultamiento de información. b) clases. c) análisis y diseño orientados a objetos (A/DOO). d) herencia. e) El Lenguaje unificado de modelado (UML). f) atributos.

Ejercicios

1.4 Complete las siguientes oraciones:
a) La unidad lógica de la computadora que recibe información desde el exterior de la computadora para que ésta la utilice se llama _____ .
b) El proceso de indicar a la computadora cómo resolver un problema se llama _____ .
c) _____ es un tipo de lenguaje computacional que utiliza abreviaturas del inglés para las instrucciones de lenguaje máquina.
d) _____ es una unidad lógica de la computadora que envía información que ya ha sido procesada por la computadora a varios dispositivos, de manera que pueda utilizarse fuera de la computadora.
e) _____ y _____ son unidades lógicas de la computadora que retienen información.
f) _____ es una unidad lógica de la computadora que realiza cálculos.
g) _____ es una unidad lógica de la computadora que toma decisiones lógicas.
h) Los lenguajes _____ son los más convenientes para que el programador pueda escribir programas con rapidez y facilidad.
i) Al único lenguaje que una computadora puede entender directamente se le conoce como el _____ de esa computadora.
j) _____ es una unidad lógica de la computadora que coordina las actividades de todas las demás unidades lógicas.

1.5 Complete las siguientes oraciones:
a) _____ se utiliza ahora para desarrollar aplicaciones empresariales de gran escala, para mejorar la funcionalidad de los servidores Web, para proporcionar aplicaciones para dispositivos domésticos y muchos otros fines más.
b) En un principio, _____ se hizo muy popular como lenguaje de desarrollo para el sistema operativo UNIX.
c) La compañía Web 2.0 _____ es la que tiene el crecimiento más rápido de la historia.
d) El lenguaje de programación _____ fue desarrollado por Bjarne Stroustrup a principios de la década de 1980 en los Laboratorios Bell.

1.6 Complete las siguientes oraciones:
a) Por lo general, los programas de Java pasan a través de cinco fases: _____, _____, _____, _____ y _____ .
b) Un _____ proporciona muchas herramientas que dan soporte al proceso de desarrollo de software, como los editores para escribir y editar programas, los depuradores para localizar los errores lógicos en los programas, y muchas otras características más.

c) El comando java invoca al _____, que ejecuta los programas de Java.
d) Una _____ es una aplicación de software que simula una computadora, pero oculta el sistema operativo y el hardware subyacentes de los programas que interactúan con la VM.
e) El _____ toma los archivos `.class` que contienen los códigos de bytes del programa y los transfiere a la memoria principal.
f) El _____ examina los códigos de bytes para asegurar que sean válidos.

1.7 Explique las dos fases de compilación de los programas de Java.

1.8 Es probable que usted lleve en su muñeca uno de los tipos de objetos más comunes en el mundo: un reloj. Analice cómo se aplica cada uno de los siguientes términos y conceptos a la noción de un reloj: objeto, atributos, comportamientos, clase, herencia (por ejemplo, considere un reloj despertador), abstracción, modelado, mensajes, encapsulamiento, interfaz y ocultamiento de información.

Marcar la diferencia

Hemos incluido en este libro ejercicios Marcar la diferencia, en los que le pediremos que trabaje con problemas que son de verdad importantes para los individuos, las comunidades, los países y el mundo. Para obtener más información sobre las organizaciones a nivel mundial que trabajan para marcar la diferencia, y para obtener ideas sobre proyectos de programación relacionados, visite nuestro Centro de recursos para marcar la diferencia en `www.deitel.com/makingadifference`.

1.9 ***(Prueba práctica: calculadora de impacto ambiental del carbono)*** Algunos científicos creen que las emisiones de carbono, sobre todo las que se producen al quemar combustibles fósiles, contribuyen de manera considerable al calentamiento global y que esto se puede combatir si las personas tomamos conciencia y limitamos el uso de los combustibles con base en el carbono. Las organizaciones y los individuos se preocupan cada vez más por el "impacto ambiental del carbono". Los sitios Web como Terra Pass

```
www.terrapass.com/carbon-footprint-calculator/
```

y Carbon Footprint

```
www.carbonfootprint.com/calculator.aspx
```

ofrecen calculadoras de impacto ambiental del carbono. Pruébelas para determinar el impacto que provoca usted en el ambiente debido al carbono. Los ejercicios en capítulos posteriores le pedirán que programe su propia calculadora de impacto ambiental del carbono. Como preparación, le sugerimos investigar las fórmulas para calcularlo.

1.10 ***(Prueba práctica: calculadora del índice de masa corporal)*** Según las estimaciones recientes, dos terceras partes de las personas que viven en Estados Unidos padecen de sobrepeso; la mitad de estas personas son obesas. Esto provoca aumentos considerables en el número de personas con enfermedades como la diabetes y las cardiopatías. Para determinar si una persona tiene sobrepeso o padece de obesidad, puede usar una medida conocida como índice de masa corporal (IMC). El Departamento de Salud y Servicios Humanos de Estados Unidos proporciona una calculadora del IMC en `www.nhlbisupport.com/bmi/`. Úsela para calcular su propio IMC. Un ejercicio del capítulo 2 le pedirá que programe su propia calculadora del IMC. Como preparación, le sugerimos investigar las fórmulas para calcular el IMC.

1.11 ***(Atributos de los vehículos híbridos)*** En este capítulo aprendió sobre los fundamentos de las clases. Ahora empezará a describir con detalle los aspectos de una clase conocida como "Vehículo híbrido". Los cuales se están volviendo cada vez más populares, puesto que por lo general pueden ofrecer mucho más kilometraje que los operados sólo por gasolina. Navegue en Web y estudie las características de cuatro o cinco de los autos híbridos populares en la actualidad; después haga una lista de todos los atributos relacionados con sus características de híbridos que pueda encontrar. Por ejemplo, algunos de los atributos comunes son los kilómetros por litro en ciudad y los kilómetros por litro en carretera. También puede hacer una lista de los atributos de las baterías (tipo, peso, etcétera).

1.12 ***(Neutralidad de género)*** Muchas personas desean eliminar el sexismo de todas las formas de comunicación. Usted ha recibido la tarea de crear un programa que pueda procesar un párrafo de texto y reemplazar palabras que tengan un género específico con palabras neutrales en cuanto al género. Suponiendo que recibió una lista de palabras con género específico y sus reemplazos con neutralidad de género (por ejemplo, reemplace "esposa" por "cónyuge", "hombre" por "persona", "hija" por "descendiente", y así en lo sucesivo), explique el procedimiento que utilizaría para leer un párrafo de texto y realizar estos reemplazos en forma manual. ¿Cómo podría su procedimiento generar un término extraño como

"woperchild", que aparece listado en el Diccionario Urbano (`www.urbandictionary.com`)? En el capítulo 4 aprenderá que un término más formal para "procedimiento" es "algoritmo", que especifica los pasos a realizar, además del orden en el que se deben llevar a cabo.

1.13 ***(Privacidad)*** Algunos servicios de correo electrónico en línea guardan toda la correspondencia electrónica durante cierto periodo de tiempo. Suponga que un empleado disgustado de uno de estos servicios de correo electrónico en línea publicara en Internet todas las correspondencias de correo electrónico de millones de personas, entre ellas la suya. Analice las consecuencias.

1.14 ***(Responsabilidad ética y legal del programador)*** Como programador en la industria, tal vez llegue a desarrollar software que podría afectar la salud de otras personas, o incluso sus vidas. Suponga que un error de software en uno de sus programas provocara que un paciente de cáncer recibiera una dosis excesiva durante la terapia de radiación y resultara gravemente lesionada o muriera. Analice las consecuencias.

1.15 ***(El "Flash Crash" de 2010)*** Un ejemplo de las consecuencias de nuestra excesiva dependencia con respecto a las computadoras es el denominado "flash crash", que ocurrió el 6 de mayo de 2010, cuando el mercado de valores de Estados Unidos se derrumbó de manera precipitada en cuestión de minutos, al borrarse billones de dólares de inversiones que se volvieron a recuperar pocos minutos después. Use Internet para investigar las causas de este derrumbe y analice las consecuencias que genera.

Recursos para marcar la diferencia

La *Copa Imagine de Microsoft* es una competencia global en la que los estudiantes usan la tecnología para intentar resolver algunos de los problemas más difíciles del mundo, como la sostenibilidad ambiental, acabar con la hambruna, la respuesta a emergencias, la alfabetización, combatir el HIV/SIDA y otros más. Visite `www.imaginecup.com/about` para obtener más información sobre la competencia y para aprender sobre los proyectos desarrollados por los anteriores ganadores. También encontrará varias ideas de proyectos enviadas por organizaciones de caridad a nivel mundial en `www.imaginecup.com/students/imagine-cup-solve-this`. Si desea obtener ideas para proyectos de programación que puedan marcar la diferencia, busque en Web el tema "marcar la diferencia" y visite:

`www.un.org/millenniumgoals`
El proyecto Milenio de Naciones Unidas busca soluciones para los principales problemas mundiales, como la sostenibilidad ambiental, la igualdad de sexos, la salud infantil y materna, la educación universal y otros más.

`www.ibm.com/smarterplanet/`
El sitio Web Smarter Planet de IBM® habla sobre cómo es que IBM utiliza la tecnología para resolver problemas relacionados con los negocios, la computación en la nube, la educación, la sostenibilidad y otros más.

`www.gatesfoundation.org/Pages/home.aspx`
La Fundación Bill y Melinda Gates ofrece becas a las organizaciones que trabajan para mitigar el hambre, la pobreza y las enfermedades en los países en desarrollo. En Estados Unidos, la fundación se enfoca en mejorar la educación pública, en especial para las personas con bajos recursos.

`www.nethope.org/`
NetHope es una colaboración de organizaciones humanitarias en todo el mundo, que trabajan para resolver los problemas relacionados con la tecnología, como la conectividad y la respuesta a las emergencias, entre otros.

`www.rainforestfoundation.org/home`
La Fundación Rainforest trabaja para preservar los bosques tropicales y proteger los derechos de los indígenas que consideran a estos bosques como su hogar. El sitio contiene una lista de actividades que usted puede hacer para ayudar.

`www.undp.org/`
El Programa de las Naciones Unidas para el Desarrollo (UNDP) busca soluciones a los desafíos globales, como la prevención y recuperación de crisis, la energía y el ambiente, la gobernanza democrática y otros más.

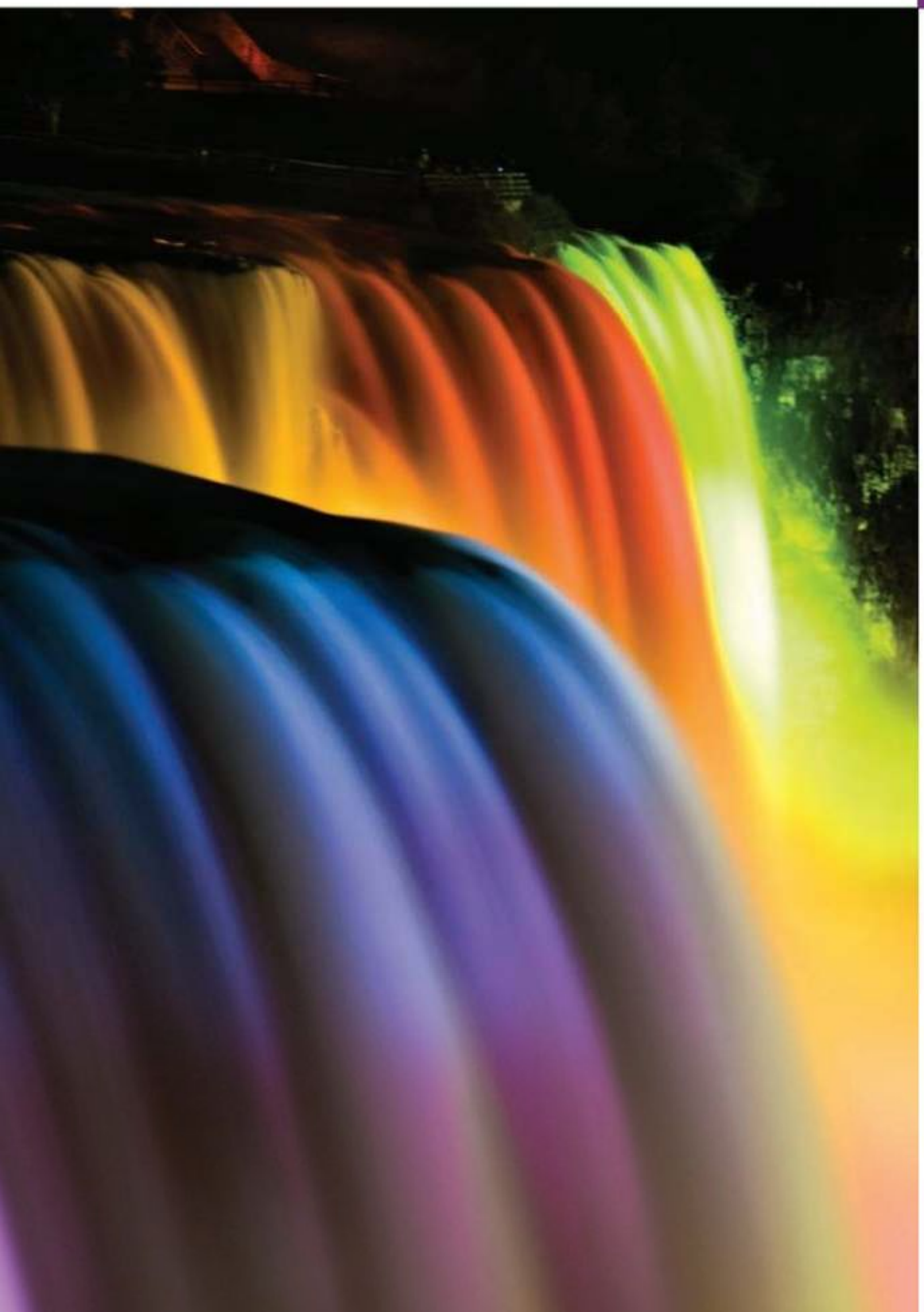
`www.unido.org`
La Organización de las Naciones Unidas para el Desarrollo Industrial (UNIDO) busca reducir la pobreza, dar a los países en desarrollo la oportunidad de participar en el comercio global y promover tanto la eficiencia de la energía como la sostenibilidad.

`www.usaid.gov/`
USAID promueve la democracia global, la salud, el crecimiento económico, la prevención de conflictos y la ayuda humanitaria, entre otras cosas.

`www.toyota.com/ideas-for-good/`
El sitio Web Ideas for Good de Toyota describe varias tecnologías de esta empresa que están haciendo la diferencia; entre éstas; su Sistema avanzado de asistencia de estacionamiento (Advanced Parking Guidance System), la tecnología Hybrid Synergy Drive®, el Sistema de ventilación operado por energía solar (Solar Powered Ventilation System), el modelo T.H.U.M.S. (Modelo humano total para la seguridad) y Touch Tracer Display. Usted puede participar en el desafío de Ideas for Good; envíe un breve ensayo o un video que describa cómo se pueden usar estas tecnologías para otros buenos propósitos.

Introducción a las aplicaciones en Java

2



¿Qué hay en un nombre? A eso a lo que llamamos rosa, si le diéramos otro nombre conservaría su misma fragancia dulce.
—William Shakespeare

Al hacer frente a una decisión, siempre me pregunto, "¿Cuál será la solución más divertida?"
—Peggy Walker

El mérito principal del lenguaje es la claridad.
—Galen

Una persona puede hacer la diferencia y cada persona debería intentarlo.
—John F. Kennedy

Objetivos

En este capítulo aprenderá a:

- Escribir aplicaciones simples en Java.
- Utilizar las instrucciones de entrada y salida.
- Familiarizarse con los tipos primitivos de Java.
- Comprender los conceptos básicos de la memoria.
- Utilizar los operadores aritméticos.
- Comprender la precedencia de los operadores aritméticos.
- Escribir instrucciones para tomar decisiones.
- Utilizar los operadores relacionales y de igualdad.

Plan general

2.1 Introducción

En este capítulo le presentaremos la programación de aplicaciones en Java. Empezaremos con ejemplos de programas que muestran mensajes en la pantalla. Después veremos un programa que obtiene dos números de un usuario, que calcula la suma y muestra el resultado. Aprenderá cómo ordenar a la computadora que realice cálculos aritméticos y guarde sus resultados para usarlos más adelante. El último ejemplo en este capítulo demuestra cómo tomar decisiones. La aplicación compara números y después muestra mensajes con los resultados.

Este capítulo utiliza herramientas del JDK para compilar y ejecutar programas. También publicamos videos Dive Into® en `www.deitel.com/books/jhtp9` para que usted pueda empezar a trabajar con los populares entornos de desarrollo integrados Eclipse y NetBeans.

2.2 Su primer programa en Java: imprimir una línea de texto

Una **aplicación** Java es un programa de computadora que se ejecuta cuando usted utiliza el **comando java** para iniciar la máquina virtual de Java (JVM). Más adelante en esta sección hablaremos sobre cómo compilar y ejecutar una aplicación de Java. Primero vamos a considerar una aplicación simple que muestra una línea de texto. En la figura 2.1 se muestra el programa, seguido de un cuadro que muestra su salida. El programa incluye números de línea, que integramos para fines académicos; *no* son parte de un programa en Java. Este ejemplo ilustra varias características importantes. Pronto veremos que la línea 9 se encarga del verdadero trabajo: mostrar la frase `Bienvenido a la programacion en Java!` en la pantalla.

```
1  // Fig. 2.1: Bienvenido1.java
2  // Programa para imprimir texto.
3
4  public class Bienvenido1
5  {
6     // el método main empieza la ejecución de la aplicación en Java
7     public static void main( String[] args )
8     {
9        System.out.println( "Bienvenido a la programacion en Java!" );
10    } // fin del método main
11 } // fin de la clase Bienvenido1
```

```
Bienvenido a la programacion en Java!
```

Fig. 2.1 | Programa para imprimir texto.

Comentarios en sus programas

Insertamos **comentarios** para **documentar los programas** y mejorar su legibilidad. El compilador de Java ignora los comentarios, de manera que la computadora *no* hace nada cuando el programa se ejecuta.

Por convención, comenzamos cada uno de los programas con un comentario, el cual indica el número de figura y el nombre del archivo. El comentario en la línea 1

```
// Fig. 2.1: Bienvenido1.java
```

Empieza con //, lo cual indica que es un **comentario de fin de línea**: termina al final de la línea en la que aparece el carácter //. Un comentario de fin de línea no necesita empezar ésta; también puede estar en medio y continuar hasta el final (como en las líneas 10 y 11). La línea 2

```
// Programa para imprimir texto.
```

es un comentario que describe el propósito del programa.

Java también cuenta con **comentarios tradicionales**, que se pueden distribuir en varias líneas, como en:

```
/* Éste es un comentario tradicional. Se puede
   dividir en varias líneas */
```

Estos comentarios comienzan y terminan con los delimitadores /* y */. El compilador ignora todo el texto entre estos delimitadores. Java incorporó los comentarios tradicionales y los de fin de línea de los lenguajes de programación C y C++, respectivamente. En este libro sólo utilizamos comentarios de fin de línea (//).

Java también cuenta con un tercer tipo de comentarios: los **comentarios Javadoc**, que están delimitados por /** y */. El compilador ignora todo el texto entre los delimitadores. Estos comentarios nos permiten incrustar la documentación de manera directa en nuestros programas, y son el formato preferido en la industria. El **programa de utilería `javadoc`** (parte del Kit de Desarrollo de Java SE) lee esos comentarios y los utiliza para preparar la documentación de su programa, en formato HTML. En el apéndice M, Creación de documentación con Javadoc, demostramos el uso de los comentarios `javadoc` y la herramienta `javadoc`.

Error común de programación 2.1

*Olvidar uno de los delimitadores de un comentario tradicional o Javadoc es un **error de sintaxis**, el cual ocurre cuando el compilador encuentra código que viola las reglas del lenguaje Java (es decir, su sintaxis). Estas reglas son similares a las reglas gramaticales de un lenguaje natural que especifican la estructura de sus oraciones. Los errores de sintaxis se conocen también como **errores del compilador, errores en tiempo de compilación o errores de compilación**, ya que el compilador los detecta durante la fase de compilación. Como respuesta, el compilador emite un mensaje de error y evita que su programa se compile.*

Buena práctica de programación 2.1

Ciertas organizaciones requieren que todos los programas comiencen con un comentario que explique su propósito, el autor, la fecha y la hora de la última modificación del mismo.

Uso de líneas en blanco

La línea 3 es una línea en blanco. Las líneas en blanco, los espacios y las tabulaciones facilitan la lectura de los programas. En conjunto se les conoce como **espacio en blanco**, el cual es ignorado por el compilador.

Buena práctica de programación 2.2

Utilice líneas en blanco y espacios para mejorar la legibilidad del programa.

Declaración de una clase

La línea 4

```
public class Bienvenido1
```

comienza una **declaración de clase** para la clase `Bienvenido1`. Todo programa en Java consiste al menos de una clase que usted (el programador) debe definir. La **palabra clave** `class` introduce una declaración de clase, que debe ir seguida de inmediato por el nombre de la clase (`Bienvenido1`). Las **palabras clave** (también conocidas como **palabras reservadas**) se conservan para uso exclusivo de Java y siempre se escriben en minúscula. En el apéndice C se muestra la lista completa de palabras clave de Java.

Nombres de clases e identificadores

Por convención, todos los nombres de clases comienzan con una letra mayúscula, y la primera letra de cada palabra en el nombre de la clase debe ir en mayúscula (por ejemplo, `EjemploDeNombreDeClase`). El nombre de una clase es un **identificador**: una serie de caracteres que pueden ser letras, dígitos, guiones bajos (_) y signos de moneda ($), que no comience con un dígito ni tenga espacios. Algunos identificadores válidos son: `Bienvenido1`, `$valor`, `_valor`, `m_campoEntrada1` y `boton7`. El nombre `7boton` no es un identificador válido, ya que comienza con un dígito, y el nombre `campo entrada` tampoco lo es debido a que contiene un espacio. Por lo general, un identificador que no empieza con una letra mayúscula no es el nombre de una clase. Java es **sensible a mayúsculas y minúsculas**; es decir, las letras mayúsculas y minúsculas son distintas, por lo que `valor` y `Valor` son distintos identificadores (pero ambos son válidos).

En los capítulos 2 al 7, cada una de las clases que definimos comienza con la palabra clave **`public`**. Por el momento vamos a considerar tan sólo que es obligatoria. Para nuestra aplicación, el nombre del archivo es `Bienvenido1.java`. En el capítulo 8 aprenderá más acerca de las clases `public` y las que no son `public`.

Error común de programación 2.2

Una clase `public` debe colocarse en un archivo que tenga el mismo nombre que la clase (en términos de ortografía y uso de mayúsculas) y la extensión `.java`; en caso contrario, ocurre un error de compilación. Por ejemplo, `public class Bienvenido` se debe colocar en un archivo llamado `Bienvenido.java`.

Una **llave izquierda** (como en la línea 5), {, comienza el cuerpo de todas las declaraciones de clases. Su correspondiente **llave derecha** (en la línea 11), }, debe terminar cada declaración de una clase. Las líneas 6 a 10 tienen sangría.

Tip para prevenir errores 2.1

Cuando escriba una llave izquierda de apertura, {, escriba de inmediato la llave derecha de cierre, }; después vuelva a colocar el cursor entre las dos llaves y aplique sangría para empezar a escribir el cuerpo. Esta práctica ayuda a prevenir errores debido a la falta de llaves. Muchos IDE insertan las llaves por usted.

Error común de programación 2.3

Es un error de sintaxis no utilizar las llaves por pares.

Buena práctica de programación 2.3

Aplique sangría a todo el cuerpo de la declaración de cada clase, usando un "nivel" de sangría entre la llave izquierda y la llave derecha, las cuales delimitan el cuerpo de la clase. Le recomendamos usar tres espacios para formar un nivel de sangría. Este formato enfatiza la estructura de la declaración de la clase, y facilita su lectura.

Buena práctica de programación 2.4

Muchos IDE insertan la sangría por usted en los lugares apropiados. También puede usar la tecla Tab *para aplicar sangría al código, pero las posiciones de los tabuladores varían entre los diversos editores de texto. La mayoría de los IDE le permiten configurar los tabuladores de tal forma que se inserte el número especificado de espacios cada vez que oprima la tecla* Tab.

Declaración de un método

La línea 6

```
// el método main empieza la ejecución de la aplicación en Java
```

es un comentario de fin de línea que indica el propósito de las líneas 7 a 10 del programa. La línea 7

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

es el punto de inicio de toda aplicación en Java. Los **paréntesis** después del identificador `main` indican que éste es un bloque de construcción del programa, al cual se le llama **método**. Las declaraciones de clases en Java por lo general contienen uno o más métodos. En una aplicación en Java, sólo uno de esos métodos *debe* llamarse `main` y hay que definirlo como se muestra en la línea 7; de no ser así, la máquina virtual de Java (JVM) no ejecutará la aplicación. Los métodos pueden realizar tareas y devolver información una vez que las hayan concluido. La palabra clave **`void`** indica que este método *no* devolverá ningún tipo de información. Más adelante veremos cómo puede un método devolver información. Por ahora, sólo copie la primera línea de `main` en sus aplicaciones en Java. En la línea 7, las palabras `String[] args` entre paréntesis son una parte requerida de la declaración del método `main`; hablaremos sobre esto en el capítulo 7.

La llave izquierda en la línea 8 comienza el **cuerpo de la declaración del método**. Su correspondiente llave derecha debe terminarlo (línea 10). La línea 9 en el cuerpo del método tiene sangría entre las llaves.

Buena práctica de programación 2.5

Aplique sangría a todo el cuerpo de la declaración de cada método, usando un "nivel" de sangría entre las llaves que delimitan el cuerpo del método. Este formato resalta la estructura del método y ayuda a que su declaración sea más fácil de leer.

Operaciones de salida con `System.out.println`

La línea 9

```
System.out.println( "Bienvenido a la programacion en Java!" );
```

indica a la computadora que realice una acción; es decir, que imprima la **cadena** de caracteres contenida entre los caracteres de comillas dobles (sin incluirlas). A una cadena también se le denomina **cadena de caracteres** o **literal de cadena**. El compilador *no* ignora los caracteres de espacio en blanco dentro de las cadenas. Éstas no pueden abarcar varias líneas de código, pero como veremos más adelante, no impide que usemos cadenas largas en nuestro código.

`System.out` se conoce como el **objeto de salida estándar**. Permite a las aplicaciones en Java mostrar información en la **ventana de comandos** desde la cual se ejecutan. En versiones recientes de Microsoft Windows, la ventana de comandos es el **Símbolo del sistema**. En UNIX/Linux/Mac OS X, la ventana de comandos se llama **ventana de terminal** o **shell**. Muchos programadores se refieren a la ventana de comandos simplemente como la **línea de comandos**.

El método **`System.out.println`** muestra (o imprime) una línea de texto en la ventana de comandos. La cadena dentro de los paréntesis en la línea 9 es el **argumento** para el método. El método `System.out.println` completa su tarea, posiciona el cursor de salida (la ubicación en donde se mostrará el siguiente carácter) al principio de la siguiente línea en la ventana de comandos. Esto es similar a lo que ocurre

cuando un usuario oprime la tecla *Intro*, al escribir en un editor de texto: el cursor aparece al principio de la siguiente línea en el documento.

Toda la línea 9, incluyendo `System.out.println`, el argumento "`Bienvenido a la programacion en Java!`" entre paréntesis y el **punto y coma** (;), se conoce como una **instrucción**. Cada instrucción termina con un punto y coma. Por lo general, un método contiene una o más instrucciones que realizan su tarea. La mayoría de las instrucciones terminan con punto y coma. Cuando se ejecuta la instrucción de la línea 9, muestra el mensaje `Bienvenido a la programacion en Java!` en la ventana de comandos.

Tip para prevenir errores 2.2

Al aprender a programar, algunas veces es conveniente "descomponer" un programa funcional, para que de esta manera pueda familiarizarse con los mensajes de error de sintaxis del compilador. Éstos no siempre indican el problema exacto en el código. Cuando se encuentre con un mensaje de error, le dará una idea de qué fue lo que lo ocasionó [trate de quitar un punto y coma o una llave del programa de la figura 2.1, y vuelva a compilarlo de manera que pueda ver los mensajes de error que se generan debido a esta omisión].

Tip para prevenir errores 2.3

Cuando el compilador reporta un error de sintaxis, éste tal vez no se encuentre en el número de línea indicado por el mensaje de error. Primero verifique la línea en la que se reportó el error; si esa línea no contiene errores de sintaxis, verifique las anteriores.

Uso de los comentarios de fin de línea en las llaves derechas para mejorar la legibilidad

Incluimos un comentario de fin de línea después de una llave derecha de cierre que termina la declaración de un método y después de una llave de cierre que finaliza la declaración de una clase. Por ejemplo, la línea 10

```
} // fin del método main
```

especifica la llave derecha de cierre del método `main`, y la línea 11

```
} // fin de la clase Bienvenido1
```

especifica la llave derecha de cierre de la clase `Bienvenido1`. Cada comentario indica el método o la clase que termina con esa llave derecha.

Compilación y ejecución de su primera aplicación de Java

Ahora estamos listos para compilar y ejecutar nuestro programa. Vamos a suponer que usted utilizará las herramientas de línea de comandos del Kit de Desarrollo de Java y no un IDE. En nuestros Centros de Recursos de Java en `www.deitel.com/ResourceCenters.html` proporcionamos vínculos a tutoriales que le ayudarán a empezar a trabajar con varias herramientas de desarrollo populares de Java, entre ellas NetBeans™, Eclipse™ y otras. También publicamos videos de NetBeans y Eclipse en `www.deitel.com/books/jhtp9/` para ayudarle a empezar a utilizar estos populares IDE.

Para compilar el programa, abra una ventana de comandos y cambie al directorio en donde está guardado el programa. La mayoría de los sistemas operativos utilizan el comando `cd` para cambiar directorios. Por ejemplo, en Windows el comando,

```
cd c:\ejemplos\cap02\fig02_01
```

cambia al directorio `fig02_01`. En UNIX/Linux/Mac OS X, el comando

```
cd ~/ejemplos/cap02/fig02_01
```

cambia al directorio `fig02_01`.

Para compilar el programa, escriba

```
javac Bienvenido1.java
```

Si el programa no contiene errores de sintaxis, el comando anterior crea un nuevo archivo llamado `Bienvenido1.class` (conocido como el **archivo de clase** para `Bienvenido1`), el cual contiene los códigos de bytes de Java independientes de la plataforma, que representan nuestra aplicación. Cuando utilicemos el comando java para ejecutar la aplicación en una plataforma específica, la JVM traducirá estos códigos de bytes en instrucciones que el sistema operativo y el hardware subyacentes puedan comprender.

Tip para prevenir errores 2.4

Cuando trate de compilar un programa, si recibe un mensaje como "`comando o nombre de archivo incorrecto`," "`javac: comando no encontrado`" o "`'javac' no se reconoce como un comando interno o externo, programa o archivo por lotes ejecutable`," entonces su instalación del software de Java no se completó en forma apropiada. Si utiliza el JDK, esto indica que la variable de entorno `PATH` *del sistema no se estableció de manera apropiada. Consulte con cuidado las instrucciones de instalación en la sección Antes de empezar este libro. En algunos sistemas, después de corregir la variable* `PATH`*, es probable que necesite reiniciar su equipo o abrir una nueva ventana de comandos para que estos ajustes tengan efecto.*

Tip para prevenir errores 2.5

Cada mensaje de error de sintaxis contiene el nombre de archivo y el número de línea en donde ocurrió el error. Por ejemplo, `Bienvenido1.java:6` *indica que ocurrió un error en la línea 6 del archivo* `Bienvenido1.java`*. El resto del mensaje proporciona información acerca del error de sintaxis.*

Tip para prevenir errores 2.6

El mensaje de error del compilador "`class Bienvenido1 is public, should be declared in a file named Welcome1.java`" indica que el nombre del archivo no coincide con el nombre de la clase public en el archivo, o que escribió mal el nombre de la clase al momento de compilarla.

La figura 2.2 muestra el programa de la figura 2.1 al ejecutarse en una ventana **Símbolo del sistema** de Microsoft® Windows® 7. Para ejecutar el programa, escriba `java Bienvenido1`. Este comando inicia la JVM, que a su vez carga el archivo `.class` para la clase `Bienvenido1`. El comando omite la extensión `.class` del nombre de archivo; de no ser así, la JVM no ejecutará el programa. La JVM llama al método `main`. A continuación, la instrucción de la línea 9 de `main` muestra "`Bienvenido a la programacion en Java!`" [*nota:* muchos entornos muestran los símbolos del sistema con fondos negros y texto blanco. En nuestro entorno ajustamos esta configuración para que nuestras capturas de pantalla fueran más legibles].

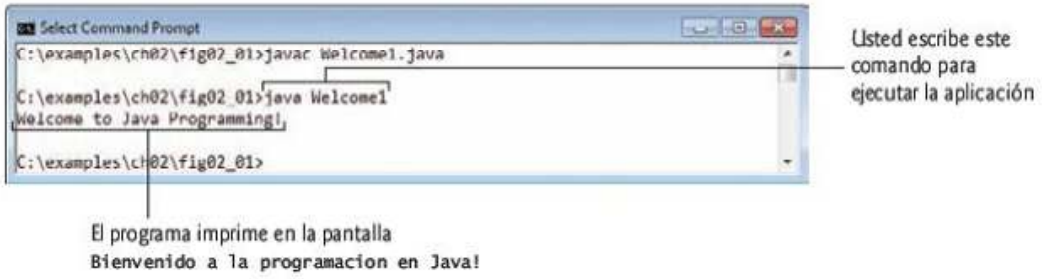


Fig. 2.2 | Ejecución de `Bienvenido1` desde el **Símbolo del sistema**.

Tip para prevenir errores 2.7

Al tratar de ejecutar un programa en Java, si recibe un mensaje como "Exception in thread "main" java.lang.NoClassDefFoundError: Bienvenido1," quiere decir que su variable de entorno CLASSPATH no está configurada de manera correcta. Consulte con cuidado las instrucciones de instalación en la sección Antes de empezar este libro. En algunos sistemas, tal vez necesite reiniciar su equipo o abrir una nueva ventana de comandos para que estos ajustes tengan efecto.

2.3 Modificación de nuestro primer programa en Java

En esta sección modificaremos el ejemplo de la figura 2.1 para imprimir texto en una línea mediante el uso de varias instrucciones, y para imprimir texto en varias líneas mediante una sola instrucción.

Cómo mostrar una sola línea de texto con varias instrucciones

Es posible mostrar la línea de texto `Bienvenido a la programacion en Java!` de varias formas. La clase `Bienvenido2`, que se muestra en la figura 2.3, utiliza dos instrucciones (líneas 9 y 10) para producir el resultado que se muestra en la figura 2.1. [*Nota:* de aquí en adelante, resaltaremos las características nuevas y las características clave en cada listado de código, como se muestra en las líneas 9 y 10 de este programa].

```
1  // Fig. 2.3: Bienvenido2.java
2  // Imprimir una línea de texto con varias instrucciones.
3
4  public class Bienvenido2
5  {
6     // el método main empieza la ejecución de la aplicación en Java
7     public static void main( String[] args )
8     {
9        System.out.print( "Bienvenido a " );
10       System.out.println( "la programacion en Java!" );
11    } // fin del método main
12 } // fin de la clase Bienvenido2
```

```
Bienvenido a la programacion en Java!
```

Fig. 2.3 | Impresión de una línea de texto con varias instrucciones.

El programa es similar al de la figura 2.1, por lo que aquí sólo hablaremos de los cambios. La línea 2

```
// Imprimir una línea de texto con varias instrucciones.
```

es un comentario de fin de línea que describe el propósito de este programa. La línea 4 comienza la declaración de la clase `Bienvenido2`. Las líneas 9 y 10 del método `main`

```
System.out.print( "Bienvenido a " );
System.out.println( "la programacion en Java!" );
```

muestran una línea de texto. La primera instrucción utiliza el método `print` de `System.out` para mostrar una cadena. Cada instrucción `print` o `println` continúa mostrando caracteres a partir de donde la última instrucción `print` o `println` dejó de mostrarlos. A diferencia de `println`, después de mostrar su argumento, `print` *no* posiciona el cursor de salida al inicio de la siguiente línea en la ventana de comandos; el siguiente carácter que muestra el programa en la ventana de comandos aparecerá *justo después* del último carácter que muestre `print`. Por lo tanto, la línea 10 coloca el primer carácter de su argumento

(la letra "l") inmediatamente después del último carácter que muestra la línea 9 (el *carácter de espacio* antes del carácter de comilla doble de cierre de la cadena).

Cómo mostrar varias líneas de texto con una sola instrucción

Una sola instrucción puede mostrar varias líneas mediante el uso de los **caracteres de nueva línea**, los cuales indican a los métodos `print` y `println` de `System.out` cuándo deben colocar el cursor de salida al inicio de la siguiente línea en la ventana de comandos. Al igual que las líneas en blanco, los espacios y los tabuladores, los caracteres de nueva línea son caracteres de espacio en blanco. El programa de la figura 2.4 muestra cuatro líneas de texto mediante el uso de caracteres de nueva línea para determinar cuándo empezar cada nueva línea. La mayor parte del programa es idéntico a los de las figuras 2.1 y 2.3.

```
1  // Fig. 2.4: Bienvenido3.java
2  // Imprimir varias lineas de texto con una sola instrucción.
3
4  public class Bienvenido3
5  {
6     // el método main empieza la ejecución de la aplicación en Java
7     public static void main( String[] args )
8     {
9        System.out.println( "Bienvenido\na\nla programacion\nen Java!" );
10    } // fin del método main
11 } // fin de la clase Bienvenido3
```

```
Bienvenido
a la
programacion
en Java!
```

Fig. 2.4 | Impresión de varias líneas de texto con una sola instrucción.

La línea 2

```
// Imprimir varias lineas de texto con una sola instrucción.
```

es un comentario que describe el propósito de este programa. La línea 4 comienza la declaración de la clase `Bienvenido3`.

La línea 9

```
System.out.println( "Bienvenido\na\nla programacion\nen Java!" );
```

muestra cuatro líneas separadas de texto en la ventana de comandos. Por lo general, los caracteres en una cadena se muestran *justo* como aparecen en las comillas dobles. Sin embargo, observe que los dos caracteres \ y n (que se repiten tres veces en la instrucción) no aparecen en la pantalla. La **barra diagonal inversa** (\) se conoce como **carácter de escape**, el cual tiene un significado especial para los métodos `print` y `println` de `System.out`. Cuando aparece una barra diagonal inversa en una cadena de caracteres, Java combina el siguiente carácter con la barra diagonal inversa para formar una **secuencia de escape**. La secuencia de escape `\n` representa el carácter de nueva línea. Cuando aparece un carácter de nueva línea en una cadena que se va a imprimir con `System.out`, el carácter de nueva línea hace que el cursor de salida de la pantalla se desplace al inicio de la siguiente línea en la ventana de comandos.

En la figura 2.5 se enumeran varias secuencias de escape comunes, con descripciones de cómo afectan la manera de mostrar caracteres en la ventana de comandos. Para obtener una lista completa de secuencias de escape, visite `java.sun.com/docs/books/jls/third_edition/html/lexical.html#3.10.6`.

Secuencia de escape	Descripción
`\n`	Nueva línea. Coloca el cursor de la pantalla al inicio de la siguiente línea.
`\t`	Tabulador horizontal. Desplaza el cursor de la pantalla hasta la siguiente posición de tabulación.
`\r`	Retorno de carro. Coloca el cursor de la pantalla al inicio de la línea actual; *no* avanza a la siguiente línea. Cualquier carácter que se imprima después del retorno de carro sobrescribe los caracteres previamente impresos en esa línea.
`\\`	Barra diagonal inversa. Se usa para imprimir un carácter de barra diagonal inversa.
`\"`	Doble comilla. Se usa para imprimir un carácter de doble comilla. Por ejemplo, `System.out.println( "\"entre comillas\"" );` displays `"entre comillas"`.

Fig. 2.5 | Algunas secuencias de escape comunes.

2.4 Cómo mostrar texto con `printf`

El método `System.out.printf` ("f" significa "formato") muestra datos con formato. La figura 2.6 usa este método para mostrar las cadenas "`Bienvenido a`" y "`la programacion en Java!`". Las líneas 9 y 10

```
System.out.printf( "%s\n%s\n",
   "Bienvenido a", "la programacion en Java!" );
```

llaman al método `System.out.printf` para mostrar la salida del programa. La llamada al método especifica tres argumentos. Cuando un método requiere varios argumentos, éstos se colocan en una **lista separada por comas**.

Buena práctica de programación 2.6

Coloque un espacio después de cada coma (,) en una lista de argumentos para que sus programas sean más legibles.

```
1  // Fig. 2.6: Bienvenido4.java
2  // Imprimir varias líneas con el método System.out.printf.
3
4  public class Bienvenido4
5  {
6     // el método main empieza la ejecución de la aplicación de Java
7     public static void main( String[] args )
8     {
9        System.out.printf( "%s\n%s\n",
10          "Bienvenido a", "la programacion en Java!" );
11    } // fin del método main
12 } // fin de la clase Bienvenido4
```

```
Bienvenido a
la programacion en Java!
```

Fig. 2.6 | Imprimir varias líneas de texto con el método `System.out.printf`.

Las líneas 9 y 10 representan sólo *una* instrucción. Java permite dividir instrucciones extensas en varias líneas. Aplicamos sangría a la línea 10 para indicar que es la *continuación* de la línea 9.

Error común de programación 2.4

Dividir una instrucción a la mitad de un identificador o de una cadena es un error de sintaxis.

El primer argumento del método `printf` es una **cadena de formato** que puede consistir en **texto fijo** y **especificadores de formato**, este método imprime el texto fijo de igual forma que `print` o `println`. Cada especificador de formato es un receptáculo para un valor, y especifica el tipo de datos a imprimir. Los especificadores de formato también pueden incluir información de formato opcional.

Empiezan con un signo porcentual (%) y van seguidos de un carácter que representa el tipo de datos. Por ejemplo, el especificador de formato `%s` es un receptáculo para una cadena. La cadena de formato en la línea 9 especifica que `printf` debe imprimir dos cadenas, y que a cada una de ellas le debe seguir un carácter de nueva línea. En la posición del primer especificador de formato, `printf` sustituye el valor del primer argumento después de la cadena de formato. En cada posición posterior del especificador de formato, `printf` sustituye el valor del siguiente argumento. Así, este ejemplo sustituye "`Bienvenido a`" por el primer `%s` y "`la programacion en Java!`" por el segundo `%s`. La salida muestra que se despliegan dos líneas de texto en pantalla.

En nuestros ejemplos presentaremos las diversas características de formato a medida que se vayan necesitando. El apéndice G presenta los detalles de cómo dar formato a la salida con `printf`.

2.5 Otra aplicación en Java: suma de enteros

Nuestra siguiente aplicación lee (o recibe como entrada) dos **enteros** (números completos, como −22, 7, 0 y 1024) que el usuario introduce mediante el teclado, después calcula la suma de los valores y muestra el resultado. Este programa debe llevar la cuenta de los números que suministra el usuario para los cálculos que el programa realiza posteriormente. Los programas recuerdan números y otros datos en la memoria de la computadora, y acceden a ellos a través de unos elementos conocidos como **variables**. El programa de la figura 2.7 demuestra estos conceptos. En la salida de ejemplo, usamos texto en negritas para identificar la entrada del usuario (por ejemplo, **45** y **72**).

```
1  // Fig. 2.7: Suma.java
2  // Programa que muestra la suma de dos números.
3  import java.util.Scanner; // el programa usa la clase Scanner
4
5  public class Suma
6  {
7     // el método main empieza la ejecución de la aplicación en Java
8     public static void main( String[] args )
9     {
10       // crea objeto Scanner para obtener la entrada de la ventana de comandos
11       Scanner entrada = new Scanner( System.in );
12
13       int numero1; // primer número a sumar
14       int numero2; // segundo número a sumar
15       int suma; // suma de numero1 y numero2
16
```

Fig. 2.7 | Programa que muestra la suma de dos números (parte 1 de 2).

```
17        System.out.print( "Escriba el primer entero: " ); // indicador
18        numero1 = entrada.nextInt(); // lee el primer número del usuario
19
20        System.out.print( "Escriba el segundo entero: " ); // indicador
21        numero2 = entrada.nextInt(); // lee el segundo número del usuario
22
23        suma = numero1 + numero2; // suma los números, después almacena el total en suma
24
25        System.out.printf( "La suma es %d\n", suma ); // muestra la suma
26     } // fin del método main
27  } // fin de la clase Suma
```

```
Escriba el primer entero: 45
Escriba el segundo entero: 72
La suma es 117
```

Fig. 2.7 | Programa que muestra la suma de dos números (parte 2 de 2).

Declaraciones `import`

Las líneas 1 y 2

```
// Fig. 2.7: Suma.java
// Programa que muestra la suma de dos números.
```

indican el número de la figura, el nombre del archivo y el propósito del programa.

Una gran fortaleza de Java es su extenso conjunto de clases predefinidas que podemos *reutilizar*, en vez de "reinventar la rueda". Estas clases se agrupan en **paquetes** (grupos con nombre de clases relacionadas) y se conocen en conjunto como la **biblioteca de clases de Java**, o **Interfaz de programación de aplicaciones de Java (API de Java)**. La línea 3

```
import java.util.Scanner; // el programa usa la clase Scanner
```

es una **declaración `import`** que ayuda al compilador a localizar una clase que se utiliza en este programa. Indica que este ejemplo utiliza la clase `Scanner` predefinida de Java (que veremos en breve) del paquete `java.util`.

Error común de programación 2.5

Todas las declaraciones `import` deben aparecer antes de la declaración de la primera clase en el archivo. Colocar una declaración `import` dentro del cuerpo de la declaración de una clase, o después de ésta, es un error de sintaxis.

Tip para prevenir errores 2.8

Por lo general, si olvida incluir una declaración `import` para una clase que utilice en su programa, se produce un error de compilación que contiene un mensaje como "`cannot find symbol`". Cuando esto ocurra, verifique que haya proporcionado las declaraciones `import` apropiadas y que los nombres en las mismas estén escritos en forma correcta, que se hayan usado las letras mayúsculas y minúsculas como se debe.

Declaración de la clase `Suma`

La línea 5

```
public class Suma
```

empieza la declaración de la clase `Suma`. El nombre de archivo para esta clase `public` debe ser `Suma.java`. Recuerde que el cuerpo de cada declaración de clase empieza con una llave izquierda de apertura (línea 6) y termina con una llave derecha de cierre (línea 27).

La aplicación empieza a ejecutarse con el método `main` (líneas 8 a la 26). La llave izquierda (línea 9) marca el inicio del cuerpo de `main`, y la correspondiente llave derecha (línea 26) marca su final. Observe que al método `main` se le aplica un nivel de sangría en el cuerpo de la clase `Suma`, y que al código en el cuerpo de `main` se le aplica otro nivel para mejorar la legibilidad.

Declaración y creación de un objeto `Scanner` *para obtener la entrada del usuario mediante el teclado*

Una **variable** es una ubicación en la memoria de la computadora, en donde se puede guardar un valor para utilizarlo después en un programa. Todas las variables *deben* declararse con un **nombre** y un **tipo** *antes* de poder usarse. El nombre de ésta, que puede ser cualquier identificador válido, permite al programa acceder al valor de la variable en memoria. El tipo de una variable especifica la clase de información que se guarda en esa ubicación de memoria. Al igual que las demás instrucciones, las instrucciones de declaración terminan con punto y coma (;).

La línea 11

```
Scanner entrada = new Scanner( System.in );
```

es una **instrucción de declaración de variable** que especifica el nombre (`entrada`) y tipo (`Scanner`) de una variable que se utiliza en este programa. Un objeto **`Scanner`** permite a un programa leer datos (por ejemplo: números y cadenas) para usarlos en un programa. Los datos pueden provenir de muchas fuentes, como un archivo en disco o desde el teclado de un usuario. Antes de usar un objeto `Scanner`, hay que crearlo y especificar el origen de la información.

El signo = en la línea 11 indica que es necesario **inicializar** la variable `entrada` tipo `Scanner` (es decir, hay que prepararla para usarla en el programa) en su declaración con el resultado de la expresión a la derecha del signo igual: `new Scanner(System.in)`. Esta expresión usa la palabra clave **`new`** para crear un objeto `Scanner` que lee los datos escritos por el usuario mediante el teclado. El **objeto de entrada estándar, `System.in`**, permite a las aplicaciones de Java leer la información escrita por el usuario. El objeto `Scanner` traduce estos bytes en tipos (como `int`) que se pueden usar en un programa.

Declaración de variables para almacenar enteros

Las instrucciones de declaración de variables en las líneas 13 a la 15

```
int numero1; // primer número a sumar
int numero2; // segundo número a sumar
int suma; // suma de numero1 y numero2
```

declaran que las variables `numero1`, `numero2` y `suma` contienen datos de tipo **`int`**; estas variables pueden contener valores enteros (números completos, como `72`, `-1127` y `0`). Estas variables no se han inicializado todavía. El rango de valores para un `int` es de –2,147,483,648 a +2,147,483,647 [*nota:* los valores `int` reales tal vez no contengan comas].

Hay otros tipos de datos como **`float`** y **`double`**, para guardar números reales, y el tipo **`char`**, para guardar datos de caracteres. Los números reales son números que contienen puntos decimales, como `3.4`, `0.0` y `-11.19`. Las variables de tipo `char` representan caracteres individuales, como una letra en mayúscula (`A`), un dígito (`7`), un carácter especial (`*` o `%`) o una secuencia de escape (como el carácter de nueva línea, `\n`). Los tipos tales como `int`, `float`, `double` y `char` se conocen como **tipos primitivos**. Los nombres de los tipos primitivos son palabras clave y deben aparecer completamente en minúsculas. El apéndice D sintetiza las características de los ocho tipos primitivos (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float` y `double`).

Es posible declarar varias variables del mismo tipo en una sola declaración, en donde los nombres de las variables se separan por comas (es decir, una lista de nombres de variables separados por comas). Por ejemplo, las líneas 13 a la 15 se pueden escribir también así:

```
int numero1, // primer número a sumar
    numero2, // segundo número a sumar
    suma; // suma de numero1 y numero2
```

Buena práctica de programación 2.7

Declare cada variable en una línea separada. Este formato permite insertar un comentario descriptivo en seguida de cada declaración.

Buena práctica de programación 2.8

Seleccionar nombres de variables significativos ayuda a que un programa se autodocumente (es decir, que sea más fácil entender el programa con sólo leerlo, en lugar de leer manuales o ver un número excesivo de comentarios).

Buena práctica de programación 2.9

Por convención, los identificadores de nombre de variables empiezan con una letra minúscula, y cada una de las palabras en el nombre, que van después de la primera, deben empezar con una mayúscula. Por ejemplo, el identificador `primerNumero` *tiene una* `N` *mayúscula en su segunda palabra,* `Numero`.

Cómo pedir la entrada al usuario

La línea 17

```
System.out.print( "Escriba el primer entero: " ); // indicador
```

utiliza `System.out.print` para mostrar el mensaje "`Escriba el primer entero:`". Este mensaje se conoce como **indicador**, ya que indica al usuario que debe realizar una acción específica. En este ejemplo utilizamos el método `print` en vez de `println` para que la entrada del usuario aparezca en la misma línea que la del indicador. En la sección 2.2 vimos que, por lo general, los identificadores que empiezan con letras mayúsculas representan nombres de clases. Por lo tanto, `System` es una clase. La clase `System` forma parte del paquete `java.lang`. Cabe mencionar que la clase `System` no se importa con una declaración `import` al principio del programa.

Observación de ingeniería de software 2.1

El paquete `java.lang` *se importa de manera predeterminada en todos los programas de Java; por ende, las clases en* `java.lang` *son las únicas en la API de Java que no requieren una declaración* `import`.

Cómo obtener un valor `int` *como entrada del usuario*

La línea 18

```
numero1 = entrada.nextInt(); // lee el primer número del usuario
```

utiliza el método `nextInt` del objeto `entrada` de la clase `Scanner` para obtener un entero del usuario mediante el teclado. En este punto, el programa espera a que el usuario escriba el número y oprima *Intro* para enviar el número al programa.

Nuestro programa asume que el usuario escribirá un valor de entero válido. De no ser así, se producirá un error lógico en tiempo de ejecución y el programa terminará. El capítulo 11, Manejo de excepciones: un análisis más detallado, habla sobre cómo hacer sus programas más robustos al permitirles manejar dichos errores. Esto también se conoce como hacer que su programa sea *tolerante a fallas*.

En la línea 18, colocamos el resultado de la llamada al método `nextInt` (un valor `int`) en la variable `numero1` mediante el uso del **operador de asignación**, `=`. La instrucción se lee como "`numero1` obtiene el valor de `entrada.nextInt()`". Al operador `=` se le llama **operador binario**, ya que tiene dos **operandos**: `numero1` y el resultado de la llamada al método `entrada.nextInt()`. Esta instrucción se llama instrucción de asignación, ya que asigna un valor a una variable. Todo lo que está a la *derecha* del operador de asignación (`=`) se evalúa siempre *antes* de realizar la asignación.

Buena práctica de programación 2.10

Al colocar espacios en cualquier lado de un operador binario, mejora la legibilidad del programa.

Cómo pedir e introducir un segundo `int`

La línea 20

```
System.out.print( "Escriba el segundo entero: " ); // indicador
```

pide al usuario que escriba el segundo entero. La línea 21

```
numero2 = entrada.nextInt(); // lee el segundo número del usuario
```

lee el segundo entero y lo asigna a la variable `numero2`.

Uso de variables en un cálculo

La línea 23

```
suma = numero1 + numero2; // suma los números, después almacena el total en suma
```

es una instrucción de asignación que calcula la suma de las variables `numero1` y `numero2`, y asigna el resultado a la variable `suma` mediante el uso del operador de asignación, `=`. La instrucción se lee como "`suma` *obtiene* el valor de `numero1 + numero2`". En general, los cálculos se realizan en instrucciones de asignación. Cuando el programa encuentra la operación de suma, utiliza los valores almacenados en las variables `numero1` y `numero2` para realizar el cálculo. En la instrucción anterior, el operador de suma es un *operador binario*; sus *dos* operandos son las variables `numero1` y `numero2`. Las partes de las instrucciones que contienen cálculos se llaman **expresiones**. De hecho, una expresión es cualquier parte de una instrucción que tiene un *valor* asociado. Por ejemplo, el valor de la expresión `numero1 + numero2` es la *suma* de los números. De manera similar, el valor de la expresión `entrada.nextInt()` es el entero escrito por el usuario.

Cómo mostrar el resultado del cálculo

Una vez realizado el cálculo, la línea 25

```
System.out.printf( "La suma es %d\n", suma ); // muestra la suma
```

utiliza el método `System.out.printf` para mostrar la `suma`. El especificador de formato **`%d`** es un receptáculo para un valor `int` (en este caso, el valor de `suma`); la letra `d` se refiere a "entero decimal". El resto de los caracteres en la cadena de formato son texto fijo. Por lo tanto, el método `printf` imprime en pantalla "`La suma es`", seguido del valor de `suma` (en la posición del especificador de formato `%d`) y una nueva línea.

También es posible realizar cálculos dentro de *instrucciones* `printf`. Podríamos haber combinado las instrucciones de las líneas 23 y 25 en la siguiente instrucción:

```
System.out.printf( "La suma es %d\n", ( numero1 + numero2 ) );
```

Los paréntesis alrededor de la expresión `numero1 + numero2` no son requeridos; se incluyen para enfatizar que el valor de *toda* la expresión se imprime en la posición del especificador de formato `%d`.

Documentación de la API de Java

Para cada nueva clase de la API de Java que utilicemos, hay que indicar el paquete en el que se ubica. Esta información nos ayuda a localizar las descripciones de cada paquete y clase en la documentación de la API de Java. Puede encontrar una versión basada en Web de esta documentación en

```
download.oracle.com/javase/6/docs/api/
```

También puede descargar esta documentación de

```
www.oracle.com/technetwork/java/javase/downloads/index.html
```

El apéndice E muestra cómo utilizar esta documentación.

2.6 Conceptos acerca de la memoria

Los nombres de variables como `numero1`, `numero2` y `suma` en realidad corresponden a ciertas ubicaciones en la memoria de la computadora. Toda variable tiene un **nombre**, un **tipo**, un **tamaño** (en bytes) y un **valor**.

En el programa de suma de la figura 2.7, cuando se ejecuta la instrucción (línea 18):

```
numero1 = entrada.nextInt(); // lee el primer número del usuario
```

el número escrito por el usuario se coloca en una ubicación de memoria que corresponde al nombre `numero1`. Suponga que el usuario escribe `45`. La computadora coloca ese valor entero en la ubicación `numero1` (figura 2.8) y sustituye al valor anterior en esa ubicación (si había uno). El valor anterior se pierde.

Fig. 2.8 | Ubicación de memoria que muestra el nombre y el valor de la variable `numero1`.

Cuando se ejecuta la instrucción (línea 21)

```
numero2 = entrada.nextInt(); // lee el segundo número del usuario
```

suponga que el usuario escribe `72`. La computadora coloca ese valor entero en la ubicación `numero2`. La memoria ahora aparece como se muestra en la figura 2.9.

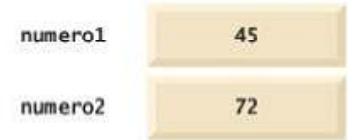


Fig. 2.9 | Ubicaciones de memoria, después de almacenar valores para `numero1` y `numero2`.

Una vez que el programa de la figura 2.7 obtiene valores para `numero1` y `numero2`, los suma y coloca el total en la variable `suma`. La instrucción (línea 23)

```
suma = numero1 + numero2; // suma los números, después almacena el total en suma
```

realiza la suma y después sustituye el valor anterior de suma. Una vez que se calcula suma, la memoria aparece como se muestra en la figura 2.10. Observe que los valores de numero1 y numero2 aparecen exactamente como antes de usarlos en el cálculo de suma. Estos valores se utilizaron, pero no se destruyeron, cuando la computadora realizó el cálculo. Por ende, cuando se lee un valor de una ubicación de memoria, el proceso es no destructivo.

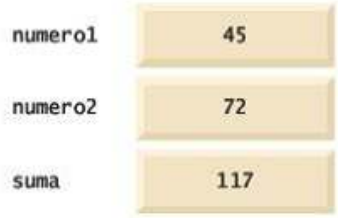


Fig. 2.10 | Ubicaciones de memoria, después de almacenar la suma de numero1 y numero2.

2.7 Aritmética

La mayoría de los programas realizan cálculos aritméticos. Los **operadores aritméticos** se sintetizan en la figura 2.11. Observe el uso de varios símbolos especiales que no se utilizan en álgebra. El **asterisco** (*) indica la multiplicación, y el signo de porcentaje (%) es el **operador residuo**, el cual describiremos en breve. Los operadores aritméticos en la figura 2.11 son operadores *binarios*, ya que funcionan con *dos* operandos. Por ejemplo, la expresión f + 7 contiene el operador binario + y los dos operandos f y 7.

Operación en Java	Operador	Expresión algebraica	Expresión en Java
Suma	+	$f + 7$	f + 7
Resta	-	$p - c$	p - c
Multiplicación	*	bm	b * m
División	/	x/y o $\frac{x}{y}$ o $x \sqrt{y}$	x / y
Residuo	%	$r \bmod s$	r % s

Fig. 2.11 | Operadores aritméticos.

La **división de enteros** produce un cociente entero. Por ejemplo, la expresión 7 / 4 da como resultado 1, y la expresión 17 / 5 da como resultado 3. Cualquier parte fraccionaria en una división de enteros simplemente se *descarta* (se *trunca*); no ocurre un redondeo. Java proporciona el operador residuo, %, el cual produce el residuo después de la división. La expresión x % y produce el residuo después de que x se divide entre y. Por lo tanto, 7 % 4 produce 3, y 17 % 5 produce 2. Este operador se utiliza más comúnmente con operandos enteros, pero también puede usarse con otros tipos aritméticos. En los ejercicios de este capítulo y de capítulos posteriores, consideramos muchas aplicaciones interesantes del operador residuo, como determinar si un número es múltiplo de otro.

Expresiones aritméticas en formato de línea recta

Las expresiones aritméticas en Java deben escribirse en **formato de línea recta** para facilitar la escritura de programas en la computadora. Por lo tanto, las expresiones como "a dividida entre b" deben escribirse como `a/b`, de manera que todas las constantes, variables y operadores aparezcan en una línea recta. La siguiente notación algebraica por lo general no es aceptable para los compiladores:

$$\frac{a}{b}$$

Paréntesis para agrupar subexpresiones

Los paréntesis se utilizan para agrupar términos en las expresiones en Java, de la misma manera que en las expresiones algebraicas. Por ejemplo, para multiplicar a por la cantidad `b + c`, escribimos

```
a * ( b + c )
```

Si una expresión contiene **paréntesis anidados**, como

```
( ( a + b ) * c )
```

se evalúa primero la expresión en el conjunto más interno de paréntesis (`a + b` en este caso).

Reglas de precedencia de operadores

Java aplica los operadores en expresiones aritméticas en una secuencia precisa, determinada por las siguientes **reglas de precedencia de operadores**, que casi siempre son las mismas que las que se utilizan en álgebra:

1. Las operaciones de multiplicación, división y residuo se aplican primero. Si una expresión contiene varias de esas operaciones, los operadores se aplican de izquierda a derecha. Los operadores de multiplicación, división y residuo tienen el mismo nivel de precedencia.

2. Las operaciones de suma y resta se aplican a continuación. Si una expresión contiene varias de esas operaciones, los operadores se aplican de izquierda a derecha. Los operadores de suma y resta tienen el mismo nivel de precedencia.

Estas reglas permiten a Java aplicar los operadores en el orden correcto.[1] Cuando decimos que los operadores se aplican de izquierda a derecha, nos referimos a su **asociatividad**. Algunos se asocian de derecha a izquierda. La figura 2.12 sintetiza estas reglas de precedencia de operadores. En el apéndice A se presenta una tabla de precedencias completa.

Operador(es)	Operación(es)	Orden de evaluación (precedencia)

`*`
`/`
`%`
	Multiplicación
	División
	Residuo
		Se evalúan primero. Si hay varios operadores de este tipo, se evalúan de izquierda a derecha.

`+` `-`	Suma Resta	Se evalúan después. Si hay varios operadores de este tipo, se evalúan de izquierda a derecha.
`=`	Asignación	Se evalúa al último.

Fig. 2.12 | Precedencia de los operadores aritméticos.

1 Utilizamos ejemplos simples para explicar el orden de evaluación de las expresiones. Se producen ligeros errores en las expresiones más complejas que veremos más adelante en el libro. Para obtener más información sobre el orden de evaluación, vea el capítulo 15 de *The Java™ Language Specification* (`java.sun.com/docs/books/jls/`).

Ejemplos de expresiones algebraicas y de Java

Ahora consideremos varias expresiones en vista de las reglas de precedencia de operadores. Cada ejemplo enlista una expresión algebraica y su equivalente en Java. El siguiente es un ejemplo de una media (promedio) aritmética de cinco términos:

Álgebra: $m = \frac{a + b + c + d + e}{5}$

Java:
```
m = ( a + b + c + d + e ) / 5;
```

Los paréntesis son obligatorios, ya que la división tiene una mayor precedencia que la suma. La cantidad completa `(a + b + c + d + e)` va a dividirse entre `5`. Si por error se omiten los paréntesis, obtenemos `a + b + c + d + e / 5`, lo cual da como resultado

$$a + b + c + d + \frac{e}{5}$$

El siguiente es un ejemplo de una ecuación de línea recta:

Álgebra: $y = mx + b$

Java:
```
y = m * x + b;
```

No se requieren paréntesis. El operador de multiplicación se aplica primero, ya que la multiplicación tiene mayor precedencia sobre la suma. La asignación ocurre al último, ya que tiene menor precedencia que la multiplicación o suma.

El siguiente ejemplo contiene las operaciones residuo (%), multiplicación, división, suma y resta:

Álgebra: $z = pr\%q + w/x - y$

Java:
```
z = p * r % q + w / x - y;
    6   1   2   4   3   5
```

Los números dentro de los círculos bajo la instrucción indican el orden en el que Java aplica los operadores. Las operaciones *, % y / se evalúan primero, en orden de izquierda a derecha (es decir, se asocian de izquierda a derecha), ya que tienen mayor precedencia que + y -. Las operaciones + y - se evalúan a continuación. Estas operaciones también se aplican de izquierda a derecha. El operador de asignación (=) se evalúa al último.

Evaluación de un polinomio de segundo grado

Para desarrollar una mejor comprensión de las reglas de precedencia de operadores, considere la evaluación de una expresión de asignación que incluye un polinomio de segundo grado $ax^2 + bx + c$:

```
y = a * x * x + b * x + c;
    6   1   2   4   3   5
```

Las operaciones de multiplicación se evalúan primero en orden de izquierda a derecha (es decir, se asocian de izquierda a derecha), ya que tienen mayor precedencia que la suma (como Java no tiene operador aritmético para la exponenciación, x^2 se representa como `x * x`. La sección 5.4 muestra una alternativa para realizar la exponenciación). A continuación se evalúan las operaciones de suma, de izquierda a derecha. Suponga que `a`, `b`, `c` y `x` se inicializan (reciben valores) como sigue: `a = 2`, `b = 3`, `c = 7` y `x = 5`. La figura 2.13 muestra el orden en el que se aplican los operadores.

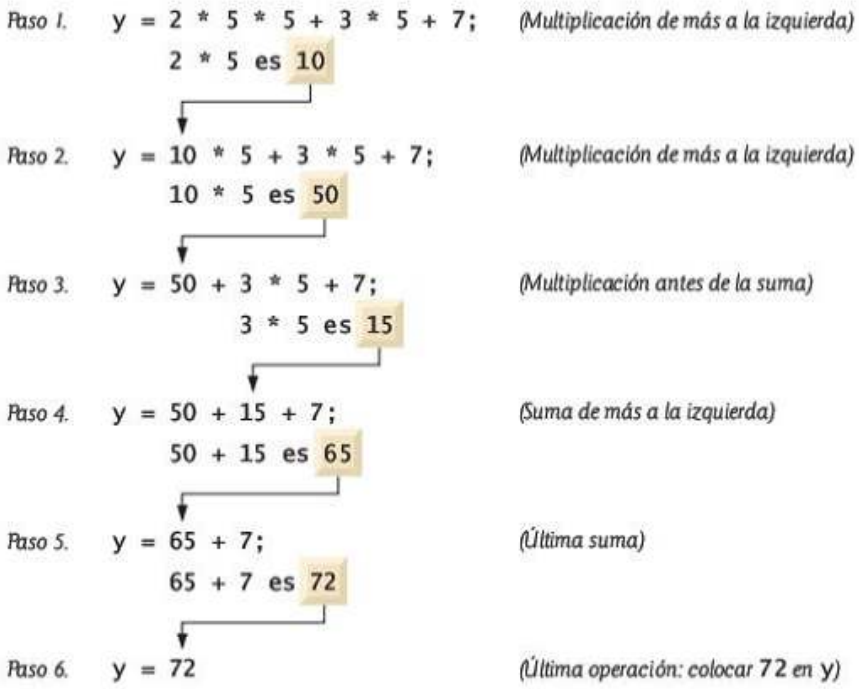


Fig. 2.13 | Orden en el cual se evalúa un polinomio de segundo grado.

Podemos usar **paréntesis redundantes** (paréntesis innecesarios) para hacer que una expresión sea más clara. Por ejemplo, la instrucción de asignación anterior podría colocarse entre paréntesis, de la siguiente manera:

```
y = ( a * x * x ) + ( b * x ) + c;
```

2.8 Toma de decisiones: operadores de igualdad y relacionales

Una **condición** es una expresión que puede ser **verdadera** (`true`) o **falsa** (`false`). Esta sección presenta la **instrucción `if` de Java**, la cual permite que un programa tome una **decisión**, con base en el valor de una condición. Por ejemplo, la condición "calificación es mayor o igual que 60" determina si un estudiante pasó o no una prueba. Si la condición en una instrucción `if` es verdadera, el cuerpo de la instrucción `if` se ejecuta. Si la condición es falsa, el cuerpo no se ejecuta. Veremos un ejemplo en breve.

Las condiciones en las instrucciones `if` pueden formarse utilizando los **operadores de igualdad** (`==` y `!=`) y los **operadores relacionales** (`>`, `<`, `>=` y `<=`) que se sintetizan en la figura 2.14. Ambos operadores de igualdad tienen el mismo nivel de precedencia, que es *menor* que la precedencia de los operadores relacionales. Los operadores de igualdad se asocian de izquierda a derecha. Todos los operadores relacionales tienen el mismo nivel de precedencia y también se asocian de izquierda a derecha.

En la figura 2.15 se utilizan seis instrucciones `if` para comparar dos enteros introducidos por el usuario. Si la condición en cualquiera de estas instrucciones `if` es verdadera, se ejecuta la instrucción asociada con esa instrucción `if`; en caso contrario, se omite la instrucción. Utilizamos un objeto `Scanner` para recibir los dos enteros del usuario y almacenarlos en las variables `numero1` y `numero2`. Después, el programa compara los números y muestra los resultados de las comparaciones que son verdaderas.

Operador estándar algebraico de igualdad o relacional	Operador de igualdad o relacional de Java	Ejemplo de condición en Java	Significado de la condición en Java
Operadores de igualdad			
=	==	x == y	x es igual que y
≠	!=	x != y	x no es igual que y
Operadores relacionales			
>	>	x > y	x es mayor que y
<	<	x < y	x es menor que y
≥	>=	x >= y	x es mayor o igual que y
≤	<=	x <= y	x es menor o igual que y

Fig. 2.14 | Operadores de igualdad y relacionales.

```
1  // Fig. 2.15: Comparacion.java
2  // Compara enteros utilizando instrucciones if, operadores relacionales
3  // y de igualdad.
4  import java.util.Scanner; // el programa utiliza la clase Scanner
5
6  public class Comparacion
7  {
8     // el método main empieza la ejecución de la aplicación en Java
9     public static void main( String[] args )
10    {
11       // crea objeto Scanner para obtener la entrada de la ventana de comandos
12       Scanner entrada = new Scanner( System.in );
13
14       int numero1; // primer número a comparar
15       int numero2; // segundo número a comparar
16
17       System.out.print( "Escriba el primer entero: " ); // indicador
18       numero1 = entrada.nextInt(); // lee el primer número del usuario
19
20       System.out.print( "Escriba el segundo entero: " ); // indicador
21       numero2 = entrada.nextInt(); // lee el segundo número del usuario
22
23       if ( numero1 == numero2 )
24          System.out.printf( "%d == %d\n", numero1, numero2 );
25
26       if ( numero1 != numero2 )
27          System.out.printf( "%d != %d\n", numero1, numero2 );
28
29       if ( numero1 < numero2 )
30          System.out.printf( "%d < %d\n", numero1, numero2 );
31
```

Fig. 2.15 | Comparación de enteros mediante instrucciones `if`, operadores de igualdad y relacionales (parte 1 de 2).

```
32       if ( numero1 > numero2 )
33          System.out.printf( "%d > %d\n", numero1, numero2 );
34
35       if ( numero1 <= numero2 )
36          System.out.printf( "%d <= %d\n", numero1, numero2 );
37
38       if ( numero1 >= numero2 )
39          System.out.printf( "%d >= %d\n", numero1, numero2 );
40    } // fin del método main
41 } // fin de la clase Comparacion
```

```
Escriba el primer entero: 777
Escriba el segundo entero: 777
777 == 777
777 <= 777
777 >= 777
```

```
Escriba el primer entero: 1000
Escriba el segundo entero: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Escriba el primer entero: 2000
Escriba el segundo entero: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

Fig. 2.15 | Comparación de enteros mediante instrucciones `if`, operadores de igualdad y relacionales (parte 2 de 2).

La declaración de la clase `Comparacion` comienza en la línea 6

```
public class Comparacion
```

El método `main` de la clase (líneas 9 a 40) empieza la ejecución del programa. La línea 12

```
Scanner entrada = new Scanner( System.in );
```

declara la variable entrada de la clase `Scanner` y le asigna un objeto `Scanner` que recibe datos de la entrada estándar (es decir, el teclado).

Las líneas 14 y 15

```
int numero1; // primer número a comparar
int numero2; // segundo número a comparar
```

declaran las variables `int` que se utilizan para almacenar los valores introducidos por el usuario.

Las líneas 17-18

```
System.out.print( "Escriba el primer entero: " ); // indicador
numero1 = entrada.nextInt(); // lee el primer número del usuario
```

piden al usuario que introduzca el primer entero y el valor, respectivamente. El valor de entrada se almacena en la variable `numero1`.

Las líneas 20-21

```
System.out.print( "Escriba el segundo entero: " ); // indicador
numero2 = entrada.nextInt(); // lee el segundo número del usuario
```

piden al usuario que introduzca el segundo entero y el valor, respectivamente. El valor de entrada se almacena en la variable `numero2`.

Las líneas 23-24

```
if ( numero1 == numero2 )
   System.out.printf( "%d == %d\n", numero1, numero2 );
```

declaran una instrucción `if` que compara los valores de las variables `numero1` y `numero2`, para determinar si son iguales o no. Una instrucción `if` siempre empieza con la palabra clave `if`, seguida de una condición entre paréntesis. Una instrucción `if` espera una instrucción en su cuerpo, pero puede contener varias instrucciones si se encierran entre un conjunto de llaves ({}). La sangría de la instrucción del cuerpo que se muestra aquí no es obligatoria, pero mejora la legibilidad del programa al enfatizar que la instrucción en la línea 24 *forma parte* de la instrucción `if` que empieza en la línea 23. La línea 24 sólo se ejecuta si los números almacenados en las variables `numero1` y `numero2` son iguales (es decir, si la condición es verdadera). Las instrucciones `if` en las líneas 26-27, 29-30, 32-33, 35-36 y 38-39 comparan a `numero1` y `numero2` con los operadores `!=`, `<`, `>`, `<=` y `>=`, respectivamente. Si la condición en una o más de las instrucciones `if` es verdadera, se ejecuta la instrucción del cuerpo correspondiente.

Error común de programación 2.6

Confundir el operador de igualdad (==) con el de asignación (=) puede producir un error lógico o de sintaxis. El operador de igualdad debe leerse como "es igual que", y el de asignación como "obtiene" u "obtiene el valor de". Para evitar confusión, algunas personas leen el operador de igualdad como "doble igual" o "igual igual".

Buena práctica de programación 2.11

Al colocar sólo una instrucción por línea en un programa, mejora su legibilidad.

No hay punto y coma (;) al final de la primera línea de cada instrucción `if`, ya que produciría un error lógico en tiempo de ejecución. Por ejemplo,

```
if ( numero1 == numero2 ); // error lógico
   System.out.printf( "%d == %d\n", numero1, numero2 );
```

sería interpretada por Java de la siguiente manera:

```
if ( numero1 == numero2 )
   ; // instrucción vacía
System.out.printf( "%d == %d\n", numero1, numero2 );
```

en donde el punto y coma que aparece por sí solo en una línea (que se conoce como **instrucción vacía**) es la instrucción que se va a ejecutar si la condición en la instrucción `if` es verdadera. Al ejecutarse la instrucción vacía, no se lleva a cabo ninguna tarea. Después el programa continúa con la instrucción de salida, que siempre se ejecuta, sin importar que la condición sea verdadera o falsa, ya que la instrucción de salida no forma parte de la instrucción `if`.

Error común de programación 2.7

Colocar un punto y coma inmediatamente después del paréntesis derecho de la condición en una instrucción `if` es, por lo general, un error lógico.

Observe el uso del espacio en blanco en la figura 2.15. Recuerde que el compilador casi siempre ignora los caracteres de espacio en blanco. Por lo tanto, las instrucciones pueden dividirse en varias líneas y espaciarse de acuerdo a las preferencias del programador, sin afectar el significado de un programa. Es incorrecto dividir identificadores y cadenas. Lo ideal es que las instrucciones se mantengan lo más reducidas posible, pero, no siempre se puede hacer esto.

Tip para prevenir errores 2.9

Una instrucción larga puede esparcirse en varias líneas. Si una sola instrucción debe dividirse entre varias líneas, los puntos que elija para hacer la división deben tener sentido, como después de una coma en una lista separada por comas, o después de un operador en una expresión larga. Si una instrucción se divide entre dos o más líneas, aplique sangría a todas las líneas subsecuentes hasta el final de la instrucción.

La figura 2.16 muestra los operadores que hemos visto hasta ahora, en orden decreciente de precedencia. Todos, con la excepción del operador de asignación, =, se asocian de izquierda a derecha. El operador de asignación, =, asocia de derecha a izquierda, por lo que una expresión como `x = y = 0` se evalúa como si se hubiera escrito así: `x = (y = 0)`, en donde primero se asigna el valor `0` a la variable `y`, y después se asigna el resultado de esa asignación, `0`, a `x`.

Buena práctica de programación 2.12

Cuando escriba expresiones que contengan muchos operadores, consulte la tabla de precedencia de operadores (apéndice A). Confirme que las operaciones en la expresión se realicen en el orden que usted espera. Si no está seguro acerca del orden de evaluación en una expresión compleja, utilice paréntesis para forzar el orden, en la misma forma que lo haría con las expresiones algebraicas.

Operadores				Asociatividad	Tipo
*	/	%		izquierda a derecha	multiplicativa
+	-			izquierda a derecha	suma
<	<=	>	>=	izquierda a derecha	relacional
==	!=			izquierda a derecha	igualdad
=				derecha a izquierda	asignación

Fig. 2.16 | Precedencia y asociatividad de los operadores descritos hasta ahora.

2.9 Conclusión

En este capítulo aprendió acerca de muchas características importantes de Java, como mostrar datos en la pantalla en un **Símbolo del sistema**, introducir datos por medio del teclado, realizar cálculos y tomar decisiones. A través de las aplicaciones que vimos en este capítulo, le presentamos los conceptos básicos de programación. Como veremos en el capítulo 3, por lo general las aplicaciones de Java contienen sólo unas cuantas líneas de código en el método `main`: casi siempre estas instrucciones crean los objetos que realizan el trabajo de la aplicación. En el capítulo 3 aprenderá a implementar sus propias clases y a usar objetos de éstas en las aplicaciones.

Resumen

Sección 2.2 Su primer programa en Java: imprimir una línea de texto

- Una aplicación de Java (pág. 38) se ejecuta cuando utilizamos el comando `java` para iniciar la JVM.

- Los comentarios (pág. 39) documentan los programas y mejoran su legibilidad. El compilador los ignora.
- Un comentario que empieza con `//` se llama comentario de fin de línea; termina al final de la línea en la que aparece.
- Los comentarios tradicionales (pág. 39) se pueden distribuir en varias líneas; están delimitados por `/*` y `*/`.
- Los comentarios `Javadoc` (pág. 39) se delimitan por `/**` y `*/`; nos permiten incrustar la documentación de los programas en el código. La herramienta `javadoc` genera páginas en HTML con base en estos comentarios.
- Un error de sintaxis (pág. 39; también conocido como error de compilador, error en tiempo de compilación o error de compilación) ocurre cuando el compilador encuentra código que viola las reglas del lenguaje Java. Es similar a un error gramatical en un lenguaje natural.
- Las líneas en blanco, los espacios y los tabuladores se conocen como espacio en blanco (pág. 39). Éste mejora la legibilidad de los programas; el compilador lo ignora.
- Las palabras clave (pág. 40) están reservadas para el uso exclusivo de Java, y siempre se escriben con letras minúsculas.
- La palabra clave `class` (pág. 40) introduce una declaración de clase.
- Por convención, todos los nombres de las clases en Java empiezan con una letra mayúscula, y la primera letra de cada palabra subsiguiente también se escribe en mayúscula (como `NombreClaseDeEjemplo`).
- El nombre de una clase de Java es un identificador: una serie de caracteres formada por letras, dígitos, guiones bajos (_) y signos de dólar ($), que no empieza con un dígito y no contiene espacios.
- Java es sensible a mayúsculas/minúsculas (pág. 40); es decir, las letras mayúsculas y minúsculas son distintas.
- El cuerpo de todas las declaraciones de clases (pág. 40) debe estar delimitado por llaves, `{` y `}`.
- La declaración de una clase `public` (pág. 40) debe guardarse en un archivo con el mismo nombre que la clase, seguido de la extensión de nombre de archivo ".`java`".
- El método `main` (pág. 41) es el punto de inicio de toda aplicación en Java, y debe empezar con:

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

  en caso contrario, la JVM no ejecutará la aplicación.
- Los métodos pueden realizar tareas y devolver información cuando las completan. La palabra clave `void` (pág. 41) indica que un método realizará una tarea, pero no devolverá información.
- Las instrucciones instruyen a la computadora para que realice acciones.
- Por lo general, a una cadena (pág. 41) entre comillas dobles se le conoce como cadena de caracteres, literal de cadena.
- El objeto de salida estándar (`System.out`; pág. 41) muestra caracteres en la ventana de comandos.
- El método `System.out.println` (pág. 41) muestra su argumento (pág. 41) en la ventana de comandos, seguido de un carácter de nueva línea para colocar el cursor de salida en el inicio de la siguiente línea.
- Para compilar un programa se utiliza el comando `javac`. Si el programa no contiene errores de sintaxis, se crea un archivo de clase (pág. 43) que contiene los códigos de bytes de Java que representan a la aplicación. La JVM interpreta estos códigos de bytes cuando ejecutamos el programa.
- Para ejecutar una aplicación, escriba la palabra `java` (pág. 38) seguida del nombre de la clase que contiene a `main`.

Sección 2.3 Modificación de nuestro primer programa en Java

- `System.out.print` (pág. 44) muestra su argumento en pantalla y coloca el cursor de salida justo después del último carácter visualizado.
- Una barra diagonal inversa (\) en una cadena es un carácter de escape (pág. 45). Java lo combina con el siguiente carácter para formar una secuencia de escape (pág. 45). La secuencia de escape `\n` (pág. 45) representa el carácter de nueva línea.

Sección 2.4 Cómo mostrar texto con `printf`

- El método `System.out.printf` (pág. 46; `f` se refiere a "formato") muestra datos con formato.

- El primer argumento del método `printf` es una cadena de formato (pág. 47) que contiene texto fijo y/o especificadores de formato. Cada especificador de formato (pág. 47) indica el tipo de datos a imprimir y es un receptáculo para el argumento correspondiente que aparece después de la cadena de formato.
- Los especificadores de formato empiezan con un signo porcentual (`%`), y van seguidos de un carácter que representa el tipo de datos. El especificador de formato `%s` (pág. 47) es un receptáculo para una cadena.

Sección 2.5 Otra aplicación en Java: suma de enteros

- Una declaración `import` (pág. 48) ayuda al compilador a localizar una clase que se utiliza en un programa.
- El extenso conjunto de clases predefinidas de Java se agrupan en paquetes (pág. 48) denominados grupos de clases. A éstos se les conoce como la biblioteca de clases de Java (pág. 48), o la Interfaz de programación de aplicaciones de Java (API de Java).
- Una variable (pág. 49) es una ubicación en la memoria de la computadora, en la cual se puede guardar un valor para usarlo más adelante en un programa. Todas las variables deben declararse con un nombre y un tipo para poder utilizarlas.
- El nombre de una variable permite al programa acceder a su valor en memoria.
- Un objeto `Scanner` (paquete `java.util`; pág. 49) permite a un programa leer datos para usarlos en éste. Antes de usar un objeto `Scanner`, el programa debe crearlo y especificar el origen de los datos.
- Las variables deben inicializarse (pág. 49) para poder usarlas en un programa.
- La expresión `new Scanner(System.in)` crea un objeto `Scanner` que lee datos desde el objeto de entrada estándar (`System.in`; pág. 49); por lo general es el teclado.
- El tipo de datos `int` (pág. 49) se utiliza para declarar variables que guardarán valores enteros. El rango de valores para un `int` es de –2,147,483,648 a +2,147,483,647.
- Los tipos `float` y `double` (pág. 49) especifican números reales con puntos decimales, como `3.4` y `-11.19`.
- Las variables de tipo `char` (pág. 49) representan caracteres individuales, como una letra mayúscula (por ejemplo, `A`), un dígito (por ejemplo, `7`), un carácter especial (por ejemplo, `*` o `%`) o una secuencia de escape (por ejemplo, el carácter de nueva línea, `\n`).
- Los tipos como `int`, `float`, `double` y `char` son tipos primitivos (pág. 49). Los nombres de los tipos primitivos son palabras clave; por ende, deben aparecer escritos sólo con letras minúsculas.
- Un indicador (pág. 50) pide al usuario que realice una acción específica.
- El método `nextInt` de `Scanner` obtiene un entero para usarlo en un programa.
- El operador de asignación, `=` (pág. 51), permite al programa dar un valor a una variable. Se llama operador binario (pág. 51), ya que tiene dos operandos.
- Las partes de las instrucciones que tienen valores se llaman expresiones (pág. 51).
- El especificador de formato `%d` (pág. 51) es un receptáculo para un valor `int`.

Sección 2.6 Conceptos acerca de la memoria

- Los nombres de las variables (pág. 52) corresponden a ubicaciones en la memoria de la computadora. Cada variable tiene un nombre, un tipo, un tamaño y un valor.
- Un valor que se coloca en una ubicación de memoria sustituye al valor anterior en esa ubicación, el cual se pierde.

Sección 2.7 Aritmética

- Los operadores aritméticos (pág. 53) son `+` (suma), `-` (resta), `*` (multiplicación), `/` (división) y `%` (residuo).
- La división de enteros (pág. 53) produce un cociente entero.
- El operador residuo, `%` (pág. 53), produce el residuo después de la división.
- Las expresiones aritméticas deben escribirse en formato de línea recta (pág. 54).
- Si una expresión contiene paréntesis anidados (pág. 54), el conjunto de paréntesis más interno se evalúa primero.

- Java aplica los operadores en las expresiones aritméticas en una secuencia precisa, la cual se determina mediante las reglas de precedencia de los operadores (pág. 54).
- Cuando decimos que los operadores se aplican de izquierda a derecha, nos referimos a su asociatividad (pág. 54). Algunos operadores se asocian de derecha a izquierda.
- Los paréntesis redundantes (pág. 56) pueden hacer que una expresión sea más clara.

Sección 2.8 Toma de decisiones: operadores de igualdad y relacionales

- La instrucción `if` (pág. 56) toma una decisión con base en el valor de esa condición (verdadero o falso).
- Las condiciones en las instrucciones `if` se pueden forman mediante el uso de los operadores de igualdad (== y !=) y relacionales (>, <, >= y <=) (pág. 56).
- Una instrucción `if` empieza con la palabra clave `if` seguida de una condición entre paréntesis, y espera una instrucción en su cuerpo.
- La instrucción vacía (pág. 59) es una instrucción que no realiza una tarea.

Ejercicios de autoevaluación

2.1 Complete las siguientes oraciones:

a) El cuerpo de cualquier método comienza con un(a) _____ y termina con un(a) _____ .
b) La instrucción _____ se utiliza para tomar decisiones.
c) _____ indica el inicio de un comentario de fin de línea.
d) _____ , _____ y _____ se conocen como espacio en blanco.
e) Las _____ están reservadas para su uso en Java.
f) Las aplicaciones en Java comienzan su ejecución en el método _____ .
g) Los métodos _____ , _____ y _____ muestran información en una ventana de comandos.

2.2 Indique si cada una de las siguientes instrucciones es *verdadera* o *falsa*. Si es *falsa*, explique por qué.

a) Los comentarios hacen que la computadora imprima el texto que va después de los caracteres // en la pantalla, al ejecutarse el programa.
b) Todas las variables deben recibir un tipo cuando se declaran.
c) Java considera que las variables `numero` y `NuMeRo` son idénticas.
d) El operador residuo (%) puede utilizarse solamente con operandos enteros.
e) Los operadores aritméticos *, /, %, + y - tienen todos el mismo nivel de precedencia.

2.3 Escriba instrucciones para realizar cada una de las siguientes tareas:

a) Declarar las variables `c`, `estaEsUnaVariable`, `q76354` y `numero` como de tipo `int`.
b) Pedir al usuario que introduzca un entero.
c) Recibir un entero como entrada y asignar el resultado a la variable `int valor`. Suponga que se puede utilizar la variable `entrada` tipo `Scanner` para recibir un valor del teclado.
d) Imprimir "`Este es un programa en Java`" en una línea de la ventana de comandos. Use el método `System.out.println`.
e) Imprimir "`Este es un programa en Java`" en dos líneas de la ventana de comandos. La primera línea debe terminar con `es un`. Use el método `System.out.println`.
f) Imprimir "`Este es un programa en Java`" en dos líneas de la ventana de comandos. La primera línea debe terminar con `es un`. Use el método `System.out.printf` y dos especificadores de formato `%s`.
g) Si la variable `numero` no es igual que `7`, mostrar "`La variable numero no es igual que 7`".

2.4 Identifique y corrija los errores en cada una de las siguientes instrucciones:

a)
```
if ( c < 7 );
        System.out.println( "c es menor que 7" );
```
b)
```
if ( c => 7 )
        System.out.println( "c es igual o mayor que 7" );
```

2.5 Escriba declaraciones, instrucciones o comentarios para realizar cada una de las siguientes tareas:

a) Indicar que un programa calculará el producto de tres enteros.
b) Crear un objeto `Scanner` llamado `entrada` que lea valores de la entrada estándar.
c) Declarar las variables `x`, `y`, `z` y `resultado` de tipo `int`.
d) Pedir al usuario que escriba el primer entero.
e) Leer el primer entero del usuario y almacenarlo en la variable `x`.
f) Pedir al usuario que escriba el segundo entero.
g) Leer el segundo entero del usuario y almacenarlo en la variable `y`.
h) Pedir al usuario que escriba el tercer entero.
i) Leer el tercer entero del usuario y almacenarlo en la variable `z`.
j) Calcular el producto de los tres enteros contenidos en las variables `x`, `y` y `z`, y asignar el resultado a la variable `resultado`.
k) Mostrar el mensaje `"El producto es "`, seguido del valor de la variable `resultado`.

2.6 Utilice las instrucciones que escribió en el ejercicio 2.5 para escribir un programa completo que calcule e imprima el producto de tres enteros.

Respuestas a los ejercicios de autoevaluación

2.1 a) llave izquierda ({), llave derecha (}). b) `if`. c) `//`. d) Caracteres de espacio, caracteres de nueva línea y tabuladores. e) Palabras clave. f) `main`. g) `System.out.print`, `System.out.println` y `System.out.printf`.

2.2 a) Falso. Los comentarios no producen ninguna acción cuando el programa se ejecuta. Se utilizan para documentar programas y mejorar su legibilidad.

b) Verdadero.
c) Falso. Java es sensible a mayúsculas y minúsculas, por lo que estas variables son distintas.
d) Falso. El operador residuo puede utilizarse también con operandos no enteros en Java.
e) Falso. Los operadores `*`, `/` y `%` tienen mayor precedencia que los operadores `+` y `-`.

2.3 a)
```
int c, estaEsUnaVariable, q76354, numero;
```
o
```
int c;
int estaEsUnaVariable;
int q76354;
int numero;
```
b)
```
System.out.print( "Escriba un entero" );
```
c)
```
valor = entrada.nextInt();
```
d)
```
System.out.println( "Este es un programa en Java" );
```
e)
```
System.out.println( "Este es un\n programa en Java" );
```
f)
```
System.out.printf( "%s\%s\n", "Este es un", "programa en Java" );
```
g)
```
if ( numero != 7 )
        System.out.println( "La variable numero no es igual que 7" );
```

2.4 a) Error: Hay un punto y coma después del paréntesis derecho de la condición (`c < 7`) en la instrucción `if`. Corrección: Quite el punto y coma que va después del paréntesis derecho. [*Nota:* como resultado, la instrucción de salida se ejecutará, sin importar que la condición en la instrucción `if` sea verdadera].

b) Error: El operador relacional `=>` es incorrecto. Corrección: Cambie `=>` a `>=`.

2.5 a)
```
// Calcula el producto de tres enteros
```
b)
```
Scanner entrada = new Scanner (System.in);
```
c)
```
int x, y, z, resultado;
```
o

```
int x;
int y;
int z;
int resultado;
```

d) `System.out.print( "Escriba el primer entero: " );`
e) `x = entrada.nextInt();`
f) `System.out.print( "Escriba el segundo entero: " );`
g) `y = entrada.nextInt();`
h) `System.out.print( "Escriba el tercer entero: " );`
i) `z = entrada.nextInt();`
j) `resultado = x * y * z;`
k) `System.out.printf( "El producto es %d\n", resultado );`

2.6 La solución para el ejercicio 2.6 es la siguiente:

```
1  // Ejemplo 2.6: Producto.java
2  // Calcular el producto de tres enteros.
3  import java.util.Scanner; // el programa usa Scanner
4
5  public class Producto
6  {
7     public static void main( String[] args )
8     {
9        // crea objeto Scanner para obtener la entrada de la ventana de comandos
10       Scanner entrada = new Scanner( System.in );
11
12       int x; // primer número introducido por el usuario
13       int y; // segundo número introducido por el usuario
14       int z; // tercer número introducido por el usuario
15       int resultado; // producto de los números
16
17       System.out.print( "Escriba el primer entero: " ); // indicador de entrada
18       x = entrada.nextInt(); // lee el primer entero
19
20       System.out.print( "Escriba el segundo entero: " ); // indicador de entrada
21       y = entrada.nextInt(); // lee el segundo entero
22
23       System.out.print( "Escriba el tercer entero: " ); // indicador de entrada
24       z = entrada.nextInt(); // lee el tercer entero
25
26       resultado = x * y * z; // calcula el producto de los números
27
28       System.out.printf( "El producto es %d\n", resultado );
29    } // fin del método main
30 } // fin de la clase Producto
```

```
Escriba el primer entero: 10
Escriba el segundo entero: 20
Escriba el tercer entero: 30
El producto es 6000
```

Ejercicios

2.7 Complete las siguientes oraciones:

a) _____ se utilizan para documentar un programa y mejorar su legibilidad.
b) Una decisión puede tomarse en un programa en Java con un(a) _____ .

c) Los cálculos se realizan normalmente mediante instrucciones _____ .
d) Los operadores aritméticos con la misma precedencia que la multiplicación son _____ y _____ .
e) Cuando los paréntesis en una expresión aritmética están anidados, se evalúa primero el conjunto _____ de paréntesis.
f) Una ubicación en la memoria de la computadora que puede contener distintos valores en diversos instantes de tiempo, durante la ejecución de un programa, se llama _____ .

2.8 Escriba instrucciones en Java que realicen cada una de las siguientes tareas:
a) Mostrar el mensaje `"Escriba un entero"`, dejando el cursor en la misma línea.
b) Asignar el producto de las variables `b` y `c` a la variable `a`.
c) Usar un comando para indicar que un programa va a realizar un cálculo de nómina de muestra.

2.9 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Los operadores en Java se evalúan de izquierda a derecha.
b) Los siguientes nombres de variables son todos válidos: `_barra_inferior_`, `m928134`, `t5`, `j7`, `sus_ventas$`, `su_$cuenta_total`, `a`, `b$`, `c`, `z` y `z2`.
c) Una expresión aritmética válida en Java sin paréntesis se evalúa de izquierda a derecha.
d) Los siguientes nombres de variables son todos inválidos: `3g`, `87`, `67h2`, `h22` y `2h`.

2.10 Suponiendo que `x = 2` y `y = 3`, ¿qué muestra cada una de las siguientes instrucciones?
a) `System.out.printf( "x = %d\n", x );`
b) `System.out.printf( "El valor de %d + %d es %d\n", x, x, ( x + x ) );`
c) `System.out.printf( "x =" );`
d) `System.out.printf( "%d = %d\n", ( x + y ), ( y + x ) );`

2.11 ¿Cuáles de las siguientes instrucciones de Java contienen variables, cuyos valores se modifican?
a) `p = i + j + k + 7;`
b) `System.out.println( "variables cuyos valores se modifican" );`
c) `System.out.println( "a = 5" );`
d) `valor = entrada.nextInt();`

2.12 Dado que $y = ax^3 + 7$, ¿cuáles de las siguientes instrucciones en Java son correctas para esta ecuación?
a) `y = a * x * x * x + 7;`
b) `y = a * x * x * ( x + 7 );`
c) `y = ( a * x ) * x * ( x + 7 );`
d) `y = ( a * x ) * x * x + 7;`
e) `y = a * ( x * x * x ) + 7;`
f) `y = a * x * ( x * x + 7 );`

2.13 Indique el orden de evaluación de los operadores en cada una de las siguientes instrucciones en Java, y muestre el valor de `x` después de ejecutar cada una de ellas:
a) `x = 7 + 3 * 6 / 2 - 1;`
b) `x = 2 % 2 + 2 * 2 - 2 / 2;`
c) `x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) );`

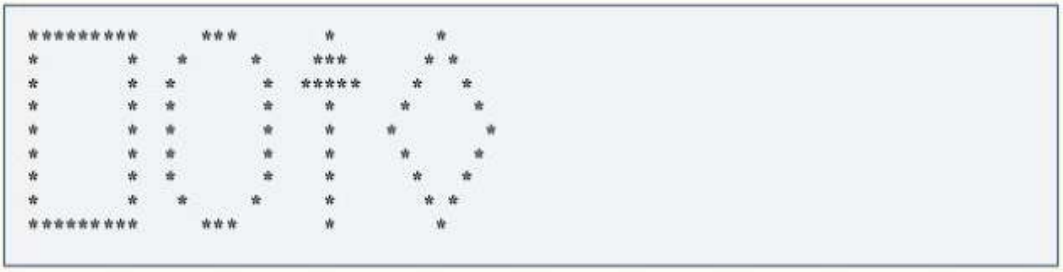
2.14 Escriba una aplicación que muestre los números del 1 al 4 en la misma línea, con cada par de números adyacentes separado por un espacio. Use las siguientes técnicas:
a) Mediante una instrucción `System.out.println`.
b) Mediante cuatro instrucciones `System.out.print`.
c) Mediante una instrucción `System.out.printf`.

2.15 ***(Aritmética)*** Escriba una aplicación que pida al usuario que escriba dos enteros, que obtenga los números del usuario e imprima la suma, producto, diferencia y cociente (división) de los números. Use las técnicas que se muestran en la figura 2.7.

2.16 ***(Comparación de enteros)*** Escriba una aplicación que pida al usuario que escriba dos enteros, que obtenga los números del usuario y muestre el más grande, seguido de las palabras "`es más grande`". Si son iguales, imprima el mensaje "`Estos números son iguales`". Utilice las técnicas que se muestran en la figura 2.15.

2.17 ***(Aritmética: menor y mayor)*** Escriba una aplicación que reciba tres enteros del usuario y muestre la suma, promedio, producto, menor y mayor de esos números. Utilice las técnicas que se muestran en la figura 2.15. [*Nota:* el cálculo del promedio en este ejercicio debe resultar en una representación entera del promedio. Por lo tanto, si la suma de los valores es 7, el promedio debe ser 2, no 2.3333...].

2.18 ***(Visualización de figuras con asteriscos)*** Escriba una aplicación que muestre un cuadro, un óvalo, una flecha y un diamante usando asteriscos (*), como se muestra a continuación:



2.19 ¿Qué imprime el siguiente código?

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

2.20 ¿Qué imprime el siguiente código?

```
System.out.println( "*" );
System.out.println( "***" );
System.out.println( "*****" );
System.out.println( "****" );
System.out.println( "**" );
```

2.21 ¿Qué imprime el siguiente código?

```
System.out.print( "*" );
System.out.print( "***" );
System.out.print( "*****" );
System.out.print( "****" );
System.out.println( "**" );
```

2.22 ¿Qué imprime el siguiente código?

```
System.out.print( "*" );
System.out.println( "***" );
System.out.println( "*****" );
System.out.print( "****" );
System.out.println( "**" );
```

2.23 ¿Qué imprime el siguiente código?

```
System.out.printf( "%s\n%s\n%s\n", "*", "***", "*****" );
```

2.24 ***(Enteros menor y mayor)*** Escriba una aplicación que lea cinco enteros y que determine e imprima los enteros mayor y menor en el grupo. Use solamente las técnicas de programación que aprendió en este capítulo.

2.25 ***(Par o impar)*** Escriba una aplicación que lea un entero y que determine e imprima si es impar o par. [*Sugerencia:* use el operador residuo. Un número par es un múltiplo de 2. Cualquier múltiplo de 2 deja un residuo de 0 cuando se divide entre 2].

2.26 ***(Múltiplos)*** Escriba una aplicación que lea dos enteros, determine si el primero es un múltiplo del segundo e imprima el resultado. [*Sugerencia:* use el operador residuo].

2.27 ***(Patrón de damas mediante asteriscos)*** Escriba una aplicación que muestre un patrón de tablero de damas, como se muestra a continuación:

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

2.28 ***(Diámetro, circunferencia y área de un círculo)*** He aquí un adelanto. En este capítulo, aprendió sobre los enteros y el tipo `int`. Java también puede representar números de punto flotante que contienen puntos decimales, como 3.14159. Escriba una aplicación que reciba del usuario el radio de un círculo como un entero, y que imprima el diámetro, la circunferencia y el área del círculo mediante el uso del valor de punto flotante 3.14159 para π. Use las técnicas que se muestran en la figura 2.7. [*Nota:* también puede utilizar la constante predefinida `Math.PI` para el valor de π. Esta constante es más precisa que el valor 3.14159. La clase `Math` se define en el paquete `java.lang`. Las clases en este paquete se importan de manera automática, por lo que no necesita importar la `clase Math` mediante la instrucción `import` para usarla.] Use las siguientes fórmulas (*r* es el radio):

diámetro = 2*r*
circunferencia = 2π*r*
área = πr^2

No almacene los resultados de cada cálculo en una variable. En vez de ello, especifique cada uno como el valor que se imprimirá en una instrucción `System.out.printf`. Los valores producidos por los cálculos del área y la circunferencia son números de punto flotante. Dichos valores pueden imprimirse con el especificador de formato `%f` en una instrucción `System.out.printf`. En el capítulo 3 aprenderá más acerca de los números de punto flotante.

2.29 ***(Valor entero de un carácter)*** He aquí otro adelanto. En este capítulo, aprendió acerca de los enteros y el tipo `int`. Java puede también representar letras en mayúsculas, en minúsculas y una considerable variedad de símbolos especiales. Cada carácter tiene su correspondiente representación entera. El conjunto de caracteres que utiliza una computadora, y las correspondientes representaciones enteras de esos caracteres, se conocen como el conjunto de caracteres de esa computadora. Usted puede indicar un valor de carácter en un programa con sólo encerrarlo entre comillas sencillas, como en `'A'`.

Puede determinar el equivalente entero de un carácter si lo antepone a la palabra `(int)`, como en

```
(int) 'A'
```

Esta forma se conoce como operador de conversión de tipo. (Aprenderá sobre estos operadores en el capítulo 4.) La siguiente instrucción imprime un carácter y su equivalente entero:

```
System.out.printf(
   "El caracter %c tiene el valor %d\n", 'A', ( (int) 'A' ) );
```

Cuando se ejecuta esta instrucción, muestra el carácter `A` y el valor `65` (del conjunto de caracteres conocido como Unicode®) como parte de la cadena. Observe que el especificador de formato `%c` es un receptáculo para un carácter (en este caso, el carácter `'A'`).

Utilizando instrucciones similares a la mostrada anteriormente en este ejercicio, escriba una aplicación que muestre los equivalentes enteros de algunas letras en mayúsculas, en minúsculas, dígitos y símbolos especiales. Muestre los equivalentes enteros de los siguientes caracteres: `A B C a b c 0 1 2 $ * + /` y el carácter en blanco.

2.30 ***(Separación de los dígitos en un entero)*** Escriba una aplicación que reciba del usuario un número compuesto por cinco dígitos, que separe ese número en sus dígitos individuales y los imprima, cada uno separado de los demás por tres espacios. Por ejemplo, si el usuario escribe el número 42339, el programa debe imprimir

```
4   2   3   3   9
```

Suponga que el usuario escribe el número correcto de dígitos. ¿Qué ocurre cuando ejecuta el programa y escribe un número con más de cinco dígitos? ¿Qué ocurre cuando ejecuta el programa y escribe un número con menos de cinco dígitos? [*Sugerencia:* es posible hacer este ejercicio con las técnicas que aprendió en este capítulo. Necesitará utilizar las operaciones de división y residuo para "seleccionar" cada dígito].

2.31 ***(Tabla de cuadrados y cubos)*** Utilizando sólo las técnicas de programación que aprendió en este capítulo, escriba una aplicación que calcule los cuadrados y cubos de los números del 0 al 10, y que imprima los valores resultantes en formato de tabla, como se muestra a continuación. [*Nota:* este programa no requiere de ningún tipo de entrada por parte del usuario].

```
numero      cuadrado      cubo
0           0             0
1           1             1
2           4             8
3           9             27
4           16            64
5           25            125
6           36            216
7           49            343
8           64            512
9           81            729
10          100           1000
```

2.32 ***(Valores negativos, positivos y cero)*** Escriba un programa que reciba cinco números, y que determine e imprima la cantidad de números negativos, positivos, y la cantidad de ceros recibidos.

Marcar la diferencia

2.33 ***(Calculadora del índice de masa corporal)*** En el ejercicio 1.10 introdujimos la calculadora del índice de masa corporal (BMI). Las fórmulas para calcular el BMI son

$$BMI = \frac{pesoEnLibras \times 703}{alturaEnPulgadas \times alturaEnPulgadas}$$

o

$$BMI = \frac{pesoEnKilogramos}{alturaEnMetros \times alturaEnMetros}$$

Cree una calculadora del BMI que lea el peso del usuario en libras y la altura en pulgadas (o, si lo prefiere, el peso del usuario en kilogramos y la altura en metros), para que luego calcule y muestre el índice de masa corporal del usuario. Muestre además la siguiente información del Departamento de Salud y Servicios Humanos/Instituto Nacional de Salud para que el usuario pueda evaluar su BMI:

```
VALORES DE BMI
Bajo peso: menos de 18.5
Normal:    entre 18.5 y 24.9
Sobrepeso: entre 25 y 29.9
Obeso:     30 o más
```

[*Nota:* en este capítulo aprendió a usar el tipo `int` para representar números enteros. Cuando se realizan los cálculos del BMI con valores `int`, se producen resultados en números enteros. En el capítulo 3 aprenderá a usar el tipo `double` para representar a los números con puntos decimales. Cuando se realizan los cálculos del BMI con valores `double`, producen números con puntos decimales; a éstos se les conoce como números de "punto flotante"].

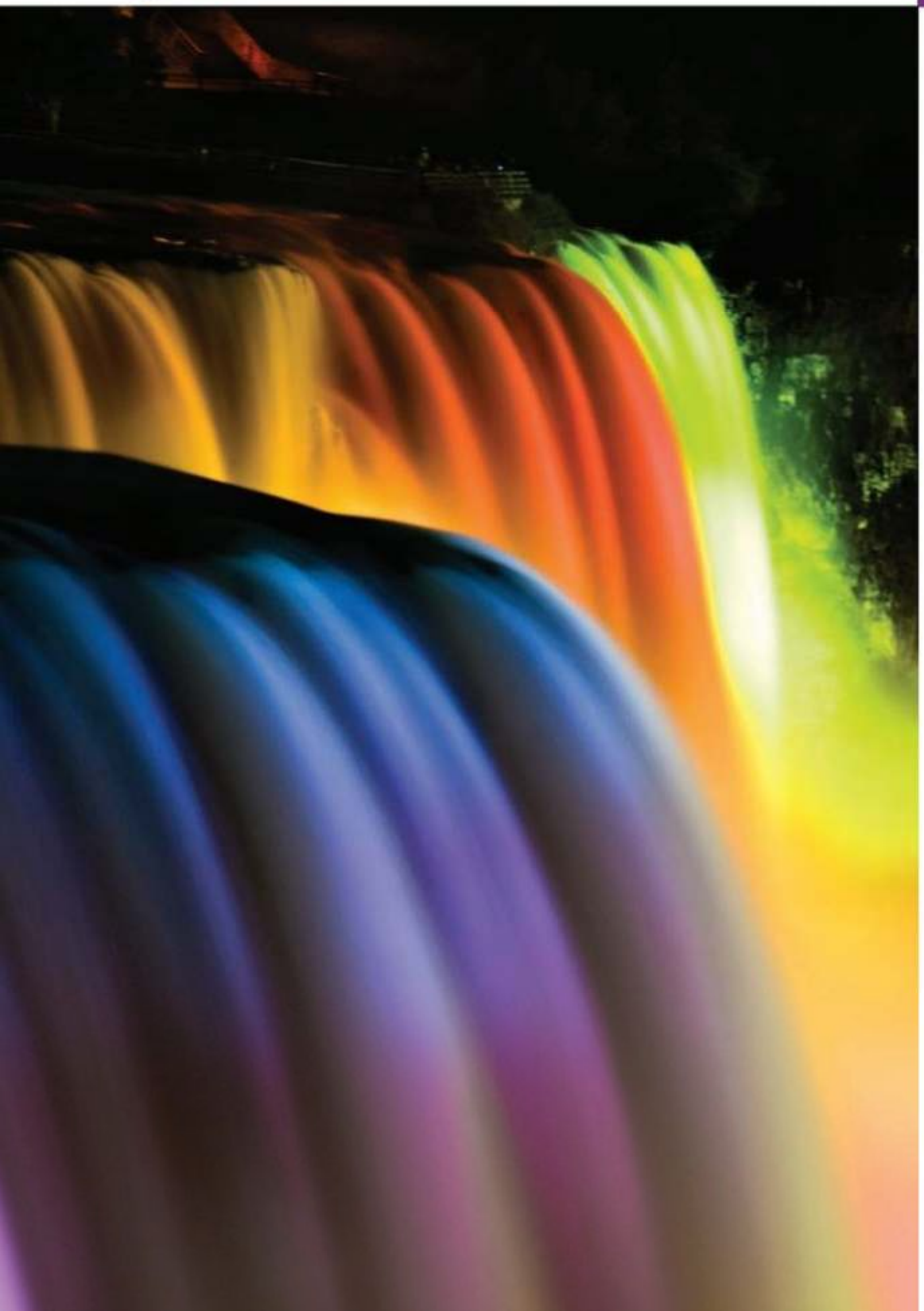
2.34 ***(Calculadora del crecimiento de la población mundial)*** Use Web para determinar la población mundial actual y la tasa de crecimiento anual de la población mundial. Escriba una aplicación que reciba estos valores como entrada y luego muestre la población mundial estimada después de uno, dos, tres, cuatro y cinco años.

2.35 ***(Calculadora de ahorro por viajes compartidos en automóvil)*** Investigue varios sitios Web de viajes compartidos en automóvil. Cree una aplicación que calcule su costo diario al conducir su automóvil, de modo que pueda estimar cuánto dinero puede ahorrar si comparte los viajes en automóvil, lo cual también tiene otras ventajas, como la reducción de las emisiones de carbono y de la congestión de tráfico. La aplicación debe recibir como entrada la siguiente información y mostrar el costo por día para el usuario por conducir al trabajo:

a) Total de kilómetros conducidos por día.
b) Costo por litro de gasolina.
c) Promedio de kilómetros por litro.
d) Cuotas de estacionamiento por día.
e) Peaje por día.

Introducción a las clases, objetos, métodos y cadenas

3



Nada puede tener valor sin ser un objeto de utilidad.
—Karl Marx

Sus servidores públicos le sirven bien.
—Adlai E. Stevenson

Usted verá algo nuevo. Dos cosas. Y las llamo Cosa Uno y Cosa Dos.
—Dr. Theodor Seuss Geisel

Objetivos

En este capítulo aprenderá a:

- Declarar una clase y utilizarla para crear un objeto.
- Implementar los comportamientos de una clase como métodos.
- Implementar los atributos de una clase como variables de instancia y propiedades.
- Llamar a los métodos de un objeto para hacer que realicen sus tareas.
- Conocer cuáles son las variables de instancia de una clase y las variables locales de un método.
- Utilizar un constructor para inicializar los datos de un objeto.
- Conocer las diferencias entre los tipos primitivos y los tipos por referencia.

Plan general

3.1 Introducción

En la sección 1.6 le presentamos la terminología básica y los conceptos acerca de la programación orientada a objetos. En este capítulo le presentaremos un marco de trabajo simple para organizar aplicaciones orientadas a objetos en Java. Por lo general, las aplicaciones que desarrollará en este libro consistirán de dos o más clases. Si se vuelve parte de un equipo de desarrollo en la industria, podría trabajar en aplicaciones que contengan cientos, o incluso miles de clases.

Primero explicaremos el concepto de las clases con un ejemplo real. Después presentaremos cinco aplicaciones para demostrarle cómo crear y utilizar sus propias clases. Los primeros cuatro ejemplos empiezan nuestro caso de estudio acerca de cómo desarrollar una clase tipo libro de calificaciones, que los instructores pueden utilizar para mantener las calificaciones de las pruebas de sus estudiantes. Ampliaremos este ejemplo práctico en los capítulos 4, 5 y 7. El último ejemplo en este capítulo introduce los números de punto flotante (números que contienen puntos decimales) en una clase tipo cuenta bancaria, la cual mantiene el saldo de un cliente.

3.2 Declaración de una clase con un método e instanciamiento de un objeto de una clase

En las secciones 2.5 y 2.8 creó un objeto de la clase *existente* `Scanner`, y después lo utilizó para leer datos mediante el teclado. En esta sección creará una *nueva* clase y después la utilizará para crear un objeto. Comenzaremos por declarar las clases `LibroCalificaciones` (figura 3.1) y `PruebaLibroCalificaciones` (figura 3.2). La clase `LibroCalificaciones` (declarada en el archivo `LibroCalificaciones.java`) se utilizará para mostrar un mensaje en la pantalla (figura 3.2), para dar la bienvenida al instructor a la aplicación del libro de calificaciones. La clase `PruebaLibroCalificaciones` (declarada en el archivo `PruebaLibroCalificaciones.java`) es una clase de aplicación en la que el método `main` creará y utilizará un objeto de la clase `LibroCalificaciones`. *Cada declaración de clase que comienza con la palabra clave* `public` *debe almacenarse en un archivo que tenga el mismo nombre que la clase, y que termine con la extensión de archivo* `.java`. Por lo tanto, las clases `LibroCalificaciones` y `PruebaLibroCalificaciones` deben declararse en archivos separados, ya que cada clase se declara como `public`.

La clase `LibroCalificaciones`

La declaración de la clase `LibroCalificaciones` (figura 3.1) contiene un método llamado `mostrarMensaje` (líneas 7-10), el cual muestra un mensaje en la pantalla. Necesitamos crear un objeto de esta clase y llamar a su método para hacer que se ejecute la línea 9 y que muestre su mensaje.

La *declaración de la clase* empieza en la línea 4. La palabra clave `public` es un **modificador de acceso**. Por ahora, simplemente declararemos cada clase como `public`. Toda declaración de clase contiene la

```
1  // Fig. 3.1: LibroCalificaciones.java
2  // Declaración de una clase con un método.
3
4  public class LibroCalificaciones
5  {
6     // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
7     public void mostrarMensaje()
8     {
9        System.out.println( "Bienvenido al Libro de calificaciones!" );
10    } // fin del método mostrarMensaje
11 } // fin de la clase LibroCalificaciones
```

Fig. 3.1 | Declaración de una clase con un método.

palabra clave `class`, seguida de inmediato por el nombre de la clase. El cuerpo de toda clase se encierra entre un par de llaves izquierda y una derecha, como en las líneas 5 y 11 de la clase `LibroCalificaciones`.

En el capítulo 2, cada clase que declaramos tenía un método llamado `main`. La clase `LibroCalificaciones` también tiene un método: `mostrarMensaje` (líneas 7-10). Recuerde que `main` es un método especial, que *siempre* es llamado, automáticamente, por la Máquina Virtual de Java (JVM) a la hora de ejecutar una aplicación. La mayoría de los métodos no se llaman en forma automática. Como veremos en breve, es necesario llamar al método `mostrarMensaje` de manera explícita para indicarle que haga su trabajo.

La declaración del método comienza con la palabra clave `public` para indicar que el método está "disponible al público": los métodos de otras clases pueden llamarlo. A continuación está el **tipo de valor de retorno** del método, el cual especifica el tipo de datos que devuelve el método a quien lo llamó después de realizar su tarea. El tipo de valor de retorno `void` indica que este método realizará una tarea pero *no* devolverá (es decir, regresará) información al **método que lo llamó**. Ya hemos utilizado métodos que devuelven información; por ejemplo, en el capítulo 2 utilizó el método `nextInt` de `Scanner` para recibir un entero escrito por el usuario desde el teclado. Cuando `nextInt` recibe un valor de entrada, devuelve ese valor para utilizarlo en el programa.

El nombre del método, `mostrarMensaje`, va después del tipo de valor de retorno. Por convención, los nombres de los métodos comienzan con una letra minúscula, y el resto de las palabras en el nombre con mayúsculas. Los paréntesis después del nombre del método indican que éste es un método. Un conjunto vacío de paréntesis, como se muestra en la línea 7, indica que este método no requiere información adicional para realizar su tarea. La línea 7 se conoce comúnmente como el **encabezado del método**. El cuerpo de cada método se delimita mediante las llaves izquierda y derecha, como en las líneas 8 y 10.

El cuerpo de un método contiene una o varias instrucciones que realizan su trabajo. En este caso, el método contiene una instrucción (línea 9) que muestra el mensaje "`Bienvenido al Libro de calificaciones!`", seguido de una nueva línea (debido a `println`) en la ventana de comandos. Una vez que se ejecuta esta instrucción, el método ha completado su trabajo.

La clase `PruebaLibroCalificaciones`

A continuación, nos gustaría utilizar la clase `LibroCalificaciones` en una aplicación. Como aprendió en el capítulo 2, el método `main` empieza la ejecución de *todas* las aplicaciones. Una clase que contiene el método `main` empieza la ejecución de una aplicación de Java. La clase `LibroCalificaciones` *no* es una aplicación, ya que *no* contiene a `main`. Por lo tanto, si trata de ejecutar `LibroCalificaciones` escribiendo `java LibroCalificaciones` en la ventana de comandos, se producirá un mensaje de error. Esto no fue un problema en el capítulo 2, ya que cada clase que declaramos tenía un método `main`. Para corregir este problema, debemos declarar una clase separada que contenga un método `main`, o colocar un método `main` en la clase `LibroCalificaciones`. Para ayudarlo a prepararse para los pro-

gramas más extensos que encontrará más adelante en este libro y en la industria, utilizamos una clase separada (`PruebaLibroCalificaciones` en este ejemplo) que contiene el método `main` para probar cada nueva clase que vayamos a crear en este capítulo. Algunos programadores se refieren a este tipo de clases como una *clase controladora*.

La declaración de la clase `PruebaLibroCalificaciones` (figura 3.2) contiene el método `main` que controlará la ejecución de nuestra aplicación. La declaración de la clase `PruebaLibroCalificaciones` empieza en la línea 4 y termina en la línea 15. La clase *sólo* contiene un método `main`, algo común en muchas clases que empiezan la ejecución de una aplicación.

```
1  // Fig. 3.2: PruebaLibroCalificaciones.java
2  // Crea un objeto LibroCalificaciones y llama a su método mostrarMensaje.
3
4  public class PruebaLibroCalificaciones
5  {
6     // el método main empieza la ejecución del programa
7     public static void main( String[] args )
8     {
9        // crea un objeto LibroCalificaciones y lo asigna a miLibroCalificaciones
10       LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones();
11
12       // llama al método mostrarMensaje de miLibroCalificaciones
13       miLibroCalificaciones.mostrarMensaje();
14    } // fin de main
15 } // fin de la clase PruebaLibroCalificaciones
```

```
Bienvenido al Libro de calificaciones!
```

Fig. 3.2 | Cómo crear un objeto de la clase `LibroCalificaciones` y llamar a su método `mostrarMensaje`.

Las líneas 7 a la 14 declaran el método `main`. Una parte clave para permitir que la JVM localice y llame al método `main` para empezar la ejecución de la aplicación es la palabra clave `static` (línea 7), la cual indica que `main` es un método `static`. *Un método `static` es especial, ya que puede llamarse sin tener que crear primero un objeto de la clase en la cual se declara ese método.* En el capítulo 6, Métodos: un análisis más detallado, analizaremos los métodos `static`.

En esta aplicación nos gustaría llamar al método `mostrarMensaje` de la clase `LibroCalificaciones` para mostrar el mensaje de bienvenida en la ventana de comandos. Por lo general, no podemos llamar a un método que pertenece a otra clase sino hasta crear un objeto de esa clase, como se muestra en la línea 10. Empezaremos por declarar la variable `miLibroCalificaciones`. El tipo de la variable es `LibroCalificaciones`: la clase que declaramos en la figura 3.1. Cada nueva *clase* que creamos se convierte en un nuevo *tipo*, que puede usarse para declarar variables y crear objetos. Usted puede declarar nuevos tipos de clases según lo necesite; ésta es una razón por la cual Java se conoce como un **lenguaje extensible**.

La variable `miLibroCalificaciones` se inicializa (línea 10) con el resultado de la **expresión de creación de instancia de clase** `new LibroCalificaciones()`. La palabra clave **new** crea un nuevo objeto de la clase especificada a la derecha de la palabra clave (es decir, `LibroCalificaciones`). Los paréntesis a la derecha de `LibroCalificaciones` son obligatorios. Como veremos en la sección 3.6, esos paréntesis en combinación con el nombre de una clase representan una llamada a un **constructor**, que es similar a un método, pero se utiliza sólo cuando se crea un objeto para *inicializar* los datos de éste. En esa sección verá que los datos pueden colocarse entre paréntesis para especificar los *valores iniciales* para los datos del objeto. Por ahora, sólo dejaremos los paréntesis vacíos.

Así como podemos usar el objeto `System.out` para llamar a sus métodos `print`, `printf` y `println`, también podemos usar el objeto `miLibroCalificaciones` para llamar a su método `mostrarMensaje`. La línea 13 llama al método `mostrarMensaje` (líneas 7-10 de la figura 3.1), mediante el uso de `miLibroCalificaciones` seguida de un **separador punto** (`.`), el nombre del método `mostrarMensaje` y un conjunto vacío de paréntesis. Esta llamada hace que el método `mostrarMensaje` realice su tarea. La llamada a este método difiere de las del capítulo 2 en las que se mostraba la información en una ventana de comandos; cada una de estas llamadas al método proporcionaba argumentos que especificaban los datos a mostrar. Al inicio de la línea 13, "`miLibroCalificaciones.`" indica que `main` debe utilizar el objeto `miLibroCalificaciones` que se creó en la línea 10. La línea 7 de la figura 3.1 indica que el método `mostrarMensaje` tiene una *lista de parámetros vacía*; es decir, `mostrarMensaje` *no* requiere información adicional para realizar su tarea. Por esta razón, la llamada al método (línea 13 de la figura 3.2) especifica un conjunto vacío de paréntesis después del nombre del método, para indicar que *no* se van a pasar *argumentos* al método `mostrarMensaje`. Cuando el método `mostrarMensaje` completa su tarea, el método `main` continúa su ejecución en la línea 14. Éste es el final del método `main`, por lo que el programa termina.

Cualquier clase puede contener un método `main`. La JVM lo invoca *sólo* en la clase que se utiliza para ejecutar la aplicación. Si una aplicación tiene varias clases que contengan `main`, el que se invoque será el de la clase nombrada en el comando `java`.

Compilación de una aplicación con varias clases

Debe compilar las clases de las figuras 3.1 y 3.2 antes de poder ejecutar la aplicación. Primero, cambie al directorio que contiene los archivos de código fuente de la aplicación. Después, escriba el comando

```
javac LibroCalificaciones.java PruebaLibroCalificaciones.java
```

para compilar *ambas* clases a la vez. Si el directorio que contiene la aplicación sólo incluye los archivos de ésta, puede compilar *todas* las clases que haya en el directorio con el comando

```
javac *.java
```

El asterisco (`*`) en `*.java` indica que deben compilarse *todos* los archivos en el directorio actual que terminen con la extensión de nombre de archivo "`.java`".

Diagrama de clases de UML para la clase `LibroCalificaciones`

La figura 3.3 presenta un **diagrama de clases de UML** para la clase `LibroCalificaciones` de la figura 3.1. En UML, cada clase se modela en un diagrama de clases en forma de un rectángulo con tres compartimientos. El compartimiento superior contiene el nombre de la clase, centrado en forma horizontal y en negrita. El compartimiento de en medio contiene los atributos de la clase, que en Java corresponden a las variables de instancia (las cuales analizaremos en la sección 3.4). En la figura 3.3, el compartimiento de en medio está vacío, ya que esta clase `LibroCalificaciones` *no* tiene atributos. El compartimiento inferior contiene las **operaciones** de la clase, que en Java corresponden a los métodos.

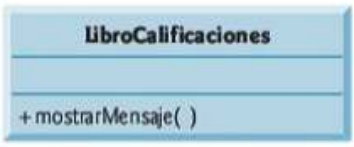


Fig. 3.3 | Diagrama de clases de UML, el cual indica que la clase `LibroCalificaciones` tiene una operación `public` llamada `mostrarMensaje`.

Para modelar las operaciones, UML lista el nombre de la operación precedido por un modificador de acceso (en este caso, +) y seguido de un conjunto de paréntesis. La clase `LibroCalificaciones` tiene un solo método llamado `mostrarMensaje`, por lo que el compartimiento inferior de la figura 3.3 lista una operación con este nombre. El método `mostrarMensaje` *no* requiere información adicional para realizar sus tareas, por lo que los paréntesis que van después del nombre del método en el diagrama de clases están *vacíos*, de igual forma que como aparecieron en la declaración del método, en la línea 7 de la figura 3.1. El signo más (+) que va antes del nombre de la operación indica que `mostrarMensaje` es una operación `pública` en UML (es decir, un método `public` en Java). Utilizaremos los diagramas de clases de UML con frecuencia para sintetizar los atributos y las operaciones de una clase.

3.3 Declaración de un método con un parámetro

En nuestra analogía del auto de la sección 1.6, hablamos sobre el hecho de que al pisar el pedal del acelerador se envía un mensaje al auto para que *realice la tarea*: que vaya más rápido. Pero, *¿qué tan rápido* debería acelerar el auto? Como sabe, cuanto más pisa el pedal, mayor será la aceleración del auto. Por lo tanto, el mensaje para el auto en realidad involucra tanto la *tarea a realizar* como *información adicional* que ayuda al auto a ejecutar su tarea. A la información adicional se le conoce como **parámetro**; el valor del parámetro ayuda al auto a determinar qué tan rápido debe acelerar. De manera similar, un método puede requerir uno o más parámetros que representan la información adicional que necesita para realizar su tarea. Los parámetros se definen en una **lista de parámetros** separada por comas, ubicada dentro de los paréntesis que van después del nombre del método. Cada parámetro debe especificar un tipo y un nombre de variable. La lista de parámetros puede contener cualquier número de éstos, o inclusive ninguno. Los paréntesis vacíos después del nombre del método (como en la línea 7 de la figura 3.1) indican que un método *no* requiere parámetros.

Argumentos para un método

La llamada a un método proporciona valores (llamados *argumentos*) para cada uno de los parámetros de ese método. Por ejemplo, el método `System.out.println` requiere un argumento que especifica los datos a mostrar en una ventana de comandos. De manera similar, para realizar un depósito en una cuenta bancaria, un método llamado `deposito` especifica un parámetro que representa el monto a depositar. Cuando se hace una llamada al método `deposito`, se asigna al parámetro del método un valor como argumento, que representa el monto a depositar. Entonces, el método realiza un depósito por ese monto.

Declaración de una clase con un método que tiene un parámetro

Ahora vamos a declarar la clase `LibroCalificaciones` (figura 3.4), con un método `mostrarMensaje` que muestra el nombre del curso como parte del mensaje de bienvenida (en la figura 3.5 podrá ver la ejecución de ejemplo). Este nuevo método requiere un parámetro que representa el nombre del curso a imprimir en pantalla.

Antes de hablar sobre las nuevas características de la clase `LibroCalificaciones`, veamos cómo se utiliza la nueva clase desde el método `main` de la clase `PruebaLibroCalificaciones` (figura 3.5). La línea 12 crea un objeto `Scanner` llamado `entrada`, para recibir el nombre del curso escrito por el usuario. La línea 15 crea el objeto `miLibroCalificaciones` de la clase `LibroCalificaciones`. La línea 18 pide al usuario que escriba el nombre de un curso. La línea 19 lee el nombre que introduce el usuario y lo asigna a la variable `nombreDelCurso`, mediante el uso del método `nextLine` de `Scanner` para realizar la operación de entrada. El usuario escribe el nombre del curso y oprime *Intro* para enviarlo al programa. Al oprimir *Intro* se inserta un carácter de nueva línea al final de los caracteres escritos por el usuario. El método `nextLine` los lee hasta encontrar la nueva línea, luego devuelve un objeto `String` que contiene los caracteres hasta la nueva línea, pero *sin* incluirla. El carácter de nueva línea se *descarta*.

```
1  // Fig. 3.4: LibroCalificaciones.java
2  // Declaración de una clase con un método que tiene un parámetro.
3
4  public class LibroCalificaciones
5  {
6     // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
7     public void mostrarMensaje( String nombreDelCurso )
8     {
9        System.out.printf( "Bienvenido al libro de calificaciones para\n%s!\n",
10          nombreDelCurso );
11    } // fin del método mostrarMensaje
12 } // fin de la clase LibroCalificaciones
```

Fig. 3.4 | Declaración de una clase con un método que tiene un parámetro.

```
1  // Fig. 3.5: PruebaLibroCalificaciones.java
2  // Crea un objeto LibroCalificaciones y pasa un objeto String
3  // a su método mostrarMensaje.
4  import java.util.Scanner; // el programa usa la clase Scanner
5
6  public class PruebaLibroCalificaciones
7  {
8     // el método main empieza la ejecución del programa
9     public static void main( String[] args )
10    {
11       // crea un objeto Scanner para obtener la entrada de la ventana de comandos
12       Scanner entrada = new Scanner( System.in );
13
14       // crea un objeto LibroCalificaciones y lo asigna a miLibroCalificaciones
15       LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones();
16
17       // pide y recibe el nombre del curso como entrada
18       System.out.println( "Escriba el nombre del curso:" );
19       String nombreDelCurso = entrada.nextLine(); // lee una línea de texto
20       System.out.println(); // imprime una línea en blanco
21
22       // llama al método mostrarMensaje de miLibroCalificaciones
23       // y pasa nombreDelCurso como argumento
24       miLibroCalificaciones.mostrarMensaje( nombreDelCurso );
25    } // fin de main
26 } // fin de la clase PruebaLibroCalificaciones
```

```
Escriba el nombre del curso:
CS101 Introduccion a la programacion en Java

Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en Java!
```

Fig. 3.5 | Cómo crear un objeto `LibroCalificaciones` y pasar un objeto `String` a su método `mostrarMensaje`.

La clase `Scanner` también cuenta con un método similar (`next`) para leer palabras individuales. Cuando el usuario oprime *Intro* después de escribir la entrada, el método `next` lee caracteres hasta encontrar un *carácter de espacio en blanco* (espacio, tabulador o nueva línea), y después devuelve un objeto

`String` que contiene los caracteres hasta el carácter de espacio en blanco (que se descarta), pero *sin* incluirlo. No se pierde toda la información que va después del primer carácter de espacio en blanco; estará disponible para que la lean otras instrucciones que llamen a los métodos de `Scanner`, más adelante en el programa. La línea 20 imprime una línea en blanco.

La línea 24 llama al método `mostrarMensaje` de `miLibroCalificaciones`. La variable `nombreDelCurso` entre paréntesis es el *argumento* que se pasa al método `mostrarMensaje`, para que éste pueda realizar su tarea. El valor de la variable `nombreDelCurso` en `main` se convierte en el valor del *parámetro* `nombreDelCurso` del método `mostrarMensaje`, en la línea 7 de la figura 3.4. Al ejecutar esta aplicación, observe que el método `mostrarMensaje` imprime en pantalla el nombre que usted escribió como parte del mensaje de bienvenida (figura 3.5).

Más sobre los argumentos y los parámetros

En la figura 3.4, la lista de parámetros de `mostrarMensaje` (línea 7) declara un parámetro que indica que el método requiere un objeto `String` para realizar su trabajo. En el instante en que se llama al método, el valor del argumento en la llamada se asigna al parámetro correspondiente (`nombreDelCurso`) en el encabezado del método. Después, el cuerpo del método utiliza el valor del parámetro `nombreDelCurso`. Las líneas 9 y 10 de la figura 3.4 muestran el valor del parámetro `nombreDelCurso`, mediante el uso del especificador de formato `%s` en la cadena de formato de `printf`. El nombre de la variable de parámetro (`nombreDelCurso` en la figura 3.4, línea 7) puede ser *igual o distinto* al nombre de la variable de argumento (`nombreDelCurso` en la figura 3.5, línea 24).

El número de argumentos en la llamada a un método *debe* coincidir con el de los parámetros en la lista de parámetros de la declaración del método. Además, los tipos de los argumentos en la llamada al método deben ser "consistentes con" los de los parámetros correspondientes en la declaración del método (como veremos en el capítulo 6, no siempre se requiere que el tipo de un argumento y el de su correspondiente parámetro sean *idénticos*). En nuestro ejemplo, la llamada al método pasa un argumento de tipo `String` (`nombreDelCurso` se declara como `String` en la línea 19 de la figura 3.5) y la declaración del método especifica un parámetro de tipo `String` (`nombreDelCurso` se declara como `String` en la línea 7 de la figura 3.4). Por lo tanto, en este ejemplo, el tipo del argumento en la llamada al método coincide exactamente con el tipo del parámetro en el encabezado del método.

Diagrama de clases de UML actualizado para la clase `LibroCalificaciones`

El diagrama de clases de UML de la figura 3.6 modela la clase `LibroCalificaciones` de la figura 3.4. Al igual que la figura 3.1, esta clase `LibroCalificaciones` contiene la operación `public` llamada `mostrarMensaje`. Sin embargo, esta versión de `mostrarMensaje` tiene un parámetro. La forma en que UML modela un parámetro es un poco distinta a la de Java, ya que lista el nombre de éste, seguido de dos puntos y su tipo entre paréntesis, después del nombre de la operación. UML tiene sus propios tipos de datos, que son similares a los de Java (pero como veremos, no todos los tipos de datos de UML tienen los mismos nombres que los correspondientes en Java). El tipo `String` de UML corresponde al tipo `String` de Java. El método `mostrarMensaje` de `LibroCalificaciones` (figura 3.4) tiene un parámetro `String` llamado `nombreDelCurso`, por lo que en la figura 3.6 se lista a `nombreDelCurso : String` entre los paréntesis que van después de `mostrarMensaje`.

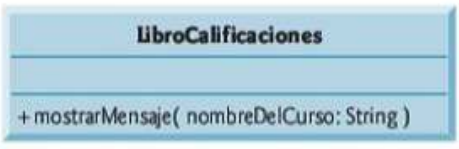


Fig. 3.6 | Diagrama de clases de UML, que indica que la clase `LibroCalificaciones` tiene una operación llamada `mostrarMensaje`, con un parámetro llamado `nombreDelCurso` de tipo `String` de UML.

Observaciones acerca del uso de las declaraciones `import`

Observe la declaración `import` en la figura 3.5 (línea 4). Esto indica al compilador que el programa utiliza la clase `Scanner`. ¿Por qué necesitamos importar la clase `Scanner`, pero no las clases `System`, `String` o `LibroCalificaciones`? Las clases `System` y `String` están en el paquete `java.lang`, que se importa de manera implícita en *todo* programa de Java, por lo que todos los programas pueden usar las clases de ese paquete *sin tener que* importarlas de manera explícita. La mayoría de las otras clases que utilizará en los programas de Java deben importarse de manera explícita.

Hay una relación especial entre las clases que se compilan en el mismo directorio en el disco, como las clases `LibroCalificaciones` y `PruebaLibroCalificaciones`. De manera predeterminada, se considera que dichas clases se encuentran en el mismo paquete; a éste se le conoce como el **paquete predeterminado**. Las clases en el mismo paquete se *importan implícitamente* en los archivos de código fuente de las otras clases en el mismo paquete. Por ende, *no* se requiere una declaración `import` cuando la clase en un paquete utiliza a otra en el mismo paquete; como cuando `PruebaLibroCalificaciones` utiliza a la clase `LibroCalificaciones`.

La declaración `import` en la línea 4 *no* es obligatoria si siempre hacemos referencia a la clase `Scanner` como `java.util.Scanner`, que contiene el *nombre completo del paquete y de la clase*. Esto se conoce como el **nombre de clase completamente calificado**. Por ejemplo, la línea 12 podría escribirse como

```
java.util.Scanner entrada = new java.util.Scanner( System.in );
```

Observación de ingeniería de software 3.1

El compilador de Java no requiere declaraciones `import` en un archivo de código fuente de Java, si se especifica el nombre de clase completamente calificado cada vez que se utilice el nombre de una clase en el código fuente. La mayoría de los programadores de Java prefieren usar declaraciones `import`.

3.4 Variables de instancia, métodos *establecer* y métodos *obtener*

En el capítulo 2 declaramos todas las variables de una aplicación en el método `main`. Las variables que se declaran en el cuerpo de un método específico se conocen como **variables locales**, y sólo se pueden utilizar en ese método. Cuando termina ese método, se pierden los valores de sus variables locales. En la sección 1.6 vimos que un objeto tiene *atributos* que lleva consigo cuando se utiliza en un programa. Los cuales existen antes de que un objeto llame a un método, al momento y después de que éste se ejecuta.

Por lo general, una clase consiste en uno o más métodos que manipulan los atributos pertenecientes a un objeto específico de la clase. Los atributos se representan como variables en la declaración de la clase. Dichas variables se llaman **campos** y se declaran *dentro* de la declaración de una clase, pero *fuera* de los cuerpos de las declaraciones de los métodos de ésta. Cuando cada objeto de una clase mantiene su propia copia de un atributo, el campo que representa a ese atributo se conoce también como **variable de instancia**; cada objeto (instancia) de la clase tiene una instancia separada de la variable en memoria. El ejemplo en esta sección demuestra una clase `LibroCalificaciones`, que contiene una variable de instancia llamada `nombreDelCurso` para representar el nombre del curso de un objeto `LibroCalificaciones` específico.

La clase `LibroCalificaciones` *con una variable de instancia, un método* establecer *y un método* obtener

En nuestra siguiente aplicación (figuras 3.7 y 3.8), la clase `LibroCalificaciones` (figura 3.7) mantiene el nombre del curso como una variable de instancia, para que pueda usarse o modificarse en cualquier momento, durante la ejecución de una aplicación. Esta clase contiene tres métodos: `establecerNombreDelCurso`, `obtenerNombreDelCurso` y `mostrarMensaje`. El método `establecerNombreDelCurso` almacena el nombre de un curso en un `LibroCalificaciones`. El método `obtenerNombreDelCurso` obtiene el nombre del curso de un `LibroCalificaciones`. El método `mostrarMensaje`, que en este caso no

especifica parámetros, sigue mostrando un mensaje de bienvenida que incluye el nombre del curso; como veremos más adelante, el método ahora obtiene el nombre del curso mediante una llamada a otro método en la misma clase: `obtenerNombreDelCurso`.

```
1  // Fig. 3.7: LibroCalificaciones.java
2  // Clase LibroCalificaciones que contiene una variable de instancia nombreDelCurso
3  // y métodos para establecer y obtener su valor.
4
5  public class LibroCalificaciones
6  {
7     private String nombreDelCurso; // nombre del curso para este LibroCalificaciones
8
9     // método para establecer el nombre del curso
10    public void establecerNombreDelCurso( String nombre )
11    {
12       nombreDelCurso = nombre; // almacena el nombre del curso
13    } // fin del método establecerNombreDelCurso
14
15    // método para obtener el nombre del curso
16    public String obtenerNombreDelCurso()
17    {
18       return nombreDelCurso;
19    } // fin del método obtenerNombreDelCurso
20
21    // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
22    public void mostrarMensaje()
23    {
24       // esta instrucción llama a obtenerNombreDelCurso para obtener el
25       // nombre del curso que representa este LibroCalificaciones
26       System.out.printf( "Bienvenido al libro de calificaciones para\n%s!\n",
27          obtenerNombreDelCurso() );
28    } // fin del método mostrarMensaje
29 } // fin de la clase LibroCalificaciones
```

Fig. 3.7 | Cómo crear un objeto `LibroCalificaciones` y pasar un objeto `String` a su método `mostrarMensaje`.

Por lo general, un instructor enseña más de un curso, cada uno con su propio nombre. La línea 7 declara que `nombreDelCurso` es una variable de tipo `String`. Como la variable se declara *en* el cuerpo de la clase, pero *fuera* de los cuerpos de los métodos de la misma (líneas 10 a la 13, 16 a la 19 y 22 a la 28), la línea 7 es una declaración para una *variable de instancia*. Cada instancia (es decir, objeto) de la clase `LibroCalificaciones` contiene una copia de cada variable de instancia. Por ejemplo, si hay dos objetos `LibroCalificaciones`, cada objeto tiene su propia copia de `nombreDelCurso`. Un beneficio de hacer de `nombreDelCurso` una variable de instancia es que todos los métodos de la clase (en este caso, `LibroCalificaciones`) pueden manipular cualquier variable de instancia que aparezca en la clase (en este caso, `nombreDelCurso`).

Los modificadores de acceso `public` *y* `private`

La mayoría de las declaraciones de variables de instancia van precedidas por la palabra clave `private` (como en la línea 7). Al igual que `public`, la palabra clave `private` es un *modificador de acceso*. *Las variables o los métodos declarados con el modificador de acceso* `private` *son accesibles sólo para los métodos de la clase en la que se declaran*. Así, la variable `nombreDelCurso` sólo puede utilizarse en los méto-

dos `establecerNombreDelCurso`, `obtenerNombreDelCurso` y `mostrarMensaje` de (cada objeto de) la clase `LibroCalificaciones`.

El proceso de declarar variables de instancia con el modificador de acceso `private` se conoce como **ocultamiento de datos**, u ocultamiento de información. Cuando un programa crea (instancia) un objeto de la clase `LibroCalificaciones`, la variable `nombreDelCurso` se *encapsula* (oculta) en el objeto, y sólo está accesible para los métodos de la clase de ese objeto. Esto evita que una clase en otra parte del programa modifique a `nombreDelCurso` por accidente. En la clase `LibroCalificaciones`, los métodos `establecerNombreDelCurso` y `obtenerNombreDelCurso` manipulan a la variable de instancia `nombreDelCurso`.

Observación de ingeniería de software 3.2

Es necesario colocar un modificador de acceso antes de cada declaración de un campo y de un método. Por lo general, las variables de instancia deben declararse como `private` y los métodos como `public`. (Es apropiado declarar ciertos métodos como `private`, si sólo van a estar accesibles para otros métodos de la clase).

Buena práctica de programación 3.1

Preferimos listar los campos de una clase primero, para que, a medida que usted lea el código, pueda ver los nombres y tipos de las variables antes de usarlas en los métodos de la clase. Es posible listar los campos de la clase en cualquier parte de la misma, fuera de las declaraciones de sus métodos, pero si se esparcen por todo el código, éste será más difícil de leer.

Los métodos `establecerNombreDelCurso` *y* `obtenerNombreDelCurso`

El método `establecerNombreDelCurso` (líneas 10 a la 13) no devuelve datos cuando completa su tarea, por lo que su tipo de valor de retorno es `void`. El método recibe un parámetro (`nombre`), el cual representa el nombre del curso que se pasará al método como un argumento. La línea 12 asigna `nombre` a la variable de instancia `nombreDelCurso`.

El método `obtenerNombreDelCurso` (líneas 16 a la 19) devuelve un `nombreDelCurso` de un objeto `LibroCalificaciones` específico. Tiene una lista de parámetros vacía, por lo que no requiere información adicional para realizar su tarea. Este método especifica que devuelve un objeto `String`; a éste se le conoce como el tipo de valor de retorno del método. Cuando se hace una llamada a un método que especifica un tipo de valor de retorno distinto de `void` y completa su tarea, devuelve un *resultado* al método que lo llamó. Por ejemplo, cuando usted va a un cajero automático (ATM) y solicita el saldo de su cuenta, espera que el ATM le devuelva un valor que representa su saldo. De manera similar, cuando una instrucción llama al método `obtenerNombreDelCurso` en un objeto `LibroCalificaciones`, la instrucción espera recibir el nombre del curso de `LibroCalificaciones` (en este caso, un objeto `String`, como se especifica en el tipo de valor de retorno de la declaración del método).

La instrucción `return` en la línea 18 pasa el valor de la variable de instancia `nombreDelCurso` de vuelta a la instrucción que llama al método `obtenerNombreDelCurso`. Ahora considere la línea 27 del método `mostrarMensaje`, que llama al método `obtenerNombreDelCurso`. Al devolver el valor, la instrucción en las líneas 26 y 27 usa ese valor para imprimir el nombre del curso. De manera similar, si tiene un método cuadrado que devuelve el cuadrado de su argumento, es de esperarse que la instrucción

```
int resultado = cuadrado( 2 );
```

devuelva 4 del método `cuadrado` y asigne 4 a la variable `resultado`. Si tiene un método `maximo` que devuelve el mayor de tres argumentos enteros, es de esperarse que la siguiente instrucción

```
int mayor = maximo( 27, 114, 51 );
```

devuelva 114 del método `maximo` y asigne 114 a la variable `mayor`.

Las instrucciones en las líneas 12 y 18 utilizan `nombreDelCurso`, *aun cuando esta variable no se declaró en ninguno de los métodos.* Podemos utilizar `nombreDelCurso` en los métodos de la clase `LibroCalificaciones`, ya que `nombreDelCurso` es una variable de instancia de la clase.

El método `mostrarMensaje`

El método `mostrarMensaje` (líneas 22 a la 28) *no* devuelve datos cuando completa su tarea, por lo que su tipo de valor de retorno es `void`. El método *no* recibe parámetros, por lo que la lista de parámetros está vacía. Las líneas 26 y 27 imprimen un mensaje de bienvenida, que incluye el valor de la variable de instancia `nombreDelCurso`, el cual se devuelve mediante la llamada al método `obtenerNombreDelCurso` en la línea 27. Observe que un método de una clase (`mostrarMensaje` en este caso) puede llamar a otro método de la *misma* clase con sólo usar su nombre (`obtenerNombreDelCurso` en este caso).

La clase `PruebaLibroCalificaciones` *que demuestra a la clase* `LibroCalificaciones`

La clase `PruebaLibroCalificaciones` (figura 3.8) crea un objeto de la clase `LibroCalificaciones` y demuestra el uso de sus métodos. La línea 14 crea un objeto `LibroCalificaciones` y lo asigna a la variable local `miLibroCalificaciones`, de tipo `LibroCalificaciones`. Las líneas 17-18 muestran el nombre inicial del curso mediante una llamada al método `obtenerNombreDelCurso` del objeto. La primera línea de la salida muestra el nombre "`null`". *A diferencia de las variables locales, que no se inicializan de manera automática, cada campo tiene un* ***valor inicial predeterminado****: un valor que Java proporciona cuando el programador no especifica el valor inicial del campo.* Por ende, no se requiere que los campos se inicialicen de manera explícita antes de usarlos en un programa, a menos que deban hacerlo con valores *distintos* de los predeterminados. El valor predeterminado para un campo de tipo `String` (como `nombreDelCurso` en este ejemplo) es `null`, de lo cual hablaremos con más detalle en la sección 3.5.

La línea 21 pide al usuario que escriba el nombre para el curso. La variable `String` local `elNombre` (declarada en la línea 22) se inicializa con el nombre del curso que escribió el usuario, el cual se devuelve mediante la llamada al método `nextLine` del objeto `Scanner` llamado `entrada`. La línea 23 llama al método `establecerNombreDelCurso` del objeto `miLibroCalificaciones` y provee `elNombre` como argumento para el método. Cuando se hace la llamada al método, el valor del argumento se asigna al parámetro `nombre` (línea 10, figura 3.7) del método `establecerNombreDelCurso` (líneas 10 a la 13, figura 3.7). Después, el valor del parámetro se asigna a la variable de instancia `nombreDelCurso` (línea 12, figura 3.7). La línea 24 (figura 3.8) salta una línea en la salida, y después la línea 27 llama al método `mostrarMensaje` del objeto `miLibroCalificaciones` para mostrar en pantalla el mensaje de bienvenida, que contiene el nombre del curso.

```
1  // Fig. 3.8: PruebaLibroCalificaciones.java
2  // Crea y manipula un objeto LibroCalificaciones.
3  import java.util.Scanner; // el programa usa la clase Scanner
4
5  public class PruebaLibroCalificaciones
6  {
7     // el método main empieza la ejecución del programa
8     public static void main( String[] args )
9     {
10       // crea un objeto Scanner para obtener la entrada de la ventana de comandos
11       Scanner entrada = new Scanner( System.in );
12
13       // crea un objeto LibroCalificaciones y lo asigna a miLibroCalificaciones
14       LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones();
15
```

Fig. 3.8 | Creación y manipulación de un objeto `LibroCalificaciones` (parte 1 de 2).

```
16        // muestra el valor inicial de nombreDelCurso
17        System.out.printf( "El nombre inicial del curso es: %s\n\n",
18           miLibroCalificaciones.obtenerNombreDelCurso() );
19
20        // pide y lee el nombre del curso
21        System.out.println( "Escriba el nombre del curso:" );
22        String elNombre = entrada.nextLine(); // lee una linea de texto
23        miLibroCalificaciones.establecerNombreDelCurso( elNombre ); // establece el nombre
          del curso
24        System.out.println(); // imprime una linea en blanco
25
26        // muestra el mensaje de bienvenida después de especificar el nombre del curso
27        miLibroCalificaciones.mostrarMensaje();
28     } // fin de main
29  } // fin de la clase PruebaLibroCalificaciones
```

```
El nombre inicial del curso es: null

Escriba el nombre del curso:
CS101 Introduccion a la programacion en Java

Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en Java!
```

Fig. 3.8 | Creación y manipulación de un objeto `LibroCalificaciones` (parte 2 de 2).

Los métodos establecer *y* obtener

Los campos `private` de una clase pueden manipularse *sólo* mediante los métodos de esa clase. Por lo tanto, un **cliente de un objeto** (es decir, cualquier clase que llame a los métodos del objeto) llama a los métodos `public` de la clase para manipular los campos `private` de un objeto de esa clase. Esto explica por qué las instrucciones en el método `main` (figura 3.8) llaman a los métodos `establecerNombreDelCurso`, `obtenerNombreDelCurso` y `mostrarMensaje` en un objeto `LibroCalificaciones`. A menudo, las clases proporcionan métodos `public` para permitir a los clientes de la clase *establecer* (asignar valores a) u *obtener* (obtener los valores de) variables de instancia `private`. Los nombres de estos métodos no necesitan empezar con *establecer* u *obtener*, pero esta convención de nomenclatura es muy recomendada en Java, y es requerida para ciertos componentes de software especiales de Java, conocidos como JavaBeans, que pueden simplificar la programación en muchos entornos de desarrollo integrados (IDE). El método que *establece* la variable de instancia `nombreDelCurso` en este ejemplo se llama `establecerNombreDelCurso`, y el método que *obtiene* su valor se llama `obtenerNombreDelCurso`.

Diagrama de clases de UML para la clase `LibroCalificaciones` *con una variable de instancia, y métodos* establecer *y* obtener

La figura 3.9 contiene un diagrama de clases de UML actualizado para la versión de la clase `LibroCalificaciones` de la figura 3.7. Este diagrama modela la variable de instancia `nombreDelCurso` de la clase `LibroCalificaciones` como un atributo en el compartimiento intermedio de la clase. UML representa a las variables de instancia como atributos, listando el nombre del atributo, seguido de dos puntos y del tipo del atributo. El tipo de UML del atributo `nombreDelCurso` es `String`. La variable de instancia `nombreDelCurso` es `private` en Java, por lo que el diagrama de clases lista un modificador de acceso de signo menos (-) en frente del nombre del atributo correspondiente. La clase `LibroCalificaciones` contiene tres métodos `public`, por lo que el diagrama de clases lista tres operaciones en el tercer compartimiento. Recuerde que el signo más (+) antes de cada nombre de operación indica que ésta es `public`. La operación `establecerNombreDelCurso` tiene un parámetro `String` llamado `nombre`. UML indica el tipo de valor de retorno de una operación colocando dos puntos y el tipo de valor de retorno después de los paréntesis que le siguen al nombre de la operación. El método `obtenerNombreDelCurso` de la clase `LibroCalifi-`

caciones (figura 3.7) tiene un tipo de valor de retorno `String` en Java, por lo que el diagrama de clases muestra un tipo de valor de retorno `String` en UML. Las operaciones `establecerNombreDelCurso` y `mostrarMensaje` *no* devuelven valores (es decir, devuelven `void` en Java), por lo que el diagrama de clases de UML *no* especifica un tipo de valor de retorno después de los paréntesis de estas operaciones.

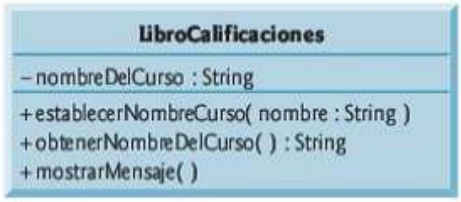


Fig. 3.9 | Diagrama de clases de UML, en el que se indica que la clase `LibroCalificaciones` tiene un atributo privado `nombreDelCurso` de tipo `String` en UML, y tres operaciones públicas: `establecerNombreDelCurso` (con un parámetro nombre de tipo `String` de UML), `obtenerNombreDelCurso` (que devuelve el tipo `String` de UML) y `mostrarMensaje`.

3.5 Comparación entre tipos primitivos y tipos por referencia

Los tipos de datos en Java se dividen en dos categorías: tipos primitivos y **tipos por referencia**. Los tipos primitivos son `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` y `double`. Todos los tipos no primitivos son tipos por referencia, por lo cual las clases, que especifican los tipos de objetos, son tipos por referencia.

Una variable de tipo primitivo puede almacenar sólo un *valor de su tipo declarado* a la vez. Por ejemplo, una variable `int` puede almacenar un número entero (como 7) a la vez. Cuando se le asigna otro valor, sustituye su valor inicial. Las variables de instancia de tipo primitivo se *inicializan de manera predeterminada*; las de los tipos `byte`, `char`, `short`, `int`, `long`, `float` y `double` se inicializan con 0, y las de tipo `boolean` se inicializan con `false`. Usted puede especificar su propio valor inicial para una variable de tipo primitivo al asignarle un valor en su declaración, como en

```
private int numeroDeEstudiantes = 10;
```

Recuerde que las variables locales *no* se inicializan de manera predeterminada.

Tip para prevenir errores 3.1

Un intento de utilizar una variable local que no se haya inicializado produce un error de compilación.

Los programas utilizan variables de tipo por referencia (que por lo general se llaman **referencias**) para almacenar las *ubicaciones* de los objetos en la memoria de la computadora. Se dice que dicha variable hace **referencia a un objeto** en el programa. Cada uno de los objetos a los que se hace referencia puede contener muchas variables de instancia. La línea 14 de la figura 3.8 crea un objeto de la clase `LibroCalificaciones`, y la variable `miLibroCalificaciones` contiene una referencia a ese objeto. *Las variables de instancia de tipo por referencia se inicializan de manera predeterminada con el valor* `null`: una palabra reservada que representa una "referencia a nada". Esto explica por qué la primera llamada a `obtenerNombreDelCurso` en la línea 18 de la figura 3.8 devolvía `null`; no se había establecido el valor de `nombreDelCurso`, por lo que se devolvía el valor inicial predeterminado `null`. En el apéndice C se muestra una lista completa de las palabras reservadas y las palabras clave.

Cuando usamos un objeto de otra clase, es obligatorio que una referencia a él **invoque** (es decir, llame) a sus métodos. En la aplicación de la figura 3.8, las instrucciones en el método `main` utilizan la

variable miLibroCalificaciones para enviar mensajes al objeto LibroCalificaciones. Estos mensajes son llamadas a métodos (como establecerNombreDelCurso y obtenerNombreDelCurso) que permiten al programa interactuar con el objeto LibroCalificaciones. Por ejemplo, la instrucción en la línea 23 utiliza a miLibroCalificaciones para enviar el mensaje establecerNombreDelCurso al objeto LibroCalificaciones. El mensaje incluye el argumento que requiere establecerNombreDelCurso para realizar su tarea. El objeto LibroCalificaciones utiliza esta información para establecer la variable de instancia nombreDelCurso. Las variables de tipo primitivo no hacen referencias a objetos, por lo que dichas variables no pueden utilizarse para invocar métodos.

Observación de ingeniería de software 3.3

El tipo declarado de una variable (por ejemplo, int, double o LibroCalificaciones) indica si la variable es de tipo primitivo o por referencia. Si el tipo de una variable no es uno de los ocho tipos primitivos, entonces es un tipo por referencia.

3.6 Inicialización de objetos mediante constructores

Como mencionamos en la sección 3.4, cuando se crea un objeto de la clase LibroCalificaciones (figura 3.7), su variable de instancia nombreCurso se inicializa con null de manera predeterminada. ¿Qué pasa si desea proporcionar el nombre de un curso a la hora de crear un objeto LibroCalificaciones? Cada clase que usted declare puede proporcionar un método especial llamado constructor, el cual puede utilizarse para inicializar un objeto de una clase al momento de crearlo. De hecho, Java *requiere* una llamada al constructor para *cada* objeto que se crea. La palabra clave new solicita memoria del sistema para almacenar un objeto, y después llama al constructor de la clase correspondiente para inicializar el objeto. La llamada se indica mediante el nombre de la clase, seguido de paréntesis. Un constructor *debe* tener el *mismo nombre* que la clase. Por ejemplo, la línea 14 de la figura 3.8 primero utiliza new para crear un objeto LibroCalificaciones. Los paréntesis vacíos después de "new LibroCalificaciones" indican una llamada sin argumentos al constructor de la clase. De manera predeterminada, el compilador proporciona un **constructor predeterminado** *sin parámetros*, en cualquier clase que no incluya un constructor en forma explícita. Cuando una clase sólo tiene el constructor predeterminado, sus variables de instancia se inicializan con sus *valores predeterminados*.

Cuando usted declara una clase, puede proporcionar su propio constructor para especificar una inicialización personalizada para los objetos de su clase. Por ejemplo, tal vez quiera especificar el nombre de un curso para un objeto LibroCalificaciones al momento de crear este objeto, como en

```
LibroCalificaciones miLibroCalificaciones =
   new LibroCalificaciones( "CS101 Introduccion a la programacion en Java" );
```

En este caso, el argumento "CS101 Introduccion a la programacion en Java" se pasa al constructor del objeto LibroCalificaciones y se utiliza para inicializar el nombreDelCurso. La instrucción anterior requiere que la clase proporcione un constructor con un parámetro String. La figura 3.10 contiene una clase LibroCalificaciones modificada con dicho constructor.

```
1  // Fig. 3.10: LibroCalificaciones.java
2  // La clase LibroCalificaciones con un constructor para inicializar el nombre del curso.
3
4  public class LibroCalificaciones
5  {
6     private String nombreDelCurso; // nombre del curso para este LibroCalificaciones
7
```

Fig. 3.10 | La clase LibroCalificaciones con un constructor para inicializar el nombre del curso (parte 1 de 2).

```
8    // el constructor inicializa nombreDelCurso con un argumento String
9    public LibroCalificaciones( String nombre ) // el nombre del constructor es el nombre
                                                 de la clase
10   {
11      nombreDelCurso = nombre; // inicializa nombreDelCurso
12   } // fin del constructor
13
14   // método para establecer el nombre del curso
15   public void establecerNombreDelCurso( String nombre )
16   {
17      nombreDelCurso = nombre; // almacena el nombre del curso
18   } // fin del método establecerNombreDelCurso
19
20   // método para obtener el nombre del curso
21   public String obtenerNombreDelCurso()
22   {
23      return nombreDelCurso;
24   } // fin del método obtenerNombreDelCurso
25
26   // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
27   public void mostrarMensaje()
28   {
29      // esta instrucción llama a obtenerNombreDelCurso para obtener el
30      // nombre del curso que este LibroCalificaciones representa
31      System.out.printf( "Bienvenido al Libro de calificaciones para\n%s!\n",
32         obtenerNombreDelCurso() );
33   } // fin del método mostrarMensaje
34 } // fin de la clase LibroCalificaciones
```

Fig. 3.10 | La clase `LibroCalificaciones` con un constructor para inicializar el nombre del curso (parte 2 de 2).

Las líneas 9 a la 12 declaran el constructor de `LibroCalificaciones`. Al igual que un método, un constructor especifica en su lista de parámetros los datos que requiere para realizar su tarea. Cuando usted crea un nuevo objeto (como haremos en la figura 3.11), estos datos se colocan en los *paréntesis que van después del nombre de la clase*. La línea 9 de la figura 3.10 indica que el constructor tiene un parámetro `String` llamado `nombre`. El `nombre` que se pasa al constructor se asigna a la variable de instancia `nombreDelCurso` en la línea 11.

La figura 3.11 inicializa los objetos `LibroCalificaciones` mediante el constructor. Las líneas 11 y 12 crean e inicializan el objeto `libroCalificaciones1` de `LibroCalificaciones`. El constructor de la clase `LibroCalificaciones` se llama con el argumento "`CS101 Introduccion a la programacion en Java`" para inicializar el nombre del curso. La expresión de creación de la instancia de la clase en las líneas 11 y 12 devuelve una referencia al nuevo objeto, el cual se asigna a la variable `libroCalificaciones1`. Las líneas 13 y 14 repiten este proceso, pero esta vez se pasa el argumento "`CS102 Estructuras de datos en Java`" para inicializar el nombre del curso para `libroCalificaciones2`. Las líneas 17 a la 20 utilizan el método `obtenerNombreDelCurso` de cada objeto para obtener los nombres de los cursos y mostrar que se inicializaron en el momento en el que se crearon los objetos. La salida confirma que cada objeto `LibroCalificaciones` mantiene su propia copia de la variable de instancia `nombreDelCurso`.

Una importante diferencia entre los constructores y los métodos es que los constructores no pueden devolver valores, por lo cual no pueden especificar un tipo de valor de retorno (ni siquiera `void`). Por lo general, los constructores se declaran como `public`. Si una clase no incluye un constructor, las variables de instancia de esa clase se inicializan con sus valores predeterminados. *Si un programador declara uno o más constructores para una clase, el compilador de Java no creará un constructor predeterminado para esa clase*. Por lo tanto, ya no podemos crear un objeto `LibroCalificaciones` con `new LibroCalificaciones()` como hicimos en los ejemplos anteriores.

```
1  // Fig. 3.11: PruebaLibroCalificaciones.java
2  // El constructor de LibroCalificaciones se utiliza para especificar el
3  // nombre del curso cada vez que se crea cada objeto LibroCalificaciones.
4
5  public class PruebaLibroCalificaciones
6  {
7     // el método main empieza la ejecución del programa
8     public static void main( String[] args )
9     {
10       // crea objeto LibroCalificaciones
11       LibroCalificaciones libroCalificaciones1 = new LibroCalificaciones(
12          "CS101 Introduccion a la programacion en Java" );
13       LibroCalificaciones libroCalificaciones2 = new LibroCalificaciones(
14          "CS102 Estructuras de datos en Java" );
15
16       // muestra el valor inicial de nombreDelCurso para cada LibroCalificaciones
17       System.out.printf( "El nombre del curso de libroCalificaciones1 es: %s\n",
18          libroCalificaciones1.obtenerNombreDelCurso() );
19       System.out.printf( "El nombre del curso de libroCalificaciones2 es: %s\n",
20          libroCalificaciones2.obtenerNombreDelCurso() );
21    } // fin de main
22 } // fin de la clase PruebaLibroCalificaciones
```

```
El nombre del curso de libroCalificaciones1 es: CS101 Introduccion a la programacion en Java
El nombre del curso de libroCalificaciones2 es: CS102 Estructuras de datos en Java
```

Fig. 3.11 | El constructor de `LibroCalificaciones` se utiliza para especificar el nombre del curso cada vez que se crea un objeto `LibroCalificaciones`.

Observación de ingeniería de software 3.4

A menos que sea aceptable la inicialización predeterminada de las variables de instancia de su clase, deberá proporcionar un constructor para asegurarse que se inicialicen en forma apropiada con valores significativos a la hora de crear cada nuevo objeto.

Agregar el constructor al diagrama de clases de UML de la clase `LibroCalificaciones`

El diagrama de clases de UML de la figura 3.12 modela la clase `LibroCalificaciones` de la figura 3.10, la cual tiene un constructor con un parámetro llamado `nombre`, de tipo `String`. Al igual que las operaciones, en un diagrama de clases, UML modela a los constructores en el tercer compartimiento de una clase.

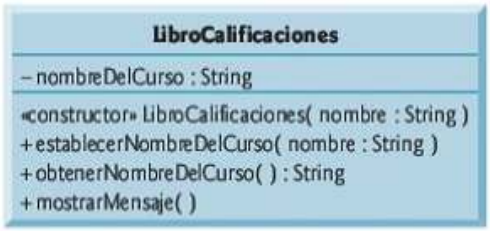


Fig. 3.12 | Diagrama de clases de UML, en el cual se indica que la clase `LibroCalificaciones` tiene un constructor con un parámetro nombre del tipo `String` de UML.

Para diferenciar a un constructor de las operaciones de una clase, UML requiere que se coloque la palabra "constructor" entre los signos « y » antes del nombre del constructor. Es *costumbre* listar los constructores *antes* de otras operaciones en el tercer compartimiento.

Constructores con varios parámetros

Algunas veces es conveniente inicializar objetos con varios elementos de datos. En el ejercicio 3.11, le pedimos que almacene el nombre del curso y del instructor en un objeto `LibroCalificaciones`. En este caso, se modifica el constructor de `LibroCalificaciones` para que reciba dos parámetros `String`, como en

```
public LibroCalificaciones( String nombreDelCurso, String nombreDelInstructor )
```

y llamamos al constructor de `LibroCalificaciones` de la siguiente manera:

```
LibroCalificaciones libroCalificaciones = new LibroCalificaciones(
   "CS101 Introduccion a la programacion en Java", "Sue Green" );
```

3.7 Los números de punto flotante y el tipo double

Ahora dejaremos por un momento nuestro caso de estudio con la clase `LibroCalificaciones` para declarar una clase llamada `Cuenta`, la cual mantiene el saldo de una cuenta bancaria. La mayoría de los saldos de las cuentas no son números enteros (por ejemplo, 0, –22 y 1024). Por esta razón, la clase `Cuenta` representa el saldo de las cuentas como un **número de punto flotante** (es decir, un número con un punto decimal, como 7.33, 0.0975 o 1000.12345). Java cuenta con dos tipos primitivos para almacenar números de punto flotante en la memoria: `float` y `double`. La principal diferencia entre ellos es que las variables tipo `double` pueden almacenar números con mayor magnitud y detalle (más dígitos a la derecha del punto decimal; lo que también se conoce como **precisión** del número) que las variables `float`.

Precisión de los números de punto flotante y requerimientos de memoria

Las variables de tipo `float` representan **números de punto flotante de precisión simple** y pueden representar hasta *siete dígitos significativos*. Las variables de tipo `double` representan **números de punto flotante de precisión doble**. Éstos requieren el doble de memoria que las variables `float` y proporcionan 15 *dígitos significativos*; aproximadamente el doble de precisión de las variables `float`. Para el rango de valores requeridos por la mayoría de los programas, debe bastar con las variables de tipo `float`, pero podemos utilizar variables tipo `double` para "ir a la segura". En algunas aplicaciones, incluso hasta las variables de tipo `double` serán inadecuadas. La mayoría de los programadores representan los números de punto flotante con el tipo `double`. De hecho, Java trata a todos los números de punto flotante que escribimos en el código fuente de un programa (como 7.33 y 0.0975) como valores `double` de manera predeterminada. Dichos valores en el código fuente se conocen como **literales de punto flotante**. En el apéndice D, Tipos primitivos, puede consultar los rangos de los valores para los tipos `float` y `double`.

Aunque los números de punto flotante no son siempre 100% precisos, tienen numerosas aplicaciones. Por ejemplo, cuando hablamos de una temperatura corporal "normal" de 36.8, no necesitamos una precisión con un número extenso de dígitos. Cuando leemos la temperatura en un termómetro como 36.8, en realidad podría ser 36.7999473210643. Si consideramos a este número simplemente como 36.8, está bien para la mayoría de las aplicaciones en las que se trabaja con las temperaturas corporales. Debido a la naturaleza imprecisa de los números de punto flotante, se prefiere el tipo `double` al tipo `float` ya que las variables `double` pueden representar números de punto flotante con más precisión. Por esta razón, utilizaremos el tipo `double` a lo largo de este libro. Para los números precisos de punto flotante, Java cuenta con la clase `BigDecimal` (paquete `java.math`).

Los números de punto flotante también surgen como resultado de la división. En la aritmética convencional, cuando dividimos 10 entre 3 el resultado es 3.3333333..., y la secuencia de números 3

se repite en forma indefinida. La computadora asigna sólo una cantidad fija de espacio para almacenar un valor de este tipo, por lo que, sin duda, el valor de punto flotante almacenado sólo puede ser una aproximación.

La clase `Cuenta` *con una variable de instancia de tipo* `double`

Nuestra siguiente aplicación (figuras 3.13 y 3.14) contiene una clase llamada `Cuenta` (figura 3.13), la cual mantiene el saldo de una cuenta bancaria. Un banco ordinario da servicio a muchas cuentas, cada una con su propio saldo, por lo que la línea 7 declara una variable de instancia, de tipo `double`, llamada `saldo`. La variable saldo es una variable de instancia, ya que está declarada en el cuerpo de la clase pero fuera de las declaraciones de los métodos de la misma (líneas 10 a la 16, 19 a la 22 y 25 a la 28). Cada instancia (objeto) de la clase `Cuenta` contiene su propia copia de `saldo`.

```
1  // Fig. 3.13: Cuenta.java
2  // La clase Cuenta con un constructor para validar e
3  // inicializar la variable de instancia saldo de tipo double.
4
5  public class Cuenta
6  {
7     private double saldo; // variable de instancia que almacena el saldo
8
9     // constructor
10    public Cuenta( double saldoInicial )
11    {
12       // valida que saldoInicial sea mayor que 0.0;
13       // si no lo es, saldo se inicializa con el valor predeterminado 0.0
14       if ( saldoInicial > 0.0 )
15          saldo = saldoInicial;
16    } // fin del constructor de Cuenta
17
18    // abona (suma) un monto a la cuenta
19    public void abonar( double monto )
20    {
21       saldo = saldo + monto; // suma el monto al saldo
22    } // fin del método abonar
23
24    // devuelve el saldo de la cuenta
25    public double obtenerSaldo()
26    {
27       return saldo; // proporciona el valor de saldo al método que hizo la llamada
28    } // fin del método obtenerSaldo
29 } // fin de la clase Cuenta
```

Fig. 3.13 | La clase `Cuenta` con un constructor para validar e inicializar la variable de instancia `saldo` de tipo `double`.

La clase tiene un constructor y dos métodos. Debido a que es común que alguien abra una cuenta para depositar dinero de inmediato, el constructor (líneas 10 a la 16) recibe un parámetro llamado `saldoInicial` de tipo `double`, el cual representa el *saldo inicial* de la cuenta. Las líneas 14 y 15 aseguran que `saldoInicial` sea mayor que `0.0`. De ser así, el valor de `saldoInicial` se asigna a la variable de instancia saldo. En caso contrario, `saldo` permanece en `0.0`, su valor inicial predeterminado.

El método `abonar` (líneas 19 a la 22) *no* devuelve datos cuando completa su tarea, por lo que su tipo de valor de retorno es `void`. El método recibe un parámetro llamado `monto`: un valor `double` que

se sumará al saldo. La línea 21 suma `monto` al valor actual de `saldo`, y después asigna el resultado a `saldo` (con lo cual se sustituye el monto del saldo anterior).

El método `obtenerSaldo` (líneas 25 a la 28) permite a los clientes de la clase (otras clases que utilicen esta clase) obtener el valor del `saldo` de un objeto `Cuenta` específico. El método especifica el tipo de valor de retorno `double` y una lista de parámetros vacía.

Observe una vez más que las instrucciones en las líneas 15, 21 y 27 utilizan la variable de instancia `saldo`, aún y cuando *no* se declaró en ninguno de los métodos. Podemos usar saldo en estos métodos, ya que es una variable de instancia de la clase.

La clase **PruebaCuenta** *que utiliza a la clase* **Cuenta**

La clase `PruebaCuenta` (figura 3.14) crea dos objetos `Cuenta` (líneas 10 y 11) y los inicializa con `50.00` y `-7.53`, respectivamente. Las líneas 14 a la 17 imprimen el saldo en cada objeto `Cuenta` mediante una llamada al método `obtenerSaldo de Cuenta`. Cuando se hace una llamada al método `obtenerSaldo` para `cuenta1` en la línea 15, se devuelve el valor del saldo de `cuenta1` de la línea 27 en la figura 3.13, y se imprime en pantalla mediante la instrucción `System.out.printf` (figura 3.14, líneas 14 y 15). De manera similar, cuando se hace la llamada al método `obtenerSaldo` para `cuenta2` en la línea 17, se devuelve el valor del saldo de `cuenta2` de la línea 27 en la figura 3.13, y se imprime en pantalla mediante la instrucción `System.out.printf` (figura 3.14, líneas 16 y 17). El saldo de `cuenta2` es `0.00`, ya que el constructor se aseguró de que la cuenta *no* pudiera empezar con un saldo negativo. El valor se imprime en pantalla mediante `printf`, con el especificador de formato `%.2f`. El especificador de formato `%f` se utiliza para imprimir valores de tipo `float` o `double`. El `.2` entre `%` y `f` representa el número de lugares decimales (2) que deben imprimirse a la derecha del punto decimal en el número de punto flotante; a esto también se le conoce como la **precisión** del número. Cualquier valor de punto flotante que se imprima con `%.2f` se redondeará a la posición de las centenas; por ejemplo, 123.457 se redondearía a 123.46, 27.333 se redondearía a 27.33 y 123.455 se redondearía a 123.46.

```
1  // Fig. 3.14: PruebaCuenta.java
2  // Entrada y salida de números de punto flotante con objetos Cuenta.
3  import java.util.Scanner;
4
5  public class PruebaCuenta
6  {
7     // el método main empieza la ejecución de la aplicación de Java
8     public static void main( String[] args )
9     {
10       Cuenta cuenta1 = new Cuenta( 50.00 ); // crea objeto Cuenta
11       Cuenta cuenta2 = new Cuenta( -7.53 ); // crea objeto Cuenta
12
13       // muestra el saldo inicial de cada objeto
14       System.out.printf( "Saldo de cuenta1: $%.2f\n",
15          cuenta1.obtenerSaldo() );
16       System.out.printf( "Saldo de cuenta2: $%.2f\n\n",
17          cuenta2.obtenerSaldo() );
18
19    // crea objeto Scanner para obtener la entrada de la ventana de comandos
20    Scanner entrada = new Scanner( System.in );
21    double montoDeposito; // deposita el monto escrito por el usuario
```

Fig. 3.14 | Entrada y salida de números de punto flotante con objetos `Cuenta` (parte 1 de 2).

```
22
23        System.out.print( "Escriba el monto a depositar para cuenta1: " ); // indicador
24        montoDeposito = entrada.nextDouble(); // obtiene entrada del usuario
25        System.out.printf( "\nsumando %.2f al saldo de cuenta1\n\n",
26           montoDeposito );
27        cuenta1.abonar( montoDeposito ); // suma al saldo de cuenta1
28
29        // muestra los saldos
30        System.out.printf( "Saldo de cuenta1: $%.2f\n",
31           cuenta1.obtenerSaldo() );
32        System.out.printf( "Saldo de cuenta2: $%.2f\n\n",
33           cuenta2.obtenerSaldo() );
34
35        System.out.print( "Escriba el monto a depositar para cuenta2: " ); // indicador
36        montoDeposito = entrada.nextDouble(); // obtiene entrada del usuario
37        System.out.printf( "\nsumando %.2f al saldo de cuenta2\n\n",
38           montoDeposito );
39        cuenta2.abonar( montoDeposito ); // suma al saldo de cuenta2
40
41        // muestra los saldos
42        System.out.printf( "Saldo de cuenta1: $%.2f\n",
43           cuenta1.obtenerSaldo() );
44        System.out.printf( "Saldo de cuenta2: $%.2f\n",
45           cuenta2.obtenerSaldo() );
46     } // fin de main
47  } // fin de la clase PruebaCuenta
```

```
Saldo de cuenta1: $50.00
Saldo de cuenta2: $0.00

Escriba el monto a depositar para cuenta1: 25.53

sumando 25.53 al saldo de cuenta1

Saldo de cuenta1: $75.53
Saldo de cuenta2: $0.00

Escriba el monto a depositar para cuenta2: 123.45

sumando 123.45 al saldo de cuenta2

Saldo de cuenta1: $75.53
Saldo de cuenta2: $123.45
```

Fig. 3.14 | Entrada y salida de números de punto flotante con objetos `Cuenta` (parte 2 de 2).

La línea 21 declara la variable local `montoDeposito` para almacenar cada monto de depósito introducido por el usuario. A diferencia de la variable de instancia `saldo` en la clase `Cuenta`, la variable local `montoDeposito` en `main` *no* se inicializa con 0.0 de manera predeterminada. Sin embargo, esta variable no necesita inicializarse aquí, ya que su valor se determinará con base a la entrada del usuario.

La línea 23 pide al usuario que escriba un monto a depositar para `cuenta1`. La línea 24 obtiene la entrada del usuario, llamando al método **`nextDouble`** del objeto `Scanner` llamado `entrada`, el cual devuelve un valor `double` introducido por el usuario. Las líneas 25 y 26 muestran el monto del depósito.

La línea 27 llama al método abonar del objeto cuenta1 y le suministra montoDeposito como argumento. Cuando se hace la llamada al método, el valor del argumento se asigna al parámetro monto (línea 19 de la figura 3.13) del método abonar (líneas 19 a la 22 de la figura 3.13); después el método abonar suma ese valor al saldo (línea 21 de la figura 3.13). Las líneas 30 a la 33 (figura 3.14) imprimen en pantalla los saldos de ambos objetos Cuenta otra vez, para mostrar que sólo se modificó el saldo de cuenta1.

La línea 35 pide al usuario que escriba un monto a depositar para cuenta2. La línea 36 obtiene la entrada del usuario, para lo cual invoca al método nextDouble del objeto Scanner llamado entrada. Las líneas 37 y 38 muestran el monto del depósito. La línea 39 llama al método abonar del objeto cuenta2 y le suministra montoDeposito como argumento; después, el método abonar suma ese valor al saldo. Por último, las líneas 42 a la 45 imprimen en pantalla los saldos de ambos objetos Cuenta otra vez, para mostrar que sólo se modificó el saldo de cuenta2.

Diagrama de clases de UML para la clase Cuenta

El diagrama de clases de UML en la figura 3.15 modela la clase Cuenta de la figura 3.13. El diagrama modela el atributo private llamado saldo con el tipo Double de UML, para que corresponda a la variable de instancia saldo de la clase, que tiene el tipo double de Java. Modela el constructor de la clase Cuenta con un parámetro saldoInicial del tipo Double de UML en el tercer compartimiento de la clase. Los dos métodos public de la clase se modelan como operaciones en el tercer compartimiento también. El diagrama también modela la operación abonar con un parámetro monto de tipo Double de UML (ya que el método correspondiente tiene un parámetro monto de tipo double en Java) y la operación obtenerSaldo con un tipo de valor de retorno Double (ya que el método correspondiente en Java devuelve un valor double).

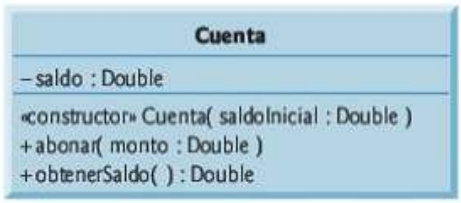


Fig. 3.15 | Diagrama de clases de UML, el cual indica que la clase Cuenta tiene un atributo private llamado saldo, con el tipo Double de UML, un constructor (con un parámetro de tipo Double de UML) y dos operaciones public: abonar (con un parámetro monto de tipo Double de UML) y obtenerSaldo (devuelve el tipo Double de UML).

3.8 (Opcional) Caso de estudio de GUI y gráficos: uso de cuadros de diálogo

Este caso de estudio opcional está diseñado para aquellos que desean empezar a conocer las poderosas herramientas de Java para crear interfaces gráficas de usuario (GUI) y gráficos antes de los principales debates de estos temas en el capítulo 14 (en el sitio Web del libro), Componentes de la GUI: Parte 1, el capítulo 15 (también en el sitio Web), Gráficos y Java 2D, y el capítulo 25, Componentes de la GUI: Parte 2 (en inglés, en el sitio Web).

El caso de estudio de GUI y gráficos aparece en 10 secciones breves (vea la figura 3.16). Cada sección introduce unos cuantos conceptos básicos y proporciona ejemplos con capturas de pantalla que muestran interacciones de ejemplo y resultados. En las primeras secciones, creará sus primeras aplicaciones gráficas. En las secciones posteriores, utilizará los conceptos de programación orientada a obje-

tos para crear una aplicación que dibuja una variedad de figuras. Cuando presentemos de manera formal las GUI en el capítulo 14, utilizaremos el ratón para elegir con exactitud qué figuras dibujar y en dónde. En el capítulo 15, agregaremos las herramientas de la API de gráficos en 2D de Java para dibujar las figuras con distintos grosores de línea y rellenos. Esperamos que este ejemplo práctico le sea informativo y divertido.

Ubicación	Título – Ejercicio(s)
Sección 3.8	Uso de cuadros de diálogo: entrada y salida básica con cuadros de diálogo
Sección 4.14	Creación de dibujos simples: mostrar y dibujar líneas en la pantalla
Sección 5.10	Dibujo de rectángulos y óvalos: uso de figuras para representar datos
Sección 6.13	Colores y figuras rellenas: dibujar un tiro al blanco y gráficos aleatorios
Sección 7.15	Dibujo de arcos: dibujar espirales con arcos
Sección 8.16	Uso de objetos con gráficos: almacenar figuras como objetos
Sección 9.8	Mostrar texto e imágenes usando etiquetas: proporcionar información de estado
Sección 10.8	Realizar dibujos usando polimorfismo: identificar las similitudes entre figuras
Ejercicio 14.17	Caso de estudio de GUI y gráficos: expansión de la interfaz
Ejercicio 15.31	Caso de estudio de GUI y gráficos: Agregar Java 2D

Fig. 3.16 | Resumen del caso de estudio de GUI y gráficos en cada capítulo.

Cómo mostrar texto en un cuadro de diálogo

Los programas que hemos presentado hasta ahora muestran su salida en la ventana de comandos. Muchas aplicaciones utilizan ventanas, o **cuadros de diálogo** (también llamados **diálogos**) para mostrar la salida. Por ejemplo, los navegadores Web como Firefox, Internet Explorer, Chrome y Safari muestran las páginas Web en sus propias ventanas. Los programas de correo electrónico le permiten escribir y leer mensajes en una ventana. Por lo general, los cuadros de diálogo son ventanas en las que los programas muestran mensajes importantes a los usuarios. La clase `JOptionPane` cuenta con cuadros de diálogo prefabricados, los cuales permiten a los programas mostrar ventanas que contengan mensajes; a dichas ventanas se les conoce como **diálogos de mensaje**. La figura 3.17 muestra el objeto String `"Bienvenido\na\nJava"` en un diálogo de mensaje.

```
1  // Fig. 3.17: Dialogo1.java
2  // Uso de JOptionPane para imprimir varias lineas en un cuadro de diálogo.
3  import javax.swing.JOptionPane; // importa la clase JOptionPane
4
5  public class Dialogo1
6  {
7     public static void main( String[] args )
8     {
9        // muestra un cuadro de diálogo con un mensaje
10       JOptionPane.showMessageDialog( null, "Bienvenido\na\nJava" );
11    } // fin de main
12 } // fin de la clase Dialogo1
```

Fig. 3.17 | Uso de `JOptionPane` para mostrar varias líneas en un cuadro de diálogo (parte 1 de 2).

Fig. 3.17 | Uso de `JOptionPane` para mostrar varias líneas en un cuadro de diálogo (parte 2 de 2).

La línea 3 indica que el programa utiliza la clase `JOptionPane` del paquete `javax.swing`. El cual contiene muchas clases que le ayudan a crear **interfaces gráficas de usuario (GUI)**. Los **componentes de la GUI** facilitan la entrada de datos al usuario del programa, y la presentación de los datos de salida. La línea 10 llama al método `showMessageDialog` de `JOptionPane` para mostrar un cuadro de diálogo que contiene un mensaje. El método requiere dos argumentos. El primero ayuda a Java a determinar en dónde colocar el cuadro de diálogo. Por lo general, un diálogo se muestra desde una aplicación de GUI con su propia ventana. El primer argumento hace referencia a esa ventana (conocida como ventana padre) y hace que el diálogo aparezca centrado sobre la ventana de la aplicación. Si el primer argumento es `null`, el cuadro de diálogo aparece en el centro de la pantalla de la computadora. El segundo argumento es el objeto `String` a mostrar en el cuadro de diálogo.

Introducción de los métodos `static`

El método `showMessageDialog` de la clase `JOptionPane` es lo que llamamos un **método `static`**. A menudo, dichos métodos definen las tareas que se utilizan con frecuencia. Por ejemplo, muchos programas muestran cuadros de diálogo, y el código para hacer esto es el mismo siempre. En vez de que usted tenga que "reinventar la rueda" y crear código para realizar esta tarea, los diseñadores de la clase `JOptionPane` declararon un método `static` que realiza esta tarea por usted. La llamada a un método `static` se realiza mediante el uso del nombre de su clase, seguido de un punto (`.`) y del nombre del método, como en

NombreClase.nombreMétodo(*argumentos*)

Observe que *no* tiene que crear un objeto de la clase `JOptionPane` para usar su método `static` llamado `showMessageDialog`. En el capítulo 6 analizaremos los métodos `static` con más detalle.

Introducir texto en un cuadro de diálogo

La aplicación de la figura 3.18 utiliza otro cuadro de diálogo `JOptionPane` predefinido, conocido como **diálogo de entrada**, el cual permite al usuario introducir datos en un programa. Éste pide el nombre del usuario, y responde con un diálogo de mensaje que contiene un saludo y el nombre introducido por el usuario.

Las líneas 10 y 11 utilizan el método `showInputDialog` de `JOptionPane` para mostrar un diálogo de entrada que contiene un indicador y un campo (conocido como **campo de texto**), en donde el usuario puede escribir texto. El argumento del método `showInputDialog` es el indicador que muestra lo que el usuario debe escribir. El usuario escribe caracteres en el campo de texto, y después hace clic en el botón **Aceptar** u oprime la tecla *Intro* para devolver el objeto `String` al programa. El método `showInputDialog` (línea 11) devuelve un objeto `String` que contiene los caracteres escritos por el usuario. Almacenamos el objeto `String` en la variable `nombre` (línea 10). [*Nota:* si oprime el botón **Cancelar** en el cuadro de diálogo u oprime *Esc*, el método devuelve `null` y el programa muestra la palabra clave "null" como el nombre].

Las líneas 14 y 15 utilizan el método `static String` llamado `format` para devolver un objeto `String` que contiene un saludo con el nombre del usuario. El método format es similar al método `System.out.printf`, excepto que `format` devuelve el objeto `String` con formato, en vez de mostrarlo en una ventana de comandos. La línea 18 muestra el saludo en un cuadro de diálogo de mensaje, como hicimos en la figura 3.17.

```
1  // Fig. 3.18: DialogoNombre.java
2  // Entrada básica con un cuadro de diálogo.
3  import javax.swing.JOptionPane;
4
5  public class DialogoNombre
6  {
7     public static void main( String[] args )
8     {
9        // pide al usuario que escriba su nombre
10       String nombre =
11          JOptionPane.showInputDialog( "Cual es su nombre?" );
12
13       // crea el mensaje
14       String mensaje =
15          String.format( "Bienvenido, %s, a la programacion en Java!", nombre );
16
17       // muestra el mensaje para dar la bienvenida al usuario por su nombre
18       JOptionPane.showMessageDialog( null, mensaje );
19    } // fin de main
20 } // fin de la clase DialogoNombre
```

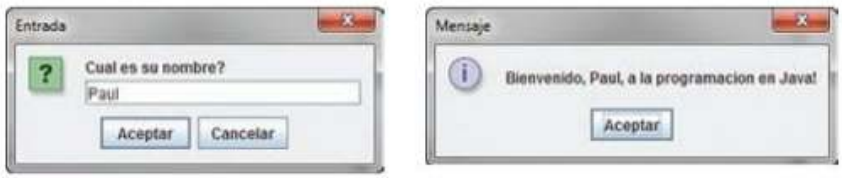


Fig. 3.18 | Cómo obtener la entrada del usuario mediante un cuadro de diálogo.

Ejercicio del ejemplo práctico de GUI y gráficos

3.1 Modifique el programa de suma en la figura 2.7 para usar la entrada y salida con base en el cuadro de diálogo con los métodos de la clase `JOptionPane`. Como el método `showInputDialog` devuelve un objeto `String`, debe convertir el objeto `String` que introduce el usuario a un `int` para usarlo en los cálculos. El método `static parseInt` de la clase `Integer` recibe un argumento `String` que representa un entero (es decir, el resultado de `JOptionPane.showInputDialog`) y devuelve el valor completo como un número `int`. El método `parseInt` es un método `static` de la clase `Integer` (del paquete `java.lang`). Si el objeto `String` no contiene un entero válido, el programa terminará con un error.

3.9 Conclusión

En este capítulo aprendió a declarar variables de instancia de una clase para mantener los datos de cada objeto, y cómo declarar métodos que operen sobre esos datos. Aprendió cómo llamar a un método para decirle que realice su tarea y cómo pasar información a los métodos en forma de argumentos. Vio la diferencia entre una variable local de un método y una variable de instancia de una clase, y que sólo las variables de instancia se inicializan en forma automática. También aprendió a utilizar el constructor de una clase para especificar los valores iniciales para las variables de instancia de un objeto. A lo largo del capítulo, vio cómo puede usarse UML para crear diagramas de clases que modelen los constructores, métodos y atributos de las clases. Por último, aprendió acerca de los números de punto flotante: cómo almacenarlos con variables del tipo primitivo `double`, cómo recibirlos en forma de datos de entrada

mediante un objeto `Scanner` y cómo darles formato con `printf` y el especificador de formato `%f` para fines de visualización. En el siguiente capítulo empezaremos nuestra introducción a las instrucciones de control, las cuales especifican el orden en el que se realizan las acciones de un programa. Utilizará estas instrucciones en sus métodos para especificar cómo deben realizar sus tareas.

Resumen

Sección 3.2 Declaración de una clase con un método e instanciamiento de un objeto de una clase

- Cada declaración de clase que empieza con el modificador de acceso `public` (pág. 72) debe almacenarse en un archivo que tenga exactamente el mismo nombre que la clase, y que termine con la extensión de nombre de archivo `.java`.
- Cada declaración de clase contiene la palabra clave `class`, seguida inmediatamente por el nombre de la clase.
- La declaración de un método que empieza con la palabra clave `public` indica que a ese método lo pueden llamar otras clases declaradas fuera de la declaración de esa clase.
- La palabra clave `void` indica que un método realizará una tarea, pero no devolverá información cuando la termine.
- Por convención, los nombres de los métodos empiezan con la primera letra en minúscula, y todas las palabras subsiguientes en el nombre empiezan con la primera letra en mayúscula.
- Los paréntesis vacíos después del nombre de un método indican que éste no requiere parámetros para realizar su tarea.
- El cuerpo de todos los métodos está delimitado por llaves izquierda y derecha (`{` y `}`).
- El cuerpo de un método contiene instrucciones que realizan la tarea de éste. Una vez que se ejecutan las instrucciones, el método ha terminado su tarea.
- Cuando intentamos ejecutar una clase, Java busca el método `main` de la clase para empezar la ejecución.
- Por lo general, no podemos llamar a un método que pertenece a otra clase, sino hasta crear un objeto de esa clase.
- Una expresión de creación de instancia de clase (pág. 74) empieza con la palabra clave `new` y crea un nuevo objeto.
- Para llamar a un método de un objeto, se pone después del nombre de la variable un separador punto (`.`; pág. 75), el nombre del método y un conjunto de paréntesis que contienen los argumentos del método.
- En UML, cada clase se modela en un diagrama de clases en forma de rectángulo con tres compartimientos. El compartimiento superior contiene el nombre de la clase, centrado horizontalmente y en negrita. El compartimiento intermedio contiene los atributos de la clase, que corresponden a los campos en Java. El compartimiento inferior contiene las operaciones de la clase (pág. 76), que corresponden a los métodos y constructores en Java.
- Para modelar las operaciones, UML enumera el nombre de la operación, seguido de un conjunto de paréntesis. Un signo más (+) enfrente del nombre de la operación indica que ésta es una operación `public` en UML (es decir, un método `public` en Java).

Sección 3.3 Declaración de un método con un parámetro

- A menudo, los métodos requieren parámetros (pág. 76) para realizar sus tareas. Dicha información adicional se proporciona mediante argumentos en las llamadas a los métodos.
- El método `nextLine` de `Scanner` (pág. 76) lee caracteres hasta encontrar una nueva línea y después devuelve los caracteres que leyó en forma de un objeto `String`.
- El método `next` de `Scanner` (pág. 77) lee caracteres hasta encontrar cualquier carácter de espacio en blanco, y después devuelve los caracteres que leyó en forma de un objeto `String`.
- Un método que requiere datos para realizar su tarea debe especificar esto en su declaración, para lo cual coloca información adicional en la lista de parámetros del método (pág. 76).
- Cada parámetro debe especificar tanto un tipo como un nombre de variable.

- Cuando se hace la llamada a un método, sus argumentos se asignan a sus parámetros. Entonces, el cuerpo del método utiliza las variables de los parámetros para acceder a los valores de los argumentos.
- Un método especifica varios parámetros en una lista separada por comas.
- El número de argumentos en la llamada a un método debe coincidir con el de los parámetros en la lista de parámetros de la declaración del método. Además, los tipos de los argumentos en la llamada al método deben ser consistentes con los de los parámetros correspondientes en la declaración de éste.
- La clase `String` está en el paquete `java.lang`, que por lo general se importa de manera implícita en todos los archivos de código fuente.
- De manera predeterminada, las clases que se compilan en el mismo directorio están en el mismo paquete. Las clases en el mismo paquete se importan implícitamente en los archivos de código fuente de las otras clases que están en el mismo paquete.
- Las declaraciones `import` no son obligatorias si usamos siempre nombres de clases completamente calificados (pág. 79).
- Para modelar un parámetro de una operación, UML lista el nombre del parámetro, seguido de dos puntos y el tipo del parámetro entre los paréntesis que van después del nombre de la operación.
- UML tiene sus propios tipos de datos, similares a los de Java. No todos los tipos de datos de UML tienen los mismos nombres que los tipos correspondientes en Java.
- El tipo `String` de UML corresponde al tipo `String` de Java.

Sección 3.4 Variables de instancia, métodos establecer *y métodos* obtener

- Las variables que se declaran en el cuerpo de un método son variables locales, y pueden utilizarse sólo en ese método.
- Por lo general, una clase consiste en uno o más métodos que manipulan los atributos (datos) pertenecientes a un objeto específico de esa clase. Dichas variables se llaman campos y se declaran dentro de la declaración de una clase, pero fuera de los cuerpos de las declaraciones de los métodos de esa clase.
- Cuando cada objeto de una clase mantiene su propia copia de un atributo, al campo correspondiente se le conoce como variable de instancia.
- Las variables o métodos declarados con el modificador de acceso `private` sólo están accesibles para los métodos de la clase en la que están declarados.
- Al proceso de declarar variables de instancia con el modificador de acceso `private` (pág. 80) se le conoce como ocultamiento de datos.
- Un beneficio de los campos es que todos los métodos de la clase pueden usarlos. Otra diferencia entre un campo y una variable local es que un campo tiene un valor inicial predeterminado (pág. 82), que Java proporciona cuando el programador no especifica el valor inicial del campo, pero una variable local no hace esto.
- El valor predeterminado para un campo de tipo `String` (o cualquier otro tipo por referencia) es `null`.
- Cuando se llama a un método que especifica un tipo de valor de retorno (pág. 73) y completa su tarea, devuelve un resultado al método que lo llamó (pág. 73).
- A menudo, las clases proporcionan métodos `public` para permitir que los clientes de la clase *establezcan* u *obtengan* variables de instancia `private` (pág. 83). Los nombres de estos métodos no necesitan comenzar con *establecer* u *obtener*, pero esta convención de nomenclatura es muy recomendada en Java, y requerida para ciertos componentes de software de Java especiales, conocidos como JavaBeans.
- UML representa a las variables de instancia como un nombre de atributo, seguido de dos puntos y el tipo del atributo.
- En UML, los atributos privados van precedidos por un signo menos (-).
- Para indicar el tipo de valor de retorno de una operación, UML coloca dos puntos y el tipo de valor de retorno después de los paréntesis que siguen del nombre de la operación.
- Los diagramas de clases de UML (pág. 75) no especifican tipos de valores de retorno para las operaciones que no devuelven valores.

Sección 3.5 Comparación entre tipos primitivos y tipos por referencia

- En Java, los tipos se dividen en dos categorías: tipos primitivos y tipos por referencia. Los tipos primitivos son `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` y `double`. Todos los demás tipos son por referencia, por lo cual, las clases que especifican los tipos de los objetos, son tipos por referencia.

- Una variable de tipo primitivo puede almacenar exactamente un valor de su tipo declarado, en un momento dado.
- Las variables de instancia de tipos primitivos se inicializan de manera predeterminada. Las variables de los tipos `byte`, `char`, `short`, `int`, `long`, `float` y `double` se inicializan con `0`. Las variables de tipo `boolean` se inicializan con `false`.
- Las variables de tipos por referencia (llamadas referencias; pág. 84) almacenan la ubicación de un objeto en la memoria de la computadora. Dichas variables hacen referencia a los objetos en el programa. El objeto al que se hace referencia puede contener muchas variables de instancia y métodos.
- Los campos de tipo por referencia se inicializan de manera predeterminada con el valor `null`.
- Para invocar a los métodos de instancia de un objeto, se requiere una referencia a éste (pág. 84). Una variable de tipo primitivo no hace referencia a un objeto, por lo cual no puede usarse para invocar a un método.

Sección 3.6 Inicialización de objetos mediante constructores

- La palabra clave `new` solicita memoria del sistema para almacenar un objeto, y después llama al constructor de la clase correspondiente (pág. 74) para inicializar el objeto.
- Un constructor puede usarse para inicializar un objeto de una clase, a la hora de crearlo.
- Los constructores pueden especificar parámetros, pero no tipos de valores de retorno.
- Si una clase no define constructores, el compilador proporciona uno predeterminado (pág. 85) sin parámetros, y las variables de instancia de la clase se inicializan con sus valores predeterminados.
- UML modela a los constructores en el tercer compartimiento de un diagrama de clases. Para diferenciar a un constructor con base en las operaciones de una clase, UML coloca la palabra "constructor" entre los signos « y » (pág. 88) antes del nombre de éste.

Sección 3.7 Los números de punto flotante y el tipo `double`

- Un número de punto flotante (pág. 88) es un número con un punto decimal. Java proporciona dos tipos primitivos para almacenar números de punto flotante (pág. 88) en la memoria: `float` y `double`. La principal diferencia entre estos tipos es que las variables `double` pueden almacenar números con mayor magnitud y detalle (a esto se le conoce como la precisión del número; pág. 88) que las variables `float`.
- Las variables de tipo `float` representan números de punto flotante de precisión simple, y tienen siete dígitos significativos. Las variables de tipo `double` representan números de punto flotante de precisión doble. Éstos requieren el doble de memoria que las variables `float` y proporcionan 15 dígitos significativos; tienen aproximadamente el doble de precisión de las variables `float`.
- Las literales de punto flotante (pág. 88) son de tipo `double` de manera predeterminada.
- El método `nextDouble` de `Scanner` (pág. 91) devuelve un valor `double`.
- El especificador de formato `%f` (pág. 90) se utiliza para mostrar valores de tipo `float` o `double`. El especificador de formato `%.2f` especifica que se deben mostrar dos dígitos de precisión (pág. 90) a la derecha del punto decimal, en el número de punto flotante.
- El valor predeterminado para un campo de tipo `double` es `0.0`, y el valor predeterminado para un campo de tipo `int` es `0`.

Ejercicios de autoevaluación

3.2 Complete las siguientes oraciones:

a) Cada declaración de clase que empieza con la palabra clave _____ debe almacenarse en un archivo que tenga exactamente el mismo nombre de la clase, y que termine con la extensión de nombre de archivo `.java`.
b) En la declaración de una clase, la palabra clave _____ va seguida inmediatamente por el nombre de la clase.
c) La palabra clave _____ solicita memoria del sistema para almacenar un objeto, y después llama al constructor de la clase correspondiente para inicializarlo.
d) Cada parámetro debe especificar un(a) _____ y un(a) _____ .
e) De manera predeterminada, se considera que las clases que se compilan en el mismo directorio están en el mismo paquete, conocido como _____ .

f) Cuando cada objeto de una clase mantiene su propia copia de un atributo, el campo que representa a este atributo se conoce también como _____ .
g) Java proporciona dos tipos primitivos para almacenar números de punto flotante en la memoria: _____ y _____ .
h) Las variables de tipo `double` representan a los números de punto flotante _____ .
i) El método _____ de la clase `Scanner` devuelve un valor `double`.
j) La palabra clave `public` es un _____ de acceso.
k) El tipo de valor de retorno _____ indica que un método no devolverá un valor.
l) El método _____ de `Scanner` lee caracteres hasta encontrar una nueva línea y después devuelve esos caracteres como un objeto `String`.
m) La clase `String` está en el paquete _____ .
n) No se requiere un(a) _____ si siempre hacemos referencia a una clase con su nombre completamente calificado.
o) Un(a) _____ es un número con un punto decimal, como 7.33, 0.0975 o 1000.12345.
p) Las variables de tipo `float` representan números de punto flotante _____ .
q) El especificador de formato _____ se utiliza para mostrar valores de tipo `float` o `double`.
r) Los tipos en Java se dividen en dos categorías: tipos _____ y tipos _____ .

3.3 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Por convención, los nombres de los métodos empiezan con la primera letra en mayúscula, en el nombre todas las palabras subsiguientes comienzan con la primera letra en mayúscula.
b) Una declaración `import` no es obligatoria cuando una clase en un paquete utiliza a otra en el mismo paquete.
c) Los paréntesis vacíos que van después del nombre de un método en su declaración indican que no requiere parámetros para realizar su tarea.
d) Las variables o los métodos declarados con el modificador de acceso `private` son accesibles sólo para los métodos de la clase en la que se declaran.
e) Una variable de tipo primitivo puede usarse para invocar un método.
f) Las variables que se declaran en el cuerpo de un método específico se conocen como variables de instancia, y pueden utilizarse en todos los métodos de la clase.
g) El cuerpo de cada método está delimitado por llaves izquierda y derecha (`{` y `}`).
h) Las variables locales de tipo primitivo se inicializan de manera predeterminada.
i) Las variables de instancia de tipo por referencia se inicializan de manera predeterminada con el valor `null`.
j) Cualquier clase que contenga `public static void main( String[] args )` puede usarse para ejecutar una aplicación.
k) El número de argumentos en la llamada a un método debe coincidir con el de parámetros en la lista de parámetros de la declaración del método.
l) Los valores de punto flotante que aparecen en código fuente se conocen como literales de punto flotante, y son de tipo `float` de manera predeterminada.

3.4 ¿Cuál es la diferencia entre una variable local y un campo?

3.5 Explique el propósito de un parámetro de un método. ¿Cuál es la diferencia entre un parámetro y un argumento?

Respuestas a los ejercicios de autoevaluación

3.1 a) `public`. b) `class`. c) `new`. d) tipo, nombre. e) paquete predeterminado. f) variable de instancia. g) `float`, `double`. h) de precisión doble. i) `nextDouble`. j) modificador. k) `void`. l) `nextLine`. m) `java.lang`. n) declaracion `import`. o) número de punto flotante. p) de precisión simple. q) `%f`. r) primitivo, por referencia.

3.2 a) Falso. Por convención, los nombres de los métodos empiezan con una primera letra en minúscula y todas las palabras subsiguientes con una letra en mayúscula. b) Verdadero. c) Verdadero. d) Verdadero. e) Falso. Una variable de tipo

primitivo no puede usarse para invocar a un método; se requiere una referencia a un objeto para invocar a sus métodos. f) Falso. Dichas variables se llaman variables locales, y sólo se pueden utilizar en el método en el que están declaradas. g) Verdadero. h) Falso. Las variables de instancia de tipo primitivo se inicializan de manera predeterminada. A cada variable local se le debe asignar un valor de manera explícita. i) Verdadero. j) Verdadero. k) Verdadero. l) Falso. Dichas literales son de tipo double de manera predeterminada.

3.3 Una variable local se declara en el cuerpo de un método, y sólo puede utilizarse desde el punto en el que se declaró, hasta el final de la declaración del método. Un campo se declara en una clase, pero no en el cuerpo de alguno de los métodos de ella. Además, los campos están accesibles para todos los métodos de la clase. (En el capítulo 8, Clases y objetos: un análisis más detallado, veremos una excepción a esto).

3.4 Un parámetro representa la información adicional que requiere un método para realizar su tarea. Cada parámetro requerido por un método está especificado en la declaración del método. Un argumento es el valor actual para un parámetro del método. Cuando se llama a un método, los valores de los argumentos se pasan a sus parámetros correspondientes para que éste pueda realizar su tarea.

Ejercicios

3.5 ***(Palabra clave new)*** ¿Cuál es el propósito de la palabra clave `new`? Explique lo que ocurre cuando se utiliza en una aplicación.

3.6 ***(Constructores predeterminados)*** ¿Qué es un constructor predeterminado? ¿Cómo se inicializan las variables de instancia de un objeto, si una clase sólo tiene un constructor predeterminado?

3.7 ***(Variables de instancia)*** Explique el propósito de una variable de instancia.

3.8 ***(Usar clases sin importarlas)*** La mayoría de las clases necesitan importarse antes de poder utilizarlas en una aplicación ¿Por qué cualquier aplicación puede utilizar las clases `System` y `String` sin tener que importarlas primero?

3.9 ***(Usar una clase sin importarla)*** Explique cómo podría un programa utilizar la clase `Scanner` sin importarla.

3.10 ***(Métodos establecer y obtener)*** Explique por qué una clase podría proporcionar un método establecer y un método obtener para una variable de instancia.

3.11 ***(Clase `LibroCalificaciones` modificada)*** Modifique la clase `LibroCalificaciones` (figura 3.10) de la siguiente manera:

a) Incluya una segunda variable de instancia `String`, que represente el nombre del instructor del curso.
b) Proporcione un método *establecer* para modificar el nombre del instructor, y un método *obtener* para conseguir el nombre.
c) Modifique el constructor para especificar dos parámetros: uno para el nombre del curso y otro para el del instructor.
d) Modifique el método `mostrarMensaje`, de tal forma que primero imprima el mensaje de bienvenida y el nombre del curso, seguidos de `"Este curso es presentado por: "` y el nombre del instructor.

Use su clase modificada en una aplicación de prueba que demuestre las nuevas capacidades que tiene.

3.12 ***(Clase `Cuenta` modificada)*** Modifique la clase `Cuenta` (figura 3.13) para proporcionar un método llamado `cargar`, que retire dinero de un objeto `Cuenta`. Asegure que el monto a cargar no exceda el saldo de `Cuenta`. Si lo hace, el saldo debe permanecer sin cambio y el método debe imprimir un mensaje que indique `"El monto a cargar excede el saldo de la cuenta."` Modifique la clase `PruebaCuenta` (figura 3.14) para probar el método `cargar`.

3.13 ***(La clase `Factura`)*** Cree una clase llamada `Factura`, que una ferretería podría utilizar para representar una factura para un artículo vendido en la tienda. Una `Factura` debe incluir cuatro piezas de información como variables de instancia: un número de pieza (tipo `String`), la descripción de la pieza (tipo `String`), la cantidad de artículos de ese tipo que se van a comprar (tipo `int`) y el precio por artículo (`double`). Su clase debe tener un constructor que inicialice las cuatro variables de instancia. Proporcione un método *establecer* y uno *obtener* para cada variable de instancia. Además, proporcione un método llamado `obtenerMontoFactura`, que calcule el monto de la factura (es decir, que multiplique la cantidad por el precio por artículo) y después lo devuelva como un valor `double`. Si la cantidad no es positiva, debe estable-

cerse en 0. Si el precio por artículo no es positivo, debe establecerse en 0.0. Escriba una aplicación de prueba llamada PruebaFactura, que demuestre las capacidades de la clase Factura.

3.14 ***(La clase Empleado)*** Cree una clase llamada Empleado, que incluya tres variables de instancia: un primer nombre (tipo String), un apellido paterno (tipo String) y un salario mensual (double). Su clase debe tener un constructor que inicialice las tres variables de instancia. Proporcione un método *establecer* y un método *obtener* para cada variable de instancia. Si el salario mensual no es positivo, no establezca su valor. Escriba una aplicación de prueba llamada PruebaEmpleado, que demuestre las capacidades de la clase Empleado. Cree dos objetos Empleado y muestre el salario anual de cada objeto. Después, proporcione a cada Empleado un aumento del 10% y muestre el salario anual de cada Empleado otra vez.

3.15 ***(La clase Fecha)*** Cree una clase llamada Fecha, que incluya tres variables de instancia: un mes (tipo int), un día (tipo int) y un año (tipo int). Su clase debe tener un constructor que inicialice las tres variables de instancia, y debe asumir que los valores que se proporcionan son correctos. Proporcione un método *establecer* y un método *obtener* para cada variable de instancia. Proporcione un método mostrarFecha, que muestre el mes, día y año, separados por barras diagonales (/). Escriba una aplicación de prueba llamada PruebaFecha, que demuestre las capacidades de la clase Fecha.

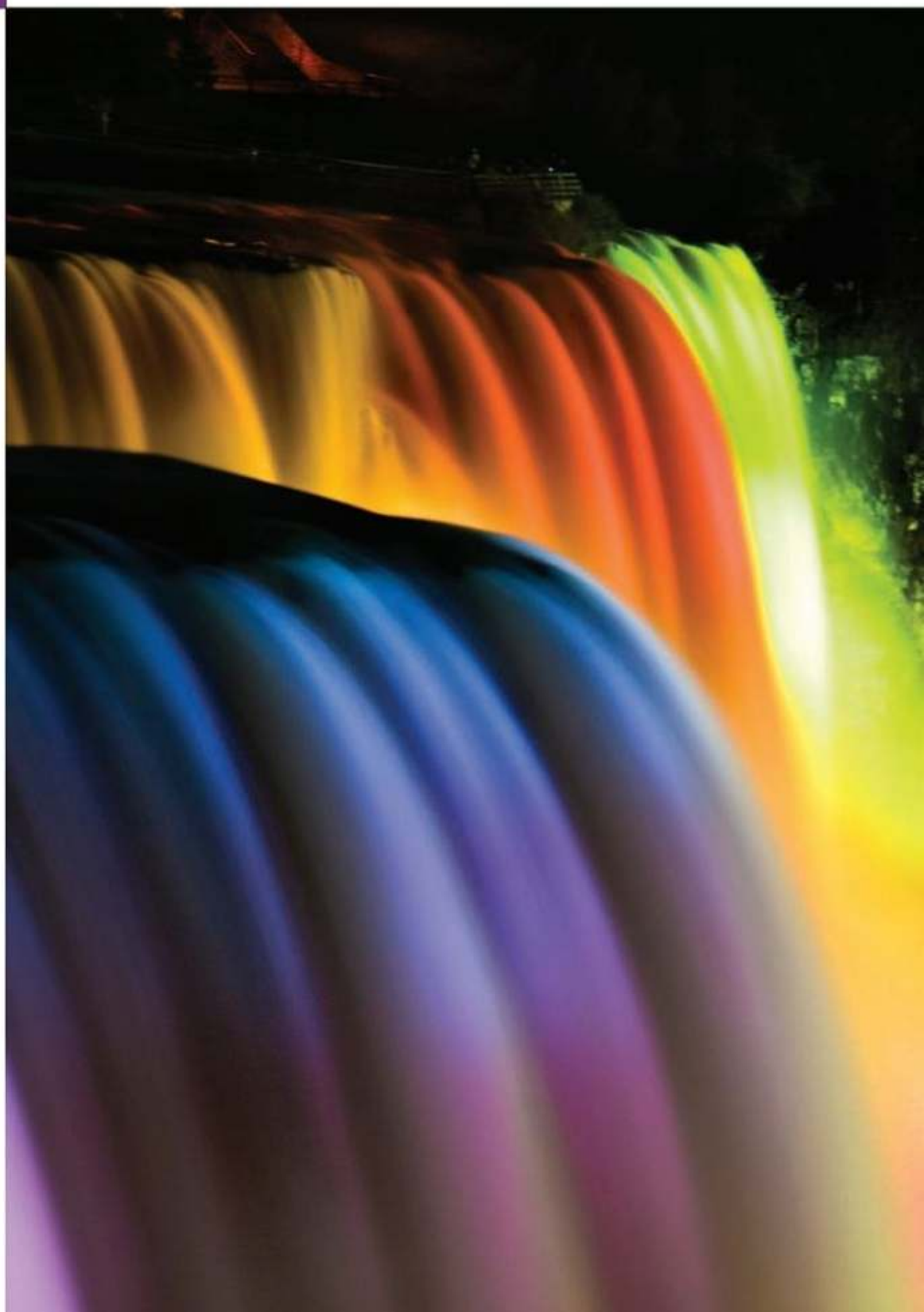
Marcar la diferencia

3.16 ***(Calculadora de la frecuencia cardiaca esperada)*** Mientras se ejercita, puede usar un monitor de frecuencia cardiaca para ver que su corazón permanezca dentro de un rango seguro sugerido por sus entrenadores y doctores. De acuerdo con la Asociación Estadounidense del Corazón (AHA) (www.americanheart.org/), la fórmula para calcular su frecuencia cardiaca máxima en pulsos por minuto es de 220 menos su edad en años. Su frecuencia cardiaca esperada es un rango que está entre el 50 y el 85% de su frecuencia cardiaca máxima. [*Nota:* estas fórmulas son estimaciones proporcionadas por la AHA. Las frecuencias cardiacas máxima y esperada pueden variar de acuerdo con la salud, condición física y sexo del individuo. Siempre debe consultar un médico o a un profesional de la salud antes de empezar o modificar un programa de ejercicios.] Cree una clase llamada FrecuenciasCardiacas. Los atributos de la clase deben incluir el primer nombre de la persona, su apellido y fecha de nacimiento (la cual debe consistir de atributos separados para el mes, día y año de nacimiento). Su clase debe tener un constructor que reciba estos datos como parámetros. Para cada atributo debe proveer métodos *establecer* y *obtener*. La clase también debe incluir un método que calcule y devuelva la edad de la persona (en años), uno que calcule y devuelva la frecuencia cardiaca máxima de esa persona, y otro que calcule y devuelva la frecuencia cardiaca esperada de la persona. Escriba una aplicación de Java que pida la información de la persona, cree una instancia de un objeto de la clase FrecuenciasCardiacas e imprima la información a partir de ese objeto (incluya el primer nombre de la persona, su apellido y fecha de nacimiento), y que después calcule e imprima la edad de la persona en (años), frecuencia cardiaca máxima y rango de frecuencia cardiaca esperada.

3.17 ***(Computarización de los registros médicos)*** Un problema relacionado con la salud que ha estado últimamente en las noticias es la computarización de los registros médicos. Esta posibilidad se está tratando con mucho cuidado, debido a las delicadas cuestiones de privacidad y seguridad, entre otras cosas. [Trataremos esas cuestiones en ejercicios posteriores.] La computarización de los registros médicos puede facilitar a los pacientes el proceso de compartir sus perfiles e historiales médicos con los diversos profesionales de la salud que consulten. Esto podría mejorar la calidad del servicio médico, ayudar a evitar conflictos de fármacos y prescripciones erróneas, reducir los costos y, en emergencias, ayudar a salvar vidas. En este ejercicio usted diseñará una clase inicial llamada PerfilMedico para una persona. Los atributos de la clase deben llevar el primer nombre de la persona, su apellido, sexo, fecha de nacimiento (que debe consistir de atributos separados para el día, mes y año de nacimiento), altura (en centímetros) y peso (en kilogramos). Su clase debe tener un constructor que reciba estos datos. Para cada atributo, debe proveer los métodos *establecer* y *obtener*. La clase también debe tener métodos que calculen y devuelvan la edad del usuario en años, la frecuencia cardiaca máxima y el rango de frecuencia cardiaca esperada (vea el ejercicio 3.16), además del índice de masa corporal (BMI; vea el ejercicio 2.33). Escriba una aplicación de Java que pida la información de la persona, cree una instancia de un objeto de la clase PerfilMedico para esa persona e imprima la información de ese objeto (debe contener el primer nombre de la persona, apellido, sexo, fecha de nacimiento, altura y peso), y que después calcule e imprima la edad de esa persona en años, junto con el BMI, la frecuencia cardiaca máxima y el rango de frecuencia cardiaca esperada. También debe mostrar la tabla de valores del BMI del ejercicio 2.33.

4

Instrucciones de control: Parte 1



Desplacémonos un lugar.
—Lewis Carroll

La rueda se convirtió en un círculo completo.
—William Shakespeare

¡Cuántas manzanas tuvieron que caer en la cabeza de Newton antes de que entendiera el suceso!
—Robert Frost

Objetivos

En este capítulo aprenderá a:

- Comprender las técnicas básicas para solucionar problemas.
- Desarrollar algoritmos mediante el proceso de refinamiento de arriba a abajo, paso a paso.
- Utilizar las estructuras de selección `if` e `if...else` para elegir entre distintas acciones alternativas.
- Usar la estructura de repetición `while` para ejecutar instrucciones de manera repetitiva dentro de un programa.
- Emplear la repetición controlada por un contador y la repetición controlada por un centinela.
- Manejar los operadores de asignación compuestos, de incremento y decremento.
- Conocer la portabilidad de los tipos de datos primitivos.

Plan general

4.1 Introducción

Antes de escribir un programa que dé solución a un problema, es imprescindible tener una comprensión detallada de todo el problema, además de una metodología cuidadosamente planeada para resolverlo. Al escribir un programa, es también esencial comprender los tipos de bloques de construcción disponibles, y emplear las técnicas comprobadas para construir programas. En este capítulo y en el 5, Instrucciones de control: Parte 2, hablaremos sobre estas cuestiones cuando presentemos la teoría y los principios de la programación estructurada. Los conceptos aquí presentados son imprescindibles para crear clases y manipular objetos.

En este capítulo le presentamos las instrucciones `if`, `if...else` y `while` de Java, tres de los bloques de construcción que le permiten especificar la lógica requerida para que los métodos realicen sus tareas. Dedicamos una parte de este capítulo (y de los capítulos 5 y 7) para desarrollar más la clase `LibroCalificaciones` que presentamos en el capítulo 3. En especial, agregamos un método a la clase `LibroCalificaciones` que utiliza instrucciones de control para calcular el promedio de un conjunto de calificaciones de estudiantes. Otro ejemplo demuestra formas adicionales de combinar instrucciones de control para resolver un problema similar. Presentamos los operadores de asignación compuestos de incremento y decremento de Java. Por último, analizamos la portabilidad de los tipos de datos primitivos de Java.

4.2 Algoritmos

Cualquier problema de computación puede resolverse ejecutando una serie de acciones en un orden específico. Un procedimiento para resolver un problema en términos de

1. las **acciones** a ejecutar y
2. el **orden** en el que se ejecutan estas acciones

se conoce como un **algoritmo**. El siguiente ejemplo demuestra que es importante especificar de manera correcta el orden en el que se ejecutan las acciones.

Considere el "algoritmo para levantarse y arreglarse" que sigue un ejecutivo para levantarse de la cama e ir a trabajar: (1) levantarse; (2) quitarse la pijama; (3) bañarse; (4) vestirse; (5) desayunar; (6) transportarse al trabajo. Esta rutina hace que el ejecutivo llegue al trabajo bien preparado para tomar decisiones importantes. Suponga que los mismos pasos se realizan en un orden ligeramente distinto: (1) levantarse; (2) quitarse la pijama; (3) vestirse; (4) bañarse; (5) desayunar; (6) transportarse al trabajo. En este caso, nuestro ejecutivo llegará al trabajo todo mojado. Al proceso de especificar el orden en el que se ejecutan las instrucciones (acciones) en un programa, se le llama **control del pro-**

grama. En este capítulo investigaremos el control de los programas mediante el uso de las **instrucciones de control** de Java.

4.3 Seudocódigo

El **seudocódigo** es un lenguaje informal que le ayuda a desarrollar algoritmos sin tener que preocuparse por los estrictos detalles de la sintaxis del lenguaje Java. El seudocódigo que presentaremos es especialmente útil para desarrollar algoritmos que se convertirán en porciones estructuradas de programas en Java. El seudocódigo es similar al lenguaje cotidiano: es conveniente y amigable con el usuario, aunque en realidad no es un lenguaje de programación de computadoras. En la figura 4.5 verá un algoritmo escrito en seudocódigo.

El seudocódigo no se ejecuta en las computadoras. En vez de ello, le ayuda a "organizar" un programa antes de que intente escribirlo en un lenguaje de programación como Java. Este capítulo presenta varios ejemplos de cómo utilizar el seudocódigo para desarrollar programas en Java.

El estilo de seudocódigo que presentaremos consiste sólo en caracteres, para que usted pueda escribirlo de una manera conveniente, con cualquier programa editor de texto. Un programa en seudocódigo preparado de manera cuidadosa puede convertirse fácilmente en su correspondiente programa en Java.

Por lo general, el seudocódigo describe sólo las instrucciones que representan las acciones que ocurren después de convertir un programa de seudocódigo a Java, y el programa se ejecuta en una computadora. Dichas acciones podrían incluir la entrada, salida o un cálculo. Por lo general no incluimos las declaraciones de variables en nuestro seudocódigo, pero algunos programadores optan por listarlas y mencionar sus propósitos al principio de su seudocódigo.

4.4 Estructuras de control

Es común en un programa que las instrucciones se ejecuten una después de otra, en el orden en que están escritas. Este proceso se conoce como **ejecución secuencial**. Varias instrucciones en Java, que pronto veremos, permiten al programador especificar que la siguiente instrucción a ejecutarse tal vez no sea la *siguiente* en la secuencia. Esto se conoce como **transferencia de control**.

Durante la década de 1960, se hizo evidente que el uso indiscriminado de las transferencias de control era el origen de muchas de las dificultades que experimentaban los grupos de desarrollo de software. A quien se señaló como culpable fue a la **instrucción goto** (utilizada en la mayoría de los lenguajes de programación de esa época), la cual permite al programador especificar la transferencia de control a uno de los muchos posibles destinos dentro de un programa. La noción de lo que conocemos como **programación estructurada** se hizo casi un sinónimo de la "eliminación del goto". [*Nota:* Java *no* tiene una instrucción `goto`; sin embargo, la palabra `goto` está *reservada* para Java y *no* debe usarse como identificador en los programas].

Las investigaciones de Bohm y Jacopini[1] demostraron que los programas podían escribirse *sin* instrucciones `goto`. El reto de la época para los programadores fue cambiar sus estilos a una "programación sin `goto`". No fue sino hasta la década de 1970 cuando los programadores tomaron en serio la programación estructurada. Los resultados fueron impresionantes. Los grupos de desarrollo de software reportaron reducciones en los tiempos de desarrollo, mayor incidencia de entregas de sistemas a tiempo y más proyectos de software finalizados sin salirse del presupuesto. La clave para estos logros fue que los programas estructurados eran más claros, más fáciles de depurar y modificar, y había más probabilidad de que estuvieran libres de errores desde el principio.

1 Bohm, C. Y G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM*, vol. 9, núm. 5, mayo de 1966, páginas 336-371.

El trabajo de Bohm y Jacopini demostró que todos los programas podían escribirse en términos de tres estructuras de control solamente: la **estructura de secuencia**, la **estructura de selección** y la **estructura de repetición**. Cuando presentemos las implementaciones de las estructuras de control en Java, nos referiremos a ellas en la terminología de la *Especificación del lenguaje Java* como "instrucciones de control".

Estructura de secuencia en Java

La estructura de secuencia está integrada en Java. A menos que se le indique lo contrario, la computadora ejecuta las instrucciones en Java una después de otra, en el orden en que estén escritas; es decir, en secuencia. El **diagrama de actividad** de la figura 4.1 ilustra una estructura de secuencia típica, en la que se realizan dos cálculos en orden. Java permite tantas acciones como deseemos en una estructura de secuencia. Como veremos pronto, en donde quiera que se coloque una sola acción, podrán colocarse varias en secuencia.

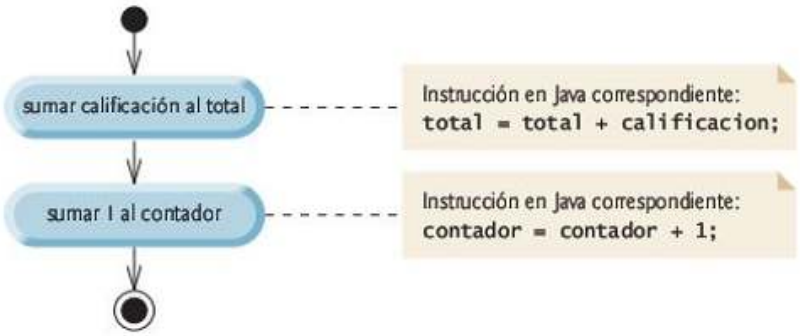


Fig. 4.1 | Diagrama de actividad de una estructura de secuencia.

Un diagrama de actividad de UML modela el **flujo de trabajo** (también conocido como la **actividad**) de una parte de un sistema de software. Dichos flujos de trabajo pueden incluir una porción de un algoritmo, como la estructura de secuencia de la figura 4.1. Los diagramas de actividad están compuestos por símbolos de propósito especial, como los **símbolos de estado de acción** (rectángulos cuyos lados izquierdo y derecho se reemplazan con arcos hacia fuera), **rombos** y **círculos pequeños**. Estos símbolos se conectan mediante **flechas de transición**, que representan el flujo de la actividad; es decir, el orden en el que deben ocurrir las acciones.

Al igual que el seudocódigo, los diagramas de actividad ayudan a los programadores a desarrollar y representar algoritmos, aunque muchos de ellos aún prefieren el seudocódigo. Los diagramas de actividad muestran con claridad cómo operan las estructuras de control. Usaremos el UML en este capítulo y en el 5 para mostrar el flujo de control en las instrucciones de control. En los capítulos 12 y 13 utilizaremos el UML en un caso de estudio de un cajero automático real.

Considere el diagrama de actividad para la estructura de secuencia de la figura 4.1. Contiene dos **estados de acción** que representan las acciones a realizar. Cada estado de acción contiene una **expresión de acción** (por ejemplo, "sumar calificación a total" o "sumar 1 al contador"), que especifica una acción particular a realizar. Otras acciones podrían incluir cálculos u operaciones de entrada/salida. Las flechas en el diagrama de actividad representan **transiciones**, las cuales indican el *orden* en el que ocurren las acciones representadas por los estados de acción. El programa que implementa las actividades ilustradas por el diagrama de la figura 4.1 primero suma `calificacion` a `total`, y después suma `1` a `contador`.

El **círculo relleno** que se encuentra en la parte superior del diagrama de actividad representa el **estado inicial**: el *inicio* del flujo de trabajo *antes* de que el programa realice las actividades modeladas.

El **círculo sólido rodeado por una circunferencia** que aparece en la parte inferior del diagrama representa el **estado final**; es decir, el *final* del flujo de trabajo *después* de que el programa realiza sus acciones.

La figura 4.1 también incluye rectángulos que tienen la esquina superior derecha doblada. En UML, a estos rectángulos se les llama **notas** (como los comentarios en Java): comentarios con explicaciones que describen el propósito de los símbolos en el diagrama. La figura 4.1 utiliza las notas de UML para mostrar el código en Java asociado con cada estado de acción. Una **línea punteada** conecta cada nota con el elemento que ésta describe. Los diagramas de actividad generalmente *no* muestran el código en Java que implementa la actividad. En este libro utilizamos las notas para mostrar cómo se relaciona el diagrama con el código en Java. Para obtener más información sobre UML, vea nuestro caso de estudio opcional (capítulos 12 y 13) o visite `www.uml.org`.

Instrucciones de selección en Java

Java tiene tres tipos de **instrucciones de selección** (las cuales se describen en este capítulo y en el 5). La instrucción `if` realiza (selecciona) una acción si la condición es verdadera, o evita la acción si la condición es falsa. La instrucción `if...else` realiza una acción si la condición es verdadera, o realiza una acción distinta si es falsa. La instrucción `switch` (capítulo 5) realiza una de entre varias acciones distintas, dependiendo del valor de una expresión.

La instrucción `if` es una **instrucción de selección simple**, ya que selecciona o ignora una *sola* acción (o, como pronto veremos, un *solo grupo de acciones*). La instrucción `if...else` se conoce como **instrucción de selección doble**, ya que selecciona entre *dos acciones distintas* (o *grupos de acciones*). La instrucción `switch` es una estructura de selección múltiple, ya que selecciona entre *diversas acciones* (o *grupos de acciones*).

Instrucciones de repetición en Java

Java cuenta con tres **instrucciones de repetición** (también llamadas **instrucciones de ciclo**) que permiten a los programas ejecutar instrucciones en forma repetida, siempre y cuando una condición (llamada la **condición de continuación del ciclo**) siga siendo verdadera. Las instrucciones de repetición son `while`, `do...while` y `for`. (En el capítulo 5 presentamos las instrucciones `do...while` y `for`). Las instrucciones `while` y `for` realizan la acción (o grupo de acciones) en sus cuerpos, cero o más veces; si en un principio la condición de continuación del ciclo es falsa, no se ejecutará la acción (o grupo de acciones). La instrucción `do...while` realiza la acción (o grupo de acciones) en su cuerpo, una o más veces. Las palabras `if`, `else`, `switch`, `while`, `do` y `for` son palabras clave en Java. En el apéndice C aparece una lista completa de las palabras clave en Java.

Resumen de las instrucciones de control en Java

Java sólo tiene tres tipos de estructuras de control, a las cuales nos referiremos de aquí en adelante como instrucciones de control: la instrucción de secuencia, las instrucciones de selección (tres tipos) y las instrucciones de repetición (tres tipos). Cada programa se forma combinando tantas de estas instrucciones como sea apropiado para el algoritmo que implemente el programa. Podemos modelar cada una de las instrucciones de control como un diagrama de actividad. Al igual que la figura 4.1, cada diagrama contiene un estado inicial y final, los cuales representan el punto de entrada y salida de la instrucción de control, respectivamente. Las **instrucciones de control de una sola entrada/una sola salida** facilitan la creación de programas; sólo tenemos que conectar el punto de salida de una al punto de entrada de la siguiente. A esto le llamamos **apilamiento de instrucciones de control**. En breve aprenderemos que sólo hay una manera alternativa de conectar las instrucciones de control: el **anidamiento de instrucciones de control**, en el cual una instrucción de control aparece *dentro* de otra. Por lo tanto, los algoritmos en los programas en Java se crean a partir de sólo tres principales tipos de instrucciones de control, que se combinan sólo de dos formas. Ésta es la esencia de la simpleza.

4.5 Instrucción if de selección simple

Los programas utilizan instrucciones de selección para elegir entre los cursos alternativos de acción. Por ejemplo, suponga que la calificación para aprobar un examen es 60. La instrucción en seudocódigo

```
Si la calificación del estudiante es mayor o igual que 60
    Imprimir "Aprobado"
```

determina si la condición "la calificación del estudiante es mayor o igual que 60" es verdadera. En caso de que sea así se imprime "Aprobado", y se "ejecuta" en orden la siguiente instrucción en seudocódigo. (Recuerde que el seudocódigo no es un verdadero lenguaje de programación). Si la condición es falsa se ignora la instrucción *Imprimir*, y se ejecuta en orden la siguiente instrucción en seudocódigo. La sangría de la segunda línea de esta instrucción de selección es opcional, pero se recomienda ya que enfatiza la estructura inherente de los programas estructurados.

La instrucción anterior *Si* en seudocódigo puede escribirse en Java de la siguiente manera:

```
if ( calificacionEstudiante >= 60 )
   System.out.println( "Aprobado" );
```

El código en Java corresponde en gran medida con el seudocódigo. Ésta es una de las propiedades que hace del seudocódigo una herramienta de desarrollo de programas tan útil.

La figura 4.2 muestra la instrucción if de selección simple. Esta figura contiene lo que quizá sea el símbolo más importante en un diagrama de actividad: el rombo o **símbolo de decisión**, el cual indica que se tomará una decisión. El flujo de trabajo continúa a lo largo de una ruta determinada por las condiciones de guardia asociadas de ese símbolo, que pueden ser verdaderas o falsas. Cada flecha de transición que sale de un símbolo de decisión tiene una **condición de guardia** (especificada entre corchetes, a un lado de la flecha de transición). Si una condición de guardia es verdadera, el flujo de trabajo entra al estado de acción al que apunta la flecha de transición. En la figura 4.2, si la calificación es mayor o igual que 60, el programa imprime "Aprobado" y luego se dirige al estado final de esta actividad. Si la calificación es menor que 60, el programa se dirige de inmediato al estado final sin mostrar ningún mensaje.

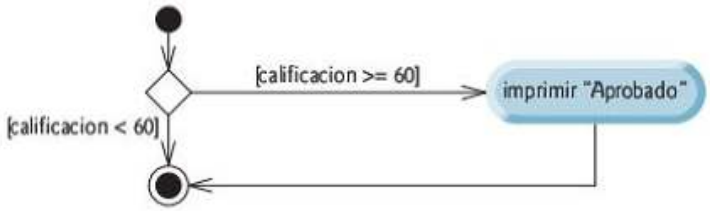


Fig. 4.2 | Diagrama de actividad en UML de la instrucción if de selección simple.

La instrucción if es una instrucción de control de una sola entrada/una sola salida. Pronto veremos que los diagramas de actividad para las instrucciones de control restantes también contienen estados iniciales, flechas de transición, estados de acción que indican las acciones a realizar, símbolos de decisión (con sus condiciones de guardia asociadas) que indican las decisiones a tomar, y estados finales.

4.6 Instrucción if...else de selección doble

La instrucción if de selección simple realiza una acción indicada sólo cuando la condición es verdadera (true); de no ser así, se evita dicha acción. La **instrucción if...else de selección doble** le permite espe-

cificar una acción a realizar cuando la condición es verdadera, y otra distinta cuando es falsa. Por ejemplo, la instrucción en seudocódigo:

```
Si la calificación del estudiante es mayor o igual que 60
    Imprimir "Aprobado"
De lo contrario
    Imprimir "Reprobado"
```

imprime "Aprobado" si la calificación del estudiante es mayor o igual que 60, y "Reprobado" si la calificación del estudiante es menor que 60. En cualquier caso, después de la impresión se "ejecuta" la siguiente instrucción en seudocódigo en la secuencia.

La instrucción anterior en seudocódigo *Si... De lo contrario* puede escribirse en Java como

```
if ( calificacion >= 60 )
   System.out.println( "Aprobado" );
else
   System.out.println( "Reprobado" );
```

El cuerpo de la instrucción `else` también tiene sangría. Cualquiera que sea la convención de sangría que usted elija, debe aplicarla de manera consistente en todos sus programas.

Buena práctica de programación 4.1

Utilice sangría en ambos cuerpos de instrucciones de una estructura `if...else`. Muchos IDE hacen esto por usted.

Buena práctica de programación 4.2

Si hay varios niveles de sangría, en cada uno debe aplicarse la misma cantidad de espacio adicional.

La figura 4.3 muestra el flujo de control en la instrucción `if...else`. Una vez más (además del estado inicial, las flechas de transición y el estado final), los símbolos en el diagrama de actividad de UML representan estados de acción y decisiones.

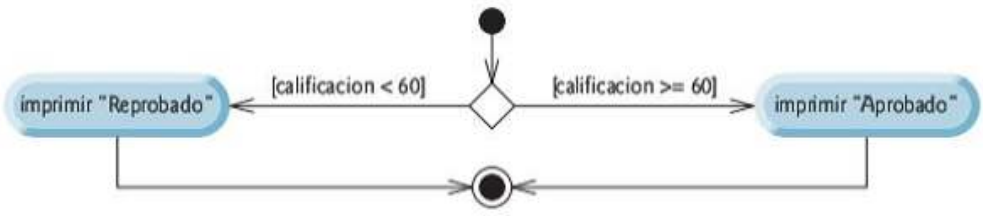


Fig. 4.3 | Diagrama de actividad de UML de la instrucción `if...else` de selección doble.

Operador condicional (?:)

Java cuenta con el **operador condicional** (`?:`), que en ocasiones puede utilizarse en lugar de una instrucción `if...else`. Éste es el único **operador ternario** en Java (un operador que utiliza tres operandos). En conjunto, los operandos y el símbolo `?:` forman una **expresión condicional**. El primer operando (a la izquierda del ?) es una **expresión booleana** (boolean) (es decir, una condición

que se evalúa a un valor boolean: true o false), el segundo operando (entre el ? y :) es el valor de la expresión condicional si la expresión booleana es verdadera, y el tercer operando (a la derecha del :) es el valor de la expresión condicional si la expresión booleana se evalúa como false. Por ejemplo, la instrucción

```
System.out.println( calificacionEstudiante>=60 ? "Aprobado" : "Reprobado" );
```

imprime el valor del argumento de la expresión condicional de println. La expresión condicional en esta instrucción produce como resultado la cadena "Aprobado" si la expresión booleana calificacionEstudiante >= 60 es verdadera, y la cadena "Reprobado" si la expresión booleana es falsa. Por lo tanto, esta instrucción con el operador condicional realiza en esencia la misma función que la instrucción if...else que se mostró anteriormente, en esta sección. Puesto que la precedencia del operador condicional es baja, es común que toda la expresión condicional se coloque entre paréntesis. Pronto veremos que las expresiones condicionales pueden usarse en algunas situaciones en las que no se pueden utilizar instrucciones if...else.

Instrucciones if...else anidadas

Un programa puede evaluar varios casos colocando instrucciones if...else dentro de otras instrucciones if...else para crear **instrucciones if...else anidadas**. Por ejemplo, el siguiente seudocódigo representa una instrucción if...else anidada que imprime A para las calificaciones de exámenes mayores o iguales a 90, B para las calificaciones en el rango de 80 a 89, C para las calificaciones en el rango de 70 a 79, D para las calificaciones en el rango de 60 a 69 y F para todas las demás calificaciones:

```
Si la calificación del estudiante es mayor o igual que 90
    Imprimir "A"
de lo contrario
    Si la calificación del estudiante es mayor o igual que 80
        Imprimir "B"
    de lo contrario
        Si la calificación del estudiante es mayor o igual que 70
            Imprimir "C"
        de lo contrario
            Si la calificación del estudiante es mayor o igual que 60
                Imprimir "D"
            de lo contrario
                Imprimir "F"
```

Este seudocódigo puede escribirse en Java como

```
if ( calificacionEstudiante >= 90 )
   System.out.println( "A" );
else
   if ( calificacionEstudiante >= 80 )
      System.out.println( "B" );
   else
      if ( calificacionEstudiante >= 70 )
         System.out.println( "C" );
      else
         if ( calificacionEstudiante >= 60 )
            System.out.println( "D" );
         else
            System.out.println( "F" );
```

Si la variable `calificacionEstudiante` es mayor o igual que 90, las primeras cuatro condiciones en la instrucción `if...else` anidada serán verdaderas, pero sólo se ejecutará la instrucción en la parte `if` de la primera instrucción `if...else`. Después de que se ejecute esa instrucción, se evita la parte `else` de la instrucción `if...else` más "externa". La mayoría de los programadores en Java prefieren escribir la instrucción `if...else` anterior así:

```
if ( calificacionEstudiante >= 90 )
   System.out.println( "A" );
else if ( calificacionEstudiante >= 80 )
   System.out.println( "B" );
else if ( calificacionEstudiante >= 70 )
   System.out.println( "C" );
else if ( calificacionEstudiante >= 60 )
   System.out.println( "D" );
else
   System.out.println( "F" );
```

Las dos formas son idénticas, excepto por el espaciado y la sangría, que el compilador ignora. La segunda forma es más popular ya que evita usar mucha sangría hacia la derecha en el código. Dicha sangría a menudo deja poco espacio en una línea de código, forzando a que las líneas se separen.

Problema del `else` suelto

El compilador de Java siempre asocia un `else` con el `if` que le precede inmediatamente, a menos que se le indique otra cosa mediante la colocación de llaves ({ y }). Este comportamiento puede ocasionar lo que se conoce como el **problema del `else` suelto**. Por ejemplo,

```
if ( x > 5 )
   if ( y > 5 )
      System.out.println( "x e y son > 5" );
else
   System.out.println( "x es <= 5" );
```

parece indicar que si `x` es mayor que 5, la instrucción `if` anidada determina si `y` es también mayor que 5. De ser así, se produce como resultado la cadena "`x e y son > 5`". De lo contrario, parece ser que si `x` no es mayor que 5, la instrucción `else` que es parte del `if...else` produce como resultado la cadena "`x es <= 5`". ¡Cuidado! Esta instrucción `if...else` anidada no se ejecuta como parece ser. El compilador en realidad interpreta la instrucción así

```
if ( x > 5 )
   if ( y > 5 )
      System.out.println( "x e y son > 5" );
   else
      System.out.println( "x es <= 5" );
```

en donde el cuerpo del primer `if` es un `if...else` anidado. La instrucción `if` más externa evalúa si `x` es mayor que 5. De ser así, la ejecución continúa evaluando si `y` es también mayor que 5. Si la segunda condición es verdadera, se muestra la cadena apropiada ("`x e y son > 5`"). No obstante, si la segunda condición es falsa se muestra la cadena "`x es <= 5`", aun cuando sabemos que `x` es mayor que 5. Además, si la condición de la instrucción `if` exterior es falsa, se omite la instrucción `if...else` interior y no se muestra nada en pantalla.

Para forzar a que la instrucción `if...else` anidada se ejecute como se tenía pensado originalmente, debe escribirse de la siguiente manera:

```
if ( x > 5 )
{
   if ( y > 5 )
      System.out.println( "x e y son > 5" );
}
else
   System.out.println( "x es <= 5" );
```

Las llaves indican que el segundo `if` se encuentra en el cuerpo del primer `if`, y que el `else` está asociado con el *primer* `if`. Los ejercicios 4.27 y 4.28 analizan con más detalle el problema del `else` suelto.

Bloques

Por lo general, la instrucción `if` espera sólo una instrucción en su cuerpo. Para incluir varias instrucciones en el cuerpo de un `if` (o en el cuerpo del `else` en una instrucción `if...else`), encierre las instrucciones entre llaves. A las instrucciones contenidas dentro de un par de llaves se le llama **bloque**. Un bloque puede colocarse en cualquier parte de un programa en donde pueda colocarse una sola instrucción.

El siguiente ejemplo incluye un bloque en la parte `else` de una instrucción `if...else`:

```
if ( calificacion >= 60 )
   System.out.println( "Aprobado" );
else
{
   System.out.println( "Reprobado" );
   System.out.println( "Debe tomar este curso otra vez." );
}
```

En este caso, si `calificacion` es menor que 60, el programa ejecuta *ambas* instrucciones en el cuerpo del `else` e imprime

```
Reprobado.
Debe tomar este curso otra vez.
```

Observe las llaves que rodean a las dos instrucciones en la cláusula `else`. Éstas son importantes. Sin ellas, la instrucción

```
System.out.println( "Debe tomar este curso otra vez." );
```

estaría fuera del cuerpo de la parte `else` de la instrucción `if...else` y se ejecutaría *sin importar* que la calificación fuera menor a 60.

Los errores de sintaxis (como cuando se omite una llave en un bloque del programa) los atrapa el compilador. Un **error lógico** (como cuando se omiten ambas llaves en un bloque del programa) tiene su efecto en tiempo de ejecución. Un **error lógico fatal** hace que un programa falle y termine antes de tiempo. Un **error lógico no fatal** permite que un programa siga ejecutándose, pero éste produce resultados incorrectos.

Así como un bloque puede colocarse en cualquier parte en donde pueda escribirse una sola instrucción, también es posible no tener instrucción alguna. En la sección 2.8 vimos que la instrucción vacía se representa con un punto y coma (;) en donde normalmente iría una instrucción.

Error común de programación 4.1

Colocar un punto y coma después de la condición en una instrucción `if` o `if...else` produce un error lógico en las instrucciones `if` de selección simple, y un error de sintaxis en las instrucciones `if...else` de selección doble (cuando la parte del `if` contiene una instrucción en el cuerpo).

4.7 Instrucción de repetición `while`

Una **instrucción de repetición** (o **de ciclo**) le permite especificar que un programa debe repetir una acción mientras cierta condición sea verdadera. La instrucción en seudocódigo

Mientras existan más artículos en mi lista de compras
Comprar el siguiente artículo y quitarlo de mi lista

describe la repetición que ocurre al ir de compras. La condición "existan más artículos en mi lista de compras" puede ser verdadera o falsa. Si es verdadera, entonces se realiza la acción "Comprar el siguiente artículo y quitarlo de mi lista". Esta acción se realizará en forma repetida mientras la condición sea verdadera. La instrucción (o instrucciones) contenida en la instrucción de repetición *Mientras* constituye el cuerpo de esta estructura, el cual puede ser una sola instrucción o un bloque. En algún momento, la condición será falsa (cuando el último artículo de la lista de compras sea adquirido y eliminado). En este punto la repetición terminará y se ejecutará la primera instrucción que esté después de la instrucción de repetición.

Como ejemplo de la **instrucción de repetición `while`** en Java, considere un segmento de programa que encuentra la primera potencia de 3 que sea mayor que 100. Suponga que la variable `producto` de tipo `int` se inicializa en 3. Cuando la siguiente instrucción `while` termine de ejecutarse, el producto contendrá el resultado:

```
while ( producto <= 100 )
   producto = 3 * producto;
```

Cuando esta instrucción `while` comienza a ejecutarse, el valor de la variable producto es 3. Cada iteración de la instrucción `while` multiplica a producto por 3, por lo que producto toma los valores de 9, 27, 81 y 243, sucesivamente. Cuando la variable producto se vuelve 243, la condición de la instrucción `while` (`producto <= 1000`) se torna falsa. Esto termina la repetición, por lo que el valor final de producto es 243. En este punto, la ejecución del programa continúa con la siguiente instrucción después de la instrucción `while`.

Error común de programación 4.2

Si no se proporciona, en el cuerpo de una instrucción `while`, una acción que ocasione que en algún momento la condición de un `while` se torne falsa, por lo general se producirá un error lógico conocido como ***ciclo infinito*** *(el ciclo nunca termina).*

El diagrama de actividad de UML de la figura 4.4 muestra el flujo de control que corresponde a la instrucción `while` anterior. Una vez más (aparte del estado inicial, las flechas de transición, un estado final y tres notas), los símbolos en el diagrama representan un estado de acción y una decisión. Este diagrama introduce el **símbolo de fusión**. UML representa al símbolo de fusión y al símbolo de decisión como rombos. El símbolo de fusión une dos flujos de actividad en uno solo. En este diagrama, el símbolo de fusión une las transiciones del estado inicial y del estado de acción, de manera que ambas fluyan en la decisión que determina si el ciclo debe empezar a ejecutarse (o seguir ejecutándose). Los símbolos de decisión y de fusión pueden diferenciarse por el número de flechas de transición "entrantes" y "salientes". Un símbolo de decisión tiene una flecha de transición que apunta hacia el rombo y dos o más que apuntan hacia fuera del rombo, para indicar las posibles transiciones desde ese punto. Además, cada flecha de transición que apunta hacia fuera de un símbolo de decisión tiene una condición de guardia junto a ella. Un símbolo de fusión tiene dos o más flechas de transición que apuntan hacia el rombo, y sólo una que apunta hacia fuera de él, para indicar múltiples flujos de actividad que se fusionan para continuar la actividad. *Ninguna* de las flechas de transición asociadas con un símbolo de fusión tiene una condición de guardia.

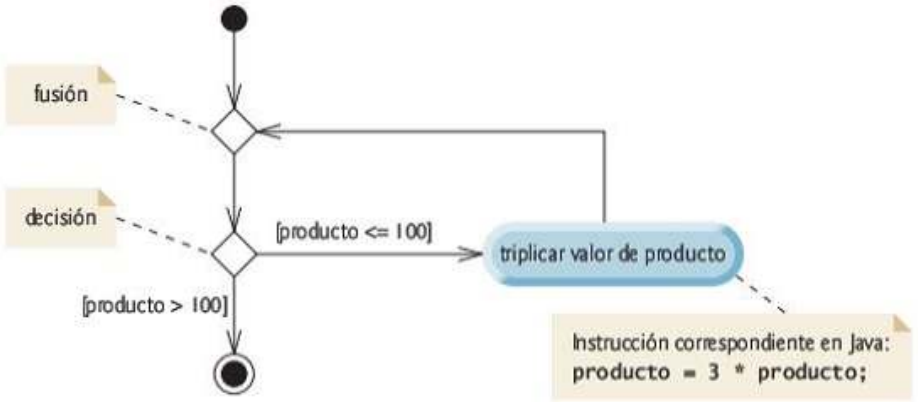


Fig. 4.4 | Diagrama de actividad de UML de la instrucción de repetición `while`.

La figura 4.4 muestra con claridad la repetición de la instrucción `while` que vimos antes en esta sección. La flecha de transición que emerge del estado de acción apunta de regreso a la fusión, desde la cual el flujo del programa regresa a la decisión que se evalúa al principio de cada iteración del ciclo. El cual sigue ejecutándose hasta que la condición de guardia `producto > 100` se vuelva verdadera. Entonces, la instrucción `while` termina (llega a su estado final) y el control pasa a la siguiente instrucción en la secuencia del programa.

4.8 Cómo formular algoritmos: repetición controlada por un contador

Para ilustrar la forma en que se desarrollan los algoritmos, modificamos la clase `LibroCalificaciones` del capítulo 3, para resolver dos variantes de un problema que promedia las calificaciones de unos estudiantes. Considere el siguiente enunciado del problema:

> *A una clase de diez estudiantes se les aplicó un examen. Las calificaciones (enteros en el rango de 0 a 100) de este examen están disponibles para usted. Determine el promedio de la clase para este examen.*

El promedio de la clase es igual que la suma de las calificaciones dividida entre el número de estudiantes. El algoritmo para resolver este problema en una computadora debe recibir como entrada cada una de las calificaciones, llevar el registro del total de las calificaciones introducidas, realizar el cálculo para promediar e imprimir el resultado.

Algoritmo de seudocódigo con repetición controlada por un contador

Emplearemos seudocódigo para enumerar las acciones que deben llevarse a cabo y especificar el orden en que deben ejecutarse. Usaremos una **repetición controlada por contador** para introducir las calificaciones, una por una. Esta técnica utiliza una variable llamada contador (o **variable de control**) para controlar el número de veces que debe ejecutarse un conjunto de instrucciones. A la repetición controlada por contador se le llama comúnmente **repetición definida**, ya que el número de repeticiones se conoce *antes* de que el ciclo comience a ejecutarse. En este ejemplo, la repetición termina cuando el contador excede a 10. Esta sección presenta un algoritmo de seudocódigo (figura 4.5) completamente desarrollado, y una versión de la clase `LibroCalificaciones` (figura 4.6) que implementa el algoritmo en un método de Java. Después presentamos una aplicación (figura 4.7) que demuestra el algoritmo en acción. En la sección 4.9 demostraremos cómo utilizar el seudocódigo para desarrollar dicho algoritmo desde cero.

Observación de ingeniería de software 4.1

La experiencia ha demostrado que la parte más difícil para la resolución de un problema en una computadora es desarrollar el algoritmo para la solución. Por lo general, una vez que se ha especificado el algoritmo correcto, el proceso de producir un programa funcional en Java a partir de dicho algoritmo es relativamente sencillo.

Observe las referencias en el algoritmo de la figura 4.5 para un total y un contador. Un total es una variable que se utiliza para acumular la suma de varios valores. Un contador es una variable que se utiliza para contar; en este caso, el contador de calificaciones indica cuál de las 10 calificaciones está a punto de escribir el usuario. Por lo general, las variables que se utilizan para guardar totales deben inicializarse en cero antes de utilizarse en un programa.

```
1  Asignar a total el valor de cero
2  Asignar al contador de calificaciones el valor de uno
3
4  Mientras que el contador de calificaciones sea menor o igual que diez
5     Pedir al usuario que introduzca la siguiente calificación
6     Obtener como entrada la siguiente calificación
7     Sumar la calificación al total
8     Sumar uno al contador de calificaciones
9
10 Asignar al promedio de la clase el total dividido entre diez
11 Imprimir el promedio de la clase
```

Fig. 4.5 | Algoritmo en seudocódigo que utiliza la repetición controlada por contador para resolver el problema del promedio de una clase.

Implementación de la repetición controlada por contador en la clase `LibroCalificaciones`

La clase `LibroCalificaciones` (figura 4.6) contiene un constructor (líneas 11 a la 14) que asigna un valor a la variable de instancia `nombreDelCurso` (declarada en la línea 8) de la clase. Las líneas 17 a la 20, 23 a la 26 y 29 a la 34 declaran los métodos `establecerNombreDelCurso`, `obtenerNombreDelCurso` y `mostrarMensaje`, respectivamente. Las líneas 37 a la 66 declaran el método `determinarPromedioClase`, el cual implementa el algoritmo para sacar el promedio de la clase, descrito por el seudocódigo de la figura 4.5.

La línea 40 declara e inicializa la variable `entrada` de tipo `Scanner`, que se utiliza para leer los valores introducidos por el usuario. Las líneas 42 a 45 declaran las variables locales `total`, `contadorCalif`, `calificacion` y `promedio` de tipo `int`. La variable calificacion almacena la entrada del usuario.

```
1  // Fig. 4.6: LibroCalificaciones.java
2  // La clase LibroCalificaciones que resuelve el problema del promedio
3  // usando la repetición controlada por un contador.
4  import java.util.Scanner; // el programa utiliza la clase Scanner
5
6  public class LibroCalificaciones
7  {
8     private String nombreDelCurso; // el nombre del curso que representa este
                                     LibroCalificaciones
```

Fig. 4.6 | Clase `LibroCalificaciones` que resuelve el problema del promedio de una clase mediante la repetición controlada por contador (parte 1 de 3).

```
9
10    // el constructor inicializa a nombreDelCurso
11    public LibroCalificaciones( String nombre )
12    {
13       nombreDelCurso = nombre; // inicializa a nombreDelCurso
14    } // fin del constructor
15
16    // método para establecer el nombre del curso
17    public void establecerNombreDelCurso( String nombre )
18    {
19       nombreDelCurso = nombre; // almacena el nombre del curso
20    } // fin del método establecerNombreDelCurso
21
22    // método para obtener el nombre del curso
23    public String obtenerNombreDelCurso()
24    {
25       return nombreDelCurso;
26    } // fin del método obtenerNombreDelCurso
27
28    // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
29    public void mostrarMensaje()
30    {
31       // obtenerNombreDelCurso obtiene el nombre del curso
32       System.out.printf( "Bienvenido al libro de calificaciones para\n%s!\n\n",
33          obtenerNombreDelCurso() );
34    } // fin del método mostrarMensaje
35
36    // determina el promedio de la clase, con base en las 10 calificaciones
         introducidas por el usuario
37    public void determinarPromedioClase()
38    {
39       // crea objeto Scanner para obtener la entrada de la ventana de comandos
40       Scanner entrada = new Scanner( System.in );
41
42       int total; // suma de las calificaciones escritas por el usuario
43       int contadorCalif; // número de la siguiente calificación a introducir
44       int calificacion; // valor de la calificación escrita por el usuario
45       int promedio; // el promedio de las calificaciones
46
47       // fase de inicialización
48       total = 0; // inicializa el total
49       contadorCalif = 1; // inicializa el contador del ciclo
50
51       // fase de procesamiento; utiliza la repetición controlada por contador
52       while ( contadorCalif <= 10 ) // itera 10 veces
53       {
54          System.out.print( "Escriba la calificación: " ); // indicador
55          calificacion = entrada.nextInt(); // lee calificación del usuario
56          total = total + calificacion; // suma calificación a total
57          contadorCalif = contadorCalif + 1; // incrementa contador en 1
58       } // fin de while
59
```

Fig. 4.6 | Clase `LibroCalificaciones` que resuelve el problema del promedio de una clase mediante la repetición controlada por contador (parte 2 de 3).

```
60          // fase de terminación
61          promedio = total / 10; // la división entera produce un resultado entero
62
63          // muestra el total y el promedio de las calificaciones
64          System.out.printf( "\nEl total de las 10 calificaciones es %d\n", total );
65          System.out.printf( "El promedio de la clase es %d\n", promedio );
66       } // fin del método determinarPromedioClase
67    } // fin de la clase LibroCalificaciones
```

Fig. 4.6 | Clase `LibroCalificaciones` que resuelve el problema del promedio de una clase mediante la repetición controlada por contador (parte 3 de 3).

Las declaraciones (en las líneas 42 a la 45) aparecen en el cuerpo del método `determinarPromedioClase`. Recuerde que las variables declaradas en el cuerpo de un método son variables locales, y sólo pueden utilizarse desde la línea de su declaración hasta la llave derecha de cierre de la declaración del método. La declaración de una variable local debe aparecer antes de que se utilice en ese método. Una variable local no puede usarse fuera del método en el que se declara.

En este libro, la clase `LibroCalificaciones` sólo lee y procesa un conjunto de calificaciones. El cálculo del promedio se realiza en el método `determinarPromedioClase`, usando variables locales; no preservamos información acerca de las calificaciones de los estudiantes en variables de instancia de la clase.

Las asignaciones (en las líneas 48 y 49) inicializan `total` a `0` y `contadorCalif` a `1`. Observe que estas inicializaciones ocurren *antes* que se utilicen las variables en los cálculos. Las variables `calificacion` y `promedio` (para la entrada del usuario y el promedio calculado, respectivamente) no necesitan inicializarse aquí; sus valores se asignarán a medida que se introduzcan o calculen más adelante en el método.

Error común de programación 4.3

Usar el valor de una variable local antes de inicializarla produce un error de compilación. Todas las variables locales deben inicializarse antes de utilizar sus valores en las expresiones.

Tip para prevenir errores 4.1

Inicialice cada contador y total, ya sea en su declaración o en una instrucción de asignación. Por lo general, los totales se inicializan a 0. Los contadores comúnmente se inicializan a 0 o a 1, dependiendo de cómo se utilicen (más adelante veremos ejemplos de cuándo usar 0 y cuándo usar 1).

La línea 52 indica que la instrucción `while` debe continuar ejecutando el ciclo (lo que también se conoce como **iterar**), siempre y cuando el valor de `contadorCalif` sea menor o igual que 10. Mientras esta condición sea verdadera, la instrucción `while` ejecutará en forma repetida las instrucciones entre las llaves que delimitan su cuerpo (líneas 54 a la 57).

La línea 54 muestra el indicador `"Escriba la calificacion:"`. La línea 55 lee el dato escrito por el usuario y lo asigna a la variable `calificacion`. Después, la línea 56 suma la nueva `calificacion` escrita por el usuario al `total`, y asigna el resultado a `total`, que sustituye su valor anterior.

La línea 57 suma `1` a `contadorCalif` para indicar que el programa ha procesado una calificación y está listo para recibir la siguiente calificación del usuario. Al incrementar a `contadorCalif` en cada iteración, en un momento dado su valor excederá a 10. En ese momento, el ciclo termina debido a que su condición (línea 52) se vuelve falsa.

Cuando el ciclo termina, la línea 61 realiza el cálculo del promedio y asigna su resultado a la variable promedio. La línea 64 utiliza el método `printf` de `System.out` para mostrar el texto `"El total de las`

10 calificaciones es ", seguido del valor de la variable total. Después, la línea 65 utiliza a printf para mostrar el texto "El promedio de la clase es ", seguido del valor de la variable promedio. Después de llegar a la línea 66, el método determinarPromedioClase devuelve el control al método que hizo la llamada (es decir, a main en PruebaLibroCalificaciones de la figura 4.7).

La clase PruebaLibroCalificaciones

La clase PruebaLibroCalificaciones (figura 4.7) crea un objeto de la clase LibroCalificaciones (figura 4.6) y demuestra sus capacidades. Las líneas 10 y 11 de la figura 4.7 crean un nuevo objeto LibroCalificaciones y lo asignan a la variable miLibroCalificaciones. El objeto String en la línea 11 se pasa al constructor de LibroCalificaciones (líneas 11 a la 14 de la figura 4.6). La línea 13 llama al método mostrarMensaje de miLibroCalificaciones para mostrar un mensaje de bienvenida al usuario. Después, la línea 14 llama al método determinarPromedioClase de miLibroCalificaciones para permitir que el usuario introduzca 10 calificaciones, para las cuales el método posteriormente calcula e imprime el promedio; el método ejecuta el algoritmo que se muestra en la figura 4.5.

```
1  // Fig. 4.7: PruebaLibroCalificaciones.java
2  // Crea un objeto LibroCalificaciones e invoca a su método obtenerPromedioClase.
3
4  public class PruebaLibroCalificaciones
5  {
6     public static void main( String[] args )
7     {
8        // crea objeto miLibroCalificaciones de la clase LibroCalificaciones y
9        // pasa el nombre del curso al constructor
10       LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
11          "CS101 Introduccion a la programacion en Java" );
12
13       miLibroCalificaciones.mostrarMensaje(); // muestra mensaje de bienvenida
14       miLibroCalificaciones.determinarPromedioClase(); // encuentra el promedio de
                                                          10 calificaciones
15    } // fin de main
16 } // fin de la clase PruebaLibroCalificaciones
```

```
Bienvenido al libro de calificaciones para
CS101 Introduccion a la programación en Java!

Escriba la calificacion: 67
Escriba la calificacion: 78
Escriba la calificacion: 89
Escriba la calificacion: 67
Escriba la calificacion: 87
Escriba la calificacion: 98
Escriba la calificacion: 93
Escriba la calificacion: 85
Escriba la calificacion: 82
Escriba la calificacion: 100

El total de las 10 calificaciones es 846
El promedio de la clase es 84
```

Fig. 4.7 | La clase PruebaLibroCalificaciones crea un objeto de la clase LibroCalificaciones (figura 4.6) e invoca a su método determinarPromedioClase.

Observaciones acerca de la división de enteros y el truncamiento

El cálculo del promedio realizado por el método `determinarPromedioClase` en respuesta a la llamada al método en la línea 14 de la figura 4.7, produce un resultado entero. La salida del programa indica que la suma de los valores de las calificaciones en la ejecución de ejemplo es 846, que al dividirse entre 10, debe producir el número de punto flotante 84.6. Sin embargo, el resultado del cálculo `total / 10` (línea 61 de la figura 4.6) es el entero 84, ya que `total` y `10` son enteros. Al dividir dos enteros se produce una **división entera**: se pierde cualquier parte fraccionaria del cálculo (es decir, se **trunca**). En la siguiente sección veremos cómo obtener un resultado de punto flotante a partir del cálculo del promedio.

Error común de programación 4.4

Suponer que la división entera redondea (en vez de truncar) puede producir resultados erróneos. Por ejemplo, 7 + 4, que produce 1.75 en la aritmética convencional, se trunca a 1 en la aritmética entera, en vez de redondearse a 2.

4.9 Cómo formular algoritmos: repetición controlada por un centinela

Generalicemos el problema, de la sección 4.8, para los promedios de una clase. Considere el siguiente problema:

> *Desarrollar un programa que calcule el promedio de una clase y procese las calificaciones para un número arbitrario de estudiantes cada vez que se ejecute.*

En el ejemplo anterior del promedio de una clase, el enunciado del problema especificó el número de estudiantes, por lo que se conocía el número de calificaciones (10) de antemano. En este ejemplo no se indica cuántas calificaciones introducirá el usuario durante la ejecución del programa. Éste debe procesar un número arbitrario de calificaciones. ¿Cómo puede el programa determinar cuándo terminar de introducir calificaciones? ¿Cómo sabrá cuándo calcular e imprimir el promedio de la clase?

Una manera de resolver este problema es utilizar un valor especial denominado **valor centinela** (también llamado **valor de señal**, **valor de prueba** o **valor de bandera**) para indicar el "fin de la introducción de datos". El usuario escribe calificaciones hasta que se haya introducido el número correcto de ellas. Después, el usuario escribe el valor centinela para indicar que no se van a introducir más calificaciones. A la **repetición controlada por centinela** a menudo se le llama **repetición indefinida**, ya que el número de repeticiones no se conoce antes de que comience la ejecución del ciclo.

Sin duda, debe elegirse un valor centinela de tal forma que no pueda confundirse con un valor de entrada permitido. Las calificaciones de un examen son enteros positivos, por lo que –1 es un valor centinela aceptable para este problema. Por lo tanto, una ejecución del programa para promediar una clase podría procesar una cadena de entradas como 95, 96, 75, 74, 89 y –1. El programa entonces calcularía e imprimiría el promedio de la clase para las calificaciones 95, 96, 75, 74 y 89; como –1 es el valor centinela, *no* debe entrar en el cálculo del promedio.

Desarrollo del algoritmo en seudocódigo con el método de refinamiento de arriba a abajo, paso a paso: el primer refinamiento (cima)

Vamos a desarrollar el programa para promediar clases con una técnica llamada **refinamiento de arriba a abajo, paso a paso**, la cual es esencial para el desarrollo de programas bien estructurados. Comenzamos con una representación en seudocódigo de la **cima**, una sola instrucción que transmite la función del programa en general:

Determinar el promedio de la clase para el examen

La cima es, en efecto, la representación *completa* de un programa. Por desgracia, pocas veces transmite los detalles suficientes como para escribir un programa en Java. Por lo tanto, ahora comenzaremos el proceso de refinamiento. Dividiremos la cima en una serie de tareas más pequeñas y las enumeraremos en el orden en el que se van a realizar. Esto arroja como resultado el siguiente **primer refinamiento**:

Inicializar variables
Introducir, sumar y contar las calificaciones del examen
Calcular e imprimir el promedio de la clase

Este refinamiento utiliza sólo la estructura de secuencia; los pasos aquí mostrados deben ejecutarse en orden, uno después del otro.

Observación de ingeniería de software 4.2

Cada refinamiento, así como la cima en sí, es una especificación completa del algoritmo; sólo varía el nivel del detalle.

Observación de ingeniería de software 4.3

Muchos programas pueden dividirse lógicamente en tres fases: una fase de inicialización, *en donde se inicializan las variables; una fase de* procesamiento, *en donde se introducen los valores de los datos y se ajustan las variables del programa según sea necesario; y una fase de* terminación, *que calcula y produce los resultados finales.*

Cómo proceder al segundo refinamiento

La anterior Observación de Ingeniería de Software es a menudo todo lo que usted necesita para el primer refinamiento en el proceso de arriba a abajo. Para avanzar al siguiente nivel de refinamiento (es decir, el **segundo refinamiento**), nos comprometemos a usar variables específicas. En este ejemplo necesitamos el total actual de los números, una cuenta de cuántos números se han procesado, una variable para recibir el valor de cada calificación, a medida que el usuario las vaya introduciendo, y una variable para almacenar el promedio calculado. La instrucción en seudocódigo

Inicializar las variables

puede mejorarse como sigue:

Inicializar total en cero
Inicializar contador en cero

Sólo las variables *total* y *contador* necesitan inicializase antes de que puedan utilizarse. Las variables *promedio* y *calificacion* (para el promedio calculado y la entrada del usuario, respectivamente) no necesitan inicializarse, ya que sus valores se reemplazarán a medida que se calculen o introduzcan.

La instrucción en seudocódigo

Introducir, sumar y contar las calificaciones del examen

requiere una estructura de repetición (es decir, un ciclo) que introduzca cada calificación en forma sucesiva. No sabemos de antemano cuántas calificaciones van a procesarse, por lo que utilizaremos la repetición controlada por centinela. El usuario introduce las calificaciones, una por una. Después de la última, el usuario mete el valor centinela. El programa lo evalúa luego de insertar cada calificación, y termina el ciclo cuando el usuario introduce el valor centinela. Entonces, el segundo refinamiento de la instrucción anterior en seudocódigo sería

```
Pedir al usuario que introduzca la primera calificación
Recibir como entrada la primera calificación (puede ser el centinela)

Mientras el usuario no haya introducido aún el centinela
    Sumar esta calificación al total actual
    Sumar uno al contador de calificaciones
    Pedir al usuario que introduzca la siguiente calificación
    Recibir como entrada la siguiente calificación (puede ser el centinela)
```

En seudocódigo no utilizamos llaves alrededor de las instrucciones que forman el cuerpo de la estructura *Mientras*. Simplemente aplicamos sangría a las instrucciones bajo el *Mientras* para mostrar que pertenecen a esta instrucción. De nuevo, el seudocódigo es solamente una herramienta informal para desarrollar programas.

La instrucción en seudocódigo

```
Calcular e imprimir el promedio de la clase
```

puede mejorarse de la siguiente manera:

```
Si el contador no es igual que cero
    Asignar al promedio el total dividido entre el contador
    Imprimir el promedio
de lo contrario
    Imprimir "No se introdujeron calificaciones"
```

Aquí tenemos cuidado de evaluar la posibilidad de una división entre cero; por lo general esto es un error lógico que, si no se detecta, haría que el programa fallara o produjera resultados inválidos. El segundo refinamiento completo del seudocódigo para el problema del promedio de una clase se muestra en la figura 4.8.

Tip para prevenir errores 4.2

Al realizar una división entre una expresión cuyo valor pudiera ser cero, debe evaluar explícitamente esta posibilidad y manejarla de manera apropiada en su programa (como imprimir un mensaje de error), en vez de permitir que ocurra el error.

```
1  Inicializar total en cero
2  Inicializar contador en cero
3
4  Pedir al usuario que introduzca la primera calificación
5  Recibir como entrada la primera calificación (puede ser el centinela)
6
7  Mientras el usuario no haya introducido aún el centinela
8      Sumar esta calificación al total actual
9      Sumar uno al contador de calificaciones
10     Pedir al usuario que introduzca la siguiente calificación
11     Recibir como entrada la siguiente calificación (puede ser el centinela)
12
13 Si el contador no es igual que cero
14     Asignar al promedio el total dividido entre el contador
15     Imprimir el promedio
16 de lo contrario
17     Imprimir "No se introdujeron calificaciones"
```

Fig. 4.8 | Algoritmo en seudocódigo del problema para promediar una clase, con una repetición controlada por centinela.

En las figuras 4.5 y 4.8 incluimos líneas en blanco y sangría en el seudocódigo para facilitar su lectura. Las líneas en blanco separan los algoritmos en sus fases y accionan las instrucciones de control; la sangría enfatiza los cuerpos de las estructuras de control.

El algoritmo en seudocódigo en la figura 4.8 resuelve el problema más general para promediar una clase. Este algoritmo se desarrolló después de aplicar dos niveles de refinamiento. En ocasiones se requieren más.

Observación de ingeniería de software 4.4

Termine el proceso de refinamiento de arriba a abajo, paso a paso, cuando haya especificado el algoritmo en seudocódigo con el detalle suficiente como para poder convertir el seudocódigo en Java. Por lo general, la implementación del programa en Java después de esto es mucho más sencilla.

Observación de ingeniería de software 4.5

Algunos programadores no utilizan herramientas de desarrollo de programas como el seudocódigo. Sienten que su meta final es resolver el problema en una computadora y que el escribir seudocódigo simplemente retarda la producción de los resultados finales. Aunque este método pudiera funcionar para problemas sencillos y conocidos, tiende a ocasionar graves errores y retrasos en proyectos grandes y complejos.

Implementación de la repetición controlada por centinela en la clase `LibroCalificaciones`

La figura 4.9 muestra la clase de Java `LibroCalificaciones` que contiene el método `determinarPromedioClase`, el cual implementa el algoritmo de la figura 4.8 en seudocódigo. Aunque cada calificación es un valor entero, existe la probabilidad de que el cálculo del promedio produzca un número con un punto decimal; en otras palabras, un número real (es decir, de punto flotante). El tipo `int` no puede representar un número de este tipo, por lo que esta clase utiliza el tipo `double` para ello.

```
1  // Fig. 4.9: LibroCalificaciones.java
2  // La clase LibroCalificaciones resuelve el problema del promedio de la clase
3  // usando la repetición controlada por un centinela.
4  import java.util.Scanner; // el programa usa la clase Scanner
5
6  public class LibroCalificaciones
7  {
8     private String nombreDelCurso; // el nombre del curso que representa este
                                     LibroCalificaciones
9
10    // el constructor inicializa a nombreDelCurso
11    public LibroCalificaciones( String nombre )
12    import java.util.Scanner; // el programa utiliza la clase Scanner
13       nombreDelCurso = nombre; // inicializa a nombreDelCurso
14    } // fin del constructor
15
16    // método para establecer el nombre del curso
17    public void establecerNombreDelCurso( String nombre )
18    {
19       nombreDelCurso = nombre; // almacena el nombre del curso
20    } // fin del método establecerNombreDelCurso
21
22    // método para obtener el nombre del curso
23    public String obtenerNombreDelCurso()
24    {
```

Fig. 4.9 | La clase `LibroCalificacion` que resuelve el problema del promedio de una clase mediante la repetición controlada por centinela (parte 1 de 3).

```
25          return nombreDelCurso;
26       } // fin del método obtenerNombreDelCurso
27
28       // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
29       public void mostrarMensaje()
30       {
31          // obtenerNombreDelCurso obtiene el nombre del curso
32          System.out.printf( "Bienvenido al libro de calificaciones para\n%s!\n\n",
33             obtenerNombreDelCurso() );
34       } // fin del método mostrarMensaje
35
36       // determina el promedio de un número arbitrario de calificaciones
37       public void determinarPromedioClase()
38       {
39       // crea objeto Scanner para obtener la entrada de la ventana de comandos
40       Scanner entrada = new Scanner( System.in );
41
42       int total; // suma de las calificaciones
43       int contadorCalif; // número de calificaciones introducidas
44       int calificacion; // valor de calificación
45       double promedio; // número con punto decimal para el promedio
46
47       // fase de inicialización
48       total = 0; // inicializa el total
49       contadorCalif = 0; // inicializa el contador del ciclo
50
51       // fase de procesamiento
52       // pide entrada y lee calificación del usuario
53       System.out.print( "Escriba calificacion o -1 para terminar: " );
54       calificacion = entrada.nextInt();
55
56       // itera hasta leer el valor centinela del usuario
57       while ( calificacion != -1 )
58       {
59          total = total + calificacion; // suma calificacion al total
60          contadorCalif = contadorCalif + 1; // incrementa el contador
61
62       // pide entrada y lee siguiente calificación del usuario
63       System.out.print( "Escriba calificacion o -1 para terminar: " );
64       calificacion = entrada.nextInt();
65       } // fin de while
66
67       // fase de terminación
68       // si el usuario introdujo por lo menos una calificación...
69       if ( contadorCalif != 0 )
70       {
71          // calcula el promedio de todas las calificaciones introducidas
72          promedio = (double) total / contadorCalif;
73
74          // muestra el total y el promedio (con dos dígitos de precisión)
75          System.out.printf( "\nEl total de las %d calificaciones introducidas es %d\n",
76             contadorCalif, total );
```

Fig. 4.9 | La clase `LibroCalificacion` que resuelve el problema del promedio de una clase mediante la repetición controlada por centinela (parte 2 de 3).

```
77            System.out.printf( "El promedio de la clase es %.2f\n", promedio );
78         } // fin de if
79         else // no se introdujeron calificaciones, por lo que se imprime el mensaje
80            apropiado
               System.out.println( "No se introdujeron calificaciones" );
81      } // fin del método determinarPromedioClase
82   } // fin de la clase LibroCalificaciones
```

Fig. 4.9 | La clase `LibroCalificacion` que resuelve el problema del promedio de una clase mediante la repetición controlada por centinela (parte 3 de 3).

En este ejemplo vemos que las estructuras de control pueden *apilarse* una encima de otra (en secuencia). La instrucción `while` (líneas 57 a 65) va seguida por una instrucción `if...else` (líneas 69 a 80) en secuencia. La mayor parte del código en este programa es igual que el código de la figura 4.6, por lo que nos concentraremos en los nuevos conceptos.

La línea 45 declara la variable `promedio` de tipo `double`, la cual nos permite guardar el promedio de la clase como un número de punto flotante. La línea 49 inicializa `contadorCalif` en 0, ya que todavía no se han introducido calificaciones. Recuerde que este programa utiliza la repetición controlada por centinela para recibir las calificaciones. Para mantener un registro preciso del número de calificaciones introducidas, el programa incrementa `contadorCalif` sólo cuando el usuario introduce una calificación válida.

Comparación entre la lógica del programa para la repetición controlada por centinela, y la repetición controlada por contador

Compare la lógica de esta aplicación para la repetición controlada por centinela con la repetición controlada por contador en la figura 4.6. En la repetición controlada por contador, cada iteración de la instrucción `while` (líneas 52 a 58 de la figura 4.6) lee un valor del usuario, para el número especificado de iteraciones. En la repetición controlada por centinela, el programa lee el primer valor (líneas 53 y 54 de la figura 4.9) antes de llegar al `while`. Este valor determina si el flujo de control del programa debe entrar al cuerpo del `while`. Si la condición del `while` es falsa, el usuario introdujo el valor centinela, por lo que el cuerpo del `while` no se ejecuta (es decir, no se introdujeron calificaciones). Si, por otro lado, la condición es verdadera, el cuerpo comienza a ejecutarse y el ciclo suma el valor de `calificacion` al `total` (línea 59). Después, las líneas 63 y 64 en el cuerpo del ciclo reciben el siguiente valor escrito por el usuario. A continuación, el control del programa se acerca a la llave derecha de terminación del cuerpo del ciclo en la línea 65, por lo que la ejecución continúa con la evaluación de la condición del `while` (línea 57). La condición utiliza el valor más reciente de calificacion que acaba de introducir el usuario, para determinar si el cuerpo del ciclo debe ejecutarse otra vez. El valor de la variable `calificacion` siempre lo introduce el usuario justo antes de que el programa evalúe la condición del `while`. Esto permite al programa determinar si el valor que acaba de introducir el usuario es el valor centinela, *antes* de que el programa procese ese valor (es decir, que lo sume al `total`). Si se introduce el valor centinela, el ciclo termina y el programa no suma –1 al `total`.

Buena práctica de programación 4.3

En un ciclo controlado por centinela, los indicadores deben recordar de manera explícita al usuario el valor que representa al centinela.

Una vez que termina el ciclo se ejecuta la instrucción `if...else` en las líneas 69 a 80. La condición en la línea 69 determina si se introdujeron calificaciones o no. Si no se introdujo ninguna, se ejecuta la parte del `else` (líneas 79 y 80) de la instrucción `if...else` que muestra el mensaje "`No se introdujeron calificaciones`", y el método devuelve el control al método que lo llamó.

Observe el bloque de la instrucción `while` en la figura 4.9 (líneas 58 a 65). Sin las llaves, el ciclo consideraría que su cuerpo sólo consiste en la primera instrucción, que suma la `calificacion` al `total`.

Las últimas tres instrucciones en el bloque quedarían fuera del cuerpo del ciclo, lo que ocasionaría que la computadora interpretara el código de manera incorrecta, como se muestra a continuación:

```
while ( calificacion != -1 )
   total = total + calificacion; // suma calificacion al total
contadorCalif = contadorCalif + 1; // incrementa el contador

// pide entrada y lee siguiente calificación del usuario
System.out.print( "Escriba calificacion o -1 para terminar: " );
calificacion = entrada.nextInt();
```

El código anterior ocasionaría un ciclo infinito en el programa si el usuario no introduce el centinela -1 como valor de entrada en la línea 54 (antes de la instrucción `while`).

Error común de programación 4.5

Omitir las llaves que delimitan a un bloque puede provocar errores lógicos, como ciclos infinitos. Para prevenir este problema, algunos programadores encierran el cuerpo de todas las instrucciones de control con llaves, aun si el cuerpo sólo contiene una instrucción.

Conversión explícita e implícita entre los tipos primitivos

Si se introdujo por lo menos una calificación, la línea 72 de la figura 4.9 calcula el promedio de las calificaciones. En la figura 4.6 vimos que la división entera produce un resultado entero. Aun y cuando la variable promedio se declara como `double` (línea 45), el cálculo

```
promedio = total / contadorCalif;
```

descarta la parte fraccionaria del cociente *antes* de asignar el resultado de la división a `promedio`. Esto ocurre debido a que `total` y `contadorCalif` son *ambos* enteros, y la división entera produce un resultado entero. Para realizar un cálculo de punto flotante con valores enteros, debemos tratar temporalmente a estos valores como números de punto flotante, para usarlos en el cálculo. Java cuenta con el **operador unario de conversión de tipo** para llevar a cabo esta tarea. La línea 72 utiliza el operador de conversión de tipo `(double)` (un operador unario) para crear una copia de punto flotante *temporal* de su operando `total` (que aparece a la derecha del operador). Utilizar un operador de conversión de tipo de esta forma es un proceso que se denomina **conversión explícita** o **conversión de tipos**. El valor almacenado en `total` sigue siendo un entero.

El cálculo ahora consiste de un valor de punto flotante (la versión temporal `double` de `total`) dividido entre el entero `contadorCalif`. Java sabe cómo evaluar sólo expresiones aritméticas en las que los tipos de los operandos sean *idénticos*. Para asegurar que los operandos sean del mismo tipo, Java realiza una operación llamada **promoción** (o **conversión implícita**) en los operandos seleccionados. Por ejemplo, en una expresión que contenga valores de los tipos `int` y `double`, los valores `int` son promovidos a valores `double` para utilizarlos en la expresión. En este ejemplo, Java promueve el valor de `contadorCalif` al tipo `double`, después el programa realiza la división de punto flotante y asigna el resultado del cálculo a `promedio`. Mientras que se aplique el operador de conversión de tipo `(double)` a cualquier variable en el cálculo, éste producirá un resultado `double`. Más adelante en el capítulo, hablaremos sobre todos los tipos primitivos. En la sección 6.7 aprenderá más acerca de las reglas de promoción.

Error común de programación 4.6

Un operador de conversión de tipo puede utilizarse para convertir entre los tipos numéricos primitivos, como `int` y `double`, y para convertir entre los tipos de referencia relacionados (como lo describiremos en el capítulo 10, Programación orientada a objetos: polimorfismo). La conversión al tipo incorrecto puede ocasionar errores de compilación o errores en tiempo de ejecución.

Un operador de conversión se forma colocando paréntesis alrededor del nombre de un tipo. Este operador es un **operador unario** (es decir, un operador que utiliza sólo un operando). Java también soporta las versiones unarias de los operadores de suma (+) y resta (-), por lo que usted puede escribir expresiones como -7 o +5. Los operadores de conversión de tipo se asocian de derecha a izquierda y tienen la misma precedencia que los demás operadores unarios, como + y -. Esta precedencia es un nivel mayor que la de los **operadores de multiplicación** *, / y %. (Consulte la tabla de precedencia de operadores en el apéndice A). En nuestras tablas de precedencia, indicamos el operador de conversión de tipos con la notación (*tipo*) para indicar que puede usarse cualquier nombre de tipo para formar un operador de conversión de tipo.

La línea 77 muestra en pantalla el promedio de la clase. En este ejemplo mostramos el promedio de la clase redondeado a la centésima más cercana. El especificador de formato `%.2f` en la cadena de control de formato de `printf` (línea 77) indica que el valor de la variable `promedio` debe mostrarse con dos dígitos de precisión a la derecha del punto decimal; esto se indica mediante el `.2` en el especificador de formato. Las tres calificaciones introducidas durante la ejecución de ejemplo de la clase `PruebaLibroCalificaciones` (figura 4.10) dan un total de 257, que produce el promedio de 85.666666.... El método `printf` utiliza la precisión en el especificador de formato para redondear el valor al número especificado de dígitos. En este programa, el promedio se redondea a la posición de las centésimas y se muestra como `85.67`.

```
1  // Fig. 4.10: PruebaLibroCalificaciones.java
2  // Crea un objeto LibroCalificaciones e invoca a su método determinarPromedioClase.
3
4  public class PruebaLibroCalificaciones
5  {
6     public static void main( String[] args )
7     {
8        // crea objeto miLibroCalificaciones de LibroCalificaciones y
9        // pasa el nombre del curso al constructor
10       LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
11          "CS101 Introduccion a la programacion en Java" );
12
13       miLibroCalificaciones.mostrarMensaje(); // muestra mensaje de bienvenida
14       miLibroCalificaciones.determinarPromedioClase(); // encuentra el promedio
                                                          de las calificaciones
15    } // fin de main
16 } // fin de la clase PruebaLibroCalificaciones
```

```
Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en Java!

Escriba calificacion o -1 para terminar: 97
Escriba calificacion o -1 para terminar: 88
Escriba calificacion o -1 para terminar: 72
Escriba calificacion o -1 para terminar: -1

El total de las 3 calificaciones introducidas es 257
El promedio de la clase es 85.67
```

Fig. 4.10 | La clase `PruebaLibroCalificaciones` crea un objeto de la clase `LibroCalificaciones` (figura 4.9) e invoca al método `determinarPromedioClase`.

4.10 Cómo formular algoritmos: instrucciones de control anidadas

En el siguiente ejemplo formularemos una vez más un algoritmo utilizando seudocódigo y el refinamiento de arriba a abajo, paso a paso, y después escribiremos el correspondiente programa en Java. Hemos

visto que las instrucciones de control pueden apilarse una encima de otra (en secuencia). En este caso de estudio examinaremos la otra forma estructurada en la que pueden conectarse las instrucciones de control, a saber, mediante el anidamiento de una instrucción de control dentro de otra.

Considere el siguiente enunciado de un problema:

> *Una universidad ofrece un curso que prepara a los estudiantes para el examen estatal de certificación del estado como corredores de bienes raíces. El año pasado, diez de los estudiantes que completaron este curso tomaron el examen. La universidad desea saber qué tan bien se desempeñaron sus estudiantes en el examen. A usted se le ha pedido que escriba un programa para sintetizar los resultados. Se le dio una lista de estos 10 estudiantes. Junto a cada nombre hay un 1 escrito, si el estudiante aprobó el examen, o un 2 si lo reprobó.*
>
> *Su programa debe analizar los resultados del examen de la siguiente manera:*
>
> 1. *Introducir cada resultado de la prueba (es decir, un 1 o un 2). Mostrar el mensaje "Escriba el resultado" en la pantalla, cada vez que el programa solicite otro resultado de la prueba.*
> 2. *Contar el número de resultados de la prueba, de cada tipo.*
> 3. *Mostrar un resumen de los resultados de la prueba, indicando el número de estudiantes que aprobaron y que reprobaron.*
> 4. *Si más de ocho estudiantes aprobaron el examen, imprimir el mensaje "Bono para el instructor!".*

Después de leer el enunciado del programa cuidadosamente, hacemos las siguientes observaciones:

1. El programa debe procesar los resultados de la prueba para 10 estudiantes. Puede usarse un ciclo controlado por contador, ya que el número de resultados de la prueba se conoce de antemano.
2. Cada resultado de la prueba tiene un valor numérico, ya sea 1 o 2. Cada vez que el programa lee un resultado de la prueba, debe determinar si el número es 1 o 2. Nosotros evaluamos un 1 en nuestro algoritmo. Si el número no es 1, suponemos que es un 2. (El ejercicio 4.24 considera las consecuencias de esta suposición).
3. Dos contadores se utilizan para llevar el registro de los resultados del examen: uno para contar el número de estudiantes que aprobaron el examen y otro para contar el número de estudiantes que reprobaron el examen.
4. Una vez que el programa ha procesado todos los resultados, debe decidir si más de ocho estudiantes aprobaron el examen.

Veamos ahora el refinamiento de arriba a abajo, paso a paso. Comencemos con la representación del seudocódigo de la cima:

```
Analizar los resultados del examen y decidir si debe pagarse un bono o no
```

Una vez más, la cima es una representación completa del programa, pero es probable que se necesiten varios refinamientos antes de que el seudocódigo pueda evolucionar de manera natural en un programa en Java.

Nuestro primer refinamiento es

```
Inicializar variables
Introducir las 10 calificaciones del examen y contar los aprobados y reprobados
Imprimir un resumen de los resultados del examen y decidir si debe pagarse un bono
```

Aquí también, aun cuando tenemos una representación *completa* del programa, es necesario refinarla. Ahora nos comprometemos con variables específicas. Se necesitan contadores para registrar los aprobados y reprobados, utilizaremos un contador para controlar el proceso de los ciclos y necesitaremos una variable para guardar la entrada del usuario. La variable en la que se almacenará la entrada del

usuario *no* se inicializa al principio del algoritmo, ya que su valor proviene del usuario durante cada iteración del ciclo.

La instrucción en seudocódigo

```
Inicializar variables
```

puede mejorarse de la siguiente manera:

```
Inicializar aprobados en cero
Inicializar reprobados en cero
Inicializar contador de estudiantes en cero
```

Observe que sólo se inicializan los contadores al principio del algoritmo.

La instrucción en seudocódigo

```
Introducir las 10 calificaciones del examen, y contar los aprobados y reprobados
```

requiere un ciclo en el que se introduzca de manera sucesiva el resultado de cada examen. Sabemos de antemano que hay precisamente 10 resultados del examen, por lo que es apropiado utilizar un ciclo controlado por contador. Dentro del ciclo (es decir, *anidado* dentro del ciclo), una estructura de selección doble determinará si cada resultado del examen es aprobado o reprobado, e incrementará el contador apropiado. Entonces, el refinamiento del seudocódigo anterior es

```
Mientras el contador de estudiantes sea menor o igual que 10
    Pedir al usuario que introduzca el siguiente resultado del examen
    Recibir como entrada el siguiente resultado del examen

    Si el estudiante aprobó
        Sumar uno a aprobados
    De lo contrario
        Sumar uno a reprobados

    Sumar uno al contador de estudiantes
```

Nosotros utilizamos líneas en blanco para aislar la estructura de control *Si...De lo contrario*, lo cual mejora la legibilidad.

La instrucción en seudocódigo

```
Imprimir un resumen de los resultados de los exámenes y decidir si debe pagarse un bono
```

puede mejorarse de la siguiente manera:

```
Imprimir el número de aprobados
Imprimir el número de reprobados

Si más de ocho estudiantes aprobaron
    Imprimir "¡Bono para el instructor!".
```

Segundo refinamiento completo en seudocódigo y conversión a la clase `Analisis`

El segundo refinamiento completo aparece en la figura 4.11. Observe que también se utilizan líneas en blanco para separar la estructura *Mientras* y mejorar la legibilidad del programa. Este seudocódigo está ahora lo bastante refinado para su conversión a Java.

La clase de Java que implementa el algoritmo en seudocódigo se muestra en la figura 4.12, junto con dos ejecuciones de ejemplo. Las líneas 13 a 16 de la clase `main` declaran las variables que utiliza el método `procesarResultadosExamen` de la clase `Analisis` para procesar los resultados del examen.

```
1  Inicializar aprobados en cero
2  Inicializar reprobados en cero
3  Inicializar contador de estudiantes en uno
4
5  Mientras el contador de estudiantes sea menor o igual que 10
6     Pedir al usuario que introduzca el siguiente resultado del examen
7     Recibir como entrada el siguiente resultado del examen
8
9     Si el estudiante aprobó
10       Sumar uno a aprobados
11    De lo contrario
12       Sumar uno a reprobados
13
14    Sumar uno al contador de estudiantes
15
16 Imprimir el número de aprobados
17 Imprimir el número de reprobados
18
19 Si más de ocho estudiantes aprobaron
20    Imprimir "Bono para el instructor!"
```

Fig. 4.11 | El seudocódigo para el problema de los resultados del examen.

Varias de estas declaraciones utilizan la habilidad de Java para incorporar la inicialización de variables en las declaraciones (a `aprobados` se le asigna `0`, a `reprobados` se le asigna `0` y a `contadorEstudiantes` se le asigna `1`). Los programas con ciclos pueden requerir de la inicialización al principio de cada repetición; por lo general, dicha reinicialización se realiza mediante instrucciones de asignación, en vez de hacerlo en las declaraciones.

Tip para prevenir errores 4.3

Inicializar las variables locales cuando se declaran ayuda al programador a evitar cualquier error de compilación que pudiera surgir, debido a los intentos por utilizar variables sin inicializar. Aunque Java no requiere que se incorporen las inicializaciones de variables locales en las declaraciones, sí requiere que se inicialicen antes de utilizar sus valores en una expresión.

La instrucción `while` (líneas 19 a 33) itera 10 veces. Durante cada iteración, el ciclo recibe y procesa un resultado del examen. Observe que la instrucción `if...else` (líneas 26 a 29) para procesar cada resultado se anida en la instrucción `while`. Si `resultado` es `1`, la instrucción `if...else` incrementa a `aprobados`; en caso contrario, asume que `resultado` es `2` e incrementa `reprobados`. La línea 32 incrementa `contadorEstudiantes` antes de que se evalúe otra vez la condición del ciclo, en la línea 19. Después de introducir 10 valores, el ciclo termina y la línea 36 muestra el número de `aprobados` y de `reprobados`. La instrucción `if` de las líneas 39 a 40 determina si más de ocho estudiantes aprobaron el examen y, de ser así, imprime el mensaje "`Bono para el instructor!`".

```
1 // Fig. 4.12: Analisis.java
2 // Analisis de los resultados de un examen, utilizando instrucciones de control anidadas.
3 import java.util.Scanner; // esta clase utiliza la clase Scanner
```

Fig. 4.12 | Análisis de los resultados del examen mediante el uso de estructuras de control anidadas (parte 1 de 3).

```
4
5  public class Analisis
6  {
7     public static void main( String[] args )
8     {
9        // crea objeto Scanner para obtener la entrada de la ventana de comandos
10       Scanner entrada = new Scanner( System.in );
11
12       // inicialización de las variables en declaraciones
13       int aprobados = 0; // número de aprobados
14       int reprobados = 0; // número de reprobados
15       int contadorEstudiantes = 1; // contador de estudiantes
16       int resultado; // un resultado del examen (obtiene el valor del usuario)
17
18       // procesa 10 estudiantes, usando ciclo controlado por contador
19       while ( contadorEstudiantes <= 10 )
20       {
21          // pide al usuario la entrada y obtiene el valor
22          System.out.print( "Escriba el resultado (1 = aprobado, 2 = reprobado): " );
23          resultado = entrada.nextInt();
24
25          // if...else anidado en la instrucción while
26          if ( resultado == 1 )           // si resultado es 1,
27             aprobados = aprobados + 1;   // incrementa aprobados;
28          else                            // de lo contrario, resultado no es 1, por lo que
29             reprobados = reprobados + 1; // incrementa reprobados
30
31          // incrementa contadorEstudiantes, para que el ciclo termine en un momento dado
32          contadorEstudiantes = contadorEstudiantes + 1;
33       } // fin de while
34
35       // fase de terminación; prepara y muestra los resultados
36       System.out.printf( "Aprobados: %d\nReprobados: %d\n", aprobados, reprobados );
37
38       // determina si más de 8 estudiantes aprobaron
39       if ( aprobados > 8 )
40          System.out.println( "Bono para el instructor!" );
41    } // fin de main
42 } // fin de la clase Analisis
```

```
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Aprobados: 9
Reprobados: 1
Bono para el instructor!
```

Fig. 4.12 | Análisis de los resultados del examen mediante el uso de estructuras de control anidadas (parte 2 de 3).

```
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Aprobados: 6
Reprobados: 4
```

Fig. 4.12 | Análisis de los resultados del examen mediante el uso de estructuras de control anidadas (parte 3 de 3).

La figura 4.12 muestra la entrada y salida de dos ejecuciones de ejemplo del programa. Durante la primera ejecución de ejemplo, la condición en la línea 39 del método `main` es `true`; más de ocho estudiantes aprobaron el examen, por lo que el programa imprime un mensaje indicando que se debe dar un bono al instructor.

Este ejemplo contiene sólo una clase; el método `main` realiza todo el trabajo. En este capítulo y en el 3 ha visto ejemplos que consisten en dos clases: una contiene los métodos que realizan tareas útiles y la otra tiene el método `main`, que crea un objeto de la otra clase y llama a sus métodos. En ocasiones, cuando no tenga sentido tratar de crear una clase reutilizable para demostrar un concepto, colocaremos todas las instrucciones del programa dentro del método `main` de una sola clase.

4.11 Operadores de asignación compuestos

Los **operadores de asignación compuestos** abrevian las expresiones de asignación. Cualquier instrucción de la forma

```
variable = variable operador expresión;
```

en donde *operador* es uno de los operadores binarios +, -, *, / o % (o alguno de los otros que veremos más adelante en el libro), puede escribirse de la siguiente forma:

```
variable operador= expresión;
```

Por ejemplo, puede abreviar la instrucción

```
c = c + 3;
```

mediante el **operador de asignación compuesto de suma**, +=, de la siguiente manera:

```
c += 3;
```

El operador += suma el valor de la expresión que está a la derecha del operador, al valor de la variable ubicada a la izquierda del operador, y almacena el resultado en la variable que se encuentra a la izquierda del operador. Por lo tanto, la expresión de asignación `c += 3` suma 3 a `c`. La figura 4.13 muestra los operadores de asignación aritméticos compuestos, algunas expresiones de ejemplo en las que se utilizan los operadores y las explicaciones de lo que estos operadores hacen.

4.12 Operadores de incremento y decremento

Java proporciona dos operadores unarios (que sintetizamos en la figura 4.14) para sumar 1, o restar 1, al valor de una variable numérica. Estos operadores son el **operador de incremento** unario, ++, y el **operador de decremento** unario, --. Un programa puede incrementar en 1 el valor de una variable

Operador de asignación	Expresión de ejemplo	Explicación	Asigna
Suponer que: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 a c
`-=`	`d -= 4`	`d = d - 4`	1 a d
`*=`	`e *= 5`	`e = e * 5`	20 a e
`/=`	`f /= 3`	`f = f / 3`	2 a f
`%=`	`g %= 9`	`g = g % 9`	3 a g

Fig. 4.13 | Operadores de asignación aritméticos compuestos.

llamada c mediante el operador de incremento, ++, en lugar de usar la expresión `c = c + 1` o `c+= 1`. A un operador de incremento o decremento que se coloca antes de una variable se le llama **operador de preincremento** o **predecremento**, respectivamente. A un operador de incremento o decremento que se coloca después de una variable se le llama **operador de postincremento** o **postdecremento**, correspondientes.

Operador	Nombre del operador	Expresión de ejemplo	Explicación
`++`	Preincremento	`++a`	Incrementar a en 1, después utilizar el nuevo valor de a en la expresión en que esta variable reside.
`++`	Postincremento	`a++`	Usar el valor actual de a en la expresión en la que esta variable reside, después incrementar a en 1.
`--`	Predecremento	`--b`	Decrementar b en 1, después utilizar el nuevo valor de b en la expresión en que esta variable reside.
`--`	Postdecremento	`b--`	Usar el valor actual de b en la expresión en la que esta variable reside, después decrementar b en 1.

Fig. 4.14 | Los operadores de incremento y decremento.

Al proceso de utilizar el operador de preincremento (o postdecremento) para sumar (o restar) 1 a una variable, se le conoce como **preincrementar** (o **predecrementar**) la variable. Esto hace que la variable se incremente (o decremente) en 1; después el nuevo valor de la variable se utiliza en la expresión en la que aparece. Al proceso de utilizar el operador de preincremento (o postdecremento) para sumar (o restar) 1 a una variable, se le conoce como **postincrementar** (o **postdecrementar**) la variable. Esto hace que el valor actual de ella se utilice en la expresión en la que aparece; después se incrementa (decrementa) el valor de la variable en 1.

Buena práctica de programación 4.4

A diferencia de los operadores binarios, los operadores unarios de incremento y decremento deben colocarse enseguida de sus operandos, sin espacios entre ellos.

La figura 4.15 demuestra la diferencia entre la versión de preincremento y la versión de predecremento del operador de incremento ++. El operador de decremento (--) funciona de manera similar.

```
1  // Fig. 4.15: Incremento.java
2  // Operadores de preincremento y postincremento.
3
4  public class Incremento
5  {
6     public static void main( String[] args )
7     {
8        int c;
9
10       // demuestra el operador de postincremento
11       c = 5; // asigna 5 a c
12       System.out.println( c );   // imprime 5
13       System.out.println( c++ ); // imprime 5, después postincrementa
14       System.out.println( c );   // imprime 6
15
16       System.out.println(); // suma de las calificaciones
17
18       // demuestra el operador de preincremento
19       c = 5; // asigna 5 a c
20       System.out.println( c );   // imprime 5
21       System.out.println( ++c ); // preincrementa y después imprime 6
22       System.out.println( c );   // imprime 6
23    } // fin de main
24 } // fin de la clase Incremento
```

```
5
5
6

5
6
6
```

Fig. 4.15 | Preincrementar y postincrementar.

La línea 11 inicializa la variable c con 5, y la línea 12 imprime el valor inicial de c. La línea 13 imprime el valor de la expresión c++. Esta expresión postincrementa la variable c, por lo que se imprime el valor original de c (5), y después el valor de c se incrementa (a 6). Por ende, la línea 13 imprime el valor inicial de c (5) otra vez. La línea 14 imprime el nuevo valor de c (6) para demostrar que, sin duda, se incrementó el valor de la variable en la línea 13.

La línea 19 restablece el valor de c a 5, y la línea 20 imprime el valor de c. La línea 21 imprime el valor de la expresión ++c. Esta expresión preincrementa a c, por lo que su valor se incrementa y después se imprime el nuevo valor (6). La línea 22 imprime el valor de c otra vez, para mostrar que sigue siendo 6 después de que se ejecuta la línea 21.

Los operadores de asignación compuestos aritméticos y los operadores de incremento y decremento pueden utilizarse para simplificar las instrucciones de los programas. Por ejemplo, las tres instrucciones de asignación de la figura 4.12 (líneas 27, 29 y 32)

```
aprobados = aprobados + 1;
reprobados = reprobados + 1;
contadorEstudiantes = contadorEstudiantes + 1;
```

pueden escribirse en forma más concisa con operadores de asignación compuestos, de la siguiente manera:

```
aprobados += 1;
reprobados += 1;
contadorEstudiantes += 1;
```

con operadores de preincremento de la siguiente forma:

```
++aprobados;
++reprobados;
++contadorEstudiantes;
```

o con operadores de postincremento de la siguiente forma:

```
aprobados++;
reprobados++;
contadorEstudiantes++;
```

Al incrementar o decrementar una variable que se encuentre en una instrucción por sí sola, las formas preincremento y postincremento tienen el mismo efecto, al igual que las formas predecremento y postdecremento. Solamente cuando una variable aparece en el contexto de una expresión más grande es cuando los operadores preincremento y postdecremento tienen distintos efectos (y lo mismo se aplica a los operadores de predecremento y postdecremento).

Error común de programación 4.7

Tratar de usar el operador de incremento o decremento en una expresión a la que no se le pueda asignar un valor es un error de sintaxis. Por ejemplo, escribir ++(x + 1) es un error de sintaxis, ya que (x + 1) no es una variable.

La figura 4.16 muestra la precedencia y la asociatividad de los operadores que presentamos. Los operadores se muestran de arriba a abajo, en orden descendente de precedencia. La segunda columna describe la asociatividad de los operadores en cada nivel de precedencia. El operador condicional (`?:`), los operadores unarios de incremento (`++`), decremento (`--`), suma (`+`) y resta (`-`), los operadores de conversión de tipo y los operadores de asignación `=`, `+=`, `-=`, `*=`, `/=` y `%=` se asocian de derecha a izquierda. Todos los demás operadores en la tabla de precedencia de operadores de la figura 4.16 se asocian de izquierda a derecha. La tercera columna enumera el tipo de cada grupo de operadores.

Operadores						Asociatividad	Tipo
`++`	`--`					derecha a izquierda	postfijo unario
`++`	`--`	`+`	`-`	`(` *tipo* `)`		derecha a izquierda	prefijo unario
`*`	`/`	`%`				izquierda a derecha	multiplicativo
`+`	`-`					izquierda a derecha	aditivo
`<`	`<=`	`>`	`>=`			izquierda a derecha	relacional
`==`	`!=`					izquierda a derecha	igualdad
`?:`						derecha a izquierda	condicional
`=`	`+=`	`-=`	`*=`	`/=`	`%=`	derecha a izquierda	asignación

Fig. 4.16 | Precedencia y asociatividad de los operadores vistos hasta ahora.

4.13 Tipos primitivos

La tabla del apéndice D enumera los ocho tipos primitivos en Java. Al igual que sus lenguajes antecesores C y C++, Java requiere que todas las variables tengan un tipo. Es por esta razón que Java se conoce como un **lenguaje fuertemente tipificado**.

En C y C++, los programadores tienen que escribir con frecuencia versiones independientes de los programas para soportar varias plataformas distintas, ya que no se garantiza que los tipos primitivos sean idénticos de computadora en computadora. Por ejemplo, un valor `int` en un equipo podría representarse mediante 16 bits (2 bytes) de memoria, en un segundo equipo mediante 32 bits (4 bytes) de memoria, y en otro mediante 64 bits (8 bytes) de memoria. En Java, los valores `int` siempre son de 32 bits (4 bytes).

Tip de portabilidad 4.1

Los tipos primitivos en Java son portables en todas las plataformas con soporte para Java.

Cada uno de los tipos del apéndice D se enumera con su tamaño en bits (hay ocho bits en un byte) y su rango de valores. Como los diseñadores de Java desean asegurar la portabilidad, utilizan estándares reconocidos a nivel internacional, tanto para los formatos de caracteres (Unicode; para más información, visite `www.unicode.org`) como para los números de punto flotante (IEEE 754; para más información, visite `grouper.ieee.org/groups/754/`).

En la sección 3.4 vimos que a las variables de tipos primitivos que se declaran fuera de un método, como campos de una clase, se les asignan valores predeterminados, a menos que se inicialicen en forma explícita. Las variables de instancia de los tipos `char`, `byte`, `short`, `int`, `long`, `float` y `double` reciben el valor `0` de manera predeterminada. Las variables de tipo `boolean` reciben el valor `false` de manera predeterminada. Las variables de instancia de tipo por referencia se inicializan de manera predeterminada con el valor `null`.

4.14 (Opcional) Caso de estudio de GUI y gráficos: creación de dibujos simples

Una característica atractiva de Java es su soporte para gráficos, el cual permite a los programadores mejorar sus aplicaciones en forma visual. Ahora le presentaremos una de las capacidades gráficas de Java: dibujar líneas. También cubriremos los aspectos básicos acerca de cómo crear una ventana para mostrar un dibujo en la pantalla de la computadora.

El sistema de coordenadas de Java

Para dibujar en Java, debe comprender su **sistema de coordenadas** (figura 4.17), un esquema para identificar puntos en la pantalla. De manera predeterminada, la esquina superior izquierda de un componente

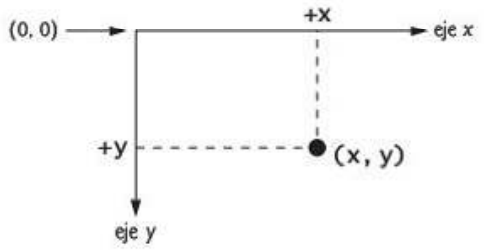


Fig. 4.17 | Sistema de coordenadas de Java. Las unidades están en pixeles.

de la GUI tiene las coordenadas (0, 0). Un par de coordenadas está compuesto por una **coordenada *x*** (la coordenada horizontal) y una **coordenada *y*** (la coordenada vertical). La coordenada *x* es la ubicación horizontal que se desplaza de izquierda a derecha. La coordenada *y* es la ubicación vertical que se desplaza de arriba hacia abajo. El **eje *x*** describe cada una de las coordenadas horizontales, y el **eje *y*** cada una de las coordenadas verticales.

Las coordenadas indican en dónde deben mostrarse los gráficos en una pantalla. Las unidades de las coordenadas se miden en **píxeles**. El término píxel significa "elemento de imagen". Un píxel es la unidad de resolución más pequeña de una pantalla.

Primera aplicación de dibujo

Nuestra primera aplicación de dibujo simplemente dibuja dos líneas. La clase `PanelDibujo` (figura 4.18) realiza el dibujo en sí, mientras que la clase `PruebaPanelDibujo` (figura 4.19) crea una ventana para mostrar el dibujo. En la clase `PanelDibujo`, las instrucciones `import` de las líneas 3 y 4 nos permiten utilizar la clase **`Graphics`** (del paquete `java.awt`), que proporciona varios métodos para dibujar texto y figuras en la pantalla, y la clase **`JPanel`** (del paquete `javax.swing`), que proporciona un área en la que podemos dibujar.

```
1  // Fig. 4.18: PanelDibujo.java
2  // Uso de drawLine para conectar las esquinas de un panel.
3  import java.awt.Graphics;
4  import javax.swing.JPanel;
5
6  public class PanelDibujo extends JPanel
7  {
8     // dibuja una x desde las esquinas del panel
9     public void paintComponent( Graphics g )
10    {
11       // llama a paintComponent para asegurar que el panel se muestre correctamente
12       super.paintComponent( g );
13
14       int anchura = getWidth(); // anchura total
15       int altura = getHeight(); // altura total
16
17       // dibuja una línea de la esquina superior izquierda a la esquina inferior derecha
18       g.drawLine( 0, 0, anchura, altura );
19
20       // dibuja una línea de la esquina inferior izquierda a la esquina superior derecha
21       g.drawLine( 0, altura, anchura, 0 );
22    } // fin del método paintComponent
23 } // fin de la clase PanelDibujo
```

Fig. 4.18 | Uso de `drawLine` para conectar las esquinas de un panel.

```
1  // Fig. 4.19: PruebaPanelDibujo.java
2  // Aplicación que muestra un PanelDibujo.
3  import javax.swing.JFrame;
4
5  public class PruebaPanelDibujo
6  {
```

Fig. 4.19 | Creación de un objeto `JFrame` para mostrar un objeto `PanelDibujo` (parte 1 de 2).

```
7     public static void main( String[] args )
8     {
9        // crea un panel que contiene nuestro dibujo
10       PanelDibujo panel = new PanelDibujo();
11
12       // crea un nuevo marco para contener el panel
13       JFrame aplicacion = new JFrame();
14
15       // establece el marco para salir cuando se cierre
16       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
17
18       aplicacion.add( panel ); // agrega el panel al marco
19       aplicacion.setSize( 250, 250 ); // establece el tamaño del marco
20       aplicacion.setVisible( true ); // hace que el marco sea visible
21    } // fin de main
22 } // fin de la clase PruebaPanelDibujo
```

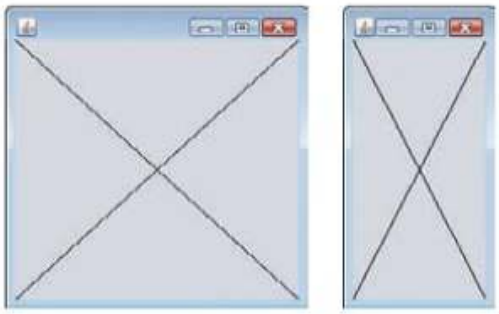


Fig. 4.19 | Creación de un objeto `JFrame` para mostrar un objeto `PanelDibujo` (parte 2 de 2).

La línea 6 utiliza la palabra clave **extends** para indicar que la clase `PanelDibujo` es un tipo mejorado de `JPanel`. La palabra clave `extends` representa algo que se denomina relación de herencia, en la cual nuestra nueva clase `PanelDibujo` empieza con los miembros existentes (datos y métodos) de la clase `JPanel`. La clase de la cual `PanelDibujo` **hereda**, `JPanel`, aparece a la derecha de la palabra clave `extends`. En esta relación de herencia, a `JPanel` se le conoce como la **superclase** y `PanelDibujo` es la **subclase**. Esto produce una clase `PanelDibujo` que tiene los atributos (datos) y comportamientos (métodos) de la clase `JPanel`, así como las nuevas características que agregaremos en nuestra declaración de la clase `PanelDibujo`; en específico, la habilidad de dibujar dos líneas a lo largo de las diagonales del panel. En el capítulo 9 explicaremos con detalle el concepto de herencia. Por ahora, sólo tiene que imitar nuestra clase `PanelDibujo` cuando cree sus propios programas de gráficos.

El método `paintComponent`

Todo `JPanel`, incluyendo nuestro `PanelDibujo`, tiene un método **`paintComponent`** (líneas 9 a 22), que el sistema llama de manera automática cada vez que necesita mostrar el objeto `JPanel`. El método `paintComponent` debe declararse como se muestra en la línea 9; de no ser así, el sistema no llamará al método. Este método se llama cuando se muestra un objeto `JPanel` por primera vez en la pantalla, cuando una ventana en la pantalla lo cubre y después lo descubre, y cuando la ventana en la que aparece cambia su tamaño. El método `paintComponent` requiere un argumento, un objeto `Graphics`, que el sistema proporciona por usted cuando llama a `paintComponent`.

La primera instrucción en cualquier método `paintComponent` que cree debe ser siempre:

```
super.paintComponent( g );
```

la cual asegura que el panel se despliegue de manera apropiada en la pantalla, antes de empezar a dibujar en él. A continuación, las líneas 14 y 15 llaman a los métodos que la clase `PanelDibujo` hereda de la clase `JPanel`. Como `PanelDibujo` extiende a `JPanel`, `PanelDibujo` puede usar cualquier método `public` de `JPanel`. Los métodos `getWidth` y `getHeight` devuelven la anchura y la altura del objeto `JPanel`, respectivamente. Las líneas 14 y 15 almacenan estos valores en las variables locales `anchura` y `altura`. Por último, las líneas 18 y 21 utilizan la variable `g` de la clase `Graphics` para llamar al método `drawLine`, y que dibuje las dos líneas. El método `drawLine` dibuja una línea entre dos puntos representados por sus cuatro argumentos. Los primeros dos son las coordenadas `x` y `y` para uno de los puntos finales de la línea, y los últimos dos son las coordenadas para el otro punto final. Si cambia de tamaño la ventana, las líneas se escalaran de manera acorde, ya que los argumentos se basan en la anchura y la altura del panel. Al cambiar el tamaño de la ventana en esta aplicación, el sistema llama a `paintComponent` para volver a dibujar el contenido de `PanelDibujo`.

La clase `PruebaPanelDibujo`

Para mostrar el `PanelDibujo` en la pantalla, debemos colocarlo en una ventana. Usted debe crear una ventana con un objeto de la clase `JFrame`. En `PruebaPanelDibujo.java` (figura 4.19), la línea 3 importa la clase `JFrame` del paquete `javax.swing`. La línea 10 en `main` crea un objeto `PanelDibujo`, el cual contiene nuestro dibujo, y la línea 13 crea un nuevo objeto `JFrame` que puede contener y mostrar nuestro panel. La línea 16 llama al método `setDefaultCloseOperation` de `JFrame` con el argumento `JFrame.EXIT_ON_CLOSE`, para indicar que la aplicación debe terminar cuando el usuario cierre la ventana. La línea 18 utiliza el **método `add`** de `JFrame` para adjuntar el objeto `PanelDibujo` al objeto `JFrame`. La línea 19 establece el tamaño del objeto `JFrame`. El método `setSize` recibe dos parámetros que representan la anchura y altura del objeto `JFrame`, respectivamente. Por último, la línea 20 muestra el objeto `JFrame` mediante una llamada a su **método `setVisible`** con el argumento `true`. Cuando se muestra el objeto `JFrame`, se hace una llamada implícita al método `paintComponent` de `PanelDibujo` (líneas 9 a 22 de la figura 4.18) y se dibujan las dos líneas (vea los resultados de ejemplo en la figura 4.19). Pruebe a cambiar el tamaño de la ventana, y podrá ver que las líneas siempre se dibujan con base en la anchura y altura actuales de la ventana.

Ejercicios del caso de estudio de GUI y gráficos

4.1 Utilizar ciclos e instrucciones de control para dibujar líneas puede producir muchos diseños interesantes.

a) Cree el diseño que se muestra en la captura de pantalla izquierda de la figura 4.20. Este diseño dibuja líneas que parten desde la esquina superior izquierda, y se despliegan hasta cubrir la mitad superior izquierda del panel. Un método es dividir la anchura y la altura en un número equivalente de pasos (nosotros descubrimos que 15 pasos es una buena cantidad). El primer punto final de una línea siempre estará en la esquina superior izquierda (0,0). El segundo punto final puede encontrarse partiendo desde la esquina inferior izquierda, y avanzando un paso vertical hacia arriba, y uno horizontal hacia la derecha. Dibuje una línea entre los dos puntos finales. Continúe avanzando un paso hacia arriba y a la derecha, para encontrar cada punto final sucesivo. La figura deberá escalarse de manera apropiada a medida que usted cambie el tamaño de la ventana.

b) Modifique su respuesta en la parte (a) para hacer que las líneas se desplieguen a partir de las cuatro esquinas, como se muestra en la captura de pantalla derecha de la figura 4.20. Las líneas de esquinas opuestas deberán intersecarse a lo largo de la parte media.

4.2 La figura 4.21 muestra dos diseños adicionales, creados mediante el uso de ciclos `while` y de `drawLine`.

a) Cree el diseño de la captura de pantalla izquierda de la figura 4.21. Empiece por dividir cada flanco en un número equivalente de incrementos (elegimos 15 de nuevo). La primera línea empieza en la esquina superior izquierda y termina un paso a la derecha, en el flanco inferior. Para cada línea sucesiva, avance hacia abajo un incremento en el flanco izquierdo, y un incremento a la derecha en el flanco inferior. Continúe dibujando líneas hasta llegar a la esquina inferior derecha. La figura deberá escalarse a medida que usted cambie el tamaño de la ventana, de manera que los puntos finales siempre toquen los flancos.

b) Modifique su respuesta en la parte (a) para reflejar el diseño en las cuatro esquinas, como se muestra en la captura de pantalla derecha de la figura 4.21.

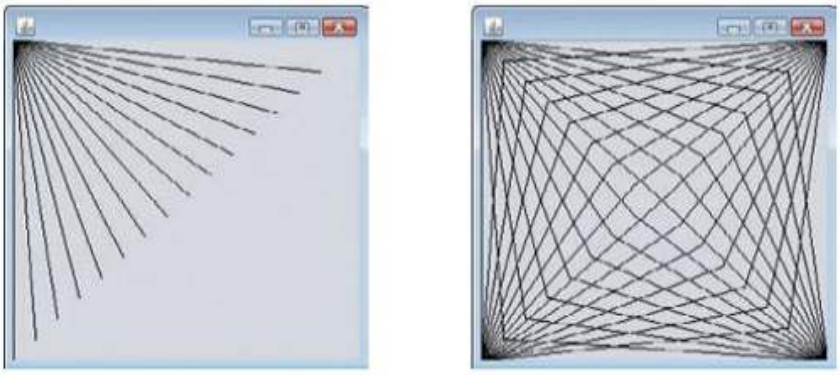


Fig. 4.20 | Despliegue de líneas desde una esquina.

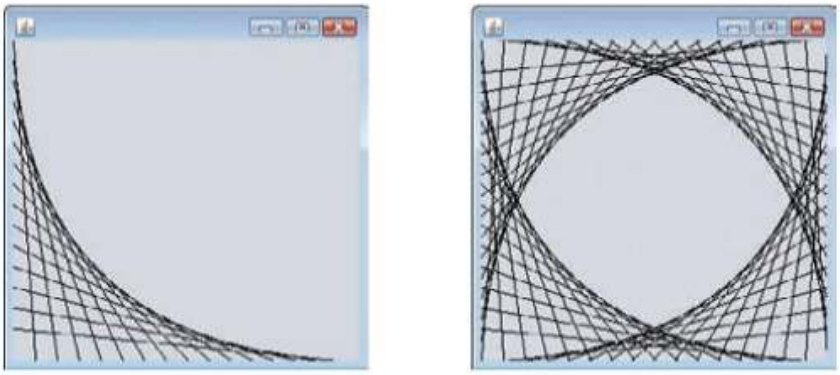


Fig. 4.21 | Arte lineal con ciclos y `drawLine`.

4.15 Conclusión

Este capítulo presentó las estrategias básicas de solución de problemas para crear clases y desarrollar métodos para ellas. Demostramos cómo construir un algoritmo (es decir, una metodología para resolver un problema), y después cómo refinarlo a través de diversas fases de desarrollo de seudocódigo, lo cual produce código en Java que puede ejecutarse como parte de un método. El capítulo demostró cómo utilizar el método de refinamiento de arriba a abajo, paso a paso, para planear las acciones específicas que debe realizar un método y el orden en el que debe hacerlo.

Sólo se requieren tres tipos de estructuras de control (secuencia, selección y repetición) para desarrollar cualquier algoritmo para solucionar un problema. En específico, este capítulo demostró el uso de la instrucción de selección simple `if`, la instrucción de selección doble `if...else` y la instrucción de repetición `while`. Estas instrucciones son algunos de los bloques de construcción que se utilizan para construir soluciones para muchos problemas. Utilizamos el apilamiento de instrucciones de control para calcular el total y el promedio de un conjunto de calificaciones de estudiantes, mediante la repetición controlada por un contador y controlada por un centinela, y utilizamos el anidamiento de instrucciones de control para analizar y tomar decisiones con base en un conjunto de resultados de un examen. Presentamos los operadores de asignación compuestos de Java, así como sus operadores de incremento y decremento. Por último, analizamos los tipos primitivos de Java. En el capítulo 5 continuaremos nuestro debate acerca de las instrucciones de control, en donde presentaremos las instrucciones `for`, `do...while` y `switch`.

Resumen

Sección 4.1 Introducción

- Antes de escribir un programa para resolver un problema, debe tener una comprensión detallada de él y una metodología cuidadosamente planeada para resolverlo. También debe comprender los bloques de construcción disponibles, y emplear las técnicas probadas para construir programas.

Sección 4.2 Algoritmos

- Cualquier problema de cómputo puede resolverse mediante la ejecución de una serie de acciones (pág. 103), en un orden específico.
- Un procedimiento para resolver un problema, en términos de las acciones a ejecutar y el orden en el que se ejecutan, se denomina algoritmo (pág. 103).
- El proceso de especificar el orden en el que se ejecutan las instrucciones en un programa se denomina control del programa (pág. 104).

Sección 4.3 Seudocódigo

- El seudocódigo (pág. 104) es un lenguaje informal, que ayuda a los programadores a desarrollar algoritmos sin tener que preocuparse por los estrictos detalles de la sintaxis del lenguaje Java.
- El seudocódigo es similar al lenguaje cotidiano; es conveniente y amigable para el usuario, pero no es un verdadero lenguaje de programación de computadoras.
- El seudocódigo ayuda al programador a "idear" un programa antes de intentar escribirlo en un lenguaje de programación, como Java.
- El seudocódigo cuidadosamente preparado puede convertirse con facilidad en su correspondiente programa en Java.

Sección 4.4 Estructuras de control

- Por lo general, las instrucciones en un programa se ejecutan, una después de la otra, en el orden en el que están escritas. A este proceso se le conoce como ejecución secuencial (pág. 104).
- Varias instrucciones de Java permiten al programador especificar que la siguiente instrucción a ejecutar no es necesariamente la siguiente en la secuencia. A esto se le conoce como transferencia de control (pág. 104).
- Bohm y Jacopini demostraron que todos los programas podían escribirse en términos de sólo tres estructuras de control (pág. 105): la estructura de secuencia, la estructura de selección y la estructura de repetición.
- El término "estructuras de control" proviene del campo de las ciencias computacionales. La *Especificación del lenguaje Java* se refiere a las "estructuras de control" como "instrucciones de control" (pág. 104).
- La estructura de secuencia está integrada en Java. A menos que se indique lo contrario, la computadora ejecuta las instrucciones de Java, una después de la otra, en el orden en el que están escritas; es decir, en secuencia.
- En cualquier parte en donde pueda colocarse una sola acción, pueden colocarse varias en secuencia.
- Los diagramas de actividad (pág. 105) forman parte de UML. Un diagrama de actividad modela el flujo de trabajo (pág. 105; también conocido como la actividad) de una parte de un sistema de software.
- Los diagramas de actividad se componen de símbolos (pág. 105) —como los símbolos de estados de acción, rombos y pequeños círculos— que se conectan mediante flechas de transición, las cuales representan el flujo de la actividad.
- Los estados de acción (pág. 105) contienen expresiones de acción que especifican las acciones específicas a realizar.
- Las flechas en un diagrama de actividad representan las transiciones, que indican el orden en el que ocurren las acciones representadas por los estados de acción.
- El círculo relleno que se encuentra en la parte superior de un diagrama de actividad representa el estado inicial de la actividad (pág. 105): el comienzo del flujo de trabajo antes de que el programa realice las acciones modeladas.
- El círculo sólido rodeado por una circunferencia, que aparece en la parte inferior del diagrama, representa el estado final (pág. 106): el término del flujo de trabajo después de que el programa realiza sus acciones.
- Los rectángulos con las esquinas superiores derechas dobladas se llaman notas en UML (pág. 106): comentarios aclaratorios que describen el propósito de los símbolos en el diagrama.

- Java tiene tres tipos de instrucciones de selección (pág. 106).
- La instrucción `if` de selección simple (pág. 106) selecciona o ignora una o más acciones.
- La instrucción `if...else` de selección doble selecciona una de dos acciones distintas, o grupos de acciones.
- La instrucción `switch` se llama instrucción de selección múltiple (pág. 106), ya que selecciona una de varias acciones distintas, o grupos de acciones.
- Java cuenta con las instrucciones de repetición (ciclos) `while`, `do...while` y `for`, las cuales permiten a los programas ejecutar instrucciones en forma repetida, siempre y cuando una condición de continuación de ciclo siga siendo verdadera.
- Las instrucciones `while` y `for` realizan la(s) acción(es) en sus cuerpos, cero o más veces; si al principio la condición de continuación de ciclo (pág. 106) es falsa, la(s) acción(es) no se ejecutará(n). La instrucción `do...while` lleva a cabo la(s) acción(es) que contiene en su cuerpo, una o más veces.
- Las palabras `if`, `else`, `switch`, `while`, `do` y `for` son palabras claves en Java. Las palabras clave no pueden utilizarse como identificadores, como los nombres de variables.
- Cada programa se forma mediante una combinación de todas las instrucciones de secuencia, selección y repetición (pág. 106) que sean apropiadas para el algoritmo que implementa ese programa.
- Las instrucciones de control de una sola entrada/una sola salida (pág. 106) se unen unas a otras mediante la conexión del punto de salida de una al punto de entrada de la siguiente. A esto se le conoce como apilamiento de instrucciones de control.
- Una instrucción de control también se puede anidar (pág. 106) dentro de otra instrucción de control.

Sección 4.5 Instrucción `if` de selección simple

- Los programas utilizan instrucciones de selección para elegir entre los cursos alternativos de acción.
- El diagrama de actividad de una instrucción `if` de selección simple contiene el símbolo de rombo, el cual indica que se tomará una decisión. El flujo de trabajo continuará a lo largo de una ruta determinada por las condiciones de guardia asociadas al símbolo (pág. 107). Si una condición de guardia es verdadera, el flujo de trabajo entra al estado de acción al que apunta la flecha de transición correspondiente.
- La instrucción `if` es una instrucción de control de una sola entrada/una sola salida.

Sección 4.6 Instrucción `if...else` de selección doble

- La instrucción `if` de selección simple realiza una acción indicada sólo cuando la condición es verdadera.
- La instrucción `if...else` de selección doble (pág. 106) realiza una acción cuando la condición es verdadera, y otra acción distinta cuando es falsa.
- El operador condicional (pág. 108, `?:`) es el único operador ternario de Java; recibe tres operandos. En conjunto, los operandos y el símbolo `?:` forman una expresión condicional (pág. 108).
- Un programa puede evaluar varios casos mediante instrucciones `if...else` anidadas (pág. 109).
- El compilador de Java asocia un `else` con el `if` que lo precede inmediatamente, a menos que se le indique otra cosa mediante la colocación de llaves.
- La instrucción `if` espera una instrucción en su cuerpo. Para incluir varias instrucciones en el cuerpo de un `if` (o en el cuerpo de un `else` para una instrucción `if...else`), encierre las instrucciones entre llaves.
- Un bloque (pág. 111) de instrucciones puede colocarse en cualquier parte en donde se pueda colocar una sola instrucción.
- Un error lógico (pág. 111) tiene su efecto en tiempo de ejecución. Un error lógico fatal (pág.111) hace que un programa falle y termine antes de tiempo. Un error lógico no fatal (pág.111) permite que un programa continúe ejecutándose, pero hace que el programa produzca resultados erróneos.
- Así como podemos colocar un bloque en cualquier parte en la que pueda colocarse una sola instrucción, también podemos usar una instrucción vacía, que se representa colocando un punto y coma (;) en donde normalmente estaría una instrucción.

Sección 4.7 Instrucción de repetición `while`

- La instrucción de repetición `while` (pág. 112) permite al programador especificar que un programa debe repetir una acción, mientras cierta condición siga siendo verdadera.

- El símbolo de fusión (pág. 112) de UML combina dos flujos de actividad en uno.
- Los símbolos de decisión y de fusión pueden diferenciarse con base en el número de flechas de transición "entrantes" y "salientes". Un símbolo de decisión tiene (pág. 107) una flecha de transición que apunta hacia el rombo, y dos o más que apuntan hacia fuera de él, para indicar las posibles transiciones desde ese punto. Cada flecha de transición que apunta hacia fuera de un símbolo de decisión tiene una condición de guardia. Un símbolo de fusión tiene dos o más flechas de transición que apuntan hacia el rombo, y sólo una que apunta hacia fuera de éste, para indicar que se fusionarán varios flujos de actividad para continuar con la actividad. Ninguna de las flechas de transición asociadas con un símbolo de fusión tiene una condición de guardia.

Sección 4.8 Cómo formular algoritmos: repetición controlada por un contador

- La repetición controlada por un contador (pág. 113) utiliza una variable llamada contador (o variable de control), para controlar el número de veces que se ejecuta un conjunto de instrucciones.
- A la repetición controlada por contador se le conoce comúnmente como repetición definida (pág. 113), ya que el número de repeticiones se conoce desde antes que empiece a ejecutarse el ciclo.
- Un total (pág. 114) es una variable que se utiliza para acumular la suma de varios valores. Por lo general, las variables que se utilizan para almacenar totales se inicializan en cero antes de usarlas en un programa.
- La declaración de una variable local debe aparecer antes de usarla en el método en el que está declarada. Una variable local no puede utilizarse fuera del método en el que se declaró.
- Al dividir dos enteros se produce una división entera (pág. 118); la parte fraccionaria del cálculo se trunca.

Sección 4.9 Cómo formular algoritmos: repetición controlada por un centinela

- En la repetición controlada por un centinela (pág. 118) se utiliza un valor especial, conocido como valor centinela (también llamado valor de señal, valor de prueba o valor de bandera) para indicar el "fin de la entrada de datos".
- Debe elegirse un valor centinela que no pueda confundirse con un valor de entrada aceptable.
- El método de refinamiento de arriba a abajo, paso a paso (pág. 118), es esencial para el desarrollo de programas bien estructurados.
- La división entre cero es un error lógico.
- Para realizar un cálculo de punto flotante con valores enteros, convierta (pág. 124) uno de los enteros al tipo `double`.
- Java sabe cómo evaluar sólo las expresiones aritméticas en las que los tipos de los operandos son idénticos. Para asegurar esto, Java realiza una operación conocida como promoción (pág. 124) sobre los operandos seleccionados.
- El operador unario de conversión de tipos (pág. 124) se forma mediante la colocación de paréntesis alrededor del nombre de un tipo.

Sección 4.11 Operadores de asignación compuestos

- Los operadores de asignación compuestos (pág. 130) abrevian las expresiones de asignación. Las instrucciones de la forma

  variable = variable operador expresión;

  en donde operador es uno de los operadores binarios +, -, *, / o %, puede escribirse en la forma

  variable operador= expresión;

- El operador += suma el valor de la expresión que está a la derecha del operador con el valor de la variable a la izquierda del operador, y almacena el resultado en la variable a la izquierda del operador.

Sección 4.12 Operadores de incremento y decremento

- El operador de incremento unario, ++, y el operador de decremento unario, --, suman 1, o restan 1, al valor de una variable numérica (pág. 130).

- Un operador de incremento o decremento que se coloca antes de una variable (pág. 131) es el operador de preincremento o predecremento, correspondiente. Un operador de incremento o decremento que se coloca después de una variable (pág. 131) es el operador de postincremento o postdecremento, respectivamente.
- El proceso de usar el operador de preincremento o predecremento para sumar o restar 1 se conoce como preincrementar o predecrementar, cada uno.
- Al preincrementar o predecrementar una variable, ésta se incrementa o decrementa por 1; después se utiliza el nuevo valor de la variable en la expresión en la que aparece.
- El proceso de usar el operador de postincremento o postdecremento para sumar o restar 1 se conoce como postincrementar o postdecrementar, respectivamente.
- Al postincrementar o postdecrementar una variable, el valor actual de ésta se utiliza en la expresión en la que aparece; después el valor de la variable se incrementa o decrementa por 1.
- Cuando se incrementa o decrementa una variable en una instrucción por sí sola, las formas de preincremento y postincremento tienen el mismo efecto, y las formas de predecremento y postdecremento también tienen el mismo efecto.

Sección 4.13 Tipos primitivos

- Java requiere que todas las variables tengan un tipo. Por ende, Java se conoce como un lenguaje fuertemente tipificado (pág. 134).
- Java usa caracteres Unicode y números de punto flotante IEEE 754.

Ejercicios de autoevaluación

4.1 Complete los siguientes enunciados:

a) Todos los programas pueden escribirse en términos de tres tipos de estructuras de control: _____ , _____ y _____ .

b) La instrucción _____ se utiliza para ejecutar una acción cuando una condición es verdadera, y otra cuando esa es falsa.

c) Al proceso de repetir un conjunto de instrucciones un número específico de veces se le llama repetición _____ .

d) Cuando no se sabe de antemano cuántas veces se repetirá un conjunto de instrucciones, se puede usar un valor _____ para terminar la repetición.

e) La estructura _____ está integrada en Java; de manera predeterminada, las instrucciones se ejecutan en el orden en el que aparecen.

f) Todas las variables de instancia de los tipos `char`, `byte`, `short`, `int`, `long`, `float` y `double` reciben el valor _____ de manera predeterminada.

g) Java es un lenguaje _____ ; requiere que todas las variables tengan un tipo.

h) Si el operador de incremento se _____ de una variable, ésta se incrementa en 1 primero, y después su nuevo valor se utiliza en la expresión.

4.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Un algoritmo es un procedimiento para resolver un problema en términos de las acciones a ejecutar y el orden en el que lo hacen.

b) Un conjunto de instrucciones contenidas dentro de un par de paréntesis se denomina bloque.

c) Una instrucción de selección especifica que una acción se repetirá, mientras cierta condición siga siendo verdadera.

d) Una instrucción de control anidada aparece en el cuerpo de otra instrucción de control.

e) Java cuenta con los operadores de asignación compuestos aritméticos `+=`, `-=`, `*=`, `/=` y `%=` para abreviar las expresiones de asignación.

f) Los tipos primitivos (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` y `double`) son portables sólo en las plataformas Windows.

g) Al proceso de especificar el orden en el que se ejecutan las instrucciones en un programa se denomina control del programa.

h) El operador de conversión de tipos unario (`double`) crea una copia entera temporal de su operando.

i) Las variables de instancia de tipo `boolean` reciben el valor `true` de manera predeterminada.
j) El seudocódigo ayuda a un programador a idear un programa, antes de tratar de escribirlo en un lenguaje de programación.

4.3 Escriba cuatro instrucciones distintas en Java, en donde cada una sume 1 a la variable entera `x`.

4.4 Escriba instrucciones en Java para realizar cada una de las siguientes tareas:
a) Usar una instrucción para asignar la suma de `x` e `y` a `z`, e incrementar `x` en 1 después del cálculo.
b) Evaluar si la variable `cuenta` es mayor que 10. De ser así, imprimir "`Cuenta es mayor que 10`".
c) Usar una instrucción para decrementar la variable `x` en 1, luego restarla a la variable `total` y almacenar el resultado en la variable `total`.
d) Calcular el residuo después de dividir `q` entre divisor, y asignar el resultado a `q`. Escriba esta instrucción de dos maneras distintas.

4.5 Escriba una instrucción en Java para realizar cada una de las siguientes tareas:
a) Declarar las variables `suma` y `x` como de tipo `int`.
b) Asignar `1` a la variable `x`.
c) Asignar `0` a la variable `suma`.
d) Sumar la variable `x` a `suma` y asignar el resultado a la variable `suma`.
e) Imprimir la cadena "`La suma es: `", seguida del valor de la variable `suma`.

4.6 Combine las instrucciones que escribió en el ejercicio 4.5 para formar una aplicación en Java que calcule e imprima la suma de los enteros del 1 al 10. Use una instrucción `while` para iterar a través de las instrucciones de cálculo e incremento. El ciclo debe terminar cuando el valor de `x` se vuelva 11.

4.7 Determine el valor de las variables en la instrucción `producto *= x++;`, después de realizar el cálculo. Suponga que todas las variables son de tipo `int` y tienen el valor 5.

4.8 Identifique y corrija los errores en cada uno de los siguientes conjuntos de código:

a)
```
while ( c <= 5 )
{
   producto *= c;
   ++c;
```

b)
```
if ( genero == 1 )
   System.out.println( "Mujer" );
else;
   System.out.println( "Hombre" );
```

4.9 ¿Qué está mal en la siguiente instrucción while?
```
while ( z >= 0 )
         suma += z;
```

Respuestas a los ejercicios de autoevaluación

4.1 a) secuencia, selección, repetición. b) `if...else`. c) controlada por contador (o definida). d) centinela, de señal, de prueba o de bandera. e) secuencia. f) 0 (cero). g) fuertemente tipificado. h) coloca antes.

4.2 a) Verdadero. b) Falso. Un conjunto de instrucciones contenidas dentro de un par de llaves ({ y }) se denomina bloque. c) Falso. Una instrucción de repetición especifica que una acción se repetirá mientras que cierta condición siga siendo verdadera. d) Verdadero. e) Verdadero. f) Falso. Los tipos primitivos (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` y `double`) son portables a través de todas las plataformas de computadora que soportan Java. g) Verdadero. h) Falso. El operador de conversión de tipos unario (`double`) crea una copia temporal de punto flotante de su operando. i) Falso. Las variables de instancia de tipo `boolean` reciben el valor `false` de manera predeterminada. j) Verdadero.

4.3
```
x = x + 1;
x += 1;
++x;
x++;
```

4.4 a) `z = x++ + y;`

b)
```
if ( cuenta > 10 )
   System.out.println( "Cuenta es mayor que 10" );
```

c) `total -= --x;`

d)
```
q %= divisor;
q = q % divisor;
```

4.5 a)
```
int suma;
int x;
```

b) `x = 1;`

c) `suma = 0;`

d) `suma += x;` o `suma = suma + x;`

e) `System.out.printf( "La suma es: %d\n", suma );`

4.6 El programa se muestra a continuación:

```
1  // Ejercicio 4.6: Calcular.java
2  // Calcula la suma de los enteros del 1 al 10
3  public class Calcular
4  {
5     public static void main( String[] args )
6     {
7        int suma;
8        int x;
9
10       x = 1;     // inicializa x en 1 para contar
11       suma = 0; // inicializa suma en 0 para el total
12
13       while ( x <= 10 ) // mientras que x sea menor o igual que 10
14       {
15          suma += x; // suma x a suma
16          ++x; // incrementa x
17       } // fin de while
18
19       System.out.printf( "La suma es: %d\n", suma );
20    } // fin de main
21 } // fin de la clase Calcular
```

```
La suma es: 55
```

4.7 `producto = 25, x = 6`

4.8 a) Error: falta la llave derecha de cierre del cuerpo de la instrucción `while`.
Corrección: Agregar una llave derecha de cierre después de la instrucción `++c;`.

b) Error: El punto y coma después de `else` produce un error lógico. La segunda instrucción de salida siempre se ejecutará.
Corrección: Quitar el punto y coma después de `else`.

4.9 El valor de la variable z nunca se cambia en la instrucción `while`. Por lo tanto, si la condición de continuación de ciclo `( z >= 0 )` es verdadera, se crea un ciclo infinito. Para evitar que ocurra un ciclo infinito, z debe decrementarse de manera que en un momento dado se vuelva menor que 0.

Ejercicios

4.10 Compare y contraste la instrucción `if` de selección simple y la instrucción de repetición `while`. ¿Cuál es la similitud en las dos instrucciones? ¿Cuál es su diferencia?

4.11 Explique lo que ocurre cuando un programa en Java trata de dividir un entero entre otro. ¿Qué ocurre con la parte fraccionaria del cálculo? ¿Cómo puede un programador evitar ese resultado?

4.12 Describa las dos formas en las que pueden combinarse las instrucciones de control.

4.13 ¿Qué tipo de repetición sería apropiada para calcular la suma de los primeros 100 enteros positivos? ¿Qué tipo de repetición sería apropiada para calcular la suma de un número arbitrario de enteros positivos? Describa brevemente cómo podría realizarse cada una de estas tareas.

4.14 ¿Cuál es la diferencia entre preincrementar y postincrementar una variable?

4.15 Identifique y corrija los errores en cada uno de los siguientes fragmentos de código. [*Nota:* puede haber más de un error en cada fragmento de código].

a)
```
if ( edad >= 65 );
   System.out.println( "Edad es mayor o igual que 65" );
else
   System.out.println( "Edad es menor que 65 )";
```

b)
```
int x = 1, total;
while ( x <= 10 )
{
   total += x;
   ++x;
}
```

c)
```
while ( x <= 100 )
   total += x;
   ++x;
```

d)
```
while ( y > 0 )
{
   System.out.println( y );
   ++y;
```

4.16 ¿Qué es lo que imprime el siguiente programa?

```
1  // Ejercicio 4.16: Misterio.java
2  public class Misterio
3  {
4     public static void main( String[] args )
5     {
6        int y;
7        int x = 1;
8        int total = 0;
9
10       while ( x <= 10 )
11       {
12          y = x * x;
13          System.out.println( y );
14          total += y;
15          ++x;
16       } // fin de while
17
18       System.out.printf( "El total es %d\n", total );
19    } // fin de main
20 } // fin de la clase Misterio
```

Para los ejercicios 4.17 a 4.20, realice cada uno de los siguientes pasos:

a) Lea el enunciado del problema.

b) Formule el algoritmo mediante un seudocódigo y el proceso de refinamiento de arriba a abajo, paso a paso.

c) Escriba un programa en Java.
d) Pruebe, depure y ejecute el programa en Java.
e) Procese tres conjuntos completos de datos.

4.17 ***(Kilometraje de gasolina)*** Los conductores se preocupan acerca del kilometraje de sus automóviles. Un conductor ha llevado el registro de varios reabastecimientos de gasolina, registrando los kilómetros conducidos y los litros usados en cada viaje. Desarrolle una aplicación en Java que reciba como entrada los kilómetros conducidos y los litros usados (ambos como enteros) por cada viaje. El programa debe calcular y mostrar los kilómetros por litro obtenidos en cada viaje, y debe imprimir el total de kilómetros por litro obtenidos en todos los reabastecimientos hasta este punto. Todos los cálculos del promedio deben producir resultados en números de punto flotante. Use la clase `Scanner` y la repetición controlada por centinela para obtener los datos del usuario.

4.18 ***(Calculadora de límite de crédito)*** Desarrolle una aplicación en Java que determine si alguno de los clientes de una tienda de departamentos se ha excedido del límite de crédito en una cuenta. Para cada cliente se tienen los siguientes datos:

a) el número de cuenta.
b) el saldo al inicio del mes.
c) el total de todos los artículos cargados por el cliente en el mes.
d) el total de todos los créditos aplicados a la cuenta del cliente en el mes.
e) el límite de crédito permitido.

El programa debe recibir como entrada cada uno de estos datos en forma de números enteros, debe calcular el nuevo saldo (= *saldo inicial* + *cargos* – *créditos*), mostrar el nuevo balance y determinar si éste excede el límite de crédito del cliente. Para los clientes cuyo límite de crédito sea excedido, el programa debe mostrar el mensaje "`Se excedió el límite de su crédito`".

4.19 ***(Calculadora de comisiones de ventas)*** Una empresa grande paga a sus vendedores mediante comisiones. Los vendedores reciben $200 por semana, más el 9% de sus ventas brutas durante esa semana. Por ejemplo, un vendedor que vende $5,000 de mercancía en una semana, recibe $200 más el 9% de 5,000, o un total de $650. Usted acaba de recibir una lista de los artículos vendidos por cada vendedor. Los valores de estos artículos son los siguientes:

```
Artículo    Valor
1           239.99
2           129.75
3            99.95
4           350.89
```

Desarrolle una aplicación en Java que reciba como entrada los artículos vendidos por un comerciante durante la última semana, y que calcule y muestre los ingresos de ese vendedor. No hay límite en cuanto al número de artículos que un representante puede vender.

4.20 ***(Calculadora del salario)*** Desarrolle una aplicación en Java que determine el sueldo bruto para cada uno de tres empleados. La empresa paga la cuota normal en las primeras 40 horas de trabajo de cada empleado, y cuota y media en todas las horas trabajadas que excedan de 40. Usted recibe una lista de los empleados de la empresa, el número de horas que trabajó cada uno la semana pasada y la tarifa por horas de cada empleado. Su programa debe recibir como entrada esta información para cada empleado, debe determinar y mostrar el sueldo bruto de cada trabajador. Utilice la clase `Scanner` para introducir los datos.

4.21 ***(Encontrar el valor más grande)*** El proceso de encontrar el valor más grande se utiliza con frecuencia en aplicaciones de computadora. Por ejemplo, un programa para determinar el ganador de un concurso de ventas recibe como entrada el número de unidades vendidas por cada vendedor. El que haya vendido más unidades es el que gana el concurso. Escriba un programa en seudocódigo y después una aplicación en Java que reciba como entrada una serie de 10 números enteros, y que determine e imprima el mayor de los números. Su programa debe utilizar cuando menos las siguientes tres variables:

a) `contador`: Un contador para contar hasta 10 (para llevar el registro de cuántos números se han introducido, y para determinar cuando se hayan procesado los 10 números).
b) `numero`: El entero más reciente introducido por el usuario.
c) `mayor`: El número más grande encontrado hasta ahora.

4.22 *(Salida tabular)* Escriba una aplicación en Java que utilice ciclos para imprimir la siguiente tabla de valores:

```
N       10*N    100*N   1000*N

1       10      100     1000
2       20      200     2000
3       30      300     3000
4       40      400     4000
5       50      500     5000
```

4.23 *(Encontrar los dos números más grandes)* Utilizando una metodología similar a la del ejercicio 4.21, encuentre los dos valores más grandes de los 10 que se introdujeron. [*Nota:* puede introducir cada número sólo una vez].

4.24 *(Validar la entrada del usuario)* Modifique el programa de la figura 4.12 para validar sus entradas. Para cualquier entrada, si el valor introducido es distinto de 1 o 2, debe seguir iterando hasta que el usuario introduzca un valor correcto.

4.25 ¿Qué es lo que imprime el siguiente programa?

```
1  // Ejercicio 4.25: Misterio2.java
2  public class Misterio2
3  {
4     public static void main( String[] args )
5     {
6        int cuenta = 1;
7
8        while ( cuenta <= 10 )
9        {
10          System.out.println( cuenta % 2 == 1 ? "****" : "++++++++" );
11          ++cuenta;
12       } // fin de while
13    } // fin de main
14 } // fin de la clase Misterio2
```

4.26 ¿Qué es lo que imprime el siguiente programa?

```
1  // Ejercicio 4.26: Misterio3.java
2  public class Misterio3
3  {
4     public static void main( String[] args )
5     {
6        int fila = 10;
7        int columna;
8
9        while ( fila >= 1 )
10       {
11          columna = 1;
12
13          while ( columna <= 10 )
14          {
15             System.out.print( fila % 2 == 1 ? "<" : ">" );
16             ++columna;
17          } // fin de while
18
19          --fila;
20          System.out.println();
21       } // fin de while
22    } // fin de main
23 } // fin de la clase Misterio3
```

4.27 ***(Problema del else suelto)*** Determine la salida de cada uno de los siguientes conjuntos de código, cuando x es 9 y y es 11, y cuando x es 11 y y es 9. Observe que el compilador ignora la sangría en un programa en Java. Además, el compilador de Java siempre asocia un `else` con el `if` que le precede inmediatamente, a menos que se le indique de otra forma mediante la colocación de llaves ({ }). A primera vista, el programador tal vez no esté seguro de cuál `if` corresponde a cuál `else`; esta situación se conoce como el "problema del `else` suelto". Hemos eliminado la sangría del siguiente código para hacer el problema más retador. [*Sugerencia:* aplique las convenciones de sangría que ha aprendido].

a)
```
if ( x < 10 )
if ( y > 10 )
System.out.println( "*****" );
else
System.out.println( "#####" );
System.out.println( "$$$$$" );
```

b)
```
if ( x < 10 )
{
if ( y > 10 )
System.out.println( "*****" );
}
else
{
System.out.println( "#####" );
System.out.println( "$$$$$" );
}
```

4.28 ***(Otro problema de else suelto)*** Modifique el código dado para producir la salida que se muestra en cada parte del problema. Utilice las técnicas de sangría apropiadas. No haga modificaciones en el código, sólo inserte llaves o modifique la sangría del código. El compilador ignora la sangría en un programa en Java. Hemos eliminado la sangría en el código dado, para hacer el problema más retador. [*Nota:* es posible que no se requieran modificaciones en algunas de las partes].

```
if ( y == 8 )
if ( x == 5 )
System.out.println( "@@@@@" );
else
System.out.println( "#####" );
System.out.println( "$$$$$" );
System.out.println( "&&&&&" );
```

a) Suponiendo que `x = 5` y `y = 8`, se produce la siguiente salida:

```
@@@@@
$$$$$
&&&&&
```

b) Suponiendo que `x = 5` y `y = 8`, se produce la siguiente salida:

```
@@@@@
```

c) Suponiendo que `x = 5` y `y = 8`, se produce la siguiente salida:

```
@@@@@
```

d) Suponiendo que `x = 5` y `y = 7`, se produce la siguiente salida. [*Nota:* las tres últimas instrucciones de salida después del `else` forman parte de un bloque].

```
#####
$$$$$
&&&&&
```

4.29 ***(Cuadrado de asteriscos)*** Escriba una aplicación que pida al usuario que introduzca el tamaño del lado de un cuadrado y que muestre un cuadrado hueco de ese tamaño, compuesto de asteriscos. Su programa debe funcionar con cuadrados que tengan lados de todas las longitudes entre 1 y 20.

4.30 ***(Palíndromos)*** Un palíndromo es una secuencia de caracteres que se lee igual al derecho y al revés. Por ejemplo, cada uno de los siguientes enteros de cinco dígitos es un palíndromo: 12321, 55555, 45554 y 11611. Escriba una aplicación que lea un entero de cinco dígitos y determine si es un palíndromo. Si el número no es de cinco dígitos, el programa debe mostrar un mensaje de error y permitir al usuario que introduzca un nuevo valor.

4.31 ***(Imprimir el equivalente decimal de un número binario)*** Escriba una aplicación que reciba como entrada un entero que contenga sólo dígitos 0 y 1 (es decir, un entero binario), y que imprima su equivalente decimal. [*Sugerencia:* use los operadores residuo y división para elegir los dígitos del número binario uno a la vez, de derecha a izquierda. En el sistema numérico decimal, el dígito más a la derecha tiene un valor posicional de 1 y el siguiente dígito a la izquierda tiene un valor posicional de 10, después 100, después 1,000, etcétera. El número decimal 234 puede interpretarse como 4 * 1 + 3 * 10 + 2 * 100. En el sistema numérico binario, el dígito más a la derecha tiene un valor posicional de 1, el siguiente dígito a la izquierda tiene un valor posicional de 2, luego 4, luego 8, etcétera. El equivalente decimal del número binario 1101 es 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8, o 1 + 0 + 4 + 8, o 13].

4.32 ***(Patrón de asteriscos en forma de tablero de damas)*** Escriba una aplicación que utilice sólo las instrucciones de salida

```
System.out.print( "* " );
System.out.print( " " );
System.out.println();
```

para mostrar el patrón de tablero de damas que se muestra a continuación. Observe que una llamada al método `System.out.println` sin argumentos hace que el programa imprima un solo carácter de nueva línea. [*Sugerencia:* se requieren estructuras de repetición].

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

4.33 ***(Múltiplos de 2 con un ciclo infinito)*** Escriba una aplicación que muestre en la ventana de comandos los múltiplos del entero 2 (a saber, 2, 4, 8, 16, 32, 64, etcétera). Su ciclo no debe terminar (es decir, debe crear un ciclo infinito). ¿Qué ocurre cuando ejecuta este programa?

4.34 ***(¿Qué está mal en este código?)*** ¿Qué está mal en la siguiente instrucción? Proporcione la instrucción correcta para sumar uno a la suma de `x` y `y`.

```
System.out.println( ++(x + y) );
```

4.35 ***(Lados de un triángulo)*** Escriba una aplicación que lea tres valores distintos de cero introducidos por el usuario, y que determine e imprima si podrían representar los lados de un triángulo.

4.36 ***(Lados de un triángulo rectángulo)*** Escriba una aplicación que lea tres enteros distintos de cero, determine e imprima si éstos podrían representar los lados de un triángulo rectángulo.

4.37 ***(Factorial)*** El factorial de un entero n no negativo se escribe como $n!$ (se pronuncia "factorial de n") y se define de la siguiente manera:

$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$ (para valores de n mayores o iguales a 1)

y

$$n! = 1 \quad (\text{para } n = 0)$$

Por ejemplo, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que es 120.

a) Escriba una aplicación que lea un entero no negativo, y calcule e imprima su factorial.

b) Escriba una aplicación que estime el valor de la constante matemática *e*, utilizando la siguiente fórmula. Deje que el usuario introduzca el número de términos a calcular.

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

c) Escriba una aplicación que calcule el valor de e^x, utilizando la siguiente fórmula. Deje que el usuario introduzca el número de términos a calcular.

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Marcar la diferencia

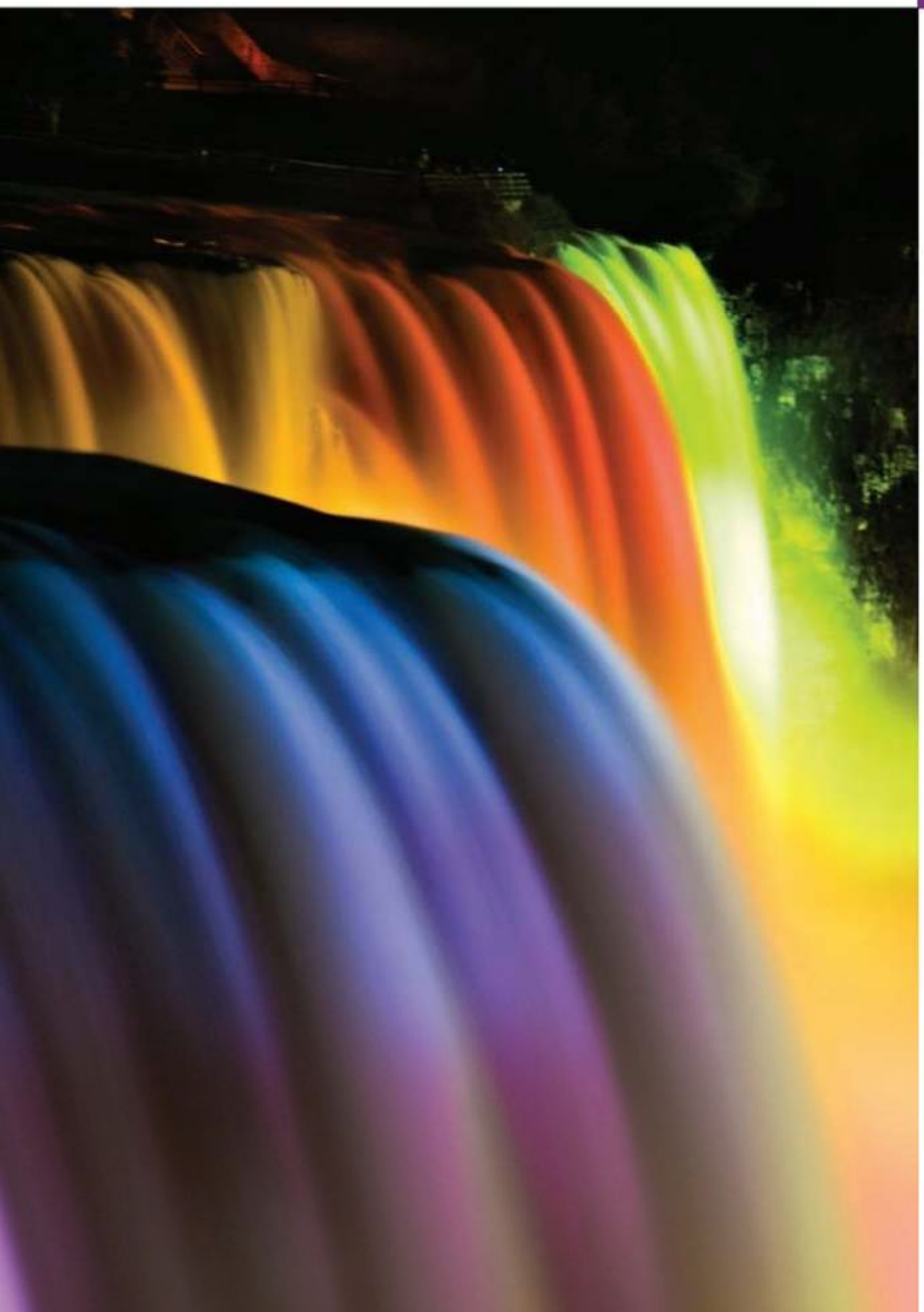
4.38 ***(Implementar la privacidad con la criptografía)*** El crecimiento explosivo de las comunicaciones de Internet y el almacenamiento de datos en computadoras conectadas a Internet, ha incrementado de manera considerable los problemas de privacidad. El campo de la criptografía se dedica a la codificación de datos para dificultar (y, mediante los esquemas más avanzados, tratar de imposibilitar) su lectura a los usuarios no autorizados. En este ejercicio, usted investigará un esquema simple para cifrar y descifrar datos. Una compañía que desea enviar datos por Internet le pidió que escribiera un programa que los cifre, de modo que se puedan transmitir con más seguridad. Todos los datos se transmiten como enteros de cuatro dígitos. Su aplicación debe leer un entero de cuatro dígitos introducido por el usuario, y cifrarlo de la siguiente manera: Reemplace cada dígito con el resultado de sumarle 7 y obtenga el residuo después de dividir el nuevo valor entre 10. Después intercambie el primer dígito con el tercero, y el segundo dígito con el cuarto. Luego imprima el entero cifrado. Escriba una aplicación separada que reciba como entrada el número entero de cuatro dígitos cifrado y lo descifre (invirtiendo el esquema de cifrado) para formar el número original. [*Proyecto de lectura opcional:* investigue la "criptografía de clave pública" en general y el esquema de clave pública específico PGP (Privacidad bastante buena). Tal vez también quiera investigar el esquema RSA, que se utiliza mucho en las aplicaciones de nivel industrial].

4.39 ***(Crecimiento de la población mundial)*** La población mundial ha crecido de manera considerable a través de los siglos. El crecimiento continuo podría, en un momento dado, desafiar los límites del aire respirable, el agua potable, la tierra cultivable y otros recursos limitados. Hay evidencia de que el crecimiento se ha reducido en años recientes, y que la población mundial podría llegar a su valor máximo en algún momento de este siglo, para luego empezar a disminuir.

Para este ejercicio, investigue en línea las cuestiones sobre el crecimiento de la población mundial. *Asegúrese de investigar varios puntos de vista.* Obtenga estimaciones de la población mundial actual y su tasa de crecimiento (el porcentaje por el cual es probable que aumente este año). Escriba un programa que calcule el crecimiento anual de la población mundial durante los siguientes 75 años, *utilizando la suposición simplificada de que la tasa de crecimiento actual permanecerá constante.* Imprima los resultados en una tabla. La primera columna debe mostrar el año, desde el año 1 hasta el año 75. La segunda columna debe mostrar la población mundial anticipada al final de ese año. La tercera columna deberá mostrar el aumento numérico en la población mundial que ocurriría ese año. Use sus resultados para determinar el año en el que el tamaño de la población será del doble del actual, si fuera a persistir la tasa de crecimiento de este año.

Instrucciones de control: Parte 2

5



La rueda se convirtió en un círculo completo.

—William Shakespeare

Toda la evolución que conocemos procede de lo vago a lo definido.

—Charles Sanders Peirce

Objetivos

En este capítulo aprenderá a:

- Conocer los fundamentos acerca de la repetición controlada por un contador.
- Utilizar las instrucciones de repetición `for` y `do...while` para ejecutar instrucciones de manera repetitiva en un programa.
- Comprender la selección múltiple utilizando la instrucción de selección `switch`.
- Utilizar las instrucciones de control de programa `break` y `continue` para alterar el flujo de control.
- Utilizar los operadores lógicos para formar expresiones condicionales complejas en instrucciones de control.

Plan general

5.1 Introducción

En este capítulo continuaremos nuestra presentación de la teoría y los principios de la programación estructurada, presentando el resto de las instrucciones de control en Java, excepto una, también demostraremos las instrucciones `for`, `do...while` y `switch` de Java. A través de una serie de ejemplos cortos en los que utilizaremos las instrucciones `while` y `for` exploraremos los fundamentos acerca de la repetición controlada por contador. Crearemos una versión de la clase `LibroCalificaciones` que utiliza una instrucción `switch` para contar el número de calificaciones equivalentes de A, B, C, D y F, en un conjunto de calificaciones numéricas introducidas por el usuario. Presentaremos las instrucciones de control de programa `break` y `continue`. Hablaremos sobre los operadores lógicos de Java, que nos permiten utilizar expresiones condicionales más complejas en las instrucciones de control. Por último, veremos un resumen de las instrucciones de control de Java y las técnicas ya probadas de solución de problemas que presentamos en éste y en el capítulo 4.

5.2 Fundamentos de la repetición controlada por contador

Esta sección utiliza la instrucción de repetición `while`, presentada en el capítulo 4, para formalizar los elementos requeridos y llevar a cabo la repetición controlada por contador. Este tipo de repetición requiere

1. una **variable de control** (o contador de ciclo)
2. el **valor inicial** de la variable de control
3. el **incremento** (o **decremento**) con el que se modifica la variable de control cada vez que pasa por el ciclo (lo que también se conoce como **cada iteración del ciclo**)
4. la **condición de continuación de ciclo**, que determina si el ciclo debe continuar o no.

Para ver estos elementos de la repetición controlada por contador, considere la aplicación de la figura 5.1, que utiliza un ciclo para mostrar los números del 1 al 10.

```
1 // Fig. 5.1: ContadorWhile.java
2 // Repetición controlada con contador, con la instrucción de repetición while.
3
4 public class ContadorWhile
5 {
6    public static void main( String[] args )
7    {
```

Fig. 5.1 | Repetición controlada con contador, con la instrucción de repetición `while` (parte 1 de 2).

```
8         int contador = 1; // declara e inicializa la variable de control
9
10        while ( contador <= 10 ) // condición de continuación de ciclo
11        {
12           System.out.printf( "%d  ", contador );
13           ++contador; // incrementa la variable de control en 1
14        } // fin de while
15
16        System.out.println(); // imprime una nueva línea
17     } // fin de main
18  } // fin de la clase ContadorWhile
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 5.1 | Repetición controlada con contador, con la instrucción de repetición `while` (parte 2 de 2).

En la figura 5.1, los elementos de la repetición controlada por contador se definen en las líneas 8, 10 y 13. La línea 8 declara la variable de control (`contador`) como un `int`, reserva espacio para esta variable en memoria y establece su valor inicial en 1. La variable contador también podría haberse declarado e inicializado con la siguientes instrucciones de declaración y asignación de variables locales:

```
int contador; // declara contador
contador = 1; // inicializa contador en 1
```

La línea 12 muestra el valor de la variable de control `contador` durante cada iteración del ciclo. La línea 13 incrementa la variable de control en 1, para cada iteración del ciclo. La condición de continuación de ciclo en la instrucción `while` (línea 10) evalúa si el valor de la variable de control es menor o igual que 10 (el valor final para el que la condición es `true`). El programa ejecuta el cuerpo de este `while` aún y cuando la variable de control sea 10. El ciclo termina cuando la variable de control es mayor que 10 (es decir, cuando `contador` se convierte en 11).

Error común de programación 5.1

Debido a que los valores de punto flotante pueden ser aproximados, controlar los ciclos con variables de punto flotante puede producir valores imprecisos del contador y pruebas de terminación imprecisas.

Tip para prevenir errores 5.1

Use enteros para controlar los ciclos de contador.

El programa de la figura 5.1 puede hacerse más conciso si inicializamos `contador` en 0 en la línea 8, y preincrementamos contador en la condición `while` de la siguiente forma:

```
while ( ++contador <= 10 ) // condición de continuación de ciclo
   System.out.printf( "%d  ", contador );
```

Este código ahorra una instrucción (y elimina la necesidad de usar llaves alrededor del cuerpo del ciclo), ya que la condición de `while` realiza el incremento antes de evaluar la condición. (En la sección 4.12 vimos que la precedencia de ++ es mayor que la de <=). La codificación en esta forma tan condensada requiere de práctica y puede hacer que el código sea más difícil de leer, depurar, modificar y mantener, así que en general, es mejor evitarla.

Observación de ingeniería de software 5.1

"Mantener las cosas simples" es un buen consejo para la mayoría del código que usted escriba.

5.3 Instrucción de repetición for

La sección 5.2 presentó los aspectos esenciales de la repetición controlada por contador. La instrucción `while` puede utilizarse para implementar cualquier ciclo controlado por un contador. Java también cuenta con la **instrucción de repetición for**, que especifica los detalles de la repetición controlada por contador en una sola línea de código. La figura 5.2 reimplementa la aplicación de la figura 5.1, usando la instrucción `for`.

```
1  // Fig. 5.2: ContadorFor.java
2  // Repetición controlada con contador, con la instrucción de repetición for.
3
4  public class ContadorFor
5  {
6     public static void main( String[] args )
7     {
8        // el encabezado de la instrucción for incluye la inicialización,
9        // la condición de continuación de ciclo y el incremento
10       for ( int contador = 1; contador <= 10; contador++ )
11          System.out.printf( "%d  ", contador );
12
13       System.out.println(); // imprime una nueva línea
14    } // fin de main
15 } // fin de la clase ContadorFor
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 5.2 | Repetición controlada por un contador, con la instrucción de repetición `for`.

Cuando la instrucción `for` (líneas 10 y 11) comienza a ejecutarse, la variable de control `contador` se declara e inicializa en 1 (en la sección 5.2 vimos que los primeros dos elementos de la repetición controlada por un contador son la variable de control y su valor inicial). A continuación, el programa verifica la condición de continuación de ciclo, `contador <= 10`, la cual se encuentra entre los dos signos de punto y coma requeridos. Como el valor inicial de `contador` es 1, al principio la condición es verdadera. Por lo tanto, la instrucción del cuerpo (línea 11) muestra el valor de la variable de control `contador`, que es 1. Después de ejecutar el cuerpo del ciclo, el programa incrementa a `contador` en la expresión `contador++`, la cual aparece a la derecha del segundo signo de punto y coma. Después, la prueba de continuación de ciclo se ejecuta de nuevo para determinar si el programa debe continuar con la siguiente iteración del ciclo. En este punto, el valor de la variable de control es 2, por lo que la condición sigue siendo verdadera (el valor final no se excede); así, el programa ejecuta la instrucción del cuerpo otra vez (es decir, la siguiente iteración del ciclo). Este proceso continúa hasta que se muestran en pantalla los números del 1 al 10 y el valor de `contador` se vuelve 11, con lo cual falla la prueba de continuación de ciclo y termina la repetición (después de 10 repeticiones del cuerpo del ciclo). Luego, el programa ejecuta la primera instrucción después del `for`; en este caso, la línea 13.

La figura 5.2 utiliza (en la línea 10) la condición de continuación de ciclo `contador <= 10`. Si usted especificara por error `contador < 10` como la condición, el ciclo sólo iteraría nueve veces. A este error lógico común se le conoce como **error por desplazamiento en uno**.

Error común de programación 5.2

Utilizar un operador relacional incorrecto o un valor final incorrecto de un contador de ciclo en la condición de continuación de ciclo de una instrucción de repetición puede producir un error por desplazamiento en uno.

Tip para prevenir errores 5.2

Utilizar el valor final en la condición de una instrucción `while` *o* `for` *con el operador relacional* `<=` *nos ayuda a evitar los errores por desplazamiento en uno. Para un ciclo que imprime los valores del 1 al 10, la condición de continuación de ciclo debe ser* `contador <= 10`*, en vez de* `contador < 10` *(lo cual produce un error por desplazamiento en uno) o* `contador <11` *(que es correcto). Muchos programadores prefieren el llamado conteo con base cero, en el cual para contar 10 veces,* `contador` *se inicializaría en cero y la prueba de continuación de ciclo sería* `contador < 10`.

Una vista más detallada del encabezado de la instrucción `for`

La figura 5.3 analiza con más detalle la instrucción `for` de la figura 5.2. A la primera línea del `for` (incluyendo la palabra clave `for` y todo lo que está entre paréntesis después de ésta), la línea 10 de la figura 5.2, se le llama algunas veces **encabezado de la instrucción for**. El encabezado del `for` "se encarga de todo": especifica cada uno de los elementos necesarios para la repetición controlada por un contador con una variable de control. Si hay más de una instrucción en el cuerpo del `for`, se requieren llaves para definir el cuerpo del ciclo.

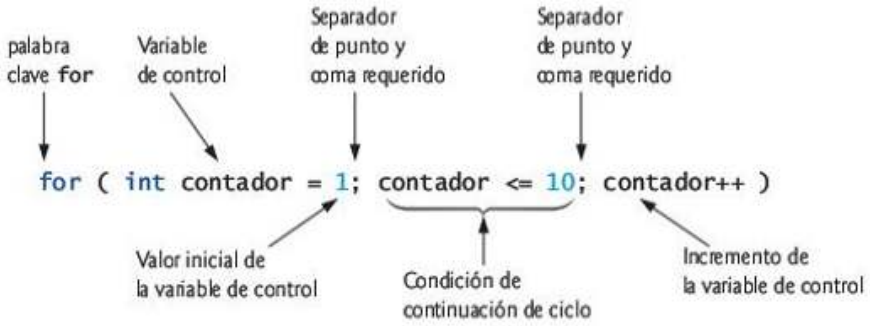


Fig. 5.3 | Componentes del encabezado de la instrucción `for`.

Formato general de una instrucción `for`

El formato general de la instrucción `for` es

```
for ( inicialización; condiciónDeContinuaciónDeCiclo; incremento )
   instrucción
```

en donde la expresión *inicialización* nombra a la variable de control de ciclo y proporciona de manera opcional su valor inicial, la *condiciónDeContinuaciónDeCiclo* determina si el ciclo debe seguir ejecutándose, y el *incremento* modifica el valor de la variable de control (posiblemente un incremento o un decremento), de manera que la condición de continuación de ciclo se vuelva falsa en un momento dado. Los dos signos de punto y coma en el encabezado del `for` son obligatorios. Si en un principio la condición de continuación de ciclo es `false`, el programa *no* ejecuta el cuerpo de la instrucción `for`. En cambio, la ejecución continúa con la instrucción después del `for`.

Representación de una instrucción `for` *con una instrucción* `while` *equivalente*

En la mayoría de los casos, la instrucción `for` puede representarse con una instrucción `while` equivalente, de la siguiente manera:

```
inicialización;
while ( condiciónDeContinuaciónDeCiclo )
{
   instrucción
   incremento;
}
```

En la sección 5.7 veremos un caso para el cual no es posible representar una instrucción `for` con una instrucción `while` equivalente.

Por lo general, las instrucciones `for` se utilizan para la repetición controlada por un contador, y las instrucciones `while` para la repetición controlada por un centinela. No obstante, `while` y `for` pueden usarse para cualquiera de los dos tipos de repetición.

Alcance de la variable de control de una instrucción `for`

Si la expresión de *inicialización* en el encabezado del `for` declara la variable de control (si el tipo de la variable de control se especifica antes del nombre de la variable, como en la figura 5.2), la variable de control puede utilizarse *sólo* en esa instrucción `for`; no existirá fuera de ella. Este uso restringido se conoce como el **alcance** de la variable. El alcance de una variable define en dónde puede emplearse en un programa. Por ejemplo, una variable local *sólo* puede usarse en el método que declara a esa variable, y *sólo* a partir del punto de declaración, hasta el final del método. En el capítulo 6, Métodos: un análisis más detallado, veremos con detenimiento el concepto de alcance.

Error común de programación 5.3

Cuando se declara la variable de control de una instrucción `for` en la sección de inicialización del encabezado del `for`, si se utiliza la variable de control después del cuerpo del `for` se produce un error de compilación.

Las expresiones en el encabezado de una instrucción `for` *son opcionales*

Las tres expresiones en un encabezado `for` son opcionales. Si se omite la *condiciónDeContinuaciónDeCiclo*, Java asume que esta condición siempre será verdadera, con lo cual se crea un ciclo infinito. Podríamos omitir la expresión de *inicialización* si el programa inicializa la variable de control antes del ciclo. Podríamos omitir la expresión de incremento si el programa calcula el incremento mediante instrucciones dentro del cuerpo del ciclo, o si no se necesita un incremento. La expresión de incremento en un `for` actúa como si fuera una instrucción independiente al final del cuerpo del `for`. Por lo tanto, las expresiones

```
contador = contador + 1
contador += 1
++contador
contador++
```

son expresiones de incremento equivalentes en una instrucción `for`. Muchos programadores prefieren `contador++`, ya que es concisa y además un ciclo `for` evalúa su expresión de incremento *después* de la ejecución de su cuerpo; por ende, la forma de postincremento parece más natural. En este caso, la variable que se incrementa no aparece en una expresión más grande, por lo que los operadores de preincremento y postdecremento tienen en realidad el mismo efecto.

Error común de programación 5.4

Colocar un punto y coma inmediatamente a la derecha del paréntesis derecho del encabezado de un `for` convierte el cuerpo de ese `for` en una instrucción vacía. Por lo general esto es un error lógico.

Tip para prevenir errores 5.3

Los ciclos infinitos ocurren cuando la condición de continuación de ciclo en una instrucción de repetición nunca se vuelve `false`. *Para evitar esta situación en un ciclo controlado por un contador, debe asegurarse que la variable de control se incremente (o decremente) durante cada iteración del ciclo. En un ciclo controlado por centinela, asegúrese que el valor centinela se introduzca en algún momento dado.*

Colocar expresiones aritméticas en el encabezado de una instrucción `for`

Las porciones correspondientes a la inicialización, la condición de continuación de ciclo y el incremento de una instrucción `for` pueden contener expresiones aritméticas. Por ejemplo, suponga que `x = 2` y `y = 10`. Si `x` y `y` no se modifican en el cuerpo del ciclo, entonces la instrucción

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

es equivalente a la instrucción

```
for ( int j = 2; j <= 80; j += 5 )
```

El incremento de una instrucción `for` también puede ser *negativo*, en cuyo caso sería un *decremento* y el ciclo contaría en orden *descendente*.

Uso de la variable de control de una instrucción `for` *en el cuerpo de las instrucciones*

Con frecuencia, los programas muestran en pantalla el valor de la variable de control o lo utilizan en cálculos dentro del cuerpo del ciclo, pero este uso no es obligatorio. Por lo general, la variable de control se utiliza para controlar la repetición sin que se le mencione dentro del cuerpo del `for`.

Tip para prevenir errores 5.4

Aunque el valor de la variable de control puede cambiarse en el cuerpo de un ciclo `for`, *evite hacerlo, ya que esta práctica puede llevarlo a cometer errores sutiles.*

Diagrama de actividad de UML para la instrucción `for`

El diagrama de actividad de UML de la instrucción `for` es similar al de la instrucción `while` (figura 4.4). La figura 5.4 muestra el diagrama de actividad de la instrucción `for` de la figura 5.2. El diagrama hace

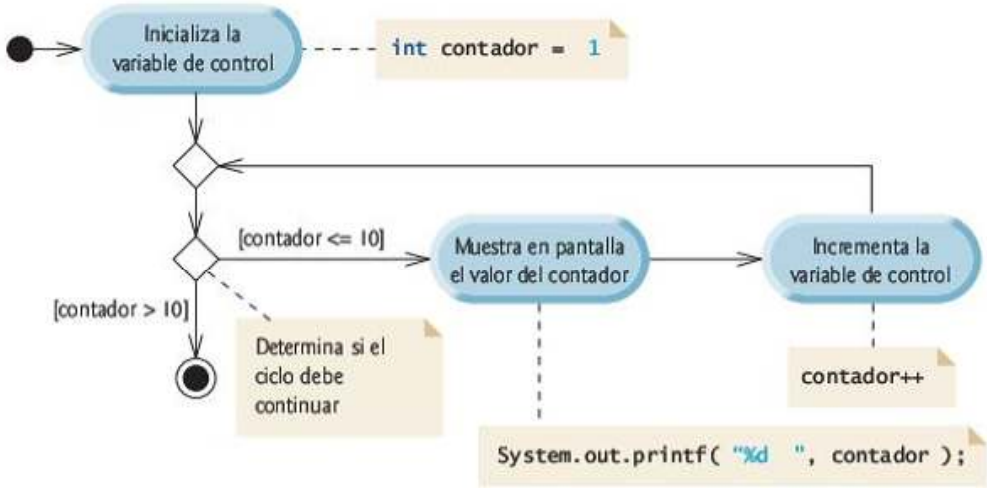


Fig. 5.4 | Diagrama de actividad de UML para la instrucción `for` de la figura 5.2.

evidente que la inicialización ocurre *sólo una vez antes* de evaluar la condición de continuación de ciclo por primera vez, y que el incremento ocurre *cada* vez que se realiza una iteración, *después* de que se ejecuta la instrucción del cuerpo.

5.4 Ejemplos sobre el uso de la instrucción for

Los siguientes ejemplos muestran técnicas para modificar la variable de control en una instrucción `for`. En cada caso, escribimos el encabezado `for` apropiado. Observe el cambio en el operador relacional para los ciclos que decrementan la variable de control.

a) Modificar la variable de control de 1 a 100 en incrementos de 1.

```
for ( int i = 1; i <= 100; i++ )
```

b) Modificar la variable de control de 100 a 1 en decrementos de 1.

```
for ( int i = 100; i >= 1; i-- )
```

c) Modificar la variable de control de 7 a 77 en incrementos de 7.

```
for ( int i = 7; i <= 77; i += 7 )
```

d) Modificar la variable de control de 20 a 2 en decrementos de 2.

```
for ( int i = 20; i >= 2; i -= 2 )
```

e) Modificar la variable de control con la siguiente secuencia de valores: 2, 5, 8, 11, 14, 17, 20.

```
for ( int i = 2; i <= 20; i += 3 )
```

f) Modificar la variable de control con la siguiente secuencia de valores: 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for ( int i = 99; i >= 0; i -= 11 )
```

Error común de programación 5.5

Utilizar el operador relacional incorrecto en la condición de continuación de un ciclo que cuente en forma regresiva (como usar `i <= 1` en vez de `i >= 1` en un ciclo que cuente en forma regresiva hasta llegar a 1) es por lo general un error lógico.

Aplicación: sumar los enteros pares del 2 al 20

Ahora consideremos dos aplicaciones de ejemplo que demuestran usos simples de la instrucción `for`. La aplicación de la figura 5.5 utiliza una instrucción `for` para sumar los enteros pares del 2 al 20 y guardar el resultado en una variable `int` llamada `total`.

```
1  // Fig. 5.5: Suma.java
2  // Sumar enteros con la instrucción for.
3
4  public class Suma
5  {
6     public static void main( String[] args )
7     {
8        int total = 0; // inicializa el total
9
```

Fig. 5.5 | Sumar enteros con la instrucción `for` (parte 1 de 2).

```
10         // total de los enteros pares del 2 al 20
11         for ( int numero = 2; numero <= 20; numero += 2 )
12            total += numero;
13
14         System.out.printf( "La suma es %d\n", total ); // muestra los resultados
15      } // fin de main
16   } // fin de la clase Suma
```

```
La suma es 110
```

Fig. 5.5 | Sumar enteros con la instrucción `for` (parte 2 de 2).

Las expresiones de *inicialización* e *incremento* pueden ser listas separadas por comas que nos permitan utilizar varias expresiones de inicialización, o varias expresiones de incremento. Por ejemplo, *aunque esto no se recomienda*, el cuerpo de la instrucción `for` en las líneas 11 y 12 de la figura 5.5 podría mezclarse con la porción del incremento del encabezado `for` mediante el uso de una coma, como se muestra a continuación:

```
for ( int numero = 2; numero <= 20; total += numero, numero += 2 )
   ; // instrucción vacía
```

Buena práctica de programación 5.1

Por legibilidad, limite el tamaño de los encabezados de las instrucciones de control a una sola línea, si es posible.

Aplicación: cálculo del interés compuesto

La siguiente aplicación utiliza la instrucción `for` para calcular el interés compuesto. Considere el siguiente problema:

> *Una persona invierte $1,000.00 en una cuenta de ahorro que produce el 5% de interés. Suponiendo que todo el interés se deposita en la cuenta, calcule e imprima el monto de dinero en la cuenta al final de cada año, durante 10 años. Use la siguiente fórmula para determinar los montos:*
>
> $a = p\,(1 + r)^n$
>
> *en donde*
>
> *p* es el monto que se invirtió originalmente (es decir, el monto principal)
> *r* es la tasa de interés anual (por ejemplo, use 0.05 para el 5%)
> *n* es el número de años
> *a* es la cantidad depositada al final del *n*-ésimo año.

La solución a este problema (figura 5.6) implica el uso de un ciclo que realiza los cálculos indicados para cada uno de los 10 años que el dinero permanece depositado. Las líneas 8 a 10 en el método `main` declaran las variables `double` llamadas `monto`, `principal` y `tasa`, e inicializan `principal` con `1000.0` y `tasa` con `0.05`. Java trata a las constantes de punto flotante, como `1000.0` y `0.05`, como de tipo `double`. De manera similar, Java trata a las constantes de números enteros, como `7` y `-22`, como de tipo `int`.

```
1  // Fig. 5.6: Interes.java
2  // Cálculo del interés compuesto con for.
3
4  public class Interes
5  {
```

Fig. 5.6 | Cálculo del interés compuesto con `for` (parte 1 de 2).

```
6     public static void main( String[] args )
7     {
8        double monto; // monto depositado al final de cada año
9        double principal = 1000.0; // monto inicial antes de los intereses
10       double tasa = 0.05; // tasa de interés
11
12       // muestra los encabezados
13       System.out.printf( "%s%20s\n", "Anio", "Monto en deposito" );
14
15       // calcula el monto en depósito para cada uno de diez años
16       for ( int anio = 1; anio <= 10; anio++ )
17       {
18          // calcula el nuevo monto para el año especificado
19          monto = principal * Math.pow( 1.0 + tasa, anio );
20
21          // muestra el año y el monto
22          System.out.printf( "%4d%,20.2f\n", anio, monto );
23       } // fin de for
24    } // fin de main
25 } // fin de la clase Interes
```

```
Anio     Monto en deposito
   1             1,050.00
   2             1,102.50
   3             1,157.63
   4             1,215.51
   5             1,276.28
   6             1,340.10
   7             1,407.10
   8             1,477.46
   9             1,551.33
  10             1,628.89
```

Fig. 5.6 | Cálculo del interés compuesto con `for` (parte 2 de 2).

Aplicar formato a las cadenas con anchuras de campo y justificación

La línea 13 imprime en pantalla los encabezados para las dos columnas de resultados. La primera columna muestra el año y la segunda, la cantidad depositada al final de ese año. Observe que utilizamos el especificador de formato `%20s` para mostrar en pantalla el objeto `String` `"Monto en deposito"`. El entero 20 después del `%` y el carácter de conversión `s` indican que el valor a imprimir debe mostrarse con una **anchura de campo** de 20; esto es, `printf` debe mostrar el valor con al menos 20 posiciones de caracteres. Si el valor a imprimir tiene una anchura menor a 20 posiciones de caracteres (en este ejemplo son 17 caracteres), el valor se **justifica a la derecha** en el campo de manera predeterminada. Si el valor `anio` a imprimir tuviera una anchura mayor a cuatro posiciones de caracteres, la anchura del campo se extendería a la derecha para dar cabida a todo el valor; esto desplazaría al campo `monto` a la derecha, con lo que se desacomodarían las columnas ordenadas de nuestros resultados tabulares. Para indicar que los valores deben imprimirse **justificados a la izquierda**, sólo hay que anteponer a la anchura de campo la **bandera de formato de signo negativo** (`-`) (por ejemplo, `%-20s`).

Realización del cálculo del interés

La instrucción `for` (líneas 16 a 23) ejecuta su cuerpo 10 veces, con lo cual la variable de control `anio` varía de 1 a 10, en incrementos de 1. Este ciclo termina cuando la variable de control `anio` se vuelve 11 (observe que anio representa a la *n* en el enunciado del problema).

Las clases proporcionan métodos que realizan tareas comunes sobre los objetos. De hecho, la mayoría de los métodos a llamar deben pertenecer a un objeto específico. Por ejemplo, para imprimir texto en la figura 5.6, la línea 13 llama al método `printf` en el objeto `System.out`. Muchas clases también cuentan con métodos que realizan tareas comunes y *no* requieren objetos. A estos métodos se les llama `static`. Por ejemplo, Java no incluye un operador de exponenciación, por lo que los diseñadores de la clase `Math` definieron el método `static` llamado `pow` para elevar un valor a una potencia. Para llamar a un método `static` debe especificar el nombre de la clase, seguido de un punto (`.`) y el nombre del método, como en

NombreClase.nombreMétodo(*argumentos*)

En el capítulo 6 aprenderá a implementar métodos `static` en sus propias clases.

Utilizamos el método `static` **`pow`** de la clase **`Math`** para realizar el cálculo del interés compuesto, en la figura 5.6. `Math.pow(`x`, `y`)` calcula el valor de x elevado a la $y^{\text{ésima}}$ potencia. El método recibe dos argumentos `double` y devuelve un valor `double`. La línea 19 realiza el cálculo $a = p(1 + r)^n$, en donde a es `monto`, p es `principal`, r es `tasa` y n es `anio`. La clase `Math` está definida en el paquete `java.lang`, por lo que no es necesario que la importe para usarla.

El cuerpo de la instrucción `for` contiene el cálculo `1.0 + tasa`, el cual aparece como argumento para el método `Math.pow`. De hecho, este cálculo produce el mismo resultado cada vez que se realiza una iteración en el ciclo, por lo tanto, repetir el cálculo en todas las iteraciones es un desperdicio.

Tip de rendimiento 5.1

En los ciclos, evite cálculos para los cuales el resultado nunca cambia; dichos cálculos por lo general deben colocarse antes del ciclo. Muchos de los sofisticados compiladores con optimización de la actualidad colocan este tipo de cálculos fuera de los ciclos en el código compilado.

Aplicar formato a números de punto flotante

Después de cada cálculo, la línea 22 imprime en pantalla el año y el monto depositado al final de ese año. El año se imprime en una anchura de campo de cuatro caracteres (según lo especificado por `%4d`). El monto se imprime como un número de punto flotante con el especificador de formato `%,20.2f`. La **bandera de formato coma** (`,`) indica que el valor de punto flotante debe imprimirse con un **separador de agrupamiento**. El separador que se utiliza realmente es específico de la configuración regional del usuario (es decir, el país). Por ejemplo, en Estados Unidos, el número se imprimirá usando comas para separar cada tres dígitos, y un punto decimal para la parte fraccionaria del número, como en 1,234.45. El número `20` en la especificación de formato indica que el valor debe imprimirse justificado a la derecha, con una anchura de campo de 20 caracteres. El `.2` especifica la precisión del número con formato; en este caso, el número se redondea a la centésima más cercana y se imprime con dos dígitos a la derecha del punto decimal.

Una advertencia sobre cómo mostrar valores redondeados

En este ejemplo declaramos las variables `monto`, `capital` y `tasa` de tipo `double`. Estamos tratando con partes fraccionales de dólares y, por ende, necesitamos un tipo que permita puntos decimales en sus valores. Por desgracia, los números de punto flotante pueden provocar problemas. He aquí una sencilla explicación de lo que puede salir mal al utilizar `double` (o `float`) para representar montos en dólares (suponiendo que los montos en dólares se muestran con dos dígitos a la derecha del punto decimal): Dos montos en dólares tipo `double` almacenados en la máquina podrían ser 14.234 (que por lo general se redondea a 14.23 para fines de mostrarlo en pantalla) y 18.763 (que por lo general se redondea a 18.67 para fines de mostrarlo en pantalla). Al sumar estos montos, producen una suma interna de 32.907, que por lo general se redondea a 32.91 para fines de mostrarlo en pantalla. Por lo tanto, sus resultados podrían aparecer como

```
  14.23
+ 18.67
-------
  32.91
```

pero una persona que sume los números individuales, como se muestran, esperaría que la suma fuera de 32.90. ¡Ha sido advertido!

Tip para prevenir errores 5.5

No utilice variables de tipo `double` *(o* `float`*) para realizar cálculos monetarios precisos. La imprecisión de los números de punto flotante puede provocar errores. En los ejercicios, aprenderá a usar enteros para realizar cálculos monetarios precisos. Java también cuenta con la clase* `java.math.BigDecimal` *para realizar cálculos monetarios precisos. Para obtener más información, visite la página* `download.oracle.com/javase/6/docs/api/java/math//BigDecimal.html`.

5.5 Instrucción de repetición do...while

La **instrucción de repetición do...while** es similar a la instrucción `while`. En ésta última el programa evalúa la condición de continuación de ciclo al principio, antes de ejecutar el cuerpo del ciclo; si la condición es falsa, el cuerpo *nunca* se ejecuta. La instrucción `do...while` evalúa la condición de continuación de ciclo *después* de ejecutar el cuerpo del ciclo; por lo tanto, *el cuerpo siempre se ejecuta por lo menos una vez*. Cuando termina una instrucción `do...while`, la ejecución continúa con la siguiente instrucción en la secuencia. La figura 5.7 utiliza una instrucción `do...while` (líneas 10 a la 14) para imprimir los números del 1 al 10.

```
1  // Fig. 5.7: PruebaDoWhile.java
2  // La instrucción de repetición do...while.
3
4  public class PruebaDoWhile
5  {
6     public static void main( String[] args )
7     {
8        int contador = 1; // inicializa contador
9
10       do
11       {
12          System.out.printf( "%d  ", contador );
13          ++contador;
14       } while ( contador <= 10 ); // fin de do...while
15
16       System.out.println(); // imprime una nueva línea
17    } // fin de main
18 } // fin de la clase PruebaDoWhile
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 5.7 | La instrucción de repetición `do...while`.

La línea 8 declara e inicializa la variable de control `contador`. Al entrar a la instrucción `do...while`, la línea 12 imprime el valor de `contador` y la 13 incrementa a `contador`. Después, el programa evalúa

la prueba de continuación de ciclo al *final* del mismo (línea 14). Si la condición es verdadera, el ciclo continúa a partir de la primera instrucción del cuerpo (línea 12). Si la condición es falsa, el ciclo termina y el programa continúa con la siguiente instrucción después del ciclo.

La figura 5.8 contiene el diagrama de actividad de UML para la instrucción do...while. Este diagrama hace evidente que la condición de continuación de ciclo no se evalúa sino hasta *después* que el ciclo ejecuta el estado de acción, por lo menos una vez. Compare este diagrama de actividad con el de la instrucción while (figura 4.4).

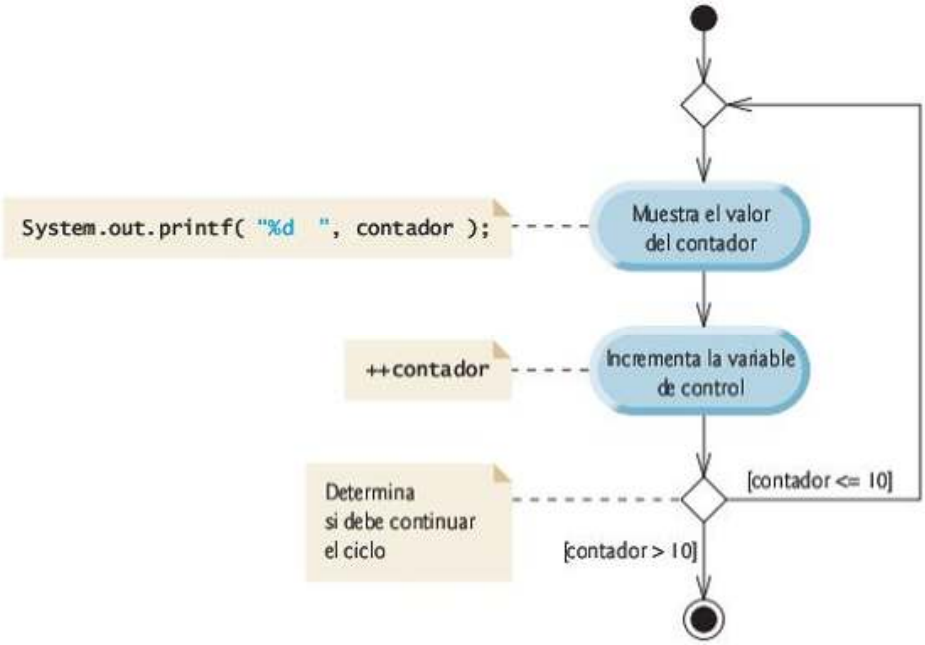


Fig. 5.8 | Diagrama de actividad de UML de la instrucción de repetición do...while.

No es necesario utilizar llaves en la estructura de repetición do...while si sólo hay una instrucción en el cuerpo. Sin embargo, la mayoría de los programadores incluyen las llaves para evitar la confusión entre las instrucciones while y do...while. Por ejemplo:

```
while ( condición )
```

es por lo general la primera línea de una instrucción while. Una instrucción do...while sin llaves, alrededor de un cuerpo con una sola instrucción, aparece así:

```
do
   instrucción
while ( condición );
```

lo cual puede ser confuso. Un lector podría malinterpretar la última línea [while (*condición*);], como una instrucción while que contiene una instrucción vacía (el punto y coma por sí solo). Por ende, la instrucción do...while con una instrucción en su cuerpo se escribe generalmente así:

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

Buena práctica de programación 5.2

Integre siempre las llaves en una instrucción do...while. Esto ayuda a eliminar la ambigüedad entre las instrucciones while y do...while que contienen sólo una instrucción.

5.6 Instrucción de selección múltiple switch

En el capítulo 4 hablamos sobre la instrucción de selección simple `if` y la instrucción de selección doble `if...else`. La **instrucción de selección múltiple switch** realiza distintas acciones, con base en los posibles valores de una **expresión entera constante** de tipo `byte`, `short`, `int` o `char`.

La clase LibroCalificaciones con la instrucción switch para contar las calificaciones A, B, C, D y F

La figura 5.9 contiene una versión mejorada de la clase `LibroCalificaciones` que presentamos en los capítulos 3 y 4. La nueva versión que presentamos ahora no sólo calcula el promedio de un conjunto de calificaciones numéricas introducidas por el usuario, sino que utiliza una instrucción `switch` para determinar si cada calificación es el equivalente de A, B, C, D o F, y para incrementar el contador de la calificación apropiada. La clase también imprime en pantalla un resumen del número de estudiantes que recibieron cada calificación. La figura 5.10 muestra las entradas y las salidas de ejemplo de la aplicación `PruebaLibroCalificaciones`, que utiliza la clase `LibroCalificaciones` para procesar un conjunto de calificaciones.

```
1  // Fig. 5.9: LibroCalificaciones.java
2  // La clase LibroCalificaciones usa la instrucción switch para contar las
   calificaciones de letras.
3  import java.util.Scanner; // el programa usa la clase Scanner
4
5  public class LibroCalificaciones
6  {
7     private String nombreDelCurso; // nombre del curso que representa este
                                     LibroCalificaciones
8     // las variables de instancia int se inicializan en 0 de manera predeterminada
9     private int total; // suma de las calificaciones
10    private int contadorCalif; // número de calificaciones introducidas
11    private int aCuenta; // cuenta de calificaciones A
12    private int bCuenta; // cuenta de calificaciones B
13    private int cCuenta; // cuenta de calificaciones C
14    private int dCuenta; // cuenta de calificaciones D
15    private int fCuenta; // cuenta de calificaciones F
16
17    // el constructor inicializa nombreDelCurso;
18    public LibroCalificaciones( String nombre )
19    {
20       nombreDelCurso = nombre; // inicializa nombreDelCurso
21    } // fin del constructor
22
23    // método para establecer el nombre del curso
24    public void establecerNombreDelCurso( String nombre )
25    {
26       nombreDelCurso = nombre; // almacena el nombre del curso
27    } // fin del método establecerNombreDelCurso
28
29    // método para obtener el nombre del curso
30    public String obtenerNombreDelCurso()
31    {
```

Fig. 5.9 | Clase `LibroCalificaciones` que utiliza una instrucción `switch` para contar las calificaciones A, B, C, D y F (parte 1 de 3).

```
32          return nombreDelCurso;
33       } // fin del método obtenerNombreDelCurso
34
35       // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
36       public void mostrarMensaje()
37       {
38          // obtenerNombreDelCurso obtiene el nombre del curso
39          System.out.printf( "Bienvenido al libro de calificaciones para\n%s!\n\n",
40             obtenerNombreDelCurso() );
41       } // fin del método mostrarMensaje
42
43       // introduce un número arbitrario de calificaciones del usuario
44       public void introducirCalif()
45       {
46          Scanner entrada = new Scanner( System.in );
47
48          int calificacion; // calificación introducida por el usuario
49
50          System.out.printf( "%s\n%s\n   %s\n   %s\n",
51             "Escriba las calificaciones enteras en el rango de 0 a 100.",
52             "Escriba el indicador de fin de archivo para terminar la entrada:",
53             "En UNIX/Linux/Mac OS X escriba <Ctrl> d y despues oprima Intro",
54             "En Windows escriba <Ctrl> z y despues oprima Intro" );
55
56          // itera hasta que el usuario introduzca el indicador de fin de archivo
57          while ( entrada.hasNext() )
58          {
59             calificacion = entrada.nextInt(); // lee calificacion
60             total += calificacion; // suma calificacion a total
61             ++contadorCalif; // incrementa el número de calificaciones
62
63             // llama al método para incrementar el contador apropiado
64             incrementarContadorCalifLetra( calificacion );
65          } // fin de while
66       } // fin del método introducirCalif
67
68       // suma 1 al contador apropiado para la calificación especificada
69       private void incrementarContadorCalifLetra( int calificacion )
70       {
71          // determina cuál calificación se introdujo
72          switch ( calificacion / 10 )
73          {
74             case 9:  // calificacion está entre 90
75             case 10: // y 100, inclusive
76                ++aCuenta; // incrementa aCuenta
77                break; // necesaria para salir del switch
78
79             case 8: // calificacion está entre 80 y 89
80                ++bCuenta; // incrementa bCuenta
81                break; // sale del switch
82
```

Fig. 5.9 | Clase `LibroCalificaciones` que utiliza una instrucción `switch` para contar las calificaciones A, B, C, D y F (parte 2 de 3).

```
83          case 7: // calificacion está entre 70 y 79
84             ++cCuenta; // incrementa cCuenta
85             break; // sale del switch
86
87          case 6: // calificacion está entre 60 y 69
88             ++dCuenta; // incrementa dCuenta
89             break; // sale del switch
90
91          default: // calificacion es menor que 60
92             ++fCuenta; // incrementa fCuenta
93             break; // opcional; de todas formas sale del switch
94       } // fin de switch
95    } // fin del método incrementarContadorCalifLetra
96
97    // muestra un reporte con base en las calificaciones introducidas por el usuario
98    public void mostrarReporteCalif()
99    {
100      System.out.println( "\nReporte de calificaciones:" );
101
102      // si el usuario introdujo por lo menos una calificación...
103      if ( contadorCalif != 0 )
104      {
105         // calcula el promedio de todas las calificaciones introducidas
106         double promedio = (double) total / contadorCalif;
107
108         // imprime resumen de resultados
109         System.out.printf( "El total de las %d calificaciones introducidas es %d\n",
110            contadorCalif, total );
111         System.out.printf( "El promedio de la clase es %.2f\n", promedio );
112         System.out.printf( "%s\n%s%d\n%s%d\n%s%d\n%s%d\n%s%d\n",
113            "Numero de estudiantes que recibieron cada calificacion:",
114            "A: ", aCuenta,   // muestra el número de calificaciones A
115            "B: ", bCuenta,   // muestra el número de calificaciones B
116            "C: ", cCuenta,   // muestra el número de calificaciones C
117            "D: ", dCuenta,   // muestra el número de calificaciones D
118            "F: ", fCuenta ); // muestra el número de calificaciones F
119      } // fin de if
120      else // no se introdujeron calificaciones, por lo que imprime el mensaje apropiado
121         System.out.println( "No se introdujeron calificaciones" );
122   } // fin del método mostrarReporteCalif
123 } // fin de la clase LibroCalificaciones
```

Fig. 5.9 | Clase `LibroCalificaciones` que utiliza una instrucción `switch` para contar las calificaciones A, B, C, D y F (parte 3 de 3).

Al igual que las versiones anteriores de la clase, `LibroCalificaciones` (figura 5.9) declara la variable de instancia `nombreDelCurso` (línea 7) y contiene los métodos `establecerNombreDelCurso` (líneas 24 a 27), `obtenerNombreDelCurso` (líneas 30 a 33) y `mostrarMensaje` (líneas 36 a 41), que establecen el nombre del curso, lo almacenan y muestran un mensaje de bienvenida al usuario, respectivamente. La clase también contiene un constructor (líneas 18 a 21) que inicializa el nombre del curso.

La clase `LibroCalificaciones` también declara las variables de instancia `total` (línea 9) y `contadorCalif` (línea 10), que llevan la cuenta de la suma de las calificaciones introducidas por el usuario y el número de calificaciones introducidas, respectivamente. Las líneas 11 a 15 declaran las variables contador para cada categoría de calificaciones. La clase `LibroCalificaciones` mantiene a `total`, `contadorCalif` y

a los cinco contadores de las letras de calificación como variables de instancia, de manera que estas variables puedan utilizarse o modificarse en cualquiera de los métodos de la clase. El constructor de la clase (líneas 18 a 21) establece sólo el nombre del curso, ya que las siete variables de instancia restantes son de tipo `int` y se inicializan con 0, de manera predeterminada.

La clase `LibroCalificaciones` (figura 5.9) contiene tres métodos adicionales: `introducirCalif`, `incrementarContadorCalifLetra` y `mostrarReporteCalif`. El método `introducirCalif` (líneas 44 a 66) lee un número arbitrario de calificaciones enteras del usuario mediante el uso de la repetición controlada por un centinela, y actualiza las variables de instancia `total` y `contadorCalif`. Este método llama al método `incrementarContadorCalifLetra` (líneas 69 a 95) para actualizar el contador de letra de calificación apropiado para cada calificación introducida. El método `mostrarReporteCalif` (líneas 98 a 122) imprime en pantalla un reporte que contiene el total de todas las calificaciones introducidas, el promedio de las mismas y el número de estudiantes que recibieron cada letra de calificación. Examinaremos estos métodos con más detalle.

El método `introducirCalif`

La línea 48 en el método `introducirCalif` declara la variable `calificacion` que almacenará la entrada del usuario. Las líneas 50 a 54 piden al usuario que introduzca calificaciones enteras y escriba el indicador de fin de archivo para terminar la entrada. El **indicador de fin de archivo** es una combinación de teclas dependiente del sistema, que el usuario introduce para indicar que no hay más datos que introducir. En el capítulo 17 (en el sitio Web), Archivos, flujos y serialización de objetos, veremos cómo se utiliza el indicador de fin de archivo cuando un programa lee su entrada desde un archivo.

En los sistemas UNIX/Linux/Mac OS X, el fin de archivo se introduce escribiendo la secuencia

```
<Ctrl> d
```

en una línea por sí sola. Esta notación significa que hay que oprimir al mismo tiempo la tecla *Ctrl* y la tecla *d*. En los sistemas Windows, para introducir el fin de archivo se escribe

```
<Ctrl> z
```

[*Nota:* en algunos sistemas, es necesario oprimir *Intro* después de escribir la secuencia de teclas de fin de archivo. Además, Windows por lo general muestra los caracteres `^Z` en la pantalla cuando se escribe el indicador de fin de archivo, como se muestra en la salida de la figura 5.10].

Tip de portabilidad 5.1

Las combinaciones de teclas para introducir el fin de archivo son dependientes del sistema.

La instrucción `while` (líneas 57 a 65) obtiene la entrada del usuario. La condición en la línea 57 llama al método `hasNext` de `Scanner` para determinar si hay más datos que introducir. Este método devuelve el valor `boolean true` si hay más datos; en caso contrario, devuelve `false`. Después, el valor devuelto se utiliza como el valor de la condición en la instrucción `while`. El método `hasNext` devuelve `false` una vez que el usuario escribe el indicador de fin de archivo.

La línea 59 recibe como entrada el valor de una calificación del usuario. La línea 60 suma `calificacion` a `total`. La línea 61 incrementa `contadorCalif`. El método `mostrarReporteCalif` de la clase utiliza estas variables para calcular el promedio de las calificaciones. La línea 64 llama al método `incrementarContadorCalifLetra` de la clase (declarado en las líneas 69 a 95) para incrementar el contador de letra de calificación apropiado, con base en la calificación numérica introducida.

El método `incrementarContadorCalifLetra`

El método `incrementarContadorCalifLetra` contiene una instrucción `switch` (líneas 72 a 94) que determina cuál contador se debe incrementar. En este ejemplo, suponemos que el usuario introduce una calificación válida en el rango de 0 a 100. Una calificación en el rango de 90 a 100 representa la A;

de 80 a 89, la B; de 70 a 79, la C; de 60 a 69, la D y de 0 a 59, la F. La instrucción `switch` consiste en un bloque que contiene una secuencia de **etiquetas case** y una instrucción **case default** opcional. Estas etiquetas se utilizan en este ejemplo para determinar cuál contador se debe incrementar, con base en la calificación.

Cuando el flujo de control llega al `switch`, el programa evalúa la expresión entre paréntesis (`calificacion / 10`) que va después de la palabra clave `switch`. A esto se le conoce como la **expresión de control** de la instrucción `switch`. El programa compara el valor de la expresión de control (que se debe evaluar como un valor entero de tipo `byte`, `char`, `short` o `int`) con cada una de las etiquetas `case`. La expresión de control de la línea 72 realiza la división entera, que *trunca la parte fraccionaria* del resultado. Por ende, cuando dividimos un valor en el rango de 0 a 100 entre 10, el resultado es siempre un valor de 0 a 10. Utilizamos varios de estos valores en nuestras etiquetas `case`. Por ejemplo, si el usuario introduce el entero `85`, la expresión de control se evalúa como 8. La instrucción `switch` compara a `8` con cada etiqueta `case`. Si ocurre una coincidencia (`case 8:` en la línea 79), el programa ejecuta las instrucciones para esa instrucción `case`. Para el entero `8`, la línea 80 incrementa a `bCuenta`, ya que una calificación entre 80 y 89 es una B. La **instrucción break** (línea 81) hace que el control del programa proceda con la primera instrucción después del `switch`; en este programa, llegamos al final del cuerpo del método `incrementarContadorCalifLetra`, por lo que el método termina y el control regresa a la línea 65 en el método `introducirCalif` (la primera línea después de la llamada a `incrementarContadorCalifLetra`). Esta línea marca el fin del cuerpo de un ciclo `while`, por lo tanto el control fluye hacia la condición del `while` (línea 57) para determinar si el ciclo debe seguir ejecutándose.

Las etiquetas `case` en nuestro `switch` evalúan explícitamente los valores `10`, `9`, `8`, `7` y `6`. Observe los casos en las líneas 74 y 75, que evalúan los valores `9` y `10` (los cuales representan la calificación A). Al listar las etiquetas case en forma consecutiva, sin instrucciones entre ellas, pueden ejecutar el mismo conjunto de instrucciones; cuando la expresión de control se evalúa como `9` o `10`, se ejecutan las instrucciones de las líneas 76 y 77. La instrucción `switch` no cuenta con un mecanismo para evaluar rangos de valores, por ello cada valor que deba evaluarse se tiene que enumerar en una etiqueta `case` separada. Cada `case` puede tener varias instrucciones. La instrucción `switch` es distinta de otras instrucciones de control, en cuanto a que *no* requiere llaves alrededor de varias instrucciones en cada `case`.

Sin instrucciones `break`, cada vez que ocurre una coincidencia en el `switch`, se ejecutan las instrucciones para ese `case` y los subsiguientes, hasta encontrar una instrucción `break` o el final del `switch`. A menudo a esto se le conoce como que las etiquetas case "se pasarían" hacia las instrucciones en las etiquetas `case` subsiguientes. (Esta característica es perfecta para escribir un programa conciso, que muestre la canción iterativa "Los Doce Días de Navidad" en el ejercicio 5.29).

Error común de programación 5.6

Olvidar una instrucción `break` cuando se necesita una en una instrucción `switch` es un error lógico.

Si no ocurre una coincidencia entre el valor de la expresión de control y una etiqueta `case`, se ejecuta el caso `default` (líneas 91 a 93). Utilizamos el caso `default` en este ejemplo para procesar todos los valores de la expresión de control que sean menores que 6; esto es, todas las calificaciones de reprobado. Si no ocurre una coincidencia y la instrucción `switch` no contiene un caso `default`, el control del programa simplemente continúa con la primera instrucción después de la instrucción `switch`.

La clase `PruebaLibroCalificaciones` *para demostrar la clase* `LibroCalificaciones`

La clase `PruebaLibroCalificaciones` (figura 5.10) crea un objeto `LibroCalificaciones` (líneas 10 y 11). La línea 13 invoca el método `mostrarMensaje` del objeto para imprimir en pantalla un mensaje de bienvenida para el usuario. La línea 14 invoca el método `introducirCalif` del objeto para leer un conjunto de calificaciones del usuario y llevar el registro de la suma de todas las calificaciones introducidas, así como el número de calificaciones. Recuerde que el método `introducirCalif` también llama al método `incrementarContadorCalifLetra` para llevar el registro del número de estudiantes que reci-

bieron cada letra de calificación. La línea 15 invoca el método `mostrarReporteCalif` de la clase `LibroCalificaciones`, el cual imprime en pantalla un reporte con base en las calificaciones introducidas (como en la ventana de entrada/salida en la figura 5.10). La línea 103 de la clase `LibroCalificaciones` (figura 5.9) determina si el usuario introdujo por lo menos una calificación; esto evita la división entre cero. De ser así, la línea 106 calcula el promedio de las calificaciones. A continuación, las líneas 109 a 118 imprimen en pantalla el total de todas las calificaciones, el promedio de la clase y el número de estudiantes que recibieron cada letra de calificación. Si no se introdujeron calificaciones, la línea 121 imprime en pantalla un mensaje apropiado. Los resultados en la figura 5.10 muestran un reporte de calificaciones de ejemplo, con base en 10 calificaciones.

```
1  // Fig. 5.10: PruebaLibroCalificaciones.java
2  // Crea un objeto LibroCalificaciones, introduce las calificaciones y muestra
   un reporte.
3
4  public class PruebaLibroCalificaciones
5  {
6     public static void main( String[] args )
7     {
8        // crea un objeto LibroCalificaciones llamado miLibroCalificaciones y
9        // pasa el nombre del curso al constructor
10       LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
11          "CS101 Introduccion a la programacion en Java" );
12
13       miLibroCalificaciones.mostrarMensaje(); // muestra un mensaje de bienvenida
14       miLibroCalificaciones.introducirCalif(); // lee calificaciones del usuario
15       miLibroCalificaciones.mostrarReporteCalif(); // muestra reporte con base en
                                                      las calificaciones
16    } // fin de main
17 } // fin de la clase PruebaLibroCalificaciones
```

```
Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en Java!

Escriba las calificaciones enteras en el rango de 0 a 100.
Escriba el indicador de fin de archivo para terminar la entrada:
   En UNIX/Linux/Mac OS X escriba <ctrl> d y despues oprima Intro
   En Windows escriba <ctrl> z y despues oprima Intro
99
92
45
57
63
71
76
85
90
100
^Z

Reporte de calificaciones:
El total de las 10 calificaciones introducidas es 778
El promedio de la clase es 77.80
```

Fig. 5.10 | Crear un objeto `LibroCalificaciones`, introducir calificaciones y mostrar el reporte de calificaciones (parte 1 de 2).

```
Numero de estudiantes que recibieron cada calificacion:
A: 4
B: 1
C: 2
D: 1
F: 2
```

Fig. 5.10 | Crear un objeto `LibroCalificaciones`, introducir calificaciones y mostrar el reporte de calificaciones (parte 2 de 2).

La clase `PruebaLibroCalificaciones` (figura 5.10) no llama de manera directa al método `incrementarContadorCalifLetra` de `LibroCalificaciones` (líneas 69 a 95 de la figura 5.9). Este método lo utiliza exclusivamente el método `introducirCalif` de la clase `LibroCalificaciones` para actualizar el contador de la calificación de letra apropiado, a medida que el usuario introduce cada nueva calificación. El método `incrementarContadorCalifLetra` existe sólo para dar soporte a las operaciones de los demás métodos de la clase `LibroCalificaciones`, por lo cual se declara como `private`.

Observación de ingeniería de software 5.2

En el capítulo 3 vimos que los métodos que se declaran con el modificador de acceso `private` pueden llamarse sólo por otros de la clase en la que están declarados los métodos `private`. Dichos métodos se conocen comúnmente como ***métodos utilitarios o métodos ayudantes****, debido a que por lo general sólo pueden llamarse mediante otros de esa clase y se utilizan para dar soporte a la operación de esos métodos.*

Diagrama de actividad de UML de la instrucción `switch`

La figura 5.11 muestra el diagrama de actividad de UML para la instrucción `switch` general. La mayoría de las instrucciones `switch` utilizan una instrucción `break` en cada `case` para terminar la instrucción `switch` después de procesar el `case`. La figura 5.11 enfatiza esto al incluir instrucciones `break` en el dia-

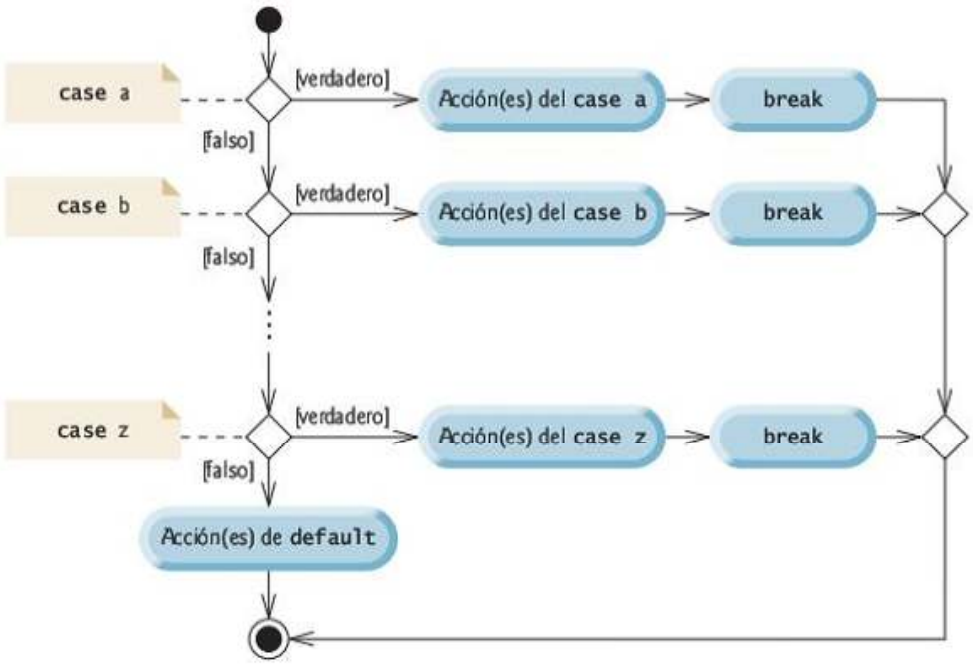


Fig. 5.11 | Diagrama de actividad de UML de la instrucción `switch` de selección múltiple con instrucciones `break`.

grama de actividad. El cual hace evidente que break al final de una etiqueta case hace que el control salga de la instrucción switch de inmediato.

No se requiere una instrucción break para la última etiqueta case del switch (o para el caso default opcional, cuando aparece al último), ya que la ejecución continúa con la siguiente instrucción que va después del switch.

Observación de ingeniería de software 5.3

Proporcione un caso default en las instrucciones switch. Al incluir un caso default usted puede enfocarse en la necesidad de procesar las condiciones excepcionales.

Buena práctica de programación 5.3

Aunque cada case y el caso default en una instrucción switch pueden ocurrir en cualquier orden, es conveniente colocar la etiqueta default al último. Cuando el caso default se lista al último, no se requiere el break para ese caso.

Notas sobre la expresión en cada `case` *de un* `switch`

Cuando utilice la instrucción switch, recuerde que cada case debe contener una expresión entera constante; es decir, cualquier combinación de constantes enteras que se evalúen como un valor entero constante (por ejemplo, –7, 0 o 221). Una constante entera es tan solo un valor entero. Además, puede utilizar **constantes tipo carácter**: caracteres específicos entre comillas sencillas, como 'A', '7' o '$', las cuales representan los valores enteros de los caracteres y las constantes enum (que presentaremos en la sección 6.10). (En el apéndice B se muestran los valores enteros de los caracteres en el conjunto de caracteres ASCII, que es un subconjunto del conjunto de caracteres Unicode utilizado por Java).

La expresión en cada case también puede ser una **variable constante**: una variable que contiene un valor que no cambia durante todo el programa. Ésta se declara mediante la palabra clave final (que describiremos en el capítulo 6). Java tiene una característica conocida como *enumeraciones*, que también presentaremos en el capítulo 6. Las constantes de enumeración también pueden utilizarse en etiquetas case. En el capítulo 10, Programación orientada a objetos: polimorfismo, presentaremos una manera más elegante de implementar la lógica del switch: utilizaremos una técnica llamada *polimorfismo* para crear programas que a menudo son más legibles, fáciles de mantener y de extender que los programas que utilizan lógica de switch.

Uso de objetos `String` *en instrucciones* `switch` *(Nuevo en Java SE 7)*

A partir de Java SE 7, es posible utilizar objetos String en la expresión de control de una instrucción switch, y en las etiquetas case. Por ejemplo, tal vez quiera usar el nombre de una ciudad para obtener el correspondiente código postal. Suponiendo que ciudad y codigoPostal sean variables String, la siguiente instrucción switch realiza esta tarea para tres ciudades:

```
switch ( ciudad )
{
    case "Maynard":
        codigoPostal = "01754";
        break;
    case "Marlborough":
        codigoPostal = "01752";
        break;
    case "Framingham":
        codigoPostal = "01701";
        break;
} // fin del switch
```

5.7 Instrucciones break y continue

Además de las instrucciones de selección y repetición, Java cuenta con las instrucciones `break` y `continue` (que presentamos en esta sección y en el apéndice O) para alterar el flujo de control. En la sección anterior mostramos cómo puede utilizarse la instrucción `break` para terminar la ejecución de una instrucción `switch`. En esta sección veremos cómo utilizar `break` en las instrucciones de repetición.

Instrucción `break`

Cuando `break` se ejecuta en una instrucción `while`, `for`, `do...while`, o `switch`, ocasiona la salida inmediata de esa instrucción. La ejecución continúa con la primera instrucción después de la instrucción de control. Los usos comunes de `break` son para escapar anticipadamente del ciclo, o para omitir el resto de una instrucción `switch` (como en la figura 5.9). La figura 5.12 demuestra el uso de una instrucción `break` para salir de un ciclo `for`.

```
1  // Fig. 5.12: PruebaBreak.java
2  // La instrucción break para salir de una instrucción for.
3  public class PruebaBreak
4  {
5     public static void main( String[] args )
6     {
7        int cuenta; // la variable de control también se usa cuando termina el ciclo
8
9        for ( cuenta = 1; cuenta <= 10; cuenta++ ) // itera 10 veces
10       {
11          if ( cuenta == 5 ) // si cuenta es 5,
12             break;          // termina el ciclo
13
14          System.out.printf( "%d ", cuenta );
15       } // fin de for
16
17       System.out.printf( "\nSalio del ciclo en cuenta = %d\n", cuenta );
18    } // fin de main
19 } // fin de la clase PruebaBreak
```

```
1 2 3 4
Salio del ciclo en cuenta = 5
```

Fig. 5.12 | Instrucción `break` para salir de una instrucción `for`.

Cuando la instrucción `if` anidada en las línea 11 y 12 dentro de la instrucción `for` (líneas 9 a 15) determina que `cuenta` es 5, se ejecuta la instrucción `break` en la línea 12. Esto termina la instrucción `for` y el programa continúa a la línea 17 (justo después de la instrucción `for`), la cual muestra un mensaje indicando el valor de la variable de control cuando terminó el ciclo. El ciclo ejecuta su cuerpo por completo sólo cuatro veces en vez de 10.

Instrucción `continue`

Cuando la instrucción `continue` se ejecuta en una instrucción `while`, `for` o `do...while`, omite las instrucciones restantes en el cuerpo del ciclo y continúa con la siguiente iteración del ciclo. En las instrucciones `while` y `do...while`, la aplicación evalúa la prueba de continuación de ciclo justo después de que se ejecuta la instrucción `continue`. En una instrucción `for` se ejecuta la expresión de incremento y después el programa evalúa la prueba de continuación de ciclo.

```
1  // Fig. 5.13: PruebaContinue.java
2  // Instrucción continue para terminar una iteración de una instrucción for.
3  public class PruebaContinue
4  {
5     public static void main( String[] args )
6     {
7        for ( int cuenta = 1; cuenta <= 10; cuenta++ ) // itera 10 veces
8        {
9           if ( cuenta == 5 ) // si cuenta es 5,
10             continue; // omite el resto del código en el ciclo
11
12          System.out.printf( "%d ", cuenta );
13       } // fin de for
14
15       System.out.println( "\nSe uso continue para omitir imprimir 5" );
16    } // fin de main
17 } // fin de la clase PruebaContinue
```

```
1 2 3 4 6 7 8 9 10
Se uso continue para omitir imprimir 5
```

Fig. 5.13 | Instrucción `continue` para terminar una iteración de una instrucción `for`.

La figura 5.13 utiliza la instrucción `continue` para omitir la instrucción de la línea 12 cuando la instrucción `if` anidada (línea 9) determina que el valor de cuenta es 5. Cuando se ejecuta la instrucción `continue`, el control del programa continúa con el incremento de la variable de control en la instrucción `for` (línea 7).

En la sección 5.3 declaramos que la instrucción `while` puede utilizarse, en la mayoría de los casos, en lugar de `for`. Esto no es verdad cuando la expresión de incremento en `while` va después de una instrucción `continue`. En este caso, el incremento no se ejecuta antes de que el programa evalúe la condición de continuación de repetición, por lo que `while` no se ejecuta de la misma manera que `for`.

Observación de ingeniería de software 5.4

Algunos programadores sienten que las instrucciones `break` y `continue` infringen la programación estructurada. Ya que pueden lograrse los mismos efectos con las técnicas de programación estructurada, estos programadores prefieren no utilizar instrucciones `break` o `continue`.

Observación de ingeniería de software 5.5

Existe una tensión entre lograr la ingeniería de software de calidad y lograr el software con mejor desempeño. A menudo, una de estas metas se logra a expensas de la otra. Para todas las situaciones excepto las que demanden el mayor rendimiento, aplique la siguiente regla empírica: primero, asegúrese de que su código sea simple y correcto; después hágalo rápido y pequeño, pero sólo si es necesario.

5.8 Operadores lógicos

Cada una de las instrucciones `if`, `if...else`, `while`, `do...while` y `for` requieren una condición para determinar cómo continuar con el flujo de control de un programa. Hasta ahora sólo hemos estudiado las condiciones simples, como `cuenta <= 10`, `numero != valorCentinela` y `total > 1000`. Las condiciones simples se expresan en términos de los operadores relacionales >, <, >= y <=, y los operadores de igualdad == y !=; cada expresión evalúa sólo una condición. Para evaluar condiciones múltiples en el

proceso de tomar una decisión, ejecutamos estas pruebas en instrucciones separadas o en instrucciones `if` o `if...else` anidadas. En ocasiones, las instrucciones de control requieren condiciones más complejas para determinar el flujo de control de un programa.

Los **operadores lógicos** de Java nos permiten formar condiciones más complejas, al *combinar* las condiciones simples. Los operadores lógicos son && (AND condicional), || (OR condicional), & (AND lógico booleano), | (OR inclusivo lógico booleano), ^ (OR exclusivo lógico booleano) y ! (NOT lógico). [*Nota:* los operadores &, | y ^ son también operadores a nivel de bits cuando se aplican a operandos enteros. En el apéndice N hablaremos sobre los operadores a nivel de bits].

Operador AND (&&) condicional

Suponga que deseamos asegurar en cierto punto de un programa que dos condiciones sean *ambas* verdaderas, antes de elegir cierta ruta de ejecución. En este caso, podemos utilizar el operador **&& (AND condicional)** de la siguiente manera:

```
if ( genero == FEMENINO && edad >= 65 )
   ++mujeresMayores;
```

Esta instrucción `if` contiene dos condiciones simples. La condición `genero == FEMENINO` compara la variable `genero` con la constante `FEMENINO` para determinar si una persona es mujer. La condición `edad >= 65` podría evaluarse para determinar si una persona es un ciudadano mayor. La instrucción `if` considera la condición combinada

```
genero == FEMENINO && edad >= 65
```

la cual es verdadera si, y sólo si ambas condiciones simples son verdaderas. En este caso, el cuerpo de la instrucción `if` incrementa a `mujeresMayores` en 1. Si una o ambas condiciones simples son falsas, el programa omite el incremento. Algunos programadores consideran que la condición combinada anterior es más legible si se agregan paréntesis redundantes, como por ejemplo:

```
( genero == FEMENINO ) && ( edad >= 65 )
```

La tabla de la figura 5.14 sintetiza el uso del operador &&. Esta tabla muestra las cuatro combinaciones posibles de valores `false` y `true` para *expresión1* y *expresión2*. A dichas tablas se les conoce como **tablas de verdad**. Java evalúa todas las expresiones que incluyen operadores relacionales, de igualdad o lógicos como `true` o `false`.

expresión1	expresión2	expresión1 && expresión2
false	false	false
false	true	false
true	false	false
true	true	true

Fig. 5.14 | Tabla de verdad del operador && (AND condicional).

Operador OR condicional (||)

Ahora suponga que deseamos asegurar que *cualquiera* o *ambas* condiciones sean verdaderas antes de elegir cierta ruta de ejecución. En este caso, utilizamos el operador || (**OR condicional**), como se muestra en el siguiente segmento de un programa:

```
if ( ( promedioSemestre >= 90 ) || ( examenFinal >= 90 ) )
   System.out.println ( "La calificacion del estudiante es A" );
```

Esta instrucción también contiene dos condiciones simples. La condición `promedioSemestre >= 90` se evalúa para determinar si el estudiante merece una A en el curso, debido a que tuvo un sólido rendimiento a lo largo del semestre. La condición `examenFinal >= 90` se evalúa para determinar si el estudiante merece una A en el curso debido a un desempeño sobresaliente en el examen final. Después, la instrucción `if` considera la condición combinada

```
( promedioSemestre >= 90 ) || ( examenFinal >= 90 )
```

y otorga una A al estudiante si *una* o *ambas* de las condiciones simples son verdaderas. La única vez que no se imprime el mensaje "`La calificación del estudiante es A`" es cuando ambas condiciones simples son *falsas*. La figura 5.15 es una tabla de verdad para el operador OR condicional (`||`). El operador `&&` tiene mayor precedencia que el operador `||`. Ambos operadores se asocian de izquierda a derecha.

expresión1	expresión2	expresión1 || expresión2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 5.15 | Tabla de verdad del operador `||` (OR condicional).

Evaluación en corto circuito de condiciones complejas

Las partes de una expresión que contienen los operadores `&&` o `||` se evalúan *sólo* hasta que se sabe si la condición es verdadera o falsa. Por ende, la evaluación de la expresión

```
( genero == FEMENINO ) && ( edad >= 65 )
```

se detiene de inmediato si `genero` no es igual que `FEMENINO` (es decir, que toda la expresión es `false`) y continúa si `genero` *es* igual que `FEMENINO` (es decir, toda la expresión podría ser aún `true` si la condición `edad >= 65` es `true`). Esta característica de las expresiones AND y OR condicional se conoce como **evaluación en corto circuito**.

Error común de programación 5.7

En las expresiones que utilizan el operador `&&`, una condición (a la cual le llamamos condición dependiente) puede requerir que otra condición sea verdadera para que la evaluación de la condición dependiente tenga significado. En este caso, la condición dependiente debe colocarse después de la otra condición, o podría ocurrir un error. Por ejemplo, en la expresión `(i != 0) && (10/i == 2)`, la segunda condición debe aparecer después de la primera, o podría ocurrir un error de división entre cero.

Operadores AND lógico booleano (`&`) y OR inclusivo lógico booleano (`|`)

Los **operadores AND lógico booleano** (`&`) y **OR inclusivo lógico booleano** (`|`) funcionan en forma idéntica a los operadores `&&` y `||`, excepto que los operadores `&` y `|` siempre evalúan ambos operandos (es decir, no realizan una evaluación en corto circuito). Por lo tanto, la expresión

```
( genero == 1 ) & ( edad >= 65 )
```

evalúa `edad >= 65`, *sin importar* que `genero` sea igual o no que `1`. Esto es útil si el operando derecho del operador AND lógico booleano o del OR inclusivo lógico booleano tiene un **efecto secundario** requerido: la modificación del valor de una variable. Por ejemplo, la expresión

```
( cumpleanios == true ) | ( ++edad >= 65 )
```

garantiza que se evalúe la condición ++edad >= 65. Por ende, la variable edad se incrementa sin importar que la expresión total sea `true` o `false`.

Tip para prevenir errores 5.6

Por cuestión de claridad, evite las expresiones con efectos secundarios en las condiciones. Los efectos secundarios quizás tengan una apariencia inteligente, pero pueden hacer que el código sea más difícil de entender y podrían llegar a producir errores lógicos sutiles.

OR exclusivo lógico booleano (^)

Una condición simple que contiene el operador OR **exclusivo lógico booleano** (^) es `true` *si y sólo si uno de sus operandos es* `true` *y el otro es* `false`. Si ambos operandos son `true` o si los dos son `false`, toda la condición es `false`. La figura 5.16 es una tabla de verdad para el operador OR exclusivo lógico booleano (^). También se garantiza que este operador evaluará *ambos* operandos.

expresión1	expresión2	expresión1 ^ expresión2
false	false	false
false	true	true
true	false	true
true	true	false

Fig. 5.16 | Tabla de verdad del operador ^ (OR exclusivo lógico booleano).

Operador lógico de negación (!)

El operador ! (**NOT lógico**, también conocido como **negación lógica** o **complemento lógico**) "invierte" el significado de una condición. A diferencia de los operadores lógicos &&, ||, &, | y ^, que son operadores *binarios* que combinan dos condiciones, el operador lógico de negación es un operador *unario* que sólo tiene una condición como operando. Este operador se coloca *antes* de una condición para elegir una ruta de ejecución si la condición original (sin el operador lógico de negación) es `false`, como en el siguiente segmento de código:

```
if ( ! ( calificacion == valorCentinela ) )
   System.out.printf( "La siguiente calificación es %d\n", calificacion );
```

que ejecuta la llamada a `printf` sólo si `calificacion` *no* es igual que `valorCentinela`. Los paréntesis alrededor de la condición `calificacion == valorCentinela` son necesarios, ya que el operador lógico de negación tiene mayor precedencia que el de igualdad.

En la mayoría de los casos, puede evitar el uso de la negación lógica si expresa la condición en forma distinta, con un operador relacional o de igualdad apropiado. Por ejemplo, la instrucción anterior también puede escribirse de la siguiente manera:

```
if ( calificacion != valorCentinela )
   System.out.printf( "La siguiente calificación es %d\n", calificacion );
```

Esta flexibilidad le puede ayudar a expresar una condición de una manera más conveniente. La figura 5.17 es una tabla de verdad para el operador lógico de negación.

expresión	!expresión
false	true
true	false

Fig. 5.17 | Tabla de verdad del operador ! (negación lógica, o NOT lógico).

Ejemplo de los operadores lógicos

La figura 5.18 demuestra el uso de operadores lógicos para producir las tablas de verdad que se describen en esta sección. Los resultados muestran la expresión boolean que se evalúo y su resultado. Utilizamos el **especificador de formato %b** para imprimir la palabra "true" o "false", con base en el valor de una expresión boolean. Las líneas 9 a 13 producen la tabla de verdad para el &&. Las líneas 16 a 20 producen la tabla de verdad para el ||. Las líneas 23 a 27 producen la tabla de verdad para el &. Las líneas 30 a 35 producen la tabla de verdad para el |. Las líneas 38 a 43 producen la tabla de verdad para el ^. Las líneas 46 a 47 producen la tabla de verdad para el !.

```
1  // Fig. 5.18: OperadoresLogicos.java
2  // Los operadores lógicos.
3
4  public class OperadoresLogicos
5  {
6     public static void main( String[] args )
7     {
8        // crea tabla de verdad para el operador && (AND condicional)
9        System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n\n",
10          "AND condicional (&&)", "false && false", ( false && false ),
11          "false && true", ( false && true ),
12          "true && false", ( true && false ),
13          "true && true", ( true && true ) );
14
15       // crea tabla de verdad para el operador || (OR condicional)
16       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n\n",
17          "OR condicional (||)", "false || false", ( false || false ),
18          "false || true", ( false || true ),
19          "true || false", ( true || false ),
20          "true || true", ( true || true ) );
21
22       // crea tabla de verdad para el operador & (AND lógico booleano)
23       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n\n",
24          "AND logico booleano (&)", "false & false", ( false & false ),
25          "false & true", ( false & true ),
26          "true & false", ( true & false ),
27          "true & true", ( true & true ) );
28
29       // crea tabla de verdad para el operador | (OR inclusivo lógico booleano)
30       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n\n",
31          "OR inclusivo logico booleano (|)",
```

Fig. 5.18 | Los operadores lógicos (parte 1 de 2).

```
32            "false | false", ( false | false ),
33            "false | true", ( false | true ),
34            "true | false", ( true | false ),
35            "true | true", ( true | true ) );
36
37         // crea tabla de verdad para el operador ^ (OR exclusivo lógico booleano)
38         System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
39            "OR exclusivo logico booleano (^)",
40            "false ^ false", ( false ^ false ),
41            "false ^ true", ( false ^ true ),
42            "true ^ false", ( true ^ false ),
43            "true ^ true", ( true ^ true ) );
44
45         // crea tabla de verdad para el operador ! (negación lógica)
46         System.out.printf( "%s\n%s: %b\n%s: %b\n", "NOT logico (!)",
47            "!false", ( !false ), "!true", ( !true ) );
48      } // fin de main
49   } // fin de la clase OperadoresLogicos
```

```
AND condicional (&&)
false && false: false
false && true: false
true && false: false
true && true: true

OR condicional (||)
false || false: false
false || true: true
true || false: true
true || true: true

AND logico booleano (&)
false & false: false
false & true: false
true & false: false
true & true: true

OR inclusivo logico booleano (|)
false | false: false
false | true: true
true | false: true
true | true: true

OR exclusivo logico booleano (^)
false ^ false: false
false ^ true: true
true ^ false: true
true ^ true: false

NOT logico (!)
!false: true
!true: false
```

Fig. 5.18 | Los operadores lógicos (parte 2 de 2).

La figura 5.19 muestra la precedencia y la asociatividad de los operadores de Java presentados hasta ahora. Los operadores se muestran de arriba hacia abajo, en orden descendente de precedencia.

Operadores	Asociatividad	Tipo
++ --	derecha a izquierda	postfijo unario
++ -- + - ! (*type*)	derecha a izquierda	prefijo unario
* / %	izquierda a derecha	multiplicativo
+ -	izquierda a derecha	aditivo
< <= > >=	izquierda a derecha	relacional
== !=	izquierda a derecha	igualdad
&	izquierda a derecha	AND lógico booleano
^	izquierda a derecha	OR exclusivo lógico booleano
|	izquierda a derecha	OR inclusivo lógico booleano
&&	izquierda a derecha	AND condicional
||	izquierda a derecha	OR condicional
?:	derecha a izquierda	condicional
= += -= *= /= %=	derecha a izquierda	asignación

Fig. 5.19 | Precedencia/asociatividad de los operadores descritos hasta ahora.

5.9 Resumen sobre programación estructurada

Así como los arquitectos diseñan edificios empleando la sabiduría colectiva de su profesión, de igual forma, los programadores diseñan programas. Nuestro campo es mucho más joven que el de la arquitectura, y nuestra sabiduría colectiva es mucho más escasa. Hemos aprendido que la programación estructurada produce programas que son más fáciles de entender, probar, depurar y modificar que los programas sin estructura, e incluso probar que son correctos en sentido matemático.

La figura 5.20 utiliza diagramas de actividad de UML para sintetizar las instrucciones de control de Java. Los estados inicial y final indican el *único punto de entrada* y el *único punto de salida* de cada instrucción de control. Si conectamos los símbolos individuales de un diagrama de actividad en forma arbitrara, existe la posibilidad de que se produzcan programas sin estructura. Por lo tanto, la profesión de la programación ha elegido un conjunto limitado de instrucciones de control que pueden combinarse sólo de dos formas simples, para crear programas estructurados.

Por cuestión de simpleza, Java incluye sólo instrucciones de control de *una sola entrada/una sola salida*; sólo hay una forma de entrar y una manera de salir de cada instrucción de control. Es sencillo conectar instrucciones de control en secuencia para formar programas estructurados. El estado final de una instrucción de control se conecta al estado inicial de la siguiente instrucción de control; es decir, se colocan una después de la otra en un programa en secuencia. A esto le llamamos *apilamiento de instrucciones de control*. Las reglas para formar programas estructurados también permiten *anidar* las instrucciones de control.

La figura 5.21 muestra las reglas para formar programas estructurados. Las reglas suponen que pueden utilizarse estados de acción para indicar cualquier tarea. Además, las reglas suponen que comenzamos con el diagrama de actividad más sencillo (figura 5.22), que consiste solamente de un estado inicial, un estado de acción, un estado final y flechas de transición.

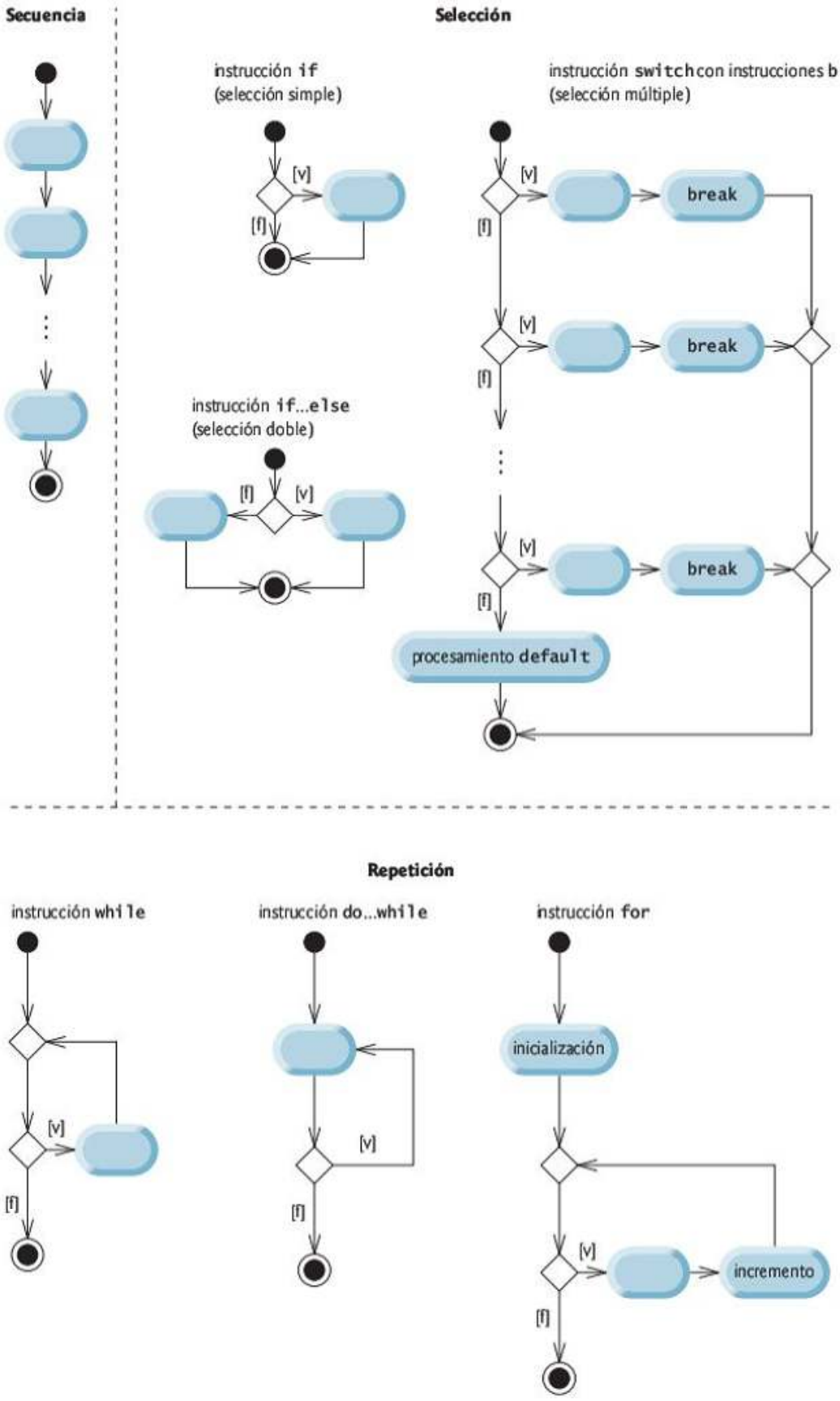


Fig. 5.20 | Instrucciones de secuencia, selección y repetición de una sola entrada/una sola salida de Java.

Reglas para formar programas estructurados
1. Comenzar con el diagrama de actividad más sencillo (figura 5.22).
2. Cualquier estado de acción puede reemplazarse por dos estados de acción en secuencia.
3. Cualquier estado de acción puede reemplazarse por cualquier instrucción de control (secuencia de estados de acción, `if`, `if...else`, `switch`, `while`, `do...while` o `for`).
4. Las reglas 2 y 3 pueden aplicarse tantas veces como se desee y en cualquier orden.

Fig. 5.21 | Reglas para formar programas estructurados.

Al aplicar las reglas de la figura 5.21, siempre se obtiene un diagrama de actividad estructurado en forma apropiada, con una agradable apariencia de bloque de construcción. Por ejemplo, si se aplica la regla 2 de manera repetida al diagrama de actividad más sencillo, se obtiene un diagrama de actividad que contiene muchos estados de acción en secuencia (figura 5.23). La regla 2 genera una pila de estructuras de control, por lo que llamaremos a la regla 2 **regla de apilamiento**. Las líneas punteadas verticales en la figura 5.23 no son parte de UML; las utilizamos para separar los cuatro diagramas de actividad que demuestran cómo se aplica la regla 2 de la figura 5.21.

Fig. 5.22 | El diagrama de actividad más simple.

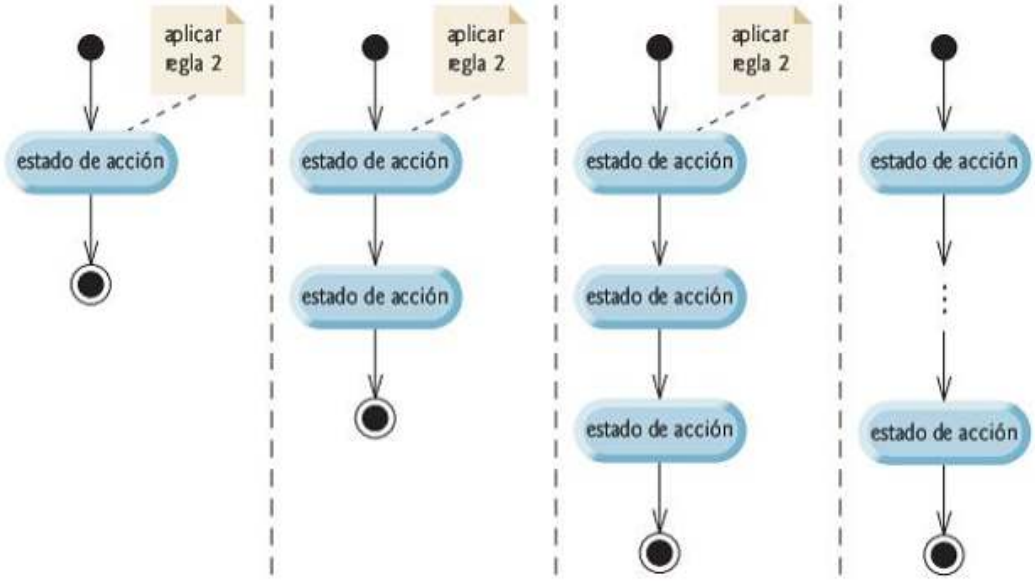


Fig. 5.23 | El resultado de aplicar la regla 2 de la figura 5.21 repetidamente al diagrama de actividad más sencillo.

La regla 3 se conoce como **regla de anidamiento**. Al aplicar la regla 3 en forma repetida al diagrama de actividad más sencillo, se obtiene un diagrama de actividad con instrucciones de control perfectamente anidadas. Por ejemplo, en la figura 5.24 el estado de acción en el diagrama de actividad más sencillo se reemplaza con una instrucción de selección doble (`if...else`). Luego la regla 3 se aplica otra vez a los estados de acción en la instrucción de selección doble, reemplazando cada uno de estos estados con una instrucción de selección doble. El símbolo punteado de estado de acción alrededor de cada una de las instrucciones de selección doble, representa el estado de acción que se reemplazó. [*Nota:* las flechas punteadas y los símbolos punteados de estado de acción que se muestran en la figura 5.24 no son parte de UML. Aquí se utilizan para ilustrar que cualquier estado de acción puede reemplazarse con un enunciado de control].

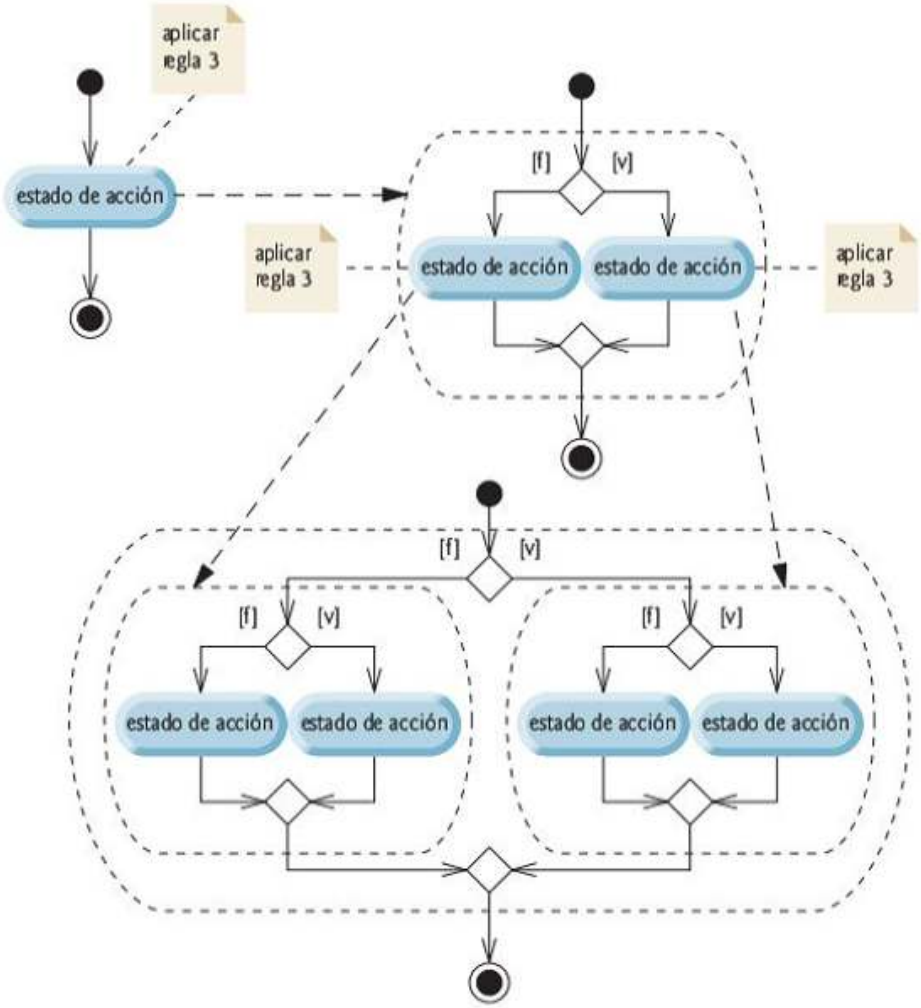


Fig. 5.24 | Aplicación en forma repetida de la regla 3 de la figura 5.21 al diagrama de actividad más sencillo.

La regla 4 genera instrucciones más grandes, más implicadas y más profundamente anidadas. Los diagramas que surgen debido a la aplicación de las reglas de la figura 5.21 constituyen el conjunto de todos los posibles diagramas de actividad estructurados y, por lo tanto, el conjunto de todos los posibles programas estructurados. La belleza de la metodología estructurada es que utilizamos *sólo siete* instrucciones de control simples de una sola entrada/una sola salida, y las ensamblamos en *sólo dos* formas simples.

Si se siguen las reglas de la figura 5.21, no podrá crearse un diagrama de actividad "sin estructura" (como el de la figura 5.25). Si usted no está seguro de que cierto diagrama sea estructurado, aplique las reglas de la figura 5.21 en orden inverso para reducirlo al diagrama de actividad más sencillo. Si puede hacerlo, entonces el diagrama original es estructurado; de lo contrario, no es estructurado.

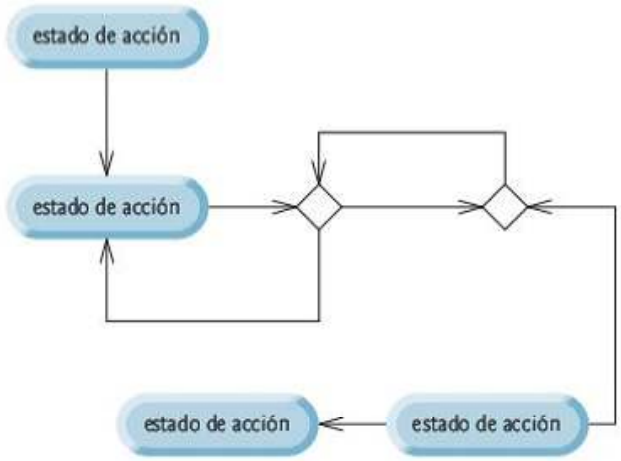


Fig. 5.25 | Diagrama de actividad "sin estructura".

La programación estructurada promueve la simpleza. Sólo se necesitan tres formas de control para implementar un algoritmo:

- Secuencia
- Selección
- Repetición

La estructura de secuencia es trivial. Simplemente enumere las instrucciones a ejecutar en el orden debido. La selección se implementa en una de tres formas:

- instrucción `if` (selección simple)
- instrucción `if...else` (selección doble)
- instrucción `switch` (selección múltiple)

De hecho, es sencillo demostrar que la instrucción `if` simple es suficiente para ofrecer *cualquier* forma de selección; todo lo que pueda hacerse con las instrucciones `if...else` y `switch` puede implementarse si se combinan instrucciones `if` (aunque tal vez no con tanta claridad y eficiencia).

La repetición se implementa en una de tres maneras:

- instrucción `while`
- instrucción `do...while`
- instrucción `for`

[*Nota:* hay una cuarta instrucción de repetición (la instrucción `for` mejorada) que veremos en la sección 7.6]. Es sencillo demostrar que la instrucción `while` es suficiente para proporcionar *cualquier* forma de repetición. Todo lo que puede hacerse con las instrucciones `do...while` y `for`, puede hacerse también con la instrucción `while` (aunque tal vez no sea tan sencillo).

Si se combinan estos resultados, se demuestra que *cualquier* forma de control necesaria en un programa en Java puede expresarse en términos de

- secuencia
- instrucción `if` (selección)
- instrucción `while` (repetición)

y que estos tres elementos pueden combinarse en sólo dos formas: *apilamiento* y *anidamiento*. Sin duda, la programación estructurada es la esencia de la simpleza.

5.10 (Opcional) Caso de estudio de GUI y gráficos: dibujo de rectángulos y óvalos

Esta sección demuestra cómo dibujar rectángulos y óvalos, mediante los métodos **`drawRect`** y **`drawOval`** de `Graphics`, respectivamente. Estos métodos se demuestran en la figura 5.26.

```
1  // Fig. 5.26: Figuras.java
2  // Demuestra cómo dibujar distintas figuras.
3  import java.awt.Graphics;
4  import javax.swing.JPanel;
5
6  public class Figuras extends JPanel
7  {
8     private int opcion; // opción del usuario acerca de cuál figura dibujar
9
10    // el constructor establece la opción del usuario
11    public Figuras( int opcionUsuario )
12    {
13       opcion = opcionUsuario;
14    } // fin del constructor de Figuras
15
16    // dibuja una cascada de figuras, empezando desde la esquina superior izquierda
17    public void paintComponent( Graphics g )
18    {
19       super.paintComponent( g );
20
21       for ( int i = 0; i < 10; i++ )
22       {
23          // elije la figura con base en la opción del usuario
24          switch ( opcion )
25          {
26             case 1: // dibuja rectángulos
27                g.drawRect( 10 + i * 10, 10 + i * 10,
28                   50 + i * 10, 50 + i * 10 );
29                break;
30             case 2: // dibuja óvalos
31                g.drawOval( 10 + i * 10, 10 + i * 10,
32                   50 + i * 10, 50 + i * 10 );
33                break;
34          } // fin del switch
35       } // fin del for
36    } // fin del método paintComponent
37 } // fin de la clase Figuras
```

Fig. 5.26 | Cómo dibujar una cascada de figuras, con base en la opción elegida por el usuario.

La línea 6 empieza la declaración de la clase para `Figuras`, que extiende a `JPanel`. La variable de instancia `opcion`, declarada en la línea 8, determina si `paintComponent` debe dibujar rectángulos u óvalos. El constructor de `Figuras` en las líneas 11 a 14 inicializa `opcion` con el valor que se pasa en el parámetro `opcionUsuario`.

El método `paintComponent` (líneas 17 a 36) realiza el dibujo actual. Recuerde que la primera instrucción en todo método `paintComponent` debe ser una llamada a `super.paintComponent`, como en la línea 19. Las líneas 21 a 35 iteran 10 veces para dibujar 10 figuras. La instrucción `switch` anidada (líneas 24 a 34) elije entre dibujar rectángulos y dibujar óvalos.

Si `opcion` es `1`, entonces el programa dibuja rectángulos. Las líneas 27 y 28 llaman al método `drawRect` de `Graphics`. El método `drawRect` requiere cuatro argumentos. Los primeros dos representan las coordenadas *x* y *y* de la esquina superior izquierda del rectángulo; los siguientes dos simbolizan la anchura y la altura del rectángulo. En este ejemplo, empezamos en la posición 10 píxeles hacia abajo y 10 píxeles a la derecha de la esquina superior izquierda, y cada iteración del ciclo avanza la esquina superior izquierda otros 10 píxeles hacia abajo y a la derecha. La anchura y la altura del rectángulo empiezan en 50 píxeles, y se incrementan por 10 píxeles en cada iteración.

Si `opcion` es 2, el programa dibuja óvalos. Crea un rectángulo imaginario llamado **rectángulo delimitador**, y dentro de éste crea un óvalo que toca los puntos medios de todos los cuatro lados. El método `drawOval` (líneas 31 y 32) requiere los mismos cuatro argumentos que el método `drawRect`. Los argumentos especifican la posición y el tamaño del rectángulo delimitador para el óvalo. Los valores que se pasan a `drawOval` en este ejemplo son exactamente los mismos valores que se pasan a `drawRect` en las líneas 27 y 28. Como la anchura y la altura del rectángulo delimitador son idénticas en este ejemplo, las líneas 27 y 28 dibujan un círculo. Como ejercicio, modifique el programa que dibuja rectángulos y óvalos, para ver cómo se relacionan `drawOval` y `drawRect`.

La clase `PruebaFiguras`

La figura 5.27 es responsable de manejar la entrada del usuario y crear una ventana para mostrar el dibujo apropiado, con base en la respuesta del usuario. La línea 3 importa a `JFrame` para manejar la pantalla, y la línea 4 importa a `JOptionPane` para manejar la entrada.

```
1  // Fig. 5.27: PruebaFiguras.java
2  // Aplicación de prueba que muestra la clase Figuras.
3  import javax.swing.JFrame;
4  import javax.swing.JOptionPane;
5
6  public class PruebaFiguras
7  {
8     public static void main( String[] args )
9     {
10       // obtiene la opción del usuario
11       String entrada = JOptionPane.showInputDialog(
12          "Escriba 1 para dibujar rectangulos\n" +
13          "Escriba 2 para dibujar ovalos" );
14
15       int opcion = Integer.parseInt( entrada ); // convierte entrada en int
16
17       // crea el panel con la entrada del usuario
18       Figuras panel = new Figuras( opcion );
```

Fig. 5.27 | Cómo obtener datos de entrada del usuario y crear un objeto `JFrame` para mostrar `Figuras` (parte 1 de 2).

```
19
20        JFrame aplicacion = new JFrame(); // crea un nuevo objeto JFrame
21
22        aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
23        aplicacion.add( panel ); // agrega el panel al marco
24        aplicacion.setSize( 300, 300 ); // establece el tamaño deseado
25        aplicacion.setVisible( true ); // muestra el marco
26     } // fin de main
27  } // fin de la clase PruebaFiguras
```

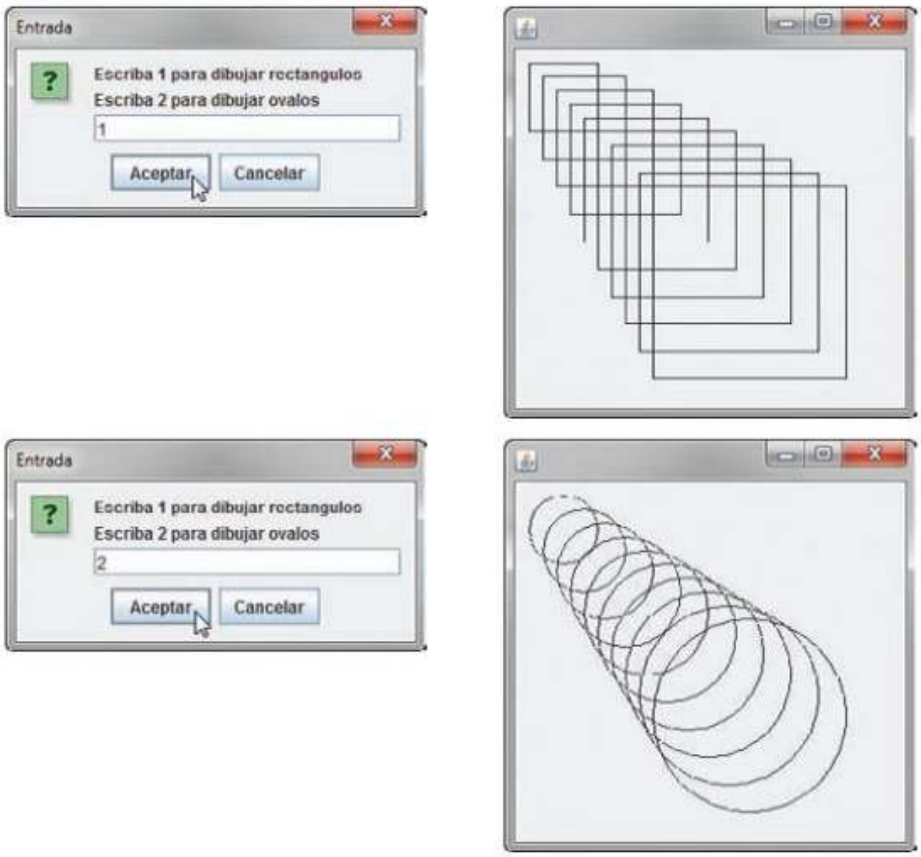


Fig. 5.27 | Cómo obtener datos de entrada del usuario y crear un objeto `JFrame` para mostrar `Figuras` (parte 2 de 2).

Las líneas 11 a 13 muestran un cuadro de diálogo al usuario y almacenan la respuesta de éste en la variable `entrada`. La línea 15 utiliza el método `parseInt` de `Integer` para convertir el objeto `String` introducido por el usuario en un `int`, y almacena el resultado en la variable `opcion`. En la línea 18 se crea una instancia de la clase `Figuras`, y se pasa la opción del usuario al constructor. Las líneas 20 a 25 realizan las operaciones estándar para crear y establecer una ventana en este caso de estudio: crear un marco, configurarlo para que la aplicación termine cuando se cierre, agregar el dibujo al marco, establecer su tamaño y hacerlo visible.

Ejercicios del caso de estudio de GUI y gráficos

5.1 Dibuje 12 círculos concéntricos en el centro de un objeto `JPanel` (figura 5.28). El círculo más interno debe tener un radio de 10 píxeles, y cada círculo sucesivo debe contar con un radio 10 píxeles mayor que el anterior. Empiece por buscar el centro del objeto `JPanel`. Para obtener la esquina superior izquierda de un círculo, avance un radio hacia arriba y un radio a la izquierda, partiendo del centro. La anchura y la altura del rectángulo delimitador es el diámetro del círculo (el doble del radio).

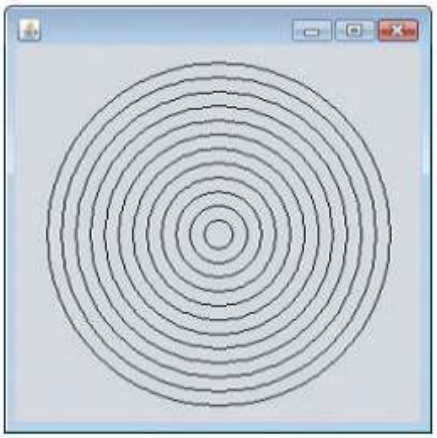


Fig. 5.28 | Cómo dibujar círculos concéntricos.

5.2 Modifique el ejercicio 5.16 de los ejercicios de fin de capítulo para leer la entrada usando cuadros de diálogo, y mostrar el gráfico de barras usando rectángulos de longitudes variables.

5.11 Conclusión

En este capítulo completamos nuestra introducción a las instrucciones de control de Java, las cuales nos permiten controlar el flujo de la ejecución en los métodos. El capítulo 4 trató acerca de las instrucciones de control `if`, `if...else` y `while` de Java. En este capítulo vimos las instrucciones `for`, `do...while` y `switch`. Aquí le mostramos que es posible desarrollar cualquier algoritmo mediante el uso de combinaciones de la estructura de la secuencia (es decir, instrucciones que se enumeran en el orden en el que deben ejecutarse), los tres tipos de instrucciones de selección (`if`, `if...else` y `switch`) y los tres tipos de instrucciones de repetición (`while`, `do...while` y `for`). En este capítulo y en el anterior hablamos de cómo combinar estos bloques de construcción para utilizar las técnicas, ya probadas, de construcción de programas y solución de problemas. En este capítulo también se introdujeron los operadores lógicos de Java, que nos permiten utilizar expresiones condicionales más complejas en las instrucciones de control. En el capítulo 6 analizaremos los métodos con más detalle.

Resumen

Sección 5.2 Fundamentos de la repetición controlada por contador

- La repetición controlada por contador (pág. 152) requiere una variable de control, el valor inicial de la variable de control, el incremento (o decremento) en base al cual se modifica la variable de control cada vez que pasa por el ciclo (lo que también se conoce como cada iteración del ciclo) y la condición de continuación de ciclo, que determina si el ciclo debe seguir ejecutándose.
- Podemos declarar e inicializar una variable en la misma instrucción.

Sección 5.3 Instrucción de repetición `for`

- La instrucción `while` puede usarse para implementar cualquier ciclo controlado por contador.
- La instrucción `for` (pág. 154) especifica todos los detalles acerca de la repetición controlada por contador, en su encabezado.
- Cuando la instrucción `for` comienza a ejecutarse, su variable de control se declara y se inicializa. Después, el programa verifica la condición de continuación de ciclo. Si al principio la condición es verdadera, el cuerpo se eje-

cuta. Después de ejecutar el cuerpo del ciclo, se ejecuta la expresión de incremento. Entonces, se lleva a cabo otra vez la prueba de continuación de ciclo, para determinar si el programa debe continuar con la siguiente iteración del ciclo.

- El formato general de la instrucción `for` es

```
for (inicialización; condiciónDeContinuacionDeCiclo; incremento)
   instrucción
```

en donde la expresión *inicialización* asigna un nombre a la variable de control del ciclo y proporciona su valor inicial, *condiciónDeContinuaciónDeCiclo* determina si el ciclo debe continuar su ejecución, e *incremento* modifica el valor de la variable de control, de manera que la condición de continuación de ciclo se vuelve falsa en un momento dado. Los dos signos de punto y coma en el encabezado `for` son obligatorios.

- La mayoría de las instrucciones `for` se pueden representar con una instrucción `while` equivalente, de la siguiente forma:

```
inicialización;
while (condiciónDeContinuaciónDeCiclo)
{
   instrucción
   incremento;
}
```

- Por lo general, las instrucciones `for` se utilizan para la repetición controlada por contador y las instrucciones `while` para la repetición controlada por centinela.
- Si la expresión de *inicialización* en el encabezado del `for` declara la variable de control, ésta sólo puede usarse en esa instrucción `for`; no existirá fuera de ella.
- Las tres expresiones en un encabezado `for` son opcionales. Si se omite la *condiciónDeContinuaciónDeCiclo*, Java asume que siempre es verdadera, con lo cual se crea un ciclo infinito. Podríamos omitir la expresión *inicialización* si el programa inicializa la variable de control antes del ciclo. Es posible omitir la expresión *incremento* si el programa calcula el incremento con instrucciones en el cuerpo del ciclo, o si no se necesita un incremento.
- La expresión de incremento en un `for` actúa como si fuera una instrucción independiente al final del cuerpo del `for`.
- Una instrucción `for` puede contar en forma descendente mediante el uso de un incremento negativo (es decir, un decremento).
- Si al principio la condición de continuación de ciclo es `false`, el programa no ejecuta el cuerpo de la instrucción `for`. En vez de ello, la ejecución continúa con la instrucción después del `for`.

Sección 5.4 Ejemplos sobre el uso de la instrucción `for`

- Java trata a las constantes de punto flotante, como `1000.0` y `0.05`, como de tipo `double`. De manera similar, trata a las constantes de números enteros, como `7` y `-22`, como de tipo `int`.
- El especificador de formato `%4s` imprime un objeto `String` con una anchura de campo (pág. 160) de 4; es decir, `printf` muestra el valor con al menos 4 posiciones de caracteres. Si el valor a imprimir es menor que 4 posiciones de caracteres de ancho, se justifica a la derecha (pág. 160) en el campo de manera predeterminada. Si el valor es mayor que 4 posiciones de ancho, la anchura del campo se expande para dar cabida al número apropiado de caracteres. Para justificar el valor a la izquierda (pág. 160), use un entero negativo que especifique la anchura del campo.
- `Math.pow(x, y)` (pág. 161) calcula el valor de x elevado a la $y^{\text{ésima}}$ potencia. El método recibe dos argumentos `double` y devuelve un valor `double`.
- La bandera de formato coma (,) (pág. 161) en un especificador de formato indica que un valor de punto flotante debe imprimirse con un separador de agrupamiento (pág. 161). El separador actual que se utiliza es específico de la configuración regional del usuario (es decir, el país). Por ejemplo, en Estados Unidos el número se imprimirá usando comas para separar cada tres dígitos, y un punto decimal para separar la parte fraccionaria del número, como en 1,234.45.
- El . en un especificador de formato indica que el entero a su derecha es la precisión del número.

Sección 5.5 Instrucción de repetición `do...while`

- La instrucción de repetición `do...while` (pág. 162) es similar a la instrucción `while`. En la instrucción `while`, el programa evalúa la condición de continuación de ciclo al principio del ciclo, antes de ejecutar su cuerpo; si la condición es

falsa, el cuerpo nunca se ejecuta. La instrucción `do...while` evalúa la condición de continuación de ciclo después de ejecutar el cuerpo del ciclo; por lo tanto, el cuerpo siempre se ejecuta por lo menos una vez.

Sección 5.6 Instrucción de selección múltiple `switch`

- La instrucción `switch` (pág. 164) realiza distintas acciones, con base en los posibles valores de una expresión entera constante (un valor constante de tipo `byte`, `short`, `int` o `char`, pero no `long`).
- El indicador de fin de archivo (pág. 167) es una combinación de teclas dependiente del sistema, que termina la entrada del usuario. En los sistemas UNIX/Linux/Mac OS X, el fin de archivo se introduce escribiendo la secuencia *<Ctrl> d* en una línea por sí sola. Esta notación significa que hay que oprimir al mismo tiempo la tecla *Ctrl* y la tecla *d*. En los sistemas Windows, el fin de archivo se puede introducir escribiendo *<Ctrl> z*.
- El método `hasNext` de `Scanner` (pág. 167) determina si hay más datos qué introducir. Este método devuelve el valor `boolean` `true` si hay más datos; en caso contrario, devuelve `false`. Mientras no se haya escrito el indicador de fin de archivo, el método `hasNext` devolverá `true`.
- La instrucción `switch` consiste en un bloque que contiene una secuencia de etiquetas `case` (pág. 168) y un caso `default` opcional (pág. 168).
- Cuando el flujo de control llega a un `switch`, el programa evalúa la expresión de control del `switch` y compara su valor con cada etiqueta `case`. Si ocurre una coincidencia, el programa ejecuta las instrucciones para esa etiqueta `case`.
- Al enumerar etiquetas `case` en forma consecutiva, sin instrucciones entre ellas, permitimos que ejecuten el mismo conjunto de instrucciones.
- Todo valor que desee evaluar en un `switch` debe enumerarse en una etiqueta `case` separada.
- Cada `case` puede tener varias instrucciones, y no es necesario colocarlas entre llaves.
- Sin las instrucciones `break`, cada vez que ocurre una coincidencia en el `switch`, las instrucciones para ese `case` y los `case` subsiguientes se ejecutarán hasta llegar a una instrucción `break` o al final de la instrucción `switch`.
- Si no ocurre una coincidencia entre el valor de la expresión de control y una etiqueta `case`, se ejecuta el caso `default` opcional. Si no ocurre una coincidencia y la instrucción `switch` no tiene un caso `default`, el control del programa simplemente continúa con la primera instrucción después del `switch`.
- A partir de Java SE 7, es posible usar objetos `String` en la expresión de control de una instrucción `switch` y las etiquetas `case`.

Sección 5.7 Instrucciones `break` *y* `continue`

- Cuando la instrucción `break` (pág. 168) se ejecuta en una instrucción `while`, `for`, `do...while` o `switch`, provoca la salida inmediata de esa instrucción.
- Cuando la instrucción `continue` (pág. 172) se ejecuta en una instrucción `while`, `for` o `do...while`, omite el resto de las instrucciones en el cuerpo del ciclo y continúa con la siguiente iteración del mismo. En las instrucciones `while` y `do...while`, el programa evalúa la prueba de continuación de ciclo de inmediato. En una instrucción `for`, se ejecuta la expresión de incremento y después el programa evalúa la prueba de continuación de ciclo.

Sección 5.8 Operadores lógicos

- Las condiciones simples se expresan en términos de los operadores relacionales `>`, `<`, `>=` y `<=`, y los operadores de igualdad `==` y `!=`, y cada expresión sólo evalúa una condición.
- Los operadores lógicos (pág. 174) nos permiten formar condiciones más complejas, mediante la combinación de condiciones simples. Los operadores lógicos son `&&` (AND condicional), `||` (OR condicional), `&` (AND lógico booleano), `|` (OR inclusivo lógico booleano), `^` (OR exclusivo lógico booleano) y `!` (NOT lógico).
- Para asegurar que dos condiciones sean, *ambas*, verdaderas, utilice el operador `&&` (AND condicional). Si una o ambas condiciones simples son falsas, la expresión completa es falsa.
- Para asegurar que una *o* ambas condiciones sean verdaderas, utilice el operador `||` (OR condicional), que se evalúa como verdadero si una o las dos condiciones simples son verdaderas.

- Una condición que utiliza los operadores `&&` o `||` (pág. 174) utiliza la evaluación de corto circuito (pág. 175); se evalúan sólo hasta que se conoce si la condición es verdadera o falsa.
- Los operadores `&` y `|` (pág. 175) funcionan de manera idéntica a los operadores `&&` y `||`, sólo que siempre evalúan ambos operandos.
- Una condición simple que contiene el operador OR exclusivo lógico booleano (`^`; pág. 176) es `true` *si, y sólo si uno de sus operandos es* `true` *y el otro es* `false`. Si ambos operandos son `true` o `false`, toda la condición es `false`. También se garantiza que este operador evaluará los dos operandos.
- El operador `!` unario (NOT lógico, pág. 176) "invierte" el significado de una condición.

Ejercicios de autoevaluación

5.1 *(Complete los espacios en blanco)* Complete los siguientes enunciados:

a) Por lo general, las instrucciones _____ se utilizan para la repetición controlada por contador y las instrucciones _____ se utilizan para la repetición controlada por centinela.

b) La instrucción `do...while` evalúa la condición de continuación de ciclo _____ ejecutar el cuerpo del ciclo; por lo tanto, el cuerpo siempre se ejecuta por lo menos una vez.

c) La instrucción _____ selecciona una de varias acciones, con base en los posibles valores de una variable o expresión entera.

d) Cuando se ejecuta la instrucción _____ en una instrucción de repetición, se omite el resto de las instrucciones en el cuerpo del ciclo y se continúa con la siguiente iteración del ciclo.

e) El operador _____ se puede utilizar para asegurar que ambas condiciones sean verdaderas, antes de elegir cierta ruta de ejecución.

f) Si al principio, la condición de continuación de ciclo en un encabezado `for` es _____ , el programa no ejecuta el cuerpo de la instrucción `for`.

g) Los procedimientos que realizan tareas comunes y no requieren objetos se llaman métodos _____ .

5.2 *(Preguntas de falso/verdadero)* Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) El caso `default` es requerido en la instrucción de selección `switch`.

b) La instrucción `break` es requerida en el último caso de una instrucción de selección `switch`.

c) La expresión `( ( x > y ) && ( a < b ) )` es verdadera si `x > y` es verdadera, o si `a < b` es verdadera.

d) Una expresión que contiene el operador `||` es verdadera si uno o ambos de sus operandos son verdaderos.

e) La bandera de formato coma (`,`) en un especificador de formato (por ejemplo, `%,20.2f`) indica que un valor debe imprimirse con un separador de miles.

f) Para evaluar un rango de valores en una instrucción `switch`, use un guión corto (–) entre los valores inicial y final del rango en una etiqueta `case`.

g) Al enumerar las instrucciones `case` en forma consecutiva, sin instrucciones entre ellas, pueden ejecutar el mismo conjunto de órdenes.

5.3 *(Escriba una instrucción)* Escriba una instrucción o un conjunto de instrucciones en Java, para realizar cada una de las siguientes tareas:

a) Sumar los enteros impares entre 1 y 99, utilizando una instrucción `for`. Suponga que se han declarado las variables enteras `suma` y `cuenta`.

b) Calcular el valor de `2.5` elevado a la potencia de `3`, mediante el método `pow`.

c) Imprimir los enteros del 1 al 20, utilizando un ciclo `while` y la variable contador `i`. Suponga que la variable `i` se ha declarado, pero no se ha inicializado. Imprima solamente cinco enteros por línea. [*Sugerencia:* use el cálculo `i % 5`. Cuando el valor de esta expresión sea 0, imprima un carácter de nueva línea; de lo contrario, imprima un carácter de tabulación. Suponga que este código es una aplicación. Utilice el método `System.out.println()` para producir el carácter de nueva línea, y el método `System.out.print( '\t' )` para producir el carácter de tabulación].

d) Repita la parte (c), usando una instrucción `for`.

5.4 *(Encuentre el error)* Encuentre el error en cada uno de los siguientes segmentos de código, y explique cómo corregirlo:

a)
```
i = 1;

while ( i <= 10 );
   ++i;
}
```

b)
```
for ( k = 0.1; k != 1.0; k += 0.1 )
   System.out.println( k );
```

c)
```
switch ( n )
{
   case 1:
      System.out.println( "El número es 1" );
   case 2:
      System.out.println( "El número es 2" );
      break;
   default:
      System.out.println( "El número no es 1 ni 2" );
      break;
}
```

d)
```
El siguiente código debe imprimir los valores 1 a 10:
n = 1;
while ( n < 10 )
   System.out.println( n++ );
```

Respuestas a los ejercicios de autoevaluación

5.1 a) `for`, `while`. b) después de. c) `switch`. d) `continue`. e) `&&` (AND condicional). f) `false`. g) `static`.

5.2 a) Falso. El caso `default` es opcional. Si no se necesita una acción predeterminada, entonces no hay necesidad de un caso `default`. b) Falso. La instrucción `break` se utiliza para salir de la instrucción `switch`. La instrucción `break` no se requiere para el último caso en una instrucción `switch`. c) Falso. Ambas expresiones relacionales deben ser verdaderas para que toda la expresión sea verdadera, cuando se utilice el operador `&&`. d) Verdadero. e) Verdadero. f) Falso. La instrucción `switch` no cuenta con un mecanismo para evaluar rangos de valores, por lo que todo valor que deba examinarse se debe enumerar en una etiqueta `case` por separado. g) Verdadero.

5.3 a)
```
suma = 0;
for ( cuenta = 1; cuenta <= 99; cuenta += 2 )
   suma += cuenta;
```

b)
```
double resultado = Math.pow( 2.5, 3 );
```

c)
```
i = 1;

while ( i <= 20 )
{
   System.out.print( i );

   if ( i % 5 == 0 )
      System.out.println();
   else
      System.out.print( ' ' );

   ++i;
}
```

d)
```
for ( i = 1; i <= 20; i++ )
{
   System.out.print( i );
```

```
   if ( i % 5 == 0 )
      System.out.println();
   else
      System.out.print( ' ' );
}
```

5.4 a) Error: el punto y coma después del encabezado `while` provoca un ciclo infinito, y falta una llave izquierda.
Corrección: reemplazar el punto y coma por una llave izquierda ({), o eliminar tanto el punto y coma (;) como la llave derecha (}).

b) Error: utilizar un número de punto flotante para controlar una instrucción `for` tal vez no funcione, ya que los números de punto flotante se representan sólo de manera aproximada en la mayoría de las computadoras.
Corrección: utilice un entero, y realice el cálculo apropiado para poder obtener los valores deseados:

```
for ( k = 1; k != 10; k++ )
   System.out.println( (double) k / 10 );
```

c) Error: el código que falta es la instrucción `break` en las instrucciones del primer `case`.
Corrección: agregue una instrucción `break` al final de las instrucciones para el primer `case`. Esta omisión no es necesariamente un error, si el programador desea que la instrucción del `case 2:` se ejecute siempre que lo haga la instrucción del `case 1:`.

d) Error: se está utilizando un operador relacional inadecuado en la condición de continuación de la instrucción de repetición `while`.
Corrección: use <= en vez de <, o cambie el 10 a 11.

Ejercicios

5.5 Describa los cuatro elementos básicos de la repetición controlada por contador.

5.6 Compare y contraste las instrucciones de repetición `while` y `for`.

5.7 Hable sobre una situación en la que sería más apropiado usar una instrucción `do...while` que una instrucción `while`. Explique por qué.

5.8 Compare y contraste las instrucciones `break` y `continue`.

5.9 Encuentre y corrija el(los) error(es) en cada uno de los siguientes fragmentos de código:

a)
```
For ( i = 100, i >= 1, i++ )
   System.out.println( i );
```

b) El siguiente código debe imprimirse sin importar si el valor entero es par o impar:

```
switch ( value % 2 )
{
   case 0:
      System.out.println( "Entero par" );

   case 1:
      System.out.println( "Entero impar" );
}
```

c) El siguiente código debe imprimir los enteros impares del 19 al 1:

```
for ( i = 19; i >= 1; i += 2 )
   System.out.println( i );
```

d) El siguiente código debe imprimir los enteros pares del 2 al 100:

```
contador = 2;

do
{
   System.out.println( contador );
   contador += 2;
} While ( contador < 100 );
```

5.10 ¿Qué es lo que hace el siguiente programa?

```
1  // Ejercicio 5.10: Imprimir.java
2  public class Imprimir
3  {
4     public static void main( String[] args )
5     {
6        for ( int i = 1; i <= 10; i++ )
7        {
8           for ( int j = 1; j <= 5; j++ )
9              System.out.print( '@' );
10
11          System.out.println();
12       } // fin del for exterior
13    } // fin de main
14 } // fin de la clase Imprimir
```

5.11 ***(Buscar el valor menor)*** Escriba una aplicación que encuentre el menor de varios enteros. Suponga que el primer valor leído especifica el número de valores que el usuario introducirá.

5.12 ***(Calcular el producto de enteros impares)*** Escriba una aplicación que calcule el producto de los enteros impares del 1 al 15.

5.13 ***(Factoriales)*** Los factoriales se utilizan con frecuencia en los problemas de probabilidad. El factorial de un entero positivo n (se escribe como $n!$ y se pronuncia "factorial de n") es igual al producto de los enteros positivos del 1 a n. Escriba una aplicación que calcule los factoriales del 1 al 20. Use el tipo `long`. Muestre los resultados en formato tabular. ¿Qué dificultad podría impedir que usted calculara el factorial de 100?

5.14 ***(Programa modificado del interés compuesto)*** Modifique la aplicación de interés compuesto de la figura 5.6, repitiendo sus pasos para las tasas de interés del 5, 6, 7, 8, 9 y 10%. Use un ciclo `for` para variar la tasa de interés.

5.15 ***(Programa para imprimir un triángulo)*** Escriba una aplicación que muestre los siguientes patrones por separado, uno debajo del otro. Use ciclos `for` para generar los patrones. Todos los asteriscos (`*`) deben imprimirse mediante una sola instrucción de la forma `System.out.print( '*' );`, la cual hace que los asteriscos se impriman uno al lado del otro. Puede utilizar una instrucción de la forma `System.out.println();` para posicionarse en la siguiente línea. Puede usar una instrucción de la forma `System.out.print(' ');` para mostrar un espacio para los últimos dos patrones. No debe haber ninguna otra instrucción de salida en el programa. [*Sugerencia:* los últimos dos patrones requieren que cada línea empiece con un número apropiado de espacios en blanco].

```
(a)         (b)         (c)         (d)
*           **********  **********      *
**          *********   *********      **
***         ********    ********      ***
****        *******     *******      ****
*****       ******      ******      *****
******      *****       *****      ******
*******     ****        ****      *******
********    ***         ***      ********
*********   **          **      *********
**********  *           *      **********
```

5.16 ***(Programa para imprimir gráficos de barra)*** Una aplicación interesante de las computadoras es dibujar gráficos convencionales y de barra. Escriba una aplicación que lea cinco números, cada uno entre 1 y 30. Por cada número leído, su programa debe mostrar ese número de asteriscos adyacentes. Por ejemplo, si su programa lee el número 7, debe mostrar `*******`. Muestre las barras de asteriscos *después* de leer los cinco números.

5.17 *(Calcular las ventas)* Un vendedor minorista en línea vende cinco productos cuyos precios de venta son los siguientes: producto 1, $2.98; producto 2, $4.50; producto 3, $9.98; producto 4, $4.49 y producto 5, $6.87. Escriba una aplicación que lea una serie de pares de números, como se muestra a continuación:

a) número del producto;
b) cantidad vendida.

Su programa debe utilizar una instrucción `switch` para determinar el precio de venta de cada producto. Debe calcular y mostrar el valor total de venta de todos los productos vendidos. Use un ciclo controlado por centinela para determinar cuándo debe el programa dejar de iterar para mostrar los resultados finales.

5.18 *(Programa modificado del interés compuesto)* Modifique la aplicación de la figura 5.6, de manera que se utilicen sólo enteros para calcular el interés compuesto. [*Sugerencia:* trate todas las cantidades monetarias como números enteros de centavos. Luego divida el resultado en su porción de dólares y su porción de centavos, utilizando las operaciones de división y residuo, respectivamente. Inserte un punto entre las porciones de dólares y centavos].

5.19 Suponga que `i = 1, j = 2, k = 3` y `m = 2`. ¿Qué es lo que imprime cada una de las siguientes instrucciones?

a) `System.out.println( i == 1 );`
b) `System.out.println( j == 3 );`
c) `System.out.println( ( i >= 1 ) && ( j < 4 ) );`
d) `System.out.println( ( m <= 99 ) & ( k < m ) );`
e) `System.out.println( ( j >= i ) || ( k == m ) );`
f) `System.out.println( ( k + m < j ) | ( 3 - j >= k ) );`
g) `System.out.println( !( k > m ) );`

5.20 *(Calcular el valor de π)* Calcule el valor de π a partir de la serie infinita

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} + \frac{4}{7} + \frac{4}{9} + \frac{4}{11} + \cdots$$

Imprima una tabla que muestre el valor aproximado de π, calculando los primeros 200,000 términos de esta serie. ¿Cuántos términos tiene que utilizar para obtener un valor que comience con 3.14159?

5.21 *(Triples de Pitágoras)* Un triángulo recto puede tener lados cuyas longitudes sean valores enteros. El conjunto de tres valores enteros para las longitudes de los lados de un triángulo recto se conoce como triple de Pitágoras. Las longitudes de los tres lados deben satisfacer la relación que establece que la suma de los cuadrados de dos lados es igual al cuadrado de la hipotenusa. Escriba una aplicación que muestre una tabla de los triples de Pitágoras para `lado1`, `lado2` y la `hipotenusa`, que no sean mayores de 500. Use un ciclo `for` triplemente anidado para probar todas las posibilidades. Este método es un ejemplo de la computación de "fuerza bruta". En cursos de ciencias computacionales más avanzados aprenderá que existen muchos problemas interesantes para los cuales no hay otra metodología algorítmica conocida, más que el uso de la fuerza bruta.

5.22 *(Programa modificado para imprimir triángulos)* Modifique el ejercicio 5.15 para combinar su código de los cuatro triángulos separados de asteriscos, de manera que los cuatro patrones se impriman uno al lado del otro. [*Sugerencia:* utilice astutamente los ciclos `for` anidados].

5.23 *(Leyes de De Morgan)* En este capítulo, hemos hablado sobre los operadores lógicos `&&`, `&`, `||`, `|`, `^` y `!`. Algunas veces, las leyes de De Morgan pueden hacer que sea más conveniente para nosotros expresar una expresión lógica. Estas leyes establecen que la expresión `!(`*condición1* `&&` *condición2*`)` es lógicamente equivalente a la expresión `(!`*condición1* `||` *!condición2*`)`. También establecen que la expresión `!(`*condición1* `||` *condición2*`)` es lógicamente equivalente a la expresión `(!`*condición1* `&&` *!condición2*`)`. Use las leyes de De Morgan para escribir expresiones equivalentes para cada una de las siguientes expresiones, luego escriba una aplicación que demuestre que, tanto la expresión original como la nueva expresión, producen en cada caso el mismo valor:

a) `!( x < 5 ) && !( y >= 7 )`
b) `!( a == b ) || !( g != 5 )`
c) `!( ( x <= 8 ) && ( y > 4 ) )`
d) `!( ( i > 4 ) || ( j <= 6 ) )`

5.24 ***(Programa para imprimir rombos)*** Escriba una aplicación que imprima la siguiente figura de rombo. Puede utilizar instrucciones de salida que impriman un solo asterisco (*), un solo espacio o un solo carácter de nueva línea. Maximice el uso de la repetición (con instrucciones for anidadas), y minimice el número de instrucciones de salida.

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

5.25 ***(Programa modificado para imprimir rombos)*** Modifique la aplicación que escribió en el ejercicio 5.24, para que lea un número impar en el rango de 1 a 19, de manera que especifique el número de filas en el rombo. Su programa debe entonces mostrar un rombo del tamaño apropiado.

5.26 Una crítica de las instrucciones break y continue es que ninguna es estructurada. En realidad, estas instrucciones pueden reemplazarse en todo momento por instrucciones estructuradas, aunque hacerlo podría ser inadecuado. Describa, en general, cómo eliminaría las instrucciones break de un ciclo en un programa, para reemplazarlas con alguna de las instrucciones estructuradas equivalentes. [*Sugerencia:* la instrucción break se sale de un ciclo desde el cuerpo de éste. La otra forma de salir es que falle la prueba de continuación de ciclo. Considere utilizar en la prueba de continuación de ciclo una segunda prueba que indique una "salida anticipada debido a una condición de 'interrupción'."] Use la técnica que desarrolló aquí para eliminar la instrucción break de la aplicación de la figura 5.12.

5.27 ¿Qué hace el siguiente segmento de programa?

```
for ( i = 1; i <= 5; i++ )
{
   for ( j = 1; j <= 3; j++ )
   {
      for ( k = 1; k <= 4; k++ )
         System.out.print( '*' );

      System.out.println();
   } // fin del for interior

   System.out.println();
} // fin del for exterior
```

5.28 Describa, en general, cómo eliminaría las instrucciones continue de un ciclo en un programa, para reemplazarlas con uno de sus equivalentes estructurados. Use la técnica que desarrolló aquí para eliminar la instrucción continue del programa de la figura 5.13.

5.29 ***(Canción "Los Doce Días de Navidad")*** Escriba una aplicación que utilice instrucciones de repetición y switch para imprimir la canción "Los Doce Días de Navidad". Una instrucción switch debe utilizarse para imprimir el día (es decir, "primer", "segundo", etcétera). Una instrucción switch separada debe utilizarse para imprimir el resto de cada verso. Visite el sitio Web http://en.wikipedia.org/wiki/The_Twelve_Days_of_Christmas_(song) para obtener la letra completa de la canción.

Marcar la diferencia

5.30 ***(Examen rápido sobre hechos del calentamiento global)*** La controversial cuestión del calentamiento global obtuvo una gran publicidad gracias a la película "An Inconvenient Truth" en la que aparece el anterior vicepresidente Al Gore. El señor Gore y una red de científicos de Naciones Unidas (N.U.), el Panel Intergubernamental sobre el Cambio Climático, compartieron el Premio Nobel de la Paz de 2007 en reconocimiento por "sus esfuerzos al generar y diseminar un

mayor conocimiento sobre el cambio climatológico provocado por el hombre". Investigue *ambos* lados de la cuestión del calentamiento global en línea (tal vez quiera buscar frases como "escépticos del calentamiento global"). Cree un examen rápido de opción múltiple con cinco preguntas sobre el calentamiento global; cada pregunta debe tener cuatro posibles respuestas (enumeradas del 1 al 4). Sea objetivo y trate de representar con imparcialidad ambos lados del asunto. Después escriba una aplicación que administre el examen rápido, calcule el número de respuestas correctas (de cero a cinco) y devuelva un mensaje al usuario. Si éste responde de manera correcta a las cinco preguntas, imprima el mensaje "Excelente"; si responde a cuatro, imprima "Muy bien"; si responde a tres o menos, imprima "Es tiempo de aprender más sobre el calentamiento global", e incluya una lista de algunos de los sitios Web en donde encontró esos hechos.

5.31 ***(Alternativas para el plan fiscal: el "impuesto justo")*** Existen muchas propuestas para que los impuestos sean más justos. Consulte la iniciativa FairTax (impuestos justos) de Estados Unidos en el sitio:

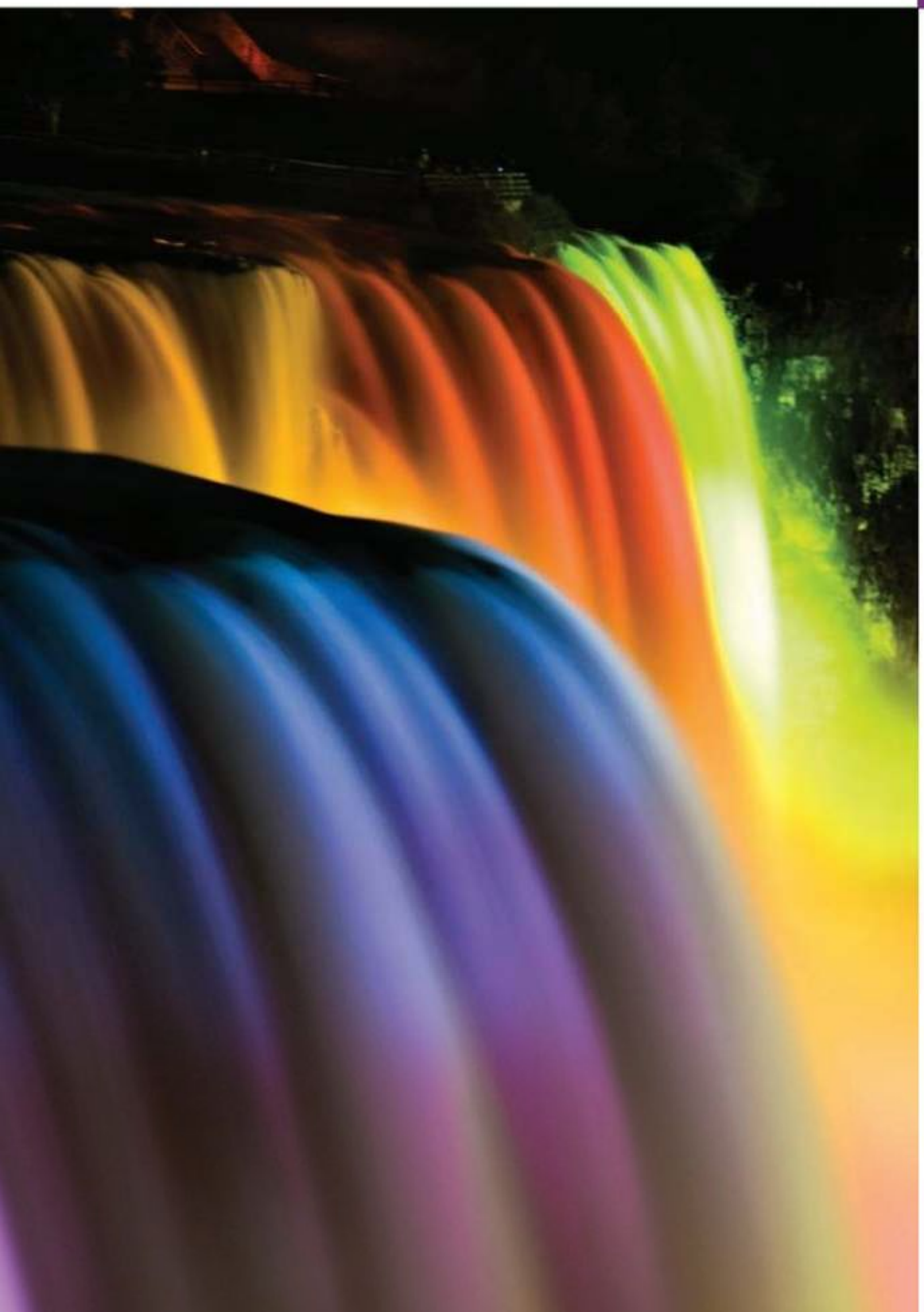
```
www.fairtax.org/site/PageSer PageServer?pagename=calculator
```

Investigue cómo funciona la iniciativa FairTax que se propone. Nuestra sugerencia es eliminar los impuestos sobre los ingresos y otros más a favor de un 23% de impuestos sobre el consumo en todos los productos y servicios que usted compre. Algunos opositores a FairTax cuestionan la cifra del 23% y dicen que, debido a la forma en que se calculan los impuestos, sería más preciso decir que la tasa sea del 30%; revise esto con cuidado. Escriba un programa que pida al usuario que introduzca sus gastos en diversas categorías de gastos disponibles (por ejemplo, alojamiento, comida, ropa, transporte, educación, servicios médicos, vacaciones), y que después imprima el impuesto FairTax estimado que esa persona pagaría.

5.32 ***(Crecimiento de la base de usuarios de Facebook)*** De acuerdo con CNNMoney.com, Facebook llegó a los 500 millones de usuarios en julio de 2010, y su base de usuarios ha estado creciendo a una tasa del 5% mensual. Use la técnica del cálculo del crecimiento compuesto que aprendió en la figura 5.6 y, suponiendo que continúe esta tasa de crecimiento, ¿cuántos meses tardará Facebook en aumentar su base de usuarios a mil millones? ¿Cuántos meses tardará Facebook en aumentar su base de usuarios a dos mil millones (que, al momento de escribir este libro, era el número total de personas en Internet)?

Métodos: un análisis más detallado

6



E pluribus unum.
(Uno compuesto de muchos).
—Virgilio

¡Oh! volvió a llamar ayer,
ofreciéndome volver.
—William Shakespeare

Llámame Ismael.
—Herman Melville

Respóndeme en una palabra.
—William Shakespeare

Hay un punto en el cual los métodos se devoran a sí mismos.
—Frantz Fanon

Objetivos

En este capítulo aprenderá a:

- Conocer cómo se asocian los métodos y los campos `static` con las clases, en vez de los objetos.
- Comprender cómo se soporta el mecanismo de llamada/retorno de los métodos mediante la pila de llamadas a métodos.
- Conocer cómo los paquetes agrupan las clases relacionadas.
- Utilizar la generación de números aleatorios para implementar aplicaciones para juegos.
- Comprender cómo se limita la visibilidad de las declaraciones a regiones específicas de los programas.
- Conocer acerca de la sobrecarga de métodos y cómo crear métodos sobrecargados.

Plan general

6.1 Introducción

La experiencia ha demostrado que la mejor manera de desarrollar y mantener un programa extenso es construirlo a partir de pequeñas piezas sencillas, o **módulos**. A esta técnica se le llama **divide y vencerás**. Los métodos, que presentamos en el capítulo 3, le ayudan a dividir los programas en módulos. En este capítulo, estudiaremos los métodos con más detalle. Haremos énfasis en cómo declarar y utilizar métodos para facilitar el diseño, la implementación, operación y el mantenimiento de programas extensos.

En breve verá que es posible llamar a ciertos métodos, conocidos como `static`, sin necesidad de que exista un objeto de la clase a la que pertenecen. Aprenderá a declarar un método con más de un parámetro, y sabrá cómo Java es capaz de llevar el rastro de qué método se ejecuta en un momento dado, cómo se mantienen las variables locales de los métodos en memoria y cómo sabe un método a dónde regresar una vez que termina su ejecución.

Hablaremos, brevemente, sobre las técnicas de simulación mediante la generación de números aleatorios y desarrollaremos una versión de un juego de dados conocido como "craps", el cual utiliza la mayoría de las técnicas de programación que ha aprendido hasta este capítulo. Además, aprenderá a declarar valores que no pueden cambiar (es decir, constantes) en sus programas.

Muchas de las clases que utilizará o creará mientras desarrolla aplicaciones tendrán más de un método con el mismo nombre. Esta técnica, conocida como sobrecarga, se utiliza para implementar métodos que realizan tareas similares, para argumentos de distintos tipos, o para un número distinto de argumentos.

En el capítulo 18, Recursividad (en el sitio Web), continuaremos nuestro debate sobre los métodos. La recursividad proporciona una manera intrigante de pensar acerca de los métodos y los algoritmos.

6.2 Módulos de programas en Java

Para escribir programas en Java, se combinan los nuevos métodos y clases con los métodos y clases predefinidos, que están disponibles en la **Interfaz de Programación de Aplicaciones de Java** (también conocida como la **API de Java** o **biblioteca de clases de Java**) y en diversas bibliotecas de clases. Por lo general, las clases relacionadas están agrupadas en paquetes, de manera que se pueden importar a los

programas y reutilizarse. En el capítulo 8 aprenderá a agrupar sus propias clases en paquetes. La API de Java proporciona una vasta colección de clases predefinidas que contienen métodos para realizar cálculos matemáticos comunes, manipulaciones de cadenas, manipulaciones de caracteres, operaciones de entrada/salida, operaciones de bases de datos, operaciones de red, procesamiento de archivos, comprobación de errores y muchas otras tareas útiles.

Observación de ingeniería de software 6.1

Procure familiarizarse con la vasta colección de clases y métodos que proporciona la API de Java (`download.oracle.com/javase/6/docs/api/`). En la sección 6.8 presentaremos las generalidades acerca de varios paquetes comunes. En el apéndice E, le explicaremos cómo navegar por la documentación de la API. Evite reinventar la rueda. Cuando sea posible, reutilice las clases y métodos de la API de Java. Esto reduce el tiempo de desarrollo de los programas y evita que se introduzcan errores de programación.

Los métodos (también conocidos como **funciones** o **procedimientos** en otros lenguajes) permiten al programador dividir un programa en módulos, por medio de la separación de sus tareas en unidades autónomas. Usted ha declarado métodos en todos los programas que ha escrito. Las instrucciones en los cuerpos de los métodos se escriben sólo una vez, se ocultan de otros métodos y se pueden reutilizar desde varias ubicaciones en un programa.

Una razón para dividir un programa en módulos mediante los métodos es el enfoque "divide y vencerás", que hace que el desarrollo de programas sea más fácil de administrar, ya que se pueden construir programas a partir de piezas pequeñas y simples. Otra razón es la **reutilización de software** (usar los métodos existentes como bloques de construcción para crear nuevos programas). A menudo se pueden crear programas a partir de métodos estandarizados, en vez de tener que crear código personalizado. Por ejemplo, en los programas anteriores no tuvimos que definir cómo leer datos del teclado; Java proporciona estas herramientas en la clase `Scanner`. Una tercera razón es para evitar la repetición de código. El proceso de dividir un programa en métodos significativos hace que el programa sea más fácil de depurar y mantener.

Observación de ingeniería de software 6.2

Para promover la reutilización de software, cada método debe limitarse de manera que realice una sola tarea bien definida, y su nombre debe expresar esa tarea con efectividad.

Tip para prevenir errores 6.1

Un método pequeño que lleva a cabo una tarea es más fácil de probar y depurar que uno más grande que realiza muchas tareas.

Observación de ingeniería de software 6.3

Si no puede elegir un nombre conciso que exprese la tarea de un método, tal vez esté tratando de realizar diversas tareas en un mismo método. Por lo general, es mejor dividirlo en varias declaraciones de métodos más pequeños.

Como sabe, un método se invoca mediante una llamada, y cuando el método que se llamó completa su tarea, devuelve un resultado, o simplemente el control al método que lo llamó. Una analogía a esta estructura de programa es la forma jerárquica de la administración (figura 6.1). Un jefe (el solicitante) pide a un trabajador (el método llamado) que realice una tarea y que le reporte (devuelva) los resultados después de completar la tarea. El método jefe no sabe cómo el método trabajador realiza sus tareas designadas. Tal vez el trabajador llame a otros métodos trabajadores, sin que lo sepa el jefe. Este "ocultamiento" de los detalles de implementación fomenta la buena ingeniería de software. La figura 6.1 muestra al método `jefe` comunicándose con varios métodos trabajadores en forma jerárquica. El método `jefe` divide las responsabilidades entre los diversos métodos `trabajador`. Aquí `trabajador1` actúa como "método jefe" de `trabajador4` y `trabajador5`.

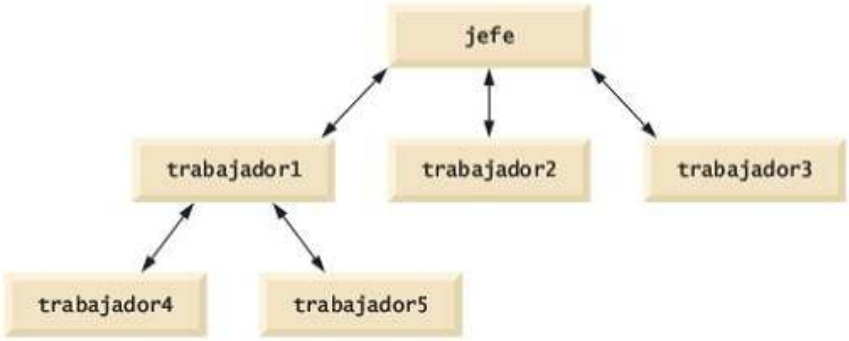


Fig. 6.1 | Relación jerárquica entre el método jefe y los métodos trabajadores.

6.3 Métodos `static`, campos `static` y la clase `Math`

Aunque la mayoría de los métodos se ejecutan en respuesta a las llamadas a métodos en objetos específicos, éste no es siempre el caso. Algunas veces un método realiza una tarea que no depende del contenido de ningún objeto. Dicho método se aplica a la clase en la que está declarado como un todo, y se conoce como método `static` o **método de clase**. Es común que las clases contengan métodos `static` convenientes para realizar tareas comunes. Por ejemplo, recuerde que en la figura 5.6 utilizamos el método `static` `pow` de la clase `Math` para elevar un valor a una potencia. Para declarar un método como `static`, coloque la palabra clave `static` antes del tipo de valor de retorno en la declaración del método. Para cualquier clase importada en su programa, puede llamar a cualquier método `static` especificando el nombre de la clase en la que está declarado el método, seguido de un punto (`.`) y del nombre del método, como sigue:

```
NombreClase.nombreMétodo( argumentos )
```

Aquí utilizaremos varios métodos de la clase `Math` para presentar el concepto de los métodos `static`. La clase `Math` cuenta con una colección de métodos que nos permiten realizar cálculos matemáticos comunes. Por ejemplo, podemos calcular la raíz cuadrada de `900.0` con una llamada al siguiente método `static`:

```
Math.sqrt( 900.0 )
```

La expresión anterior se evalúa como `30.0`. El método `sqrt` recibe un argumento de tipo `double` y devuelve un resultado del mismo tipo. Para imprimir el valor de la llamada anterior al método en una ventana de comandos, podríamos escribir la siguiente instrucción:

```
System.out.println( Math.sqrt( 900.0 ) );
```

En esta instrucción, el valor que devuelve `sqrt` se convierte en el argumento para el método `println`. Observe que no hubo necesidad de crear un objeto `Math` antes de llamar al método `sqrt`. Observe también que todos los métodos de la clase `Math` son `static`; por lo tanto, cada uno se llama anteponiendo al nombre del método el nombre de la clase `Math` y el separador punto (`.`).

Observación de ingeniería de software 6.4

La clase `Math` es parte del paquete `java.lang`, que el compilador importa de manera implícita, por lo que no es necesario importarla para utilizar sus métodos.

Los argumentos para los métodos pueden ser constantes, variables o expresiones. Si `c = 13.0`, `d = 3.0` y `f = 4.0`, entonces la instrucción

```
System.out.println( Math.sqrt( c + d * f ) );
```

calcula e imprime la raíz cuadrada de `13.0 + 3.0 * 4.0 = 25.0`; a saber, `5.0`. La figura 6.2 sintetiza varios de los métodos de la clase `Math`. En la figura, *x* y *y* son de tipo `double`.

Método	Descripción	Ejemplo

`abs(x)`	valor absoluto de *x*	`abs( 23.7 )` es `23.7`
		`abs( 0.0 )` es `0.0`
		`abs( -23.7 )` es `23.7`

`ceil(x)`	redondea *x* al entero más pequeño que no sea menor de *x*	`ceil( 9.2 )` es `10.0` `ceil( -9.8 )` es `-9.0`
`cos(x)`	coseno trigonométrico de *x* (*x* está en radianes)	`cos( 0.0 )` es `1.0`
`exp(x)`	método exponencial e^x	`exp( 1.0 )` es `2.71828` `exp( 2.0 )` es `7.38906`
`floor(x)`	redondea *x* al entero más grande que no sea mayor de *x*	`floor( 9.2 )` es `9.0` `floor( -9.8 )` es `-10.0`
`log(x)`	logaritmo natural de *x* (base *e*)	`log( Math.E )` es `1.0` `log( Math.E * Math.E )` es `2.0`
`max(x, y)`	el valor más grande de *x* y *y*	`max( 2.3, 12.7 )` es `12.7` `max( -2.3, -12.7 )` es `-2.3`
`min(x, y)`	el valor más pequeño de *x* y *y*	`min( 2.3, 12.7 )` es `2.3` `min( -2.3, -12.7 )` es `-12.7`
`pow(x, y)`	*x* elevado a la potencia *y* (x^y)	`pow( 2.0, 7.0 )` es `128.0` `pow( 9.0, 0.5 )` es `3.0`
`sin(x)`	seno trigonométrico de *x* (*x* está en radianes)	`sin( 0.0 )` es `0.0`
`sqrt(x)`	raíz cuadrada de *x*	`sqrt( 900.0 )` es `30.0`
`tan(x)`	tangente trigonométrica de *x* (*x* está en radianes)	`tan( 0.0 )` es `0.0`

Fig. 6.2 | Métodos de la clase `Math`.

Constantes `PI` y `E` de la clase `Math`

La clase `Math` declara dos campos que representan unas constantes matemáticas de uso común: `Math.PI` y `Math.E`. La constante `Math.PI` (3.141592653589793) es la proporción de la circunferencia de un círculo con su diámetro. La constante `Math.E` (2.718281828459045) es el valor de la base para los logaritmos naturales (que se calculan con el método `static log` de la clase `Math`). Estos campos se declaran en la clase `Math` con los modificadores `public`, `final` y `static`. Al hacerlos `public`, usted puede utilizar estos campos en sus propias clases. Cualquier campo declarado con la palabra clave `final` es constante; su valor no puede modificarse después de inicializar el campo. Tanto `PI` como `E` se declaran como `final`, ya que sus valores nunca cambian. Al hacer a estos campos `static`, se puede acceder a ellos mediante el nombre de clase `Math` y un separador de punto (.), justo igual que los métodos de la clase `Math`. En la sección 3.4 vimos que, cuando cada objeto de una clase mantiene su propia copia de un atributo, el campo que representa a ese atributo se conoce también como variable de instancia: cada objeto (instancia) de la clase tiene una instancia separada de la variable en memoria. Hay campos para los cuales cada objeto de una clase *no* tiene una instancia separada de ese campo. Éste es el caso con los campos `static`, que se conocen también como **variables de clase**. Cuando se crean ob-

jetos de una clase que contiene campos `static`, todos los objetos de la clase comparten una copia de los campos `static` de esa clase. En conjunto, las variables de clase (es decir, las variables `static`) y las variables de instancia representan a los campos de una clase. En la sección 8.11 aprenderá más acerca de los campos `static`.

¿Por qué el método `main` se declara como `static`?

Cuando se ejecuta la máquina virtual de Java (JVM) con el comando java, la JVM trata de invocar al método `main` de la clase que usted le especifica, cuando no se han creado objetos de esa clase. Al declarar a `main` como `static`, la JVM puede invocar a `main` sin tener que crear una instancia de la clase. Cuando usted ejecuta su aplicación, especifica el nombre de su clase como un argumento para el comando `java`, como sigue

```
java NombreClase argumento1 argumento2 ...
```

La JVM carga la clase especificada por *NombreClase* y utiliza el nombre de esa clase para invocar al método `main`. En el comando anterior, *NombreClase* es un **argumento de línea de comandos** para la JVM, que le indica cuál clase debe ejecutar. Después del *NombreClase*, también puede especificar una lista de objetos `String` (separados por espacios) como argumentos de línea de comandos, que la JVM pasará a su aplicación. Dichos argumentos pueden utilizarse para especificar opciones (por ejemplo, un nombre de archivo) para ejecutar la aplicación. Como veremos en el capítulo 7, Arreglos y objetos `ArrayList`, su aplicación puede acceder a esos argumentos de línea de comandos y utilizarlos para personalizar la aplicación.

6.4 Declaración de métodos con múltiples parámetros

A menudo, los métodos requieren más de una pieza de información para realizar sus tareas. Ahora le mostraremos cómo escribir métodos con varios parámetros.

La aplicación de la figuras 6.3 utiliza un método llamado `maximo` para determinar y devolver el mayor de tres valores `double`. En `main`, las líneas 14 a la 18 piden al usuario que introduzca tres valores `double`, y después los leen. La línea 21 llama al método `maximo` (declarado en las líneas 28 a 41) para determinar el mayor de los tres valores que recibe como argumentos. Cuando el método `maximo` devuelve el resultado a la línea 21, el programa asigna el valor de retorno de `maximo` a la variable local `resultado`. Después, la línea 24 imprime el valor máximo. Al final de esta sección, hablaremos sobre el uso del operador + en la línea 24.

```
1  // Fig. 6.3: BuscadorMaximo.java
2  // Método maximo, declarado por el programador, con tres parámetros double.
3  import java.util.Scanner;
4
5  public class BuscadorMaximo
6  {
7     // obtiene tres valores de punto flotante y determina el valor máximo
8     public static void main( String[] args )
9     {
10       // crea objeto Scanner para introducir datos desde la ventana de comandos
11       Scanner entrada = new Scanner( System.in );
12
13       // pide y recibe como entrada tres valores de punto flotante
14       System.out.print(
15          "Escriba tres valores de punto flotante, separados por espacios: " );
```

Fig. 6.3 | Método `maximo`, declarado por el programador, que tiene tres parámetros `double` (parte 1 de 2).

```
16         double numero1 = entrada.nextDouble(); // lee el primer valor double
17         double numero2 = entrada.nextDouble(); // lee el segundo valor double
18         double numero3 = entrada.nextDouble(); // lee el tercer valor double
19
20         // determina el valor máximo
21         double resultado = maximo( numero1, numero2, numero3 );
22
23         // muestra el valor máximo
24         System.out.println( "El maximo es: " + resultado );
25      } // fin de main
26
27      // devuelve el máximo de sus tres parámetros double
28      public static double maximo( double x, double y, double z )
29      {
30         double valorMaximo = x; // asume que x es el mayor para empezar
31
32         // determina si y es mayor que valorMaximo
33         if ( y > valorMaximo )
34            valorMaximo = y;
35
36         // determina si z es mayor que valorMaximo
37         if ( z > valorMaximo )
38            valorMaximo = z;
39
40         return valorMaximo;
41      } // fin del método maximo
42   } // fin de la clase BuscadorMaximo
```

```
Escriba tres valores de punto flotante, separados por espacios: 9.35 2.74 5.1
El maximo es: 9.35
```

```
Escriba tres valores de punto flotante, separados por espacios: 5.8 12.45 8.32
El maximo es: 12.45
```

```
Escriba tres valores de punto flotante, separados por espacios: 6.46 4.12 10.54
El maximo es: 10.54
```

Fig. 6.3 | Método `maximo`, declarado por el programador, que tiene tres parámetros `double` (parte 2 de 2).

Las palabras clave `public` *y* `static`

La declaración del método `maximo` comienza con la palabra clave `public` para indicar que el método está "disponible para el público": puede llamarse desde los métodos de otras clases. La palabra clave `static` permite al método `main` (otro método `static`) llamar a `maximo`, como se muestra en la línea 21, sin tener que calificar el nombre del método con el nombre de la clase `MaximumFinder`; los métodos `static` en la misma clase pueden llamarse unos a otros de manera directa. Cualquier otra clase que utilice a `maximo` debe calificar por completo el nombre del método, con el nombre de la clase.

El método `maximo`

Considere la declaración del método `maximo` (líneas 28 a 41). La línea 28 indica que el método devuelve un valor `double`, que el nombre del método es `maximo` y requiere tres parámetros `double` (`x`, `y` y `z`) para

realizar su tarea. Los parámetros múltiples se especifican como una lista separada por comas. Cuando se hace la llamada a `maximo` en la línea 21, los parámetros `x`, `y` y `z` se inicializan con los valores de los argumentos `numero1`, `numero2` y `numero3`, respectivamente. Debe haber un argumento en la llamada al método para cada parámetro en la declaración del método. Además, cada argumento debe ser *consistente* con el tipo del parámetro correspondiente. Por ejemplo, un parámetro de tipo `double` puede recibir valores como 7.35, 22 o –0.03456, pero no objetos `String` como `"hola"`, ni los valores `boolean`os `true` o `false`. En la sección 6.7 veremos los tipos de argumentos que pueden proporcionarse en la llamada a un método para cada parámetro de un tipo primitivo.

Para determinar el valor máximo, comenzamos con la suposición de que el parámetro `x` contiene el valor más grande, por lo que la línea 30 declara la variable local `valorMaximo` y la inicializa con el valor del parámetro `x`. Desde luego, es posible que el parámetro `y` o `z` contenga el valor más grande, por lo que debemos comparar cada uno de estos valores con `valorMaximo`. La instrucción `if` en las líneas 33 y 34 determina si `y` es mayor que `valorMaximo`. De ser así, la línea 34 asigna `y` a `valorMaximo`. La instrucción `if` en las líneas 37 y 38 determina si `z` es mayor que `valorMaximo`. De ser así, la línea 38 asigna `z` a `valorMaximo`. En este punto, el mayor de los tres valores reside en `valorMaximo`, por lo que la línea 40 devuelve ese valor a la línea 21. Cuando el control del programa regresa al punto en donde se llamó al método `maximo`, los parámetros `x`, `y` y `z` de `maximo` ya no existen en la memoria.

Observación de ingeniería de software 6.5

Los métodos pueden devolver a lo máximo un valor, pero el valor devuelto puede ser una referencia a un objeto que contenga muchos valores.

Observación de ingeniería de software 6.6

Las variables deben declararse como campos de una clase sólo si se requiere su uso en más de un método de la clase, o si el programa debe almacenar sus valores entre las llamadas a los métodos de ella.

Error común de programación 6.1

Declarar parámetros del mismo tipo para un método, como `float x, y` en vez de `float x, float y` es un error de sintaxis; se requiere un tipo para cada parámetro en la lista de parámetros.

Implementación del método `maximo` mediante la reutilización del método `Math.max`

Todo el cuerpo de nuestro método para encontrar el valor máximo podría también implementarse mediante dos llamadas a `Math.max`, como se muestra a continuación:

```
return Math.max( x, Math.max( y, z ) );
```

La primera llamada a `Math.max` especifica los argumentos `x` y `Math.max( y, z )`. *Antes* de poder llamar a cualquier método, todos sus argumentos deben evaluarse para determinar sus valores. Si un argumento es una llamada a un método, es necesario realizarla para determinar su valor de retorno. Por lo tanto, en la instrucción anterior, primero se evalúa `Math.max( y, z )` para determinar el máximo entre `y` y `z`. Después el resultado se pasa como el segundo argumento para la otra llamada a `Math.max`, que devuelve el mayor de sus dos argumentos. Éste es un buen ejemplo de la reutilización de software: buscamos el mayor de los tres valores reutilizando `Math.max`, el cual busca el mayor de dos valores. Observe lo conciso de este código, en comparación con las líneas 30 a 38 de la figura 6.3.

Ensamblado de cadenas mediante la concatenación

Java permite crear objetos `String` mediante el uso de los operadores `+` o `+=` para formar objetos `String` más grandes. A esto se le conoce como **concatenación de objetos `String`**. Cuando ambos operandos del

operador + son objetos `String`, el operador + crea un nuevo objeto `String` en el cual los caracteres del operando derecho se colocan al final de los caracteres en el operando izquierdo. Por ejemplo, la expresión `"hola " + "a todos"` crea el objeto `String "hola a todos"`.

En la línea 24 de la figura 6.3, la expresión `"El maximo es: " + resultado` utiliza el operador + con operandos de tipo `String` y `double`. *Cada valor primitivo y cada objeto en Java tienen una representación `String`.* Cuando uno de los operandos del operador + es un objeto `String`, el otro se convierte en `String` y después se *concatenan* los dos. En la línea 24, el valor `double` se convierte en su representación `String` y se coloca al final del objeto `String "El maximo es: "`. Si hay ceros a la derecha en un valor `double`, éstos se *descartan* cuando el número se convierte en objeto `String`; por ejemplo, el número 9.3500 se representaría como 9.35.

Los valores primitivos que se utilizan en la concatenación de objetos `String` se convierten en objetos `String`. Si un valor `boolean` se concatena con un objeto `String`, se convierte en el objeto `String "true"` o `"false"`. Todos los objetos tienen un método llamado `toString` que devuelve una representación `String` del objeto. (Hablaremos con más detalle sobre el método `toString` en los siguientes capítulos). Cuando se concatena un objeto con un `String`, se hace una llamada implícita al método `toString` de ese objeto para obtener la representación `String` del mismo. Es posible llamar al método `toString` en forma explícita.

Usted puede dividir las literales `String` grandes en varios objetos `String` más pequeños, para colocarlos en varias líneas de código y mejorar la legibilidad. En este caso, los objetos `String` pueden reensamblarse mediante el uso de la concatenación. En el capítulo 16 (en el sitio Web) hablaremos sobre los detalles de los objetos `String`.

Error común de programación 6.2

Es un error de sintaxis dividir una literal `String` en varias líneas. Si es necesario, puede dividir una literal `String` en varios objetos `String` más pequeños y utilizar la concatenación para formar la literal `String` deseada.

Error común de programación 6.3

Confundir el operador +, que se utiliza para la concatenación de cadenas, con el operador + que se utiliza para la suma, puede producir resultados extraños. Java evalúa los operandos de un operador de izquierda a derecha. Por ejemplo, si la variable entera y tiene el valor 5, la expresión `"y + 2 = " + y + 2` produce la cadena `"y + 2 = 52"`, no `"y + 2 = 7"`, ya que primero el valor de y (5) se concatena con la cadena `"y + 2 ="` y después el valor 2 se concatena con la nueva cadena `"y + 2 = 5"` más larga. La expresión `"y + 2 =" + (y + 2)` produce el resultado deseado `"y + 2 = 7"`.

6.5 Notas acerca de cómo declarar y utilizar los métodos

Hay tres formas de llamar a un método:

1. Utilizar el nombre de un método por sí solo para llamar a otro método de la *misma* clase; como `maximo( numero1, numero2, numero3 )` en la línea 21 de la figura 6.3.
2. Usar una variable que contiene una referencia a un objeto, seguida de un punto (.) y del nombre del método para llamar a un método no `static` del objeto al que se hace referencia; como la llamada al método en la línea 13 de la figura 5.10, `miLibroCalificaciones.mostrarMensaje()`, con lo cual se llama a un método de la clase `LibroCalificaciones` desde el método `main` de `PruebaLibroCalificaciones`.
3. Utilizar el nombre de la clase y un punto (.) para llamar a un método `static` de una clase, como `Math.sqrt( 900.0 )` en la sección 6.3.

Un método `static` *sólo* puede llamar directamente a otros métodos `static` de la misma clase (es decir, mediante el nombre del método por sí solo) y *sólo* puede manipular de manera directa variables `static` en la misma clase. Para acceder a los miembros no `static` de la clase, un método `static` debe usar

una referencia a un objeto de esa clase. Recuerde que los métodos `static` se relacionan con una clase como un todo, mientras que los métodos no `static` se asocian con una instancia específica (objeto) de la clase y pueden manipular las variables de instancia de ese objeto. Es posible que existan muchos objetos de una clase al mismo tiempo, cada uno con sus propias copias de las variables de instancia. Suponga que un método `static` invoca a un método no `static` en forma directa. ¿Cómo sabría el método qué variables de instancia manipular de cuál objeto? ¿Qué ocurriría si no existieran objetos de la clase en el momento en el que se invocara el método no `static`? Por lo tanto, Java no permite que un método `static` acceda de manera directa a los miembros no `static` de la misma clase.

Existen tres formas de regresar el control a la instrucción que llama a un método. Si el método no devuelve un resultado, el control regresa cuando el flujo del programa llega a la llave derecha de terminación del método, o cuando se ejecuta la instrucción

```
return;
```

Si el método devuelve un resultado, la instrucción

```
return expresión;
```

evalúa la *expresión* y después devuelve el resultado al método que hizo la llamada.

Error común de programación 6.4

Declarar un método fuera del cuerpo de la declaración de una clase, o dentro del cuerpo de otro método es un error de sintaxis.

Error común de programación 6.5

Colocar un punto y coma después del paréntesis derecho que encierra la lista de parámetros de la declaración de un método es un error de sintaxis.

Error común de programación 6.6

Volver a declarar el parámetro de un método como una variable local en el cuerpo de ese método es un error de compilación.

Error común de programación 6.7

Olvidar devolver un valor de un método que debe regresar un valor es un error de compilación. Si se especifica un tipo de valor de retorno distinto de `void`, el método debe contener una instrucción `return` que devuelva un valor consistente con el tipo de valor de retorno del método. Devolver un valor de un método cuyo tipo de valor de retorno se haya declarado como `void` es un error de compilación.

6.6 La pila de llamadas a los métodos y los registros de activación

Para comprender la forma en que Java realiza las llamadas a los métodos, necesitamos considerar primero una estructura de datos (una colección de elementos de datos relacionados) conocida como **pila**. A la que podemos considerar como una analogía de una pila de platos. Cuando se coloca un plato en ella, por lo general se coloca en la parte superior (lo que se conoce como **meter** el plato en la pila). De manera similar, cuando se extrae un plato de la pila, siempre se extrae de la parte superior (lo que se conoce como sacar el plato de la pila). Las pilas se denominan **estructuras de datos "último en entrar, primero en salir"** (**UEPS**, **LIFO** por sus siglas en inglés: **last-in, first-out**); el último elemento que se mete (inserta) en la pila es el primero que se saca (extrae) de ella.

Cuando un programa llama a un método, el método llamado debe saber cómo regresar al que lo llamó, por lo que la dirección de retorno del método que hizo la llamada se mete en la **pila de ejecución del programa** (también conocida como **pila de llamadas a los métodos**). Si ocurre una serie de llamadas a métodos, las direcciones de retorno sucesivas se meten en la pila, en el orden "último en entrar, primero en salir", para que cada método pueda regresar al que lo llamó.

La pila de ejecución del programa también contiene la memoria para las variables locales que se utilizan en cada invocación de un método, durante la ejecución de un programa. Estos datos, que se almacenan como una porción de la pila de ejecución del programa, se conocen como el **registro de activación** o **marco de pila** de la llamada a un método. Cuando se hace la llamada a un método, el registro de activación para la llamada se mete en la pila de ejecución del programa. Cuando el método regresa al que lo llamó, el registro de activación para esa llamada al método se saca de la pila y esas variables locales ya no son conocidas para el programa. Si una variable local que contiene una referencia a un objeto es la única variable en el programa con una referencia a ese objeto, cuando se saca de la pila el registro de activación que contiene a esa variable local, el programa ya no puede acceder a ese objeto, y la JVM lo eliminará de la memoria en algún momento dado, durante la "recolección de basura". En la sección 8.10 hablaremos sobre la recolección de basura.

Desde luego que la cantidad de memoria en una computadora es finita, por lo que sólo puede utilizarse cierta cantidad para almacenar los registros de activación en la pila de ejecución del programa. Si ocurren más llamadas a métodos de las que se puedan almacenar sus registros de activación, se produce un error conocido como **desbordamiento de pila**.

6.7 Promoción y conversión de argumentos

Otra característica importante de las llamadas a los métodos es la **promoción de argumentos**: convertir el valor de un argumento, si es posible, al tipo que el método espera recibir en su correspondiente parámetro. Por ejemplo, un programa puede llamar al método `sqrt` de `Math` con un argumento `int`, aun cuando el método espera recibir un argumento `double`. La instrucción

```
System.out.println( Math.sqrt( 4 ) );
```

evalúa `Math.sqrt(4)` correctamente e imprime el valor `2.0`. La lista de parámetros de la declaración del método hace que Java convierta el valor `int` `4` en el valor `double` `4.0` antes de pasar ese valor al método `sqrt`. Dichas conversiones pueden ocasionar errores de compilación, si no se satisfacen las **reglas de promoción** de Java. Estas reglas especifican qué conversiones son permitidas; esto es, qué conversiones pueden realizarse sin perder datos. En el ejemplo anterior de `sqrt`, un `int` se convierte en `double` sin modificar su valor. No obstante, la conversión de un `double` a un `int` trunca la parte fraccionaria del valor `double`; por consecuencia, se pierde parte del valor. La conversión de tipos de enteros largos a tipos de enteros pequeños (por ejemplo, de `long` a `int`) puede también producir valores modificados.

Las reglas de promoción se aplican a las expresiones que contienen valores de dos o más tipos primitivos, y a los valores de tipos primitivos que se pasan como argumentos para los métodos. Cada valor se promueve al tipo "más alto" en la expresión. En realidad, la expresión utiliza una copia temporal de cada valor; los tipos de los valores originales permanecen sin cambios. La figura 6.4 lista los tipos primitivos y los tipos a los cuales se puede promover cada uno de ellos. Las promociones válidas para un tipo dado siempre se realizan a un tipo más alto en la tabla. Por ejemplo, un `int` puede promoverse a los tipos más altos `long`, `float` y `double`.

Al convertir valores a tipos inferiores en la tabla de la figura 6.4, se producirán distintos valores si el tipo inferior no puede representar el valor del tipo superior (por ejemplo, el valor `int` 2000000 no puede representarse como un `short`, y cualquier número de punto flotante con dígitos después de su punto decimal no pueden representarse en un tipo entero como `long`, `int` o `short`). Por lo tanto, en casos en los que la información puede perderse debido a la conversión, el compilador de Java requiere que utilicemos

Tipo	Promociones válidas
`double`	Ninguna
`float`	`double`
`long`	`float` o `double`
`int`	`long`, `float` o `double`
`char`	`int`, `long`, `float` o `double`
`short`	`int`, `long`, `float` o `double` (pero no `char`)
`byte`	`short`, `int`, `long`, `float` or `double` (pero no `char`)
`boolean`	Ninguna (los valores `boolean` no se consideran números en Java)

Fig. 6.4 | Promociones permitidas para los tipos primitivos.

un operador de conversión (el cual presentamos en la sección 4.9) para forzar explícitamente la conversión; en caso contrario, ocurre un error de compilación. Eso nos permite "tomar el control" del compilador. En esencia decimos, "Sé que esta conversión podría ocasionar pérdida de información, pero para mis fines aquí, eso está bien". Suponga que el método cuadrado calcula el cuadrado de un entero y por ende requiere un argumento `int`. Para llamar a cuadrado con un argumento `double` llamado `valorDouble`, tendríamos que escribir la llamada al método de la siguiente forma:

```
cuadrado( (int) valorDouble )
```

La llamada a este método convierte explícitamente una *copia* del valor de `valorDouble` a un entero, para usarlo en el método cuadrado. Por ende, si el valor de `valorDouble` es `4.5`, el método recibe el valor `4` y devuelve `16`, no `20.25`.

Error común de programación 6.8

Convertir un valor de tipo primitivo a otro tipo primitivo puede modificar ese valor, si el nuevo tipo no es una promoción válida. Por ejemplo, convertir un valor de punto flotante a un valor entero puede introducir errores de truncamiento (pérdida de la parte fraccionaria) en el resultado.

6.8 Paquetes de la API de Java

Como hemos visto, Java contiene muchas clases predefinidas que se agrupan en categorías de clases relacionadas, llamadas paquetes. En conjunto, nos referimos a estos paquetes como la Interfaz de programación de aplicaciones de Java (API de Java), o biblioteca de clases de Java. Uno de los puntos más fuertes de Java se debe a las miles de clases de la API. Algunos paquetes clave de la API de Java se describen en la figura 6.5, que representa sólo una pequeña parte de los componentes reutilizables en la API de Java.

El conjunto de paquetes disponibles en Java SE 6 es bastante extenso. Además de los paquetes sintetizados en la figura 6.5, Java SE 6 incluye paquetes para gráficos complejos, interfaces gráficas de usuario avanzadas, impresión, redes avanzadas, seguridad, procesamiento de bases de datos, multimedia, accesibilidad (para personas con discapacidades), programación concurrente, criptografía, procesamiento de XML y muchas otras funciones. Para una visión general de los paquetes en Java SE 6, visite:

```
download.oracle.com/javase/6/docs/api/overview-summary.html
```

Además, muchos otros paquetes están disponibles también para descargarse en `java.sun.com`.

Paquete	Descripción
`java.applet`	El **Paquete Applet de Java** contiene una clase y varias interfaces requeridas para crear applets de Java: programas que se ejecutan en los navegadores Web. En el capítulo 23, Applets y Java Web Start (en inglés, en el sitio Web), hablaremos sobre los applets; en el capítulo 10, Programación orientada a objetos: polimorfismo, trataremos el tema de las interfaces.
`java.awt`	El **Paquete Abstract Window Toolkit de Java** contiene las clases e interfaces requeridas para crear y manipular interfaces GUI en las primeras versiones de Java. En las versiones actuales, por lo general se utilizan los componentes de la GUI de Swing que se incluyen en los paquetes `javax.swing`. (Algunos elementos del paquete `java.awt` se describen en el capítulo 14, Componentes de la GUI: Parte 1, en el capítulo 15, Gráficos y Java 2D [ambos en el sitio Web], y en el capítulo 25, Componentes de la GUI: Parte 2 [en inglés, en el sitio Web]).
`java.awt.event`	El **Paquete Abstract Window Toolkit Event de Java** contiene clases e interfaces que habilitan el manejo de eventos para componentes de la GUI en los paquetes `java.awt` y `javax.swing`. (Vea el capítulo 14, Componentes de la GUI: Parte 1 [en el sitio Web], y el capítulo 25, Componentes de la GUI: Parte 2 [en inglés, en el sitio Web]).
`java.awt.geom`	El **Paquete Formas 2D de Java** contiene clases e interfaces para trabajar con las herramientas de gráficos bidimensionales avanzadas de Java (vea el capítulo 15, Gráficos y Java 2D [en el sitio Web]).
`java.io`	El **Paquete de Entrada/Salida de Java** contiene clases e interfaces que permiten a los programas recibir datos de entrada y mostrar datos de salida. (Vea el capítulo 17, Archivos, flujos y serialización de objetos [en el sitio Web]).
`java.lang`	El **Paquete del Lenguaje Java** contiene clases e interfaces (descritas a lo largo de este texto) requeridas por muchos programas de Java. Este paquete es importado por el compilador en todos los programas.
`java.net`	El **Paquete de Red de Java** contiene clases e interfaces que permiten a los programas comunicarse mediante redes de computadoras, como Internet. (Vea el capítulo 27, Redes [en inglés, en el sitio Web]).
`java.sql`	El **Paquete JDBC** contiene clases e interfaces para trabajar con bases de datos (vea el capítulo 28, Acceso a bases de datos con JDBC [en inglés, en el sitio Web]).
`java.util`	El **Paquete de Utilerías de Java** contiene clases e interfaces utilitarias, que permiten acciones como manipulaciones de fecha y hora, procesamiento de números aleatorios (clase `Random`), y el almacenamiento y procesamiento de grandes cantidades de datos (vea el capítulo 20, Colecciones de genéricos [en inglés, en el sitio Web]).
`java.util.concurrent`	El **Paquete de Concurrencia de Java** contiene clases e interfaces utilitarias para implementar programas que puedan realizar varias tareas en paralelo (vea el capítulo 26, Multihilos [en inglés, en el sitio Web]).
`javax.media`	El **Paquete del Marco de Trabajo de Medios de Java** contiene clases e interfaces para trabajar con las herramientas multimedia de Java (vea el capítulo 24, Multimedia: applets y aplicaciones [en inglés en el sitio Web]).
`javax.swing`	El **Paquete de Componentes GUI Swing de Java** contiene clases e interfaces para los componentes de la GUI Swing de Java, los cuales ofrecen soporte para interfaces GUI portables (vea el capítulo 14, Componentes de la GUI: Parte 1 [en el sitio Web], y el capítulo 25, Componentes de la GUI: Parte 2 [en inglés, en el sitio Web]).

Fig. 6.5 | Paquetes de la API de Java (un subconjunto) (parte 1 de 2).

Paquete	Descripción
`javax.swing.event`	El **Paquete Swing Event de Java** contiene clases e interfaces que permiten el manejo de eventos (por ejemplo, responder a los clics del ratón) para los componentes de la GUI en el paquete `javax.swing` (vea el capítulo 14, Componentes de la GUI: Parte 1 [en el sitio Web], y el capítulo 25, Componentes de la GUI: Parte 2 [en inglés, en el sitio Web]).
`javax.xml.ws`	El **Paquete JAX-WS** contiene clases e interfaces para trabajar con los servicios Web en Java (vea el capítulo 31, servicios Web [en inglés, en el sitio Web]).

Fig. 6.5 | Paquetes de la API de Java (un subconjunto) (parte 2 de 2).

Puede localizar información adicional acerca de los métodos de una clase predefinida de Java en la documentación para la API de Java, en `download.oracle.com/javase/6/docs/api/`. Cuando visite este sitio, haga clic en el vínculo **Index** para ver un listado en orden alfabético de todas las clases y los métodos en la API de Java. Localice el nombre de la clase y haga clic en su vínculo para ver la descripción en línea de la clase. Haga clic en el vínculo **METHOD** para ver una tabla de los métodos de la clase. Cada método `static` se enlistará con la palabra "`static`" antes del tipo de valor de retorno del método.

6.9 Caso de estudio: generación de números aleatorios

Ahora analizaremos de manera breve una parte divertida de un tipo popular de aplicaciones de la programación: simulación y juegos. En ésta y en la siguiente sección desarrollaremos un programa de juego bien estructurado con varios métodos. El programa utiliza la mayoría de las instrucciones de control presentadas hasta este punto en el libro, e introduce varios conceptos de programación nuevos.

Hay algo en el ambiente de un casino de apuestas que anima a las personas: desde las elegantes mesas de caoba y fieltro para tirar dados, hasta las máquinas tragamonedas. Es el **elemento de azar**, la posibilidad de que la suerte convierta un bolsillo lleno de dinero en una montaña de riquezas. El elemento de azar puede introducirse en un programa mediante un objeto de la clase `Random` (paquete `java.util`), o mediante el método `static` llamado `random`, de la clase `Math`. Los objetos de la clase `Random` pueden producir valores aleatorios de tipo `boolean`, `byte`, `float`, `double`, `int`, `long` y gaussianos, mientras que el método `random` de la clase `Math` puede producir sólo valores de tipo `double` en el rango $0.0 \leq x < 1.0$, donde x es el valor regresado por el método `random`. En los siguientes ejemplos, usamos objetos de tipo `Random` para producir valores aleatorios.

Es posible crear un nuevo objeto generador de números aleatorios de la siguiente manera:

```
Random numerosAleatorios = new Random();
```

Después, este objeto puede usarse para generar valores `boolean`, `byte`, `float`, `double`, `int`, `long` y gaussianos; aquí sólo hablaremos sobre los valores `int` aleatorios. Para obtener más información sobre la clase `Random`, vaya a `download.oracle.com/javase/6/docs/api/java/util/Random.html`.

Considere la siguiente instrucción:

```
int valorAleatorio = numerosAleatorios.nextInt();
```

El método `nextInt` de la clase `Random` genera un valor `int` aleatorio en el rango de –2,147,483,648 a +2,147,483,647, inclusive. Si de verdad produce valores aleatorios, entonces cualquier valor en ese rango debería tener una oportunidad (o probabilidad) igual de ser elegido cada vez que se llame al método `nextInt`. Los números son en realidad **números seudoaleatorios** (una secuencia de valores produci-

dos por un cálculo matemático complejo). Ese cálculo utiliza la hora actual del día (que, desde luego, cambia de manera constante) para **sembrar** el generador de números aleatorios, de tal forma que cada ejecución de un programa produzca una secuencia distinta de valores aleatorios.

El rango de valores producidos directamente por el método `nextInt` es a menudo distinto del rango de valores requeridos en una aplicación de Java particular. Por ejemplo, un programa que simula el lanzamiento de una moneda sólo requiere 0 para "águila" y 1 para "sol". Un programa para simular el tiro de un dado de seis lados requeriría enteros aleatorios en el rango de 1 a 6. Un programa que adivine en forma aleatoria el siguiente tipo de nave espacial (de cuatro posibilidades distintas) que volará a lo largo del horizonte en un videojuego requeriría números aleatorios en el rango de 1 a 4. Para casos como éstos, la clase `Random` cuenta con otra versión del método `nextInt`, que recibe un argumento `int` y devuelve un valor desde 0 hasta el valor del argumento (pero sin incluirlo). Por ejemplo, para simular el lanzamiento de monedas, la siguiente instrucción devuelve 0 o 1.

```
int valorAleatorio = numerosAleatorios.nextInt( 2 );
```

Tirar un dado de seis lados

Para demostrar los números aleatorios, desarrollaremos un programa que simula 20 tiros de un dado de seis lados, y que muestra el valor de cada tiro. Para empezar, usaremos `nextInt` para producir valores aleatorios en el rango de 0 a 5, como se muestra a continuación:

```
cara = numerosAleatorios.nextInt( 6 );
```

El argumento 6 (que se conoce como el **factor de escala**) representa el número de valores únicos que `nextInt` debe producir (en este caso, seis: 0, 1, 2, 3, 4 y 5). A esta manipulación se le conoce como **escalar** el rango de valores producidos por el método `nextInt` de `Random`.

Un dado de seis lados tiene los números del 1 al 6 en sus caras, no del 0 al 5. Por lo tanto, **desplazamos** el rango de números producidos sumando un **valor de desplazamiento** (en este caso, 1) a nuestro resultado anterior, como en

```
cara = 1 + numerosAleatorios.nextInt( 6 );
```

El valor de desplazamiento (1) especifica el *primer* valor en el conjunto deseado de enteros aleatorios. La instrucción anterior asigna a `cara` un entero aleatorio en el rango de 1 a 6.

La figura 6.6 muestra dos resultados de ejemplo, los cuales confirman que los resultados del cálculo anterior son enteros en el rango de 1 a 6, y que cada ejecución del programa puede producir una secuencia distinta de números aleatorios. La línea 3 importa la clase `Random` del paquete `java.util`. La línea 9 crea el objeto `numerosAleatorios` de la clase `Random` para producir valores aleatorios. La línea 16 se ejecuta 20 veces en un ciclo para tirar el dado. La instrucción `if` (líneas 21 y 22) en el ciclo empieza una nueva línea de salida después de cada cinco números.

```
1  // Fig. 6.6: EnterosAleatorios.java
2  // Enteros aleatorios desplazados y escalados.
3  import java.util.Random; // el programa usa la clase Random
4
5  public class EnterosAleatorios
6  {
7     public static void main( String[] args )
8     {
9        Random numerosAleatorios = new Random(); // generador de números aleatorios
```

Fig. 6.6 | Enteros aleatorios desplazados y escalados (parte 1 de 2).

```
10         int cara; // almacena cada entero aleatorio generado
11
12         // itera 20 veces
13         for ( int contador = 1; contador <= 20; contador++ )
14         {
15            // elige entero aleatorio del 1 al 6
16            cara = 1 + numerosAleatorios.nextInt( 6 );
17
18            System.out.printf( "%d  ", cara ); // muestra el valor generado
19
20            // si contador es divisible entre 5, empieza una nueva línea de salida
21            if ( contador % 5 == 0 )
22               System.out.println();
23         } // fin de for
24      } // fin de main
25   } // fin de la clase EnterosAleatorios
```

```
1  5  3  6  2
5  2  6  5  2
4  4  4  2  6
3  1  6  2  2
```

```
6  5  4  2  6
1  2  5  1  3
6  3  2  2  1
6  4  2  6  4
```

Fig. 6.6 | Enteros aleatorios desplazados y escalados (parte 2 de 2).

Tirar un dado de seis lados 6,000,000 veces

Para mostrar que los números que produce `nextInt` ocurren con una probabilidad aproximadamente igual, simularemos 6,000,000 tiros de un dado con la aplicación de la figura 6.7. Cada entero de 1 a 6 debe aparecer cerca de 1,000,000 veces.

```
1  // Fig. 6.7: TirarDado.java
2  // Tirar un dado de seis lados 6,000,000 veces.
3  import java.util.Random;
4
5  public class TirarDado
6  {
7     public static void main( String[] args )
8     {
9        Random numerosAleatorios = new Random(); // generador de números aleatorios
10
11       int frecuencia1 = 0; // cuenta de veces que se tiró 1
12       int frecuencia2 = 0; // cuenta de veces que se tiró 2
13       int frecuencia3 = 0; // cuenta de veces que se tiró 3
14       int frecuencia4 = 0; // cuenta de veces que se tiró 4
15       int frecuencia5 = 0; // cuenta de veces que se tiró 5
16       int frecuencia6 = 0; // cuenta de veces que se tiró 6
17
```

Fig. 6.7 | Tirar un dado de seis lados 6,000,000 veces (parte 1 de 2).

```
18         int cara; // almacena el valor que se tiró más recientemente
19
20         // sintetiza los resultados de tirar un dado 6,000,000 veces
21         for ( int tiro = 1; tiro <= 6000000; tiro++ )
22         {
23            cara = 1 + numerosAleatorios.nextInt( 6 ); // número del 1 al 6
24
25            // determina el valor del tiro de 1 a 6 e incrementa el contador apropiado
26            switch ( cara )
27            {
28               case 1:
29                  ++frecuencia1; // incrementa el contador de 1s
30                  break;
31               case 2:
32                  ++frecuencia2; // incrementa el contador de 2s
33                  break;
34               case 3:
35                  ++frecuencia3; // incrementa el contador de 3s
36                  break;
37               case 4:
38                  ++frecuencia4; // incrementa el contador de 4s
39                  break;
40               case 5:
41                  ++frecuencia5; // incrementa el contador de 5s
42                  break;
43               case 6:
44                  ++frecuencia6; // incrementa el contador de 6s
45                  break; // opcional al final del switch
46            } // fin de switch
47         } // fin de for
48
49         System.out.println( "Cara\tFrecuencia" ); // encabezados de salida
50         System.out.printf( "1\t%d\n2\t%d\n3\t%d\n4\t%d\n5\t%d\n6\t%d\n",
51            frecuencia1, frecuencia2, frecuencia3, frecuencia4,
52            frecuencia5, frecuencia6 );
53      } // fin de main
54   } // fin de la clase TirarDado
```

```
Cara    Frecuencia
1       999501
2       1000412
3       998262
4       1000820
5       1002245
6       998760
```

```
Cara    Frecuencia
1       999647
2       999557
3       999571
4       1000376
5       1000701
6       1000148
```

Fig. 6.7 | Tirar un dado de seis lados 6,000,000 veces (parte 2 de 2).

Como se muestra en los dos bloques de resultados, al escalar y desplazar los valores producidos por el método `nextInt`, el programa puede simular de manera realista el tiro de un dado de seis lados. La aplicación utiliza instrucciones de control anidadas (la instrucción `switch` está anidada dentro de `for`) para determinar el número de ocurrencias de cada lado del dado. La instrucción `for` (líneas 21 a 47) itera 6,000,000 veces. Durante cada iteración, la línea 23 produce un valor aleatorio del 1 al 6. Después, ese valor se utiliza como la expresión de control (línea 26) de la instrucción `switch` (líneas 26 a 46). Con base en el valor de cara, la instrucción `switch` incrementa una de las seis variables contadores durante cada iteración del ciclo. Cuando veamos los arreglos en el capítulo 7, ¡le mostraremos una forma elegante de reemplazar toda la instrucción `switch` en este programa con una sola instrucción! Esta instrucción `switch` no tiene un caso `default`, ya que hemos creado una etiqueta `case` para todos los posibles valores que puede producir la expresión en la línea 23. Ejecute el programa y observe los resultados. Como verá, cada vez que ejecute el programa, producirá distintos resultados.

6.9.1 Escalamiento y desplazamiento generalizados de números aleatorios

Anteriormente simulamos el tiro de un dado de seis caras con la instrucción

```
cara = 1 + numerosAleatorios.nextInt( 6 );
```

Esta instrucción siempre asigna a la variable `cara` un entero en el rango `1 ≤ cara ≤ 6`. La amplitud de este rango (es decir, el número de enteros consecutivos en él) es 6, y el número inicial en el rango es 1. En la instrucción anterior, la amplitud del rango se determina con base en el número 6 que se pasa como argumento para el método `nextInt` de `Random`, y que el número inicial del rango es el número 1 que se suma a `numerosAleatorios.nextInt(6)`. Podemos generalizar este resultado de la siguiente manera:

```
numero = valorDesplazamiento + numerosAleatorios.nextInt( factorEscala );
```

en donde *valorDesplazamiento* especifica el primer número en el rango deseado de enteros consecutivos y *factorEscala* determina cuántos números hay en el rango.

También es posible elegir enteros al azar, a partir de conjuntos de valores distintos a los rangos de enteros consecutivos. Por ejemplo, para obtener un valor aleatorio de la secuencia 2, 5, 8, 11 y 14, podríamos utilizar la siguiente instrucción:

```
numero = 2 + 3 * numerosAleatorios.nextInt( 5 );
```

En este caso, `numerosAleatorios.nextInt(5)` produce valores en el rango de 0 a 4. Cada valor producido se multiplica por 3 para producir un número en la secuencia 0, 3, 6, 9 y 12. Después sumamos 2 a ese valor para desplazar el rango de valores y obtener un valor de la secuencia 2, 5, 8, 11 y 14. Podemos generalizar este resultado así:

```
numero = valorDesplazamiento +
   diferenciaEntreValores * numerosAleatorios.nextInt( factorEscala );
```

en donde *valorDesplazamiento* especifica el primer número en el rango deseado de valores, *diferenciaEntreValores* representa la diferencia constante entre números consecutivos en la secuencia y *factorEscala* especifica cuántos números hay en el rango.

6.9.2 Repetitividad de números aleatorios para prueba y depuración

Los métodos de la clase `Random` en realidad generan números seudoaleatorios con base en cálculos matemáticos complejos. La secuencia de números parece ser aleatoria. El cálculo que produce los números seudoaleatorios utiliza la hora del día como **valor de semilla** para cambiar el punto inicial de la secuen-

cia. Cada nuevo objeto `Random` se siembra a sí mismo con un valor con base en el reloj del sistema computacional al momento en que se crea el objeto, con lo cual se permite que cada ejecución de un programa produzca una secuencia distinta de números aleatorios.

Al depurar una aplicación, algunas veces es útil repetir la misma secuencia exacta de números seudoaleatorios durante cada ejecución del programa. Esta repetitividad nos permite probar que la aplicación esté funcionando para una secuencia específica de números aleatorios, antes de evaluar el programa con distintas secuencias de números aleatorios. Cuando la repetitividad es importante, podemos crear un objeto `Random` de la siguiente manera:

```
Random numerosAleatorios = new Random( valorSemilla );
```

El argumento `valorSemilla` (de tipo `long`) siembra el cálculo del número aleatorio. Si se utiliza siempre el mismo valor para `valorSemilla`, el objeto `Random` produce la misma secuencia de números. Para establecer la semilla de un objeto `Random` en cualquier momento durante la ejecución de un programa, podemos llamar al método `set` del objeto, como en

```
numerosAleatorios.set( valorSemilla );
```

Tip para prevenir errores 6.2

Mientras un programa esté en desarrollo, cree el objeto `Random` con un valor de semilla específico para producir una secuencia repetible de números aleatorios cada vez que se ejecute el programa. Si se origina un error lógico, corrija el error y evalúe el programa otra vez con el mismo valor de semilla; esto le permitirá reconstruir la misma secuencia de números aleatorios que produjeron el error. Una vez que se hayan eliminado los errores lógicos, cree el objeto `Random` sin utilizar un valor de semilla, para que el objeto `Random` genere una nueva secuencia de números aleatorios cada vez que se ejecute el programa.

6.10 Caso de estudio: un juego de probabilidad (introducción a las enumeraciones)

Un juego de azar popular es el juego de dados conocido como "Craps", el cual se juega en casinos y callejones por todo el mundo. Las reglas del juego son simples:

> *Un jugador tira dos dados. Cada uno tiene seis caras, las cuales contienen uno, dos, tres cuatro, cinco y seis puntos negros, respectivamente. Una vez que los dados dejan de moverse, se calcula la suma de los puntos negros en las dos caras superiores. Si la suma es 7 u 11 en el primer tiro, el jugador gana. Si la suma es 2, 3 o 12 en el primer tiro (llamado "Craps"), el jugador pierde (es decir, la "casa" gana). Si la suma es 4, 5, 6, 8, 9 o 10 en el primer tiro, esta suma se convierte en el "punto" del jugador. Para ganar, el jugador debe seguir tirando los dados hasta que salga otra vez "su punto" (es decir, que tire ese mismo valor de punto). El jugador pierde si tira un 7 antes de llegar a su punto.*

La figura 6.8 simula el juego Craps, en donde se utilizan varios métodos para definir la lógica del juego. En el método `main` (líneas 21 a 65) llama al método `tirarDado` (líneas 68 a 81) según sea necesario para tirar los dos dados y calcular su suma. Los resultados de ejemplo muestran que se ganó y perdió en el primer tiro, y se ganó y perdió en un tiro subsiguiente.

```
1  // Fig. 6.8: Craps.java
2  // La clase Craps simula el juego de dados "craps".
3  import java.util.Random;
```

Fig. 6.8 | La clase `Craps` simula el juego de dados "Craps" (parte 1 de 3).

```
4
5  public class Craps
6  {
7     // crea un generador de números aleatorios para usarlo en el método tirarDado
8     private static final Random numerosAleatorios = new Random();
9
10    // enumeración con constantes que representan el estado del juego
11    private enum Estado { CONTINUA, GANO, PERDIO };
12
13    // constantes que representan tiros comunes del dado
14    private static final int DOS_UNOS = 2;
15    private static final int TRES = 3;
16    private static final int SIETE = 7;
17    private static final int ONCE = 11;
18    private static final int DOCE = 12;
19
20    // ejecuta un juego de craps
21    public static void main( String[] args )
22    {
23       int miPunto = 0; // punto si no gana o pierde en el primer tiro
24       Estado estadoJuego; // puede contener CONTINUA, GANO o PERDIO
25
26       int sumaDeDados = tirarDados(); // primer tiro de los dados
27
28       // determina el estado del juego y el punto con base en el primer tiro
29       switch ( sumaDeDados )
30       {
31          case SIETE: // gana con 7 en el primer tiro
32          case ONCE: // gana con 11 en el primer tiro
33             estadoJuego = Estado.GANO;
34             break;
35          case DOS_UNOS: // pierde con 2 en el primer tiro
36          case TRES: // pierde con 3 en el primer tiro
37          case DOCE: // pierde con 12 en el primer tiro
38             estadoJuego = Estado.PERDIO;
39             break;
40          default: // no ganó ni perdió, por lo que guarda el punto
41             estadoJuego = Estado.CONTINUA; // no ha terminado el juego
42             miPunto = sumaDeDados; // guarda el punto
43             System.out.printf( "El punto es %d\n", miPunto );
44             break; // opcional al final del switch
45       } // fin de switch
46
47       // mientras el juego no esté terminado
48       while ( estadoJuego == Estado.CONTINUA ) // no GANO ni PERDIO
49       {
50          sumaDeDados = tirarDados(); // tira los dados de nuevo
51
52          // determina el estado del juego
53          if ( sumaDeDados == miPunto ) // gana haciendo un punto
54             estadoJuego = Estado.GANO;
```

Fig. 6.8 | La clase `Craps` simula el juego de dados "Craps" (parte 2 de 3).

```
55            else
56               if ( sumaDeDados == SIETE ) // pierde al tirar 7 antes del punto
57                  estadoJuego = Estado.PERDIO;
58         } // fin de while
59
60         // muestra mensaje de que ganó o perdió
61         if ( estadoJuego == Estado.GANO )
62            System.out.println( "El jugador gana" );
63         else
64            System.out.println( "El jugador pierde" );
65      } // fin del método jugar
66
67      // tira los dados, calcula la suma y muestra los resultados
68      public static int tirarDados()
69      {
70         // elige valores aleatorios para los dados
71         int dado1 = 1 + numerosAleatorios.nextInt( 6 ); // primer tiro del dado
72         int dado2 = 1 + numerosAleatorios.nextInt( 6 ); // segundo tiro del dado
73
74         int suma = dado1 + dado2; // suma de los valores de los dados
75
76         // muestra los resultados de este tiro
77         System.out.printf( "El jugador tiro %d + %d = %d\n",
78            dado1, dado2, suma );
79
80         return suma; // devuelve la suma de los dados
81      } // fin del método tirarDados
82   } // fin de la clase Craps
```

```
El jugador tiro 5 + 6 = 11
El jugador gana
```

```
El jugador tiro 5 + 4 = 9
El punto es 9
El jugador tiro 4 + 2 = 6
El jugador tiro 3 + 6 = 9
El jugador gana
```

```
El jugador tiro 1 + 2 = 3
El jugador pierde
```

```
El jugador tiro 2 + 6 = 8
El punto es 8
El jugador tiro 5 + 1 = 6
El jugador tiro 2 + 1 = 3
El jugador tiro 1 + 6 = 7
El jugador pierde
```

Fig. 6.8 | La clase Craps simula el juego de dados "Craps" (parte 3 de 3).

El método `tirarDados`

En las reglas del juego, el jugador debe tirar dos dados en el primer tiro y hacer lo mismo en todos los tiros subsiguientes. Declaramos el método `tirarDados` (figura 6.8, líneas 68 a 81) para tirar el dado y calcular e imprimir su suma. El método `tirarDados` se declara una vez, pero se llama desde dos lugares (líneas 26 y 50) en `main`, el cual contiene la lógica para un juego completo de Craps. El método `tirarDados` no tiene argumentos, por lo cual su lista de parámetros está vacía. Cada vez que se llama, `tirarDados` devuelve la suma de los dados, por lo que se indica el tipo de valor de retorno `int` en el encabezado del método (línea 68). Aunque las líneas 71 y 72 se ven iguales (excepto por los nombres de los dados), no necesariamente producen el mismo resultado. Cada una de estas instrucciones produce un valor aleatorio en el rango de 1 a 6. La variable `numerosAleatorios` (que se utiliza en las líneas 71 y 72) no se declara en el método. En cambio, se declara como una variable `private static final` de la clase y se inicializa en la línea 8. Esto nos permite crear un objeto `Random` que se reutiliza en cada llamada a `tirarDados`. Si hubiera un programa con múltiples instancias de la clase `Craps`, todos compartirían este objeto `Random`.

Variables locales del método `main`

El juego es razonablemente complejo. El jugador puede ganar o perder en el primer tiro, o en cualquier tiro subsiguiente. El método `main` (líneas 21 a 65) utiliza a la variable local `miPunto` (línea 23) para almacenar el "punto" si el jugador no gana o pierde en el primer tiro, a la variable local `estadoJuego` (línea 24) para llevar el registro del estado del juego en general y a la variable local `sumaDeDados` (línea 26) para almacenar la suma de los dados para el tiro más reciente. La variable `miPunto` se inicializa con `0` para asegurar que la aplicación se compile. Si no inicializa `miPunto`, el compilador genera un error ya que `miPunto` no recibe un valor en *todas* las etiquetas `case` de la instrucción `switch` y, en consecuencia, el programa podría tratar de utilizar `miPunto` antes de que se le asigne un valor. En contraste, a `estadoJuego` *se le asigna* un valor en *cada* etiqueta `case` de la instrucción `switch`; por lo tanto, se garantiza que se inicialice antes de usarse.

El tipo `enum Estado`

La variable local `estadoJuego` (línea 24) se declara como de un nuevo tipo llamado `Estado` (que declaramos en la línea 11). El tipo `Estado` es un miembro `private` de la clase `Craps`, ya que sólo se utiliza en esa clase. `Estado` es un tipo conocido como **enumeración**, que en su forma más simple declara un conjunto de constantes representadas por identificadores. Una enumeración es un tipo especial de clase, que se introduce mediante la palabra clave **`enum`** y un nombre para el tipo (en este caso, `Estado`). Al igual que con una clase, las llaves delimitan el cuerpo de una declaración de `enum`. Dentro de las llaves hay una lista, separada por comas, de **constantes de enumeración**, cada una de las cuales representa un valor único. Los identificadores en una `enum` deben ser únicos. En el capítulo 8 aprenderá más acerca de las enumeraciones.

Buena práctica de programación 6.1

Es una convención utilizar sólo letras mayúsculas en los nombres de las constantes de enumeración. Esto hace que resalten y nos recuerdan que las constantes de enumeración no son variables.

A las variables de tipo `Estado` se les puede asignar sólo una de las tres constantes declaradas en la enumeración (línea 11), o se producirá un error de compilación. Cuando el jugador gana el juego, el programa asigna a la variable local `estadoJuego` el valor `Estado.GANO` (líneas 33 y 54). Cuando el jugador pierde el juego, la aplicación asigna a la variable local `estadoJuego` el valor `Estado.PERDIO` (líneas 38 y 57). En cualquier otro caso, el programa asigna a la variable local `estadoJuego` el valor `Estado.CONTINUA` (línea 41) para indicar que el juego no ha terminado y hay que tirar los dados otra vez.

Buena práctica de programación 6.2

El uso de constantes de enumeración (como `Estado.GANO`, `Estado.PERDIO` y `Estado.CONTINUA`) en vez de valores enteros literales (como 0, 1 y 2) puede hacer que los programas sean más fáciles de leer y de mantener.

Lógica del método `main`

La línea 26 en el método `main` llama a `tirarDados`, el cual elige dos valores aleatorios del 1 al 6, muestra el valor del primer dado, el valor del segundo dado y la suma de los dos dados, y devuelve esa suma. Después el método `main` entra a la instrucción `switch` (líneas 29 a 45), que utiliza el valor de `sumaDeDados` de la línea 26 para determinar si el jugador ganó o perdió el juego, o si debe continuar con otro tiro. Los valores que ocasionan que se gane o pierda el juego en el primer tiro se declaran como constantes `private static final int` en las líneas 14 a 18. Los nombres de los identificadores utilizan los términos comunes en el casino para estas sumas. Estas constantes, al igual que las constantes `enum`, se declaran todas con letras mayúsculas por convención, para que resalten en el programa. Las líneas 31 a 34 determinan si el jugador ganó en el primer tiro con `SIETE(7)` u `ONCE(11)`. Las líneas 35 a 39 determinan si el jugador perdió en el primer tiro con `DOS_UNOS(2)`, `TRES(3)` o `DOCE(12)`. Después del primer tiro, si el juego no se ha terminado, el caso `default` (líneas 40 a 44) establece `estadoJuego` en `Estado.CONTINUA`, guarda `sumaDeDados` en `miPunto` y muestra el punto.

Si aún estamos tratando de "hacer nuestro punto" (es decir, el juego continúa de un tiro anterior), se ejecutan las líneas 48 a 58. En la línea 50 se tira el dado otra vez. Si `sumaDeDados` concuerda con `miPunto` (línea 53), la línea 54 establece `estadoJuego` en `Estado.GANO` y el ciclo termina, ya que el juego está terminado. Si `sumaDeDados` es igual a `SIETE(7)` (línea 56), la línea 57 asigna el valor `Estado.PERDIO` a `estadoJuego` y el ciclo termina, ya que se acabó el juego. Cuando termina el juego, las líneas 61 a 64 muestran un mensaje en el que se indica si el jugador ganó o perdió, y el programa termina.

El programa utiliza los diversos mecanismos de control que hemos visto antes. La clase `Craps` utiliza dos métodos: `main` y `tirarDados` (que se llama dos veces desde `main`), y las instrucciones de control `switch`, `while`, `if...else` e `if` anidado. Observe también el uso de múltiples etiquetas `case` en la instrucción `switch` para ejecutar las mismas instrucciones para las sumas de `SIETE` y `ONCE` (líneas 31 y 32), y para las sumas de `DOS_UNOS`, `TRES` y `DOCE` (líneas 35 a 37).

Por qué algunas constantes no se definen como constantes `enum`

Tal vez se esté preguntando por qué declaramos las sumas de los dados como constantes `private static final int` en vez de constantes `enum`. La respuesta está en el hecho de que el programa debe comparar la variable `int` llamada `sumaDeDados` (línea 26) con estas constantes para determinar el resultado de cada tiro. Suponga que declaramos constantes que contengan `enum Suma` (por ejemplo, `Suma.DOS_UNOS`) para representar las cinco sumas utilizadas en el juego, y que después utilizamos estas constantes en la instrucción `switch` (líneas 29 a 45). Hacer esto evitaría que pudiéramos usar `sumaDeDados` como la expresión de control de la instrucción `switch`, ya que Java *no* permite que un `int` se compare con una constante de enumeración. Para lograr la misma funcionalidad que el programa actual, tendríamos que utilizar una variable `sumaActual` de tipo `Suma` como expresión de control para el `switch`. Por desgracia, Java no proporciona una manera fácil de convertir un valor `int` en una constante `enum` específica. Esto podría hacerse mediante una instrucción `switch` separada. Sin duda, esto sería complicado y no mejoraría la legibilidad del programa (lo cual echaría a perder el propósito de usar una `enum`).

6.11 Alcance de las declaraciones

Ya hemos visto declaraciones de varias entidades de Java como las clases, los métodos, las variables y los parámetros. Las declaraciones introducen nombres que pueden utilizarse para hacer referencia a dichas

entidades de Java. El **alcance** de una declaración es la porción del programa que puede hacer referencia a la entidad declarada por su nombre. Se dice que dicha entidad está "dentro del alcance" para esa porción del programa. En esta sección introduciremos varias cuestiones importantes relacionadas con el alcance.

Las reglas básicas de alcance son las siguientes:

1. El alcance de la declaración de un parámetro es el cuerpo del método en el que aparece la declaración.
2. El alcance de la declaración de una variable local es desde el punto en el cual aparece la declaración, hasta el final de ese bloque.
3. El alcance de la declaración de una variable local que aparece en la sección de inicialización del encabezado de una instrucción `for` es el cuerpo de la instrucción `for` y las demás expresiones en el encabezado.
4. El alcance de un método o campo de una clase es todo el cuerpo de la clase. Esto permite a los métodos no `static` de la clase utilizar los campos y otros métodos de ésta.

Cualquier bloque puede contener declaraciones de variables. Si una variable local o parámetro en un método tiene el mismo nombre que un campo de la clase, el campo se "oculta" hasta que el bloque termina su ejecución; a esto se le llama **ocultación de variables** (**shadowing**). En el capítulo 8 veremos cómo acceder a los campos ocultos.

Tip para prevenir errores 6.3

Use nombres distintos para los campos y las variables locales, para ayudar a evitar los errores lógicos sutiles que se producen cuando se hace la llamada a un método y una variable local de éste oculta un campo con el mismo nombre en la clase.

La figura 6.9 demuestra las cuestiones de alcance con los campos y las variables locales. La línea 7 declara e inicializa el campo `x` en `1`. Este campo se oculta en cualquier bloque (o método) que declare una variable local llamada `x`. El método `main` (líneas 11 a 23) declara una variable local `x` (línea 13) y la inicializa en `5`. El valor de esta variable local se imprime para mostrar que el campo `x` (cuyo valor es `1`) se oculta en el método `main`. El programa declara otros dos métodos: `usarVariableLocal` (líneas 26 a 35) y `usarCampo` (líneas 38 a 45); cada uno de ellos no tiene argumentos y no devuelve resultados. El método `main` llama a cada método dos veces (líneas 17 a 20). El método `usarVariableLocal` declara la variable local `x` (línea 28). Cuando se llama por primera vez a `usarVariableLocal` (línea 17), crea una variable local `x` y la inicializa en `25` (línea 28), muestra en pantalla el valor de `x` (líneas 30 y 31), incrementa `x` (línea 32) y muestra en pantalla el valor de `x` otra vez (líneas 33 y 34). Cuando se llama a `usarVariableLocal` por segunda vez (línea 19), vuelve a crear la variable local `x` y la reinicializa con `25`, por lo que la salida de cada llamada a `usarVariableLocal` es idéntica.

```
1  // Fig. 6.9: Alcance.java
2  // La clase Alcance demuestra los alcances de los campos y las variables locales.
3
4  public class Alcance
5  {
6     // campo accesible para todos los métodos de esta clase
7     private static int x = 1;
8
```

Fig. 6.9 | La clase `Alcance` demuestra los alcances de los campos y las variables locales (parte 1 de 2).

```
9      // el método main crea e inicializa la variable local x
10     // y llama a los métodos usarVariableLocal y usarCampo
11     public static void main( String[] args )
12     {
13        int x = 5; // la variable local x del método oculta al campo x
14
15        System.out.printf( "la x local en main es %d\n", x );
16
17        usarVariableLocal(); // usarVariableLocal tiene la x local
18        usarCampo(); // usarCampo usa el campo x de la clase Alcance
19        usarVariableLocal(); // usarVariableLocal reinicia a la x local
20        usarCampo(); // el campo x de la clase Alcance retiene su valor
21
22        System.out.printf( "\nla x local en main es %d\n", x );
23     } // fin de main
24
25     // crea e inicializa la variable local x durante cada llamada
26     public static void usarVariableLocal()
27     {
28        int x = 25; // se inicializa cada vez que se llama a usarVariableLocal
29
30        System.out.printf(
31           "\nla x local al entrar al metodo usarVariableLocal es %d\n", x );
32        ++x; // modifica la variable x local de este método
33        System.out.printf(
34           "la x local antes de salir del metodo usarVariableLocal es %d\n", x );
35     } // fin del método usarVariableLocal
36
37     // modifica el campo x de la clase Alcance durante cada llamada
38     public static void usarCampo()
39     {
40        System.out.printf(
41           "\nel campo x al entrar al metodo usarCampo es %d\n", x );
42        x *= 10; // modifica el campo x de la clase Alcance
43        System.out.printf(
44           "el campo x antes de salir del metodo usarCampo es %d\n", x );
45     } // fin del método usarCampo
46  } // fin de la clase Alcance
```

```
la x local en el metodo iniciar es 5

la x local al entrar al metodo usarVariableLocal es 25
la x local antes de salir del metodo usarVariableLocal es 26

el campo x al entrar al metodo usarCampo es 1
el campo x antes de salir del metodo usarCampo es 10

la x local al entrar al metodo usarVariableLocal es 25
la x local antes de salir del metodo usarVariableLocal es 26

el campo x al entrar al metodo usarCampo es 10
el campo x antes de salir del metodo usarCampo es 100

la x local en el metodo iniciar es 5
```

Fig. 6.9 | La clase `Alcance` demuestra los alcances de los campos y las variables locales (parte 2 de 2).

El método `usarCampo` no declara variables locales. Por lo tanto, cuando hace referencia a `x`, se utiliza el campo `x` (línea 7) de la clase. Cuando el método `usarCampo` se llama por primera vez (línea 18), muestra en pantalla el valor (1) del campo `x` (líneas 40 y 41), multiplica el campo `x` por 10 (línea 42) y muestra en pantalla el valor (10) del campo `x` otra vez (líneas 43 y 44) antes de regresar. La siguiente vez que se llama al método `usarCampo` (línea 20), el campo `x` tiene el valor modificado de 10, por lo que el método muestra en pantalla un 10 y después un 100. Por último, en el método `main`, el programa muestra en pantalla el valor de la variable local `x` otra vez (línea 22), para mostrar que ninguna de las llamadas a los métodos modificó la variable local `x` de iniciar, ya que todos los métodos hicieron referencia a las variables llamadas `x` en otros alcances.

6.12 Sobrecarga de métodos

Pueden declararse métodos con el mismo nombre en la misma clase, siempre y cuando tengan distintos conjuntos de parámetros (que se determinan con base en el número, tipos y orden de los parámetros). A esto se le conoce como **sobrecarga de métodos**. Cuando se hace una llamada a un método sobrecargado, el compilador de Java selecciona el método apropiado mediante un análisis del número, tipos y orden de los argumentos en la llamada. Por lo general, la sobrecarga de métodos se utiliza para crear varios métodos con el *mismo* nombre que realicen la *misma* tarea o tareas similares, pero con distintos tipos o números de argumentos. Por ejemplo, los métodos `abs`, `min` y `max` de `Math` (sintetizados en la sección 6.3) se sobrecargan con cuatro versiones cada uno:

1. Uno con dos parámetros `double`.
2. Uno con dos parámetros `float`.
3. Uno con dos parámetros `int`.
4. Uno con dos parámetros `long`.

Nuestro siguiente ejemplo demuestra cómo declarar e invocar métodos sobrecargados. En el capítulo 8 demostraremos los constructores sobrecargados.

Declaración de métodos sobrecargados

La clase `SobrecargaMetodos` (figura 6.10) incluye dos versiones sobrecargadas del método `cuadrado`: una que calcula el cuadrado de un `int` (y devuelve un `int`) y otra que calcula el cuadrado de un `double` (y devuelve un `double`). Aunque estos métodos tienen el mismo nombre, además de listas de parámetros y cuerpos similares, podemos considerarlos simplemente como métodos diferentes. Puede ser útil si consideramos los nombres de los métodos como "`cuadrado de int`" y "`cuadrado de double`", respectivamente.

```
1  // Fig. 6.10: SobrecargaMetodos.java
2  // Declaraciones de métodos sobrecargados.
3
4  public class SobrecargaMetodos
5  {
6     // prueba los métodos cuadrado sobrecargados
7     public static void main( String[] args )
8     {
9        System.out.printf( "El cuadrado del entero 7 es %d\n", cuadrado( 7 ) );
10       System.out.printf( "El cuadrado del double 7.5 es %f\n", cuadrado( 7.5 ) );
11    } // fin de main
12
```

Fig. 6.10 | Declaraciones de métodos sobrecargados (parte 1 de 2).

```
13    // método cuadrado con argumento int
14    public static int cuadrado( int valorInt )
15    {
16       System.out.printf( "\nSe llamo a cuadrado con argumento int: %d\n",
17          valorInt );
18       return valorInt * valorInt;
19    } // fin del método cuadrado con argumento int
20
21    // método cuadrado con argumento double
22    public static double cuadrado( double valorDouble )
23    {
24       System.out.printf( "\nSe llamo a cuadrado con argumento double: %f\n",
25          valorDouble );
26       return valorDouble * valorDouble;
27    } // fin del método cuadrado con argumento double
28 } // fin de la clase SobrecargaMetodos
```

```
Se llamo a cuadrado con argumento int: 7
El cuadrado del entero 7 es 49

Se llamo a cuadrado con argumento double: 7.500000
El cuadrado del double 7.5 es 56.250000
```

Fig. 6.10 | Declaraciones de métodos sobrecargados (parte 2 de 2).

La línea 9 invoca al método `cuadrado` con el argumento `7`. Los valores enteros literales se tratan como de tipo `int`, por lo que la llamada al método en la línea 9 invoca a la versión de `cuadrado` de las líneas 14 a 19, la cual especifica un parámetro `int`. De manera similar, la línea 10 invoca al método `cuadrado` con el argumento `7.5`. Los valores de las literales de punto flotante se tratan como de tipo `double`, por lo que la llamada al método en la línea 10 invoca a la versión de `cuadrado` de las líneas 22 a 27, la cual especifica un parámetro `double`. Cada método imprime en pantalla primero una línea de texto, para mostrar que se llamó al método apropiado en cada caso. Los valores en las líneas 10 y 24 se muestran con el especificador de formato `%f`. No especificamos una precisión en ninguno de los dos casos. De manera predeterminada, los valores de punto flotante se muestran con seis dígitos de precisión, si ésta no se especifica en el especificador de formato.

Cómo se diferencian los métodos sobrecargados entre sí

El compilador diferencia los métodos sobrecargados con base en su **firma**: una combinación del nombre del método y del número, tipos y orden de sus parámetros. Si el compilador sólo se fijara en los nombres de los métodos durante la compilación, el código de la figura 6.10 sería ambiguo; el compilador no sabría cómo distinguir entre los dos métodos `cuadrado` (líneas 14 a 19 y 22 a 27). De manera interna, el compilador utiliza nombres de métodos más largos que incluyen el nombre del método original, el tipo de cada parámetro y el orden exacto de ellos para determinar si los métodos en una clase son únicos en esa clase.

Por ejemplo, en la figura 6.10 el compilador podría utilizar el nombre lógico "`cuadrado de int`" para el método `cuadrado` que especifica un parámetro `int`, y el método "`cuadrado de double`" para el método `cuadrado` que determina un parámetro `double` (los nombres reales que utiliza el compilador son más complicados). Si la declaración de `metodo1` empieza así:

```
void metodo1( int a, float b )
```

entonces el compilador podría usar el nombre lógico "`metodo1 de int y float`". Si los parámetros se especificaran así

```
void metodo1( float a, int b )
```

entonces el compilador podría usar el nombre lógico "metodo1 de float e int". El orden de los tipos de los parámetros es importante; el compilador considera que los dos encabezados anteriores de metodo1 son distintos.

Tipos de valores de retorno de los métodos sobrecargados

Al hablar sobre los nombres lógicos de los métodos que utiliza el compilador, no mencionamos los tipos de valores de retorno de los métodos. *Las llamadas a los métodos no pueden diferenciarse con base en el tipo de valor de retorno.* Si usted tuviera métodos sobrecargados cuya única diferencia estuviera en los tipos de valor de retorno y llamara a uno de los métodos en una instrucción individual, como:

```
cuadrado( 2 );
```

el compilador *no* podría determinar la versión del método a llamar, ya que se ignora el valor de retorno. Cuando dos métodos tienen la misma firma pero distintos tipos de valores de retorno, genera un mensaje de error para indicar que el método ya está definido en la clase. Los métodos sobrecargados *pueden* tener tipos de valor de retorno distintos si tienen distintas listas de parámetros. Además, los métodos sobrecargados *no* necesitan tener el mismo número de parámetros.

Error común de programación 6.9

Declarar métodos sobrecargados con listas de parámetros idénticas es un error de compilación, sin importar que los tipos de los valores de retorno sean distintos.

6.13 (Opcional) Caso de estudio de GUI y gráficos: colores y figuras rellenas

Aunque podemos crear muchos diseños interesantes sólo con líneas y figuras básicas, la clase Graphics proporciona muchas herramientas más. Las siguientes dos herramientas que presentaremos son los colores y las figuras rellenas. Al agregar color se enriquecen los dibujos que ve un usuario en la pantalla de la computadora. Las figuras se pueden rellenar con colores sólidos.

Los colores que se muestran en las pantallas de las computadoras se definen con base en sus componentes rojo, verde y azul (conocidos como **valores RGB**), los cuales tienen valores enteros de 0 a 255. Cuanto más alto sea el valor de un componente específico, más intensidad de color tendrá esa figura. Java utiliza la clase **Color** en el paquete java.awt para representar colores mediante sus valores RGB. Por conveniencia, la clase Color (paquete java.awt) contiene 13 objetos static Color predefinidos: BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE y YELLOW. Para acceder a éstos objetos predefinidos, se utiliza el nombre de la clase y un punto (.), como en Color.RED. La clase Color también contiene un constructor de la forma:

```
public Color( int r, int g, int b )
```

de manera que podemos crear colores personalizados con sólo especificar los valores de los componentes rojo, verde y azul.

Los métodos **fillRect** y **fillOval** de Graphics dibujan rectángulos y óvalos rellenos, cada uno. Estos dos métodos tienen los mismos parámetros que drawRect y drawOval; los primeros dos parámetros son las coordenadas para la esquina superior izquierda de la figura, mientras que los otros dos parámetros determinan su anchura y su altura. El ejemplo de las figuras 6.11 y 6.12 demuestra los colores y las figuras rellenas, al dibujar y mostrar una cara sonriente amarilla en la pantalla.

```
1  // Fig. 6.11: DibujarCaraSonriente.java
2  // Demuestra las figuras rellenas.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class DibujarCaraSonriente extends JPanel
8  {
9     public void paintComponent( Graphics g )
10    {
11       super.paintComponent( g );
12
13       // dibuja la cara
14       g.setColor( Color.YELLOW );
15       g.fillOval( 10, 10, 200, 200 );
16
17       // dibuja los ojos
18       g.setColor( Color.BLACK );
19       g.fillOval( 55, 65, 30, 30 );
20       g.fillOval( 135, 65, 30, 30 );
21
22       // dibuja la boca
23       g.fillOval( 50, 110, 120, 60 );
24
25       // convierte la boca en una sonrisa
26       g.setColor( Color.YELLOW );
27       g.fillRect( 50, 110, 120, 30 );
28       g.fillOval( 50, 120, 120, 40 );
29    } // fin del método paintComponent
30 } // fin de la clase DibujarCaraSonriente
```

Fig. 6.11 | Dibujar cara sonriente usando colores y formas de relleno.

Las instrucciones `import` en las líneas 3 a 5 de la figura 6.11 importan las clases `Color`, `Graphics` y `JPanel`. La clase `DibujarCaraSonriente` (líneas 7 a 30) utiliza la clase `Color` para especificar los colores, y utiliza la clase `Graphics` para dibujar.

La clase `JPanel` proporciona de nuevo el área en la que vamos a dibujar. La línea 14 en el método `paintComponent` utiliza el método `setColor` de `Graphics` para establecer el color actual para dibujar en `Color.YELLOW`. El método `setColor` requiere un argumento, el `Color` a establecer como el color para dibujar. En este caso, utilizamos el objeto predefinido `Color.YELLOW`.

La línea 15 dibuja un círculo con un diámetro de 200 para representar la cara; cuando los argumentos anchura y altura son idénticos, el método `fillOval` dibuja un círculo. A continuación, la línea 18 establece el color en `Color.BLACK`, y las líneas 19 y 20 dibujan los ojos. La línea 23 dibuja la boca como un óvalo, pero esto no es exactamente lo que queremos.

Para crear una cara feliz, vamos a "retocar" la boca. La línea 26 establece el color en `Color.YELLOW`, de manera que cualquier figura que dibujemos se mezcle con la cara. La línea 27 dibuja un rectángulo con la mitad de altura que la boca. Esto "borra" la mitad superior de la boca, dejando sólo la mitad inferior. Para crear una mejor sonrisa, la línea 28 dibuja otro óvalo para cubrir ligeramente la porción superior de la boca. La clase `PruebaDibujarCaraSonriente` (figura 6.12) crea y muestra un objeto `JFrame` que contiene el dibujo. Cuando se muestra el objeto `JFrame`, el sistema llama al método `paintComponent` para dibujar la cara sonriente.

```
1  // Fig. 6.12: PruebaDibujarCaraSonriente.java
2  // Aplicación de prueba que muestra una cara sonriente.
3  import javax.swing.JFrame;
4
5  public class PruebaDibujarCaraSonriente
6  {
7     public static void main( String[] args )
8     {
9        DibujarCaraSonriente panel = new DibujarCaraSonriente();
10       JFrame aplicacion = new JFrame();
11
12       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13       aplicacion.add( panel );
14       aplicacion.setSize( 230, 250 );
15       aplicacion.setVisible( true );
16    } // fin de main
17 } // fin de la clase PruebaDibujarCaraSonriente
```

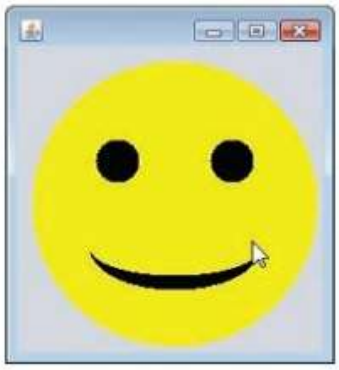


Fig. 6.12 | Creación de un objeto `JFrame` para mostrar una cara sonriente.

Ejercicios del caso de estudio de GUI y gráficos

6.1 Con el método `fillOval`, dibuje un tiro al blanco que alterne entre dos colores aleatorios, como en la figura 6.13. Use el constructor `Color( int r, int g, int b )` con argumentos aleatorios para generar colores aleatorios.

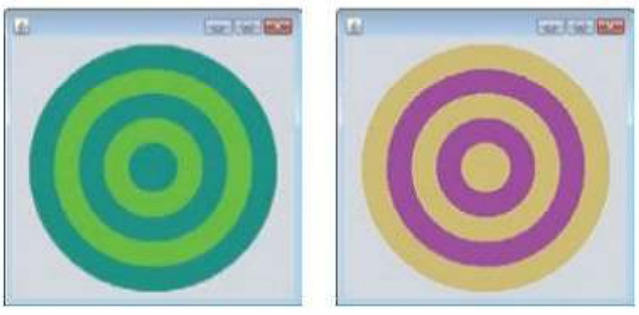


Fig. 6.13 | Un tiro al blanco con dos colores alternantes al azar.

6.2 Cree un programa para dibujar 10 figuras rellenas al azar en colores, posiciones y tamaños aleatorios (figura 6.14). El método `paintComponent` debe contener un ciclo que itere 10 veces. En cada iteración, el ciclo debe determinar si se dibujará un rectángulo o un óvalo relleno, crear un color aleatorio y elegir tanto las coordenadas como las medidas al azar. Las coordenadas deben elegirse con base en la anchura y la altura del panel. Las longitudes de los lados deben limitarse a la mitad de la anchura o altura de la ventana.

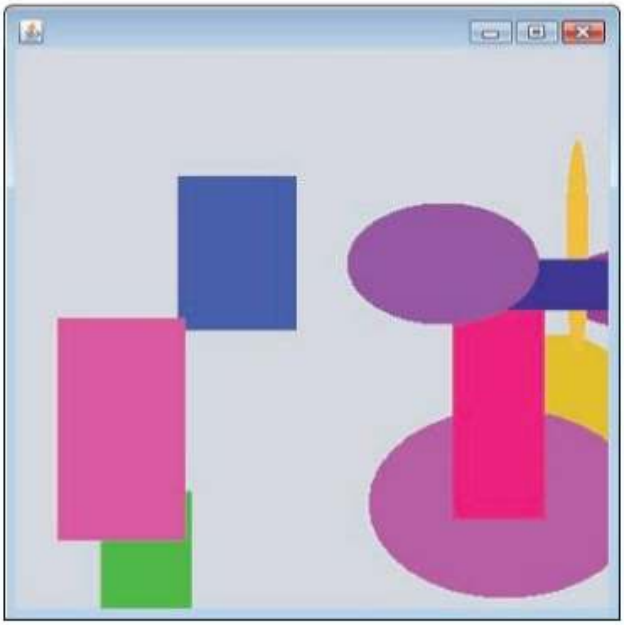


Fig. 6.14 | Figuras generadas al azar.

6.14 Conclusión

En este capítulo aprendió más acerca de las declaraciones de métodos. También conoció la diferencia entre los métodos `static` y los no `static`, y le mostramos cómo llamar a los métodos `static`, anteponiendo al nombre del método el nombre de la clase en la cual aparece, y el separador punto (`.`). Aprendió a utilizar los operadores + y += para realizar concatenaciones de cadenas. Vimos cómo la pila de llamada a los métodos y los registros de activación llevan la cuenta de los métodos que se han llamado y a dónde debe regresar cada método cuando completa su tarea. También hablamos sobre las reglas de promoción de Java para realizar conversiones implícitas entre tipos primitivos, y cómo realizar conversiones explícitas con operadores de conversión de tipos. Después aprendió acerca de algunos de los paquetes más utilizados en la API de Java.

Vio cómo declarar constantes con nombre, mediante los tipos `enum` y las variables `private static final`. Utilizó la clase `Random` para generar números aleatorios, que pueden usarse para simulaciones. También aprendió acerca del alcance de los campos y las variables locales en una clase. Por último, aprendió que varios métodos en una clase pueden sobrecargarse, al proporcionar métodos con el mismo nombre y distintas firmas. Dichos métodos pueden usarse para realizar las mismas tareas, o similares, mediante distintos tipos o números de parámetros.

En el capítulo 7 aprenderá a mantener listas y tablas de datos en arreglos. Verá una implementación más elegante de la aplicación que tira un dado 6,000,000 veces, y dos versiones mejoradas de nuestro caso de estudio `LibroCalificaciones` que estudió en los capítulos 3 a 5. También aprenderá cómo acceder a los argumentos de línea de comandos de una aplicación, los cuales se pasan al método `main` cuando una aplicación comienza su ejecución.

Resumen

Sección 6.1 Introducción

- La experiencia ha demostrado que la mejor forma de desarrollar y mantener un programa extenso es construirlo a partir de piezas pequeñas y simples, o módulos. A esta técnica se le conoce como "divide y vencerás" (pág. 198).

Sección 6.2 Módulos de programas en Java

- Los métodos se declaran dentro de las clases. Las cuales por lo general se agrupan en paquetes para que puedan importarse en los programas y reutilizarse.
- Los métodos nos permiten dividir un programa en módulos, al separar sus tareas en unidades autocontenidas. Las instrucciones en un método se escriben sólo una vez, y se ocultan de los demás métodos.
- Utilizar los métodos existentes como bloques de construcción para crear nuevos programas es una forma de reutilización del software (pág. 199), que nos permite evitar repetir código dentro de un programa.

Sección 6.3 Métodos `static`, campos `static` y la clase `Math`

- Una llamada a un método especifica el nombre del método a llamar y proporciona los argumentos que el método al que se llamó requiere para realizar su tarea. Cuando termina la llamada al método, éste devuelve un resultado o simplemente devuelve el control al método que lo llamó.
- Una clase puede contener métodos `static` para realizar tareas comunes que no requieren un objeto de la clase. Cualquier información que pueda requerir un método `static` para realizar sus tareas se le puede enviar en forma de argumentos, en una llamada al método. Para llamar a un método `static`, se especifica el nombre de la clase en la cual está declarado el método, seguido de un punto (.) y del nombre del método, como en

  NombreClase.nombreMétodo(argumentos)

- La clase `Math` cuenta con métodos `static` para realizar cálculos matemáticos comunes.
- La constante `Math.PI` (pág. 201; 3.141592653589793) es la relación entre la circunferencia de un círculo y su diámetro. La constante `Math.E` (pág. 201; 2.718281828459045) es el valor de la base para los logaritmos naturales (que se calculan con el método `static log` de `Math`).
- `Math.PI` y `Math.E` se declaran con los modificadores `public`, `final` y `static`. Al hacerlos `public`, puede usar estos campos en sus propias clases. Cualquier campo declarado con la palabra clave `final` (pág. 201) es constante; su valor no se puede modificar una vez que se inicializa el campo. Tanto `PI` como `E` se declaran final, ya que sus valores nunca cambian. Al hacer a estos campos `static`, se puede acceder a ellos a través del nombre de la clase `Math` y un separador punto (.), justo igual que con los métodos de la clase `Math`.
- Todos los objetos de una clase comparten una copia de los campos `static` de ésta. En conjunto, las variables de clase (pág. 201) y de instancia de la clase representan los campos de la clase.
- Al ejecutar la máquina virtual de Java (JVM) con el comando `java`, la JVM carga la clase que usted especifica y utiliza el nombre de esa clase para invocar al método `main`. Puede especificar argumentos de línea de comandos adicionales (pág. 202), que la JVM pasará a su aplicación.
- Puede colocar un método `main` en cualquier clase que declare; el comando java sólo llamará al método `main` en la clase que usted utilice para ejecutar la aplicación.

Sección 6.4 Declaración de métodos con múltiples parámetros

- Cuando se hace una llamada a un método, el programa crea una copia de los valores de los argumentos del método y los asigna a los parámetros correspondientes del mismo. Cuando el control del programa regresa al punto en el que se hizo la llamada al método, los parámetros del mismo se eliminan de la memoria.
- Un método puede devolver a lo más un valor, pero el valor devuelto podría ser una referencia a un objeto que contenga muchos valores.
- Las variables deben declararse como campos de una clase, sólo si se requieren para usarlos en más de un método, o si el programa debe guardar sus valores entre distintas llamadas a los métodos de la clase.

- Cuando un método tiene más de un parámetro, se especifican como una lista separada por comas. Debe haber un argumento en la llamada al método para cada parámetro en su declaración. Además, cada argumento debe ser consistente con el tipo del parámetro correspondiente. Si un método no acepta argumentos, la lista de parámetros está vacía.
- Los objetos `String` se pueden concatenar (pág. 204) mediante el uso del operador +, que coloca los caracteres del operando derecho al final de los que están en el operando izquierdo.
- Cada valor primitivo y objeto en Java tiene una representación `String`. Cuando se concatena un objeto con un `String`, el objeto se convierte en un `String` y después, los dos `String` se concatenan.
- Si un valor `boolean` se concatena con un objeto `String`, se utiliza la palabra `"true"` o la palabra `"false"` para representar el valor `boolean`.
- Todos los objetos en Java tienen un método especial, llamado `toString`, el cual devuelve una representación `String` del contenido del objeto. Cuando se concatena un objeto con un `String`, la JVM llama de manera implícita al método `toString` del objeto, para obtener la representación `String` del mismo.
- Es posible dividir las literales `String` extensas en varias literales `String` más pequeñas, colocarlas en varias líneas de código para mejorar la legibilidad, y después vuelven a ensamblar las literales `String` mediante la concatenación.

Sección 6.5 Notas acerca de cómo declarar y utilizar los métodos

- Hay tres formas de llamar a un método: usar el nombre de un método por sí solo para llamar a otro de la misma clase; usar una variable que contenga una referencia a un objeto, seguida de un punto (.) y del nombre del método, para llamar a un método del objeto al que se hace referencia; y usar el nombre de la clase y un punto (.) para llamar a un método `static` de una clase.
- Hay tres formas de devolver el control a una instrucción que llama a un método. Si el método no devuelve un resultado, el control regresa cuando el flujo del programa llega a la llave derecha de terminación del método, o cuando se ejecuta la instrucción

  ```
  return;
  ```

  si el método devuelve un resultado, la instrucción

  ```
  return expression;
  ```

  evalúa la expresión, y después regresa de inmediato el valor resultante al método que hizo la llamada.

Sección 6.6 La pila de llamadas a los métodos y los registros de activación

- Las pilas (pág. 206) se conocen como estructuras de datos tipo "último en entrar, primero en salir (UEPS)"; el último elemento que se mete (inserta) en la pila es el primer elemento que se saca (extrae) de ella.
- Un método al que se llama debe saber cómo regresar al que lo llamó, por lo que la dirección de retorno del método que hace la llamada se mete en la pila de ejecución del programa cuando se llama al método. Si ocurre una serie de llamadas, las direcciones de retorno sucesivas se meten en la pila, en el orden último en entrar, primero en salir, de manera que el último método en ejecutarse sea el primero en regresar al método que lo llamó.
- La pila de ejecución del programa (pág. 207) contiene la memoria para las variables locales que se utilizan en cada invocación de un método, durante la ejecución de un programa. Este dato se conoce como el registro de activación, o marco de pila, de la llamada al método. Cuando se hace una llamada a un método, el registro de activación para ella se mete en la pila de ejecución del programa. Cuando el método regresa al que lo llamó, el registro de activación para esta llamada al método se saca de la pila, y las variables locales ya no son conocidas para el programa.
- Si hay más llamadas a métodos de las que se puedan almacenar en sus registros de activación en la pila de ejecución del programa, se produce un error conocido como desbordamiento de pila (pág. 207). La aplicación se compilará correctamente, pero su ejecución producirá un desbordamiento de pila.

Sección 6.7 Promoción y conversión de argumentos

- La promoción de argumentos (pág. 207) convierte el valor de un argumento al tipo que el método espera recibir en su parámetro correspondiente.

- Las reglas de promoción (pág. 207) se aplican a las expresiones que contienen valores de dos o más tipos primitivos, y a los valores de tipos primitivos que se pasan como argumentos para los métodos. Cada valor se promueve al tipo "más alto" en la expresión. En casos en los que se puede perder información debido a la conversión, el compilador de Java requiere que utilicemos un operador de conversión de tipos para obligar a que ocurra la conversión en forma explícita.

Sección 6.9 Caso de estudio: generación de números aleatorios

- Los objetos de la clase `Random` (pág. 210; paquete `java.util`) pueden producir valores aleatorios. El método `random` de `Math` puede producir valores `double` en el rango `0.0` $\leq x <$ `1.0`, en donde x es el valor devuelto.
- El método `nextInt` de `Random` (pág. 210) genera un valor `int` aleatorio en el rango de –2,147,483,648 a +2,147,483,647. Los valores devueltos por `nextInt` son en realidad números seudoaleatorios (pág. 210): una secuencia de valores producidos por un cálculo matemático complejo. Ese cálculo utiliza la hora actual del día para sembrar (pág. 211) el generador de números aleatorios, de tal forma que cada ejecución del programa produzca una secuencia diferente de valores aleatorios.
- La clase `Random` cuenta con otra versión del método `nextInt`, la cual recibe un argumento `int` y devuelve un valor desde 0 hasta el valor del argumento (pero sin incluirlo).
- Los números aleatorios en un rango (pág. 211) pueden generarse mediante

  `numero` = *valorDesplazamiento* + `numerosAleatorios.nextInt(`*factorEscala*`);`

  en donde *valorDesplazamiento* especifica el primer número en el rango deseado de enteros consecutivos, y *factorEscala* especifica cuántos números hay en el rango.
- Los números aleatorios pueden elegirse a partir de rangos de enteros no consecutivos, como en

  `numero` = *valorDesplazamiento* +
  diferenciaEntreValores `*` `numerosAleatorios.nextInt(`*factorEscala*`);`

  en donde *valorDesplazamiento* especifica el primer número en el rango de valores, *diferenciaEntreValores* representa la diferencia entre números consecutivos en la secuencia y *factorEscala* especifica cuántos números hay en el rango.
- Para depurar, algunas veces es conveniente repetir la misma secuencia de números seudoaleatorios durante cada ejecución del programa. Para ello, hay que pasar un valor entero `long` al constructor del objeto `Random`. Si se utiliza la misma semilla cada vez que se ejecuta el programa, se produce la misma secuencia de números aleatorios.

Sección 6.10 Caso de estudio: un juego de probabilidad (introducción a las enumeraciones)

- Una enumeración (pág. 218) se introduce mediante la palabra clave `enum` y el nombre de un tipo. Al igual que con cualquier clase, las llaves (`{` y `}`) delimitan el cuerpo de una declaración `enum`. Dentro de las llaves hay una lista separada por comas de constantes de enumeración, cada una de las cuales representa un valor único. Los identificadores en una `enum` deben ser únicos. A las variables de tipo `enum` sólo se les pueden asignar constantes de ese tipo `enum`.
- Las constantes también pueden declararse como variables `private static final int`. Y se declaran todas con letras mayúsculas por convención, para hacer que resalten en el programa.

Sección 6.11 Alcance de las declaraciones

- El alcance (pág. 220) es la porción del programa en la que se puede hacer referencia a una entidad, como una variable o un método, por su nombre. Se dice que dicha entidad está "dentro del alcance" para esa porción del programa.
- El alcance de la declaración de un parámetro es el cuerpo del método en el que aparece esa declaración.
- El alcance de la declaración de una variable local es desde el punto en el que aparece la declaración, hasta el final de ese bloque.
- El alcance de la declaración de una variable local que aparece en la sección de inicialización del encabezado de una instrucción `for` es el cuerpo de la instrucción `for`, junto con las demás expresiones en el encabezado.
- El alcance de un método o campo de una clase es todo el cuerpo de la clase. Esto permite que los métodos de una clase utilicen nombres simples para llamar a los demás métodos de la clase y acceder a los campos de la misma.

- Cualquier bloque puede contener declaraciones de variables. Si una variable local o parámetro en un método tiene el mismo nombre que un campo, éste se oculta (pág. 220) hasta que el bloque termina de ejecutarse.

Sección 6.12 Sobrecarga de métodos

- Java permite métodos sobrecargados (pág. 222) en una clase, siempre y cuando tengan distintos conjuntos de parámetros (lo cual se determina con base en el número, orden y tipos de los parámetros).
- Los métodos sobrecargados se distinguen por sus firmas (pág. 223): combinaciones de los nombres de los métodos y el número, tipos y orden de sus parámetros.

Ejercicios de autoevaluación

6.1 Complete las siguientes oraciones:

a) Un método se invoca con un _____ .
b) A una variable que se conoce sólo dentro del método en el que está declarada, se le llama _____ .
c) La instrucción _____ en un método llamado puede usarse para regresar el valor de una expresión, al método que hizo la llamada.
d) La palabra clave _____ indica que un método no devuelve ningún valor.
e) Los datos pueden agregarse o eliminarse sólo desde _____ de una pila.
f) Las pilas se conocen como estructuras de datos _____ : el último elemento que se mete (inserta) en la pila es el primer elemento que se saca (extrae) de ella.
g) Las tres formas de regresar el control de un método llamado a un solicitante son _____ , _____ y _____ .
h) Un objeto de la clase _____ produce números aleatorios.
i) La pila de ejecución del programa contiene la memoria para las variables locales en cada invocación de un método, durante la ejecución de un programa. Estos datos, almacenados como una parte de la pila de ejecución del programa, se conocen como _____ o _____ de la llamada al método.
j) Si hay más llamadas a métodos de las que puedan almacenarse en la pila de ejecución del programa, se produce un error conocido como _____ .
k) El _____ de una declaración es la porción del programa que puede hacer referencia a la entidad en la declaración, por su nombre.
l) Es posible tener varios métodos con el mismo nombre, en donde cada uno opere con distintos tipos o números de argumentos. A esta característica se le llama _____ de métodos.
m) La pila de ejecución del programa también se conoce como la pila de _____ .

6.2 Para la clase `Craps` de la figura 6.8, indique el alcance de cada una de las siguientes entidades:

a) la variable `numerosAleatorios`.
b) la variable `dado1`.
c) el método `tirarDado`.
d) el método `main`.
e) la variable `sumaDeDados`.

6.3 Escriba una aplicación que pruebe si los ejemplos de las llamadas a los métodos de la clase `Math` que se muestran en la figura 6.2 realmente producen los resultados indicados.

6.4 Cuál es el encabezado para cada uno de los siguientes métodos:

a) El método `hipotenusa`, que toma dos argumentos de punto flotante con doble precisión, llamados `lado1` y `lado2`, y que devuelve un resultado de punto flotante, con doble precisión.
b) El método menor, que toma tres enteros `x`, `y` y `z`, y devuelve un entero.
c) El método instrucciones, que no toma argumentos y no devuelve ningún valor. (*Nota:* estos métodos se utilizan comúnmente para mostrar instrucciones a un usuario).
d) El método `intAFloat`, que toma un argumento entero llamado `numero` y devuelve un resultado de punto flotante.

6.5 Encuentre el error en cada uno de los siguientes segmentos de programas. Explique cómo se puede corregir.

a)
```
void g()
{
   System.out.println( "Dentro del método g" );

   void h()
   {
      System.out.println( "Dentro del método h" );
   }
}
```

b)
```
int suma( int x, int y )
{
   int resultado;
   resultado = x + y;
}
```

c)
```
void f( float a );
{
   float a;
   System.out.println( a );
}
```

d)
```
void producto()
{
   int a = 6, b = 5, c = 4, resultado;
   resultado = a * b * c;
   System.out.printf( "El resultado es %d\n", resultado );
return resultado;
}
```

6.6 Escriba una aplicación completa en Java que pida al usuario el radio de tipo `double` de una esfera, y que llame al método `volumenEsfera` para calcular y mostrar el volumen de esa esfera. Utilice la siguiente asignación para calcular el volumen:

```
double volumen = ( 4.0 / 3.0 ) * Math.PI * Math.pow( radio, 3 )
```

Respuestas a los ejercicios de autoevaluación

6.1 a) llamada a un método. b) variable local. c) `return`. d) `void`. e) cima. f) último en entrar, primero en salir (UEPS). g) `return;` o `return` *expresión*; o encontrar la llave derecha de cierre de un método. h) `Random`. i) registro de activación, marco de pila. j) desbordamiento de pila. k) alcance. l) sobrecarga de métodos. m) llamadas a métodos.

6.2 a) el cuerpo de la clase. b) el bloque que define el cuerpo del método `tirarDado`. c) el cuerpo de la clase. d) el cuerpo de la clase. e) el bloque que define el cuerpo del método `main`.

6.3 La siguiente solución demuestra el uso de los métodos de la clase `Math` de la figura 6.2:

```
1  // Ejercicio 6.3: PruebaMath.java
2  // Prueba de los métodos de la clase Math.
3
4  public class PruebaMath
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf( "Math.abs( 23.7 ) = %f\n", Math.abs( 23.7 ) );
9        System.out.printf( "Math.abs( 0.0 ) = %f\n", Math.abs( 0.0 ) );
10       System.out.printf( "Math.abs( -23.7 ) = %f\n", Math.abs( -23.7 ) );
11       System.out.printf( "Math.ceil( 9.2 ) = %f\n", Math.ceil( 9.2 ) );
```

```
12        System.out.printf( "Math.ceil( -9.8 ) = %f\n", Math.ceil( -9.8 ) );
13        System.out.printf( "Math.cos( 0.0 ) = %f\n", Math.cos( 0.0 ) );
14        System.out.printf( "Math.exp( 1.0 ) = %f\n", Math.exp( 1.0 ) );
15        System.out.printf( "Math.exp( 2.0 ) = %f\n", Math.exp( 2.0 ) );
16        System.out.printf( "Math.floor( 9.2 ) = %f\n", Math.floor( 9.2 ) );
17        System.out.printf( "Math.floor( -9.8 ) = %f\n",
18          Math.floor( -9.8 ) );
19        System.out.printf( "Math.log( Math.E ) = %f\n",
20          Math.log( Math.E ) );
21        System.out.printf( "Math.log( Math.E * Math.E ) = %f\n",
22          Math.log( Math.E * Math.E ) );
23        System.out.printf( "Math.max( 2.3, 12.7 ) = %f\n",
24          Math.max( 2.3, 12.7 ) );
25        System.out.printf( "Math.max( -2.3, -12.7 ) = %f\n",
26          Math.max( -2.3, -12.7 ) );
27        System.out.printf( "Math.min( 2.3, 12.7 ) = %f\n",
28          Math.min( 2.3, 12.7 ) );
29        System.out.printf( "Math.min( -2.3, -12.7 ) = %f\n",
30          Math.min( -2.3, -12.7 ) );
31        System.out.printf( "Math.pow( 2.0, 7.0 ) = %f\n",
32          Math.pow( 2.0, 7.0 ) );
33        System.out.printf( "Math.pow( 9.0, 0.5 ) = %f\n",
34          Math.pow( 9.0, 0.5 ) );
35        System.out.printf( "Math.sin( 0.0 ) = %f\n", Math.sin( 0.0 ) );
36        System.out.printf( "Math.sqrt( 900.0 ) = %f\n",
37          Math.sqrt( 900.0 ) );
38        System.out.printf( "Math.tan( 0.0 ) = %f\n", Math.tan( 0.0 ) );
39     } // fin de main
40  } // fin de la clase PruebaMath
```

```
Math.abs( 23.7 ) = 23.700000
Math.abs( 0.0 ) = 0.000000
Math.abs( -23.7 ) = 23.700000
Math.ceil( 9.2 ) = 10.000000
Math.ceil( -9.8 ) = -9.000000
Math.cos( 0.0 ) = 1.000000
Math.exp( 1.0 ) = 2.718282
Math.exp( 2.0 ) = 7.389056
Math.floor( 9.2 ) = 9.000000
Math.floor( -9.8 ) = -10.000000
Math.log( Math.E ) = 1.000000
Math.log( Math.E * Math.E ) = 2.000000
Math.max( 2.3, 12.7 ) = 12.700000
Math.max( -2.3, -12.7 ) = -2.300000
Math.min( 2.3, 12.7 ) = 2.300000
Math.min( -2.3, -12.7 ) = -12.700000
Math.pow( 2.0, 7.0 ) = 128.000000
Math.pow( 9.0, 0.5 ) = 3.000000
Math.sin( 0.0 ) = 0.000000
Math.sqrt( 900.0 ) = 30.000000
Math.tan( 0.0 ) = 0.000000
```

6.4 a) `double hipotenusa( double lado1, double lado2 )`
b) `int menor( int x, int y, int z )`
c) `void instrucciones()`
d) `float intToFloat( int numero )`

6.5 a) Error: el método `h` está declarado dentro del método `g`.
Corrección: mueva la declaración de `h` fuera de la declaración de `g`.
b) Error: se supone que el método debe devolver un entero, pero no es así.
Corrección: elimine la variable resultado, y coloque la instrucción

```
return x + y;
```

en el método, o agregue la siguiente instrucción al final del cuerpo del método:

```
return resultado;
```

c) Error: el punto y coma que va después del paréntesis derecho de la lista de parámetros es incorrecto, y el parámetro a no debe volver a declararse en el método.
Corrección: elimine el punto y coma que va después del paréntesis derecho de la lista de parámetros, y elimine la declaración `float a;`.

d) Error: el método devuelve un valor cuando no debe hacerlo.
Corrección: cambie el tipo de valor de retorno de `void` a `int`.

6.6 La siguiente solución calcula el volumen de una esfera, utilizando el radio introducido por el usuario:

```
1  // Ejercicio 6.6: Esfera.java
2  // Calcula el volumen de una esfera.
3  import java.util.Scanner;
4
5  public class Esfera
6  {
7     // obtiene el radio del usuario y muestra el volumen de la esfera
8     public static void main( String[] args )
9     {
10       Scanner entrada = new Scanner( System.in );
11
12       System.out.print( "Escriba el radio de la esfera: " );
13       double radio = entrada.nextDouble();
14
15       System.out.printf( "El volumen es %f\n", volumenEsfera( radio ) );
16    } // fin del método determinarVolumenEsfera
17
18    // calcula y devuelve el volumen de una esfera
19    public static double volumenEsfera( double radio )
20    {
21       double volumen = ( 4.0 / 3.0 ) * Math.PI * Math.pow( radio, 3 );
22       return volumen;
23    } // fin del método volumenEsfera
24 } // fin de la clase Esfera
```

```
Escriba el radio de la esfera: 4
El volumen es 268.082573
```

Ejercicios

6.7 ¿Cuál es el valor de x después de que se ejecuta cada una de las siguientes instrucciones?

a) `x = Math.abs( 7.5 );`
b) `x = Math.floor( 7.5 );`
c) `x = Math.abs( 0.0 );`
d) `x = Math.ceil( 0.0 );`
e) `x = Math.abs( -6.4 );`
f) `x = Math.ceil( -6.4 );`
g) `x = Math.ceil( -Math.abs( -8 + Math.floor( -5.5 ) ) );`

6.8 ***(Cargos por estacionamiento)*** Un estacionamiento cobra una cuota mínima de $2.00 por estacionarse hasta tres horas. El estacionamiento cobra $0.50 adicionales por cada hora *o fracción* que se pase de tres horas. El cargo máximo para cualquier periodo dado de 24 horas es de $10.00. Suponga que ningún auto se estaciona durante más de 24 horas a la vez. Escriba una aplicación que calcule y muestre los cargos por estacionamiento para cada cliente que se haya estacionado ayer.

Debe introducir las horas de estacionamiento para cada cliente. El programa debe mostrar el cargo para el cliente actual así como calcular y mostrar el total corriente de los recibos de ayer. El programa debe utilizar el método `calcularCargos` para determinar el cargo para cada cliente.

6.9 ***(Redondeo de números)*** El método `Math.floor` se puede usar para redondear un valor al siguiente entero; por ejemplo, la instrucción,

```
y = Math.floor( x + 0.5 );
```

redondea el número `x` al entero más cercano y asigna el resultado a `y`. Escriba una aplicación que lea valores `double` y que utilice la instrucción anterior para redondear cada uno de los números a su entero más cercano. Para cada número procesado, muestre tanto el número original como el redondeado.

6.10 ***(Redondeo de números)*** Para redondear números hasta un lugar decimal específico, use una instrucción como la siguiente:

```
y = Math.floor( x * 10 + 0.5 ) / 10;
```

la cual redondea `x` en la posición de las décimas (es decir, la primera posición a la derecha del punto decimal), o:

```
y = Math.floor( x * 100 + 0.5 ) / 100;
```

que redondea `x` en la posición de las centésimas (es decir, la segunda posición a la derecha del punto decimal). Escriba una aplicación que defina cuatro métodos para redondear un número `x` en varias formas:

a) `redondearAInteger( numero )`
b) `redondearADecimas( numero )`
c) `redondearACentesimas( numero )`
d) `redondearAMilesimas( numero )`

Para cada valor leído, su programa debe mostrar el valor original, el número redondeado al entero más cercano, el número redondeado a la décima más cercana, el número redondeado a la centésima más cercana y el número redondeado a la milésima más cercana.

6.11 Responda a cada una de las siguientes preguntas:

a) ¿Qué significa elegir números "al azar"?
b) ¿Por qué es el método `nextInt` de la clase `Random` útil para simular juegos al azar?
c) ¿Por qué es a menudo necesario escalar o desplazar los valores producidos por un objeto `Random`?
d) ¿Por qué es la simulación computarizada de las situaciones reales una técnica útil?

6.12 Escriba instrucciones que asignen enteros aleatorios a la variable n en los siguientes rangos:

a) $1 \leq n \leq 2$.
b) $1 \leq n \leq 100$.
c) $0 \leq n \leq 9$.
d) $1000 \leq n \leq 1112$.
e) $-1 \leq n \leq 1$.
f) $-3 \leq n \leq 11$.

6.13 Escriba instrucciones que impriman un número al azar de cada uno de los siguientes conjuntos:

a) 2, 4, 6, 8, 10.
b) 3, 5, 7, 9, 11.
c) 6, 10, 14, 18, 22.

6.14 ***(Exponenciación)*** Escriba un método llamado `enteroPotencia(base,exponente)` que devuelva el valor de

$base^{exponente}$

Por ejemplo, `enteroPotencia(3,4)` calcula 3^4 (o `3 * 3 * 3 * 3`). Suponga que `exponente` es un entero positivo distinto de cero y que `base` es un entero. Use una instrucción `for` o `while` para controlar el cálculo. No utilice ningún método de la clase `Math`. Incorpore este método en una aplicación que lea valores enteros para `base` y `exponente`, y que realice el cálculo con el método `enteroPotencia`.

6.15 ***(Cálculo de la hipotenusa)*** Defina un método llamado `hipotenusa` que calcule la longitud de la hipotenusa de un triángulo rectángulo, cuando se proporcionen las longitudes de los otros dos lados. El método debe tomar dos argumentos

de tipo `double` y devolver la hipotenusa como un valor `double`. Incorpore este método en una aplicación que lea los valores para `lado1` y `lado2`, y que realice el cálculo con el método `hipotenusa`. Use los métodos `pow` y `sqrt` de `Math` para determinar la longitud de la hipotenusa para cada uno de los triángulos de la figura 6.15. [*Nota:* la clase `Math` también cuenta con el método `hypot` para realizar este cálculo].

Triángulo	Lado 1	Lado 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

Fig. 6.15 | Valores para los lados de los triángulos del ejercicio 6.15.

6.16 ***(Múltiplos)*** Escriba un método llamado `esMultiplo` que determine, para un par de enteros, si el segundo entero es múltiplo del primero. El método debe tomar dos argumentos enteros y devolver `true` si el segundo es múltiplo del primero, y `false` en caso contrario. [*Sugerencia:* utilice el operador residuo.] Incorpore este método en una aplicación que reciba como entrada una serie de pares de enteros (un par a la vez) y determine si el segundo valor en cada par es un múltiplo del primero.

6.17 ***(Par o impar)*** Escriba un método llamado `esPar` que utilice el operador residuo (%) para determinar si un entero dado es par. El método debe tomar un argumento entero y devolver `true` si el entero es par, y `false` en caso contrario. Incorpore este método en una aplicación que reciba como entrada una secuencia de enteros (uno a la vez), y que determine si cada uno es par o impar.

6.18 ***(Mostrar un cuadrado de asteriscos)*** Escriba un método llamado `cuadradoDeAsteriscos` que muestre un cuadrado relleno (el mismo número de filas y columnas) de asteriscos cuyo lado se especifique en el parámetro entero lado. Por ejemplo, si lado es 4, el método debe mostrar:

```
****
****
****
****
```

Incorpore este método a una aplicación que lea un valor entero para el parámetro `lado` que introduzca el usuario, y despliegue los asteriscos con el método `cuadradoDeAsteriscos`.

6.19 ***(Mostrar un cuadrado de cualquier carácter)*** Modifique el método creado en el ejercicio 6.18 para que reciba un segundo parámetro de tipo `char`, llamado `caracterRelleno`. Para formar el cuadrado, utilice el `char` que se proporciona como argumento. Por ejemplo, si `lado` es 5 y `caracterRelleno` es #, el método debe imprimir

```
#####
#####
#####
#####
#####
```

Use la siguiente instrucción (en donde entrada es un objeto `Scanner`) para leer un carácter del usuario mediante el teclado:

```
char relleno = entrada.next().charAt( 0 );
```

6.20 ***(Área de un círculo)*** Escriba una aplicación que pida al usuario el radio de un círculo y que utilice un método llamado `circuloArea` para calcular e imprimir el área.

6.21 ***(Separación de dígitos)*** Escriba métodos que realicen cada una de las siguientes tareas:

a) Calcular la parte entera del cociente, cuando el entero a se divide entre el entero b.
b) Calcular el residuo entero cuando el entero a se divide entre el entero b.

c) Utilizar los métodos desarrollados en las partes (a) y (b) para escribir un método llamado `mostrarDigitos`, que reciba un entero entre `1` y `99999`, y que lo muestre como una secuencia de dígitos, separando cada par de dígitos por dos espacios. Por ejemplo, el entero `4562` debe aparecer como

```
4  5  6  2
```

Incorpore los métodos en una aplicación que reciba como entrada un entero y que llame al método `mostrarDigitos`, pasándole el entero introducido. Muestre los resultados.

6.22 ***(Conversiones de temperatura)*** Implemente los siguientes métodos enteros:

a) El método `centigrados` que devuelve la equivalencia en grados Centígrados de una temperatura en grados Fahrenheit, mediante el cálculo

```
centigrados = 5.0 / 9.0 * ( fahrenheit - 32 );
```

b) El método `fahrenheit` que devuelve la equivalencia en grados Fahrenheit de una temperatura en grados Centígrados, con el cálculo

```
fahrenheit = 9.0 / 5.0 * centigrados + 32;
```

c) Utilice los métodos de las partes (a) y (b) para escribir una aplicación que permita al usuario, ya sea escribir una temperatura en grados Fahrenheit y mostrar su equivalente en Centigrados, o escribir una temperatura en grados Centígrados y mostrarla en grados Fahrenheit.

6.23 ***(Encuentre el mínimo)*** Escriba un método llamado `minimo3` que devuelva el menor de tres números de punto flotante. Use el método `Math.min` para implementar `minimo3`. Incorpore el método en una aplicación que reciba como entrada tres valores por parte del usuario, determine el valor menor y muestre el resultado.

6.24 ***(Números perfectos)*** Se dice que un número entero es un *número perfecto* si sus factores, incluyendo 1 (pero no el número entero), al sumarse dan como resultado el número entero. Por ejemplo, 6 es un número perfecto ya que 6 = 1 + 2 + 3. Escriba un método llamado `esPerfecto` que determine si el parámetro `numero` es un número perfecto. Use este método en una aplicación que muestre todos los números perfectos entre 1 y 1,000. Imprima los factores de cada número perfecto para confirmar que el número sea realmente perfecto. Ponga a prueba el poder de su computadora, evalué números más grandes que 1,000. Muestre los resultados.

6.25 ***(Números primos)*** Se dice que un entero es *primo* si puede dividirse solamente por 1 y por sí mismo. Por ejemplo, 2, 3, 5 y 7 son primos, pero 4, 6, 8 y 9 no. Por definición, el número 1 no es primo.

a) Escriba un método que determine si un número es primo.

b) Use este método en una aplicación que determine e imprima todos los números primos menores que 10,000. ¿Cuántos números hasta 10,000 tiene que probar para asegurarse de encontrar todos los números primos?

c) Al principio podría pensarse que $n/2$ es el límite superior para evaluar si un número n es primo, pero sólo es necesario ir hasta la raíz cuadrada de n. Vuelva a escribir el programa y ejecútelo de ambas formas.

6.26 ***(Invertir dígitos)*** Escriba un método que tome un valor entero y devuelva el número con sus dígitos invertidos. Por ejemplo, para el número 7631, el método debe regresar 1367. Incorpore el método en una aplicación que reciba como entrada un valor del usuario y muestre el resultado.

6.27 ***(Máximo común divisor)*** El *máximo común divisor* (*MCD*) de dos enteros es el entero más grande que puede dividir de manera uniforme a cada uno de los dos números. Escriba un método llamado `mcd` que devuelva el máximo común divisor de dos enteros. [*Sugerencia:* tal vez sea conveniente que utilice el algoritmo de Euclides. Puede encontrar información acerca de este algoritmo en `es.wikipedia.org/wiki/Algoritmo_de_Euclides`.] Incorpore el método en una aplicación que reciba como entrada dos valores del usuario y muestre el resultado.

6.28 Escriba un método llamado `puntosCalidad` que reciba como entrada el promedio de un estudiante y devuelva `4` si el promedio se encuentra entre 90 y 100, `3` si el promedio se encuentra entre 80 y 89, `2` si el promedio se encuentra entre 70 y 79, `1` si el promedio se encuentra entre 60 y 69, y `0` si el promedio es menor que 60. Incorpore el método en una aplicación que reciba como entrada un valor del usuario y muestre el resultado.

6.29 ***(Lanzamiento de monedas)*** Escriba una aplicación que simule el lanzamiento de monedas. Deje que el programa lance una moneda cada vez que el usuario seleccione la opción del menú "`Lanzar moneda`". Cuente el número de veces que aparezca cada uno de los lados de la moneda. Muestre los resultados. El programa debe llamar a un método separado, llamado tirar, que no tome argumentos y devuelva un valor de una `enum` llamada `Moneda` (`CARA` y `CRUZ`). [*Nota:* si el programa simula en forma realista el lanzamiento de monedas, cada lado de la moneda debe aparecer aproximadamente la mitad del tiempo).

6.30 ***(Adivine el número)*** Escriba una aplicación que juegue a "adivinar el número" de la siguiente manera: su programa elige el número a adivinar, cuando selecciona un entero aleatorio en el rango de 1 a 1,000. La aplicación muestra el indicador `Adivine un número entre 1 y 1,000`. El jugador escribe su primer intento. Si la respuesta del jugador es incorrecta, su programa debe mostrar el mensaje `Demasiado alto. Intente de nuevo.` o `Demasiado bajo. Intente de nuevo.`, para ayudar a que el jugador "se acerque" a la respuesta correcta. El programa debe pedir al usuario que escriba su siguiente intento. Cuando el usuario escriba la respuesta correcta, muestre el mensaje `Felicidades. Adivino el numero!` y permita que el usuario elija si desea jugar otra vez. [*Nota:* la técnica para adivinar empleada en este problema es similar a una búsqueda binaria, que veremos en el capítulo 19 (en el sitio Web), Búsqueda, ordenamiento y Big O].

6.31 ***(Modificación de adivine el número)*** Modifique el programa del ejercicio 6.30 para contar el número de intentos que haga el jugador. Si el número es 10 o menos, imprima el mensaje `O ya sabia usted el secreto, o tuvo suerte!` Si el jugador adivina el número en 10 intentos, imprima el mensaje `Aja! Sabia usted el secreto!` Si el jugador hace más de 10 intentos, imprima `Deberia haberlo hecho mejor!` ¿Por qué no se deben requerir más de 10 intentos? Bueno, en cada "buen intento", el jugador debe poder eliminar la mitad de los números, después la mitad de los restantes, y así en lo sucesivo.

6.32 ***(Distancia entre puntos)*** Escriba un método llamado `distancia`, para calcular la distancia entre dos puntos (*x1*, *y1*) y (*x2*, *y2*). Todos los números y valores de retorno deben ser de tipo `double`. Incorpore este método en una aplicación que permita al usuario introducir las coordenadas de los puntos.

6.33 ***(Modificación del juego de Craps)*** Modifique el programa Craps de la figura 6.8 para permitir apuestas. Inicialice la variable `saldoBanco` con $1,000. Pida al jugador que introduzca una `apuesta`. Compruebe que esa apuesta sea menor o igual que `saldoBanco` y, si no lo es, haga que el usuario vuelva a introducir la apuesta hasta que se ingrese un valor válido. Después de esto, comience un juego de Craps. Si el jugador gana, agregue la `apuesta` al `saldoBanco` e imprima el nuevo `saldoBanco`. Si pierde, reste la `apuesta` al `saldoBanco`, imprima el nuevo `saldoBanco`, compruebe si `saldoBanco` se ha vuelto cero y, de ser así, imprima el mensaje "`Lo siento. Se quedo sin fondos!`" A medida que el juego progrese, imprima varios mensajes para crear algo de "charla", como "`Oh, se esta yendo a la quiebra, verdad?`", o "`Oh, vamos, arriesguese!`", o "`La hizo en grande. Ahora es tiempo de cambiar sus fichas por efectivo!`". Implemente la "charla" como un método separado que seleccione en forma aleatoria la cadena a mostrar.

6.34 ***(Tabla de números binarios, octales y hexadecimales)*** Escriba una aplicación que muestre una tabla de los equivalentes en binario, octal y hexadecimal de los números decimales en el rango de 1 al 256. Si no está familiarizado con estos sistemas numéricos, lea el apéndice H primero.

Marcar la diferencia

A medida que disminuyen los costos de las computadoras, aumenta la posibilidad de que cada estudiante, sin importar su economía, tenga una computadora y la utilice en la escuela. Esto crea excitantes posibilidades para mejorar la experiencia educativa de todos los estudiantes a nivel mundial, según lo sugieren los siguientes cinco ejercicios. [*Nota:* vea nuestras iniciativas, como el proyecto One Laptop Per Child (`www.laptop.org`). Investigue también acerca de las laptops "verdes" o ecológicas: ¿cuáles son algunas características ecológicas clave de estos dispositivos? Investigue también la Herramienta de evaluación ambiental de productos electrónicos (`www.epeat.net`), que le puede ayudar a evaluar las características ecológicas de las computadoras de escritorio, notebooks y monitores para poder decidir qué productos comprar].

6.35 ***(Instrucción asistida por computadora)*** El uso de las computadoras en la educación se conoce como *instrucción asistida por computadora* (*CAI*). Escriba un programa que ayude aprender a multiplicar a un estudiante de escuela primaria.

Use un objeto `Random` para producir dos enteros positivos de un dígito. El programa debe entonces mostrar una pregunta al usuario, como:

```
¿Cuánto es 6 por 7?
```

El estudiante entonces debe escribir la respuesta. Luego, el programa debe verificar la respuesta del estudiante. Si es correcta, muestre el mensaje `"Muy bien!"` y haga otra pregunta de multiplicación. Si la respuesta es incorrecta, dibuje la cadena `"No. Por favor intenta de nuevo."` y deje que el estudiante intente la misma pregunta varias veces, hasta que esté correcta. Debe utilizarse un método separado para generar cada pregunta nueva. Este método debe llamarse una vez cuando la aplicación empiece a ejecutarse, y cada vez que el usuario responda correctamente a la pregunta.

6.36 ***(Instrucción asistida por computadora: reducción de la fatiga de los estudiantes)*** Un problema que se desarrolla en los entornos CAI es la fatiga de los estudiantes. Este problema puede eliminarse si se varían las contestaciones de la computadora para mantener la atención del estudiante. Modifique el programa del ejercicio 6.35 de manera que se muestren diversos comentarios para cada respuesta, de la siguiente manera:

Posibles contestaciones a una respuesta correcta:

```
Muy bien!
Excelente!
Buen trabajo!
Sigue asi!
```

Contestaciones a una respuesta incorrecta:

```
No. Por favor intenta de nuevo.
Incorrecto. Intenta una vez mas.
No te rindas!
No. Sigue intentando.
```

Use la generación de números aleatorios para elegir un número entre 1 y 4 que se utilice para seleccionar una de las cuatro contestaciones apropiadas a cada respuesta correcta o incorrecta. Use una instrucción `switch` para emitir las contestaciones.

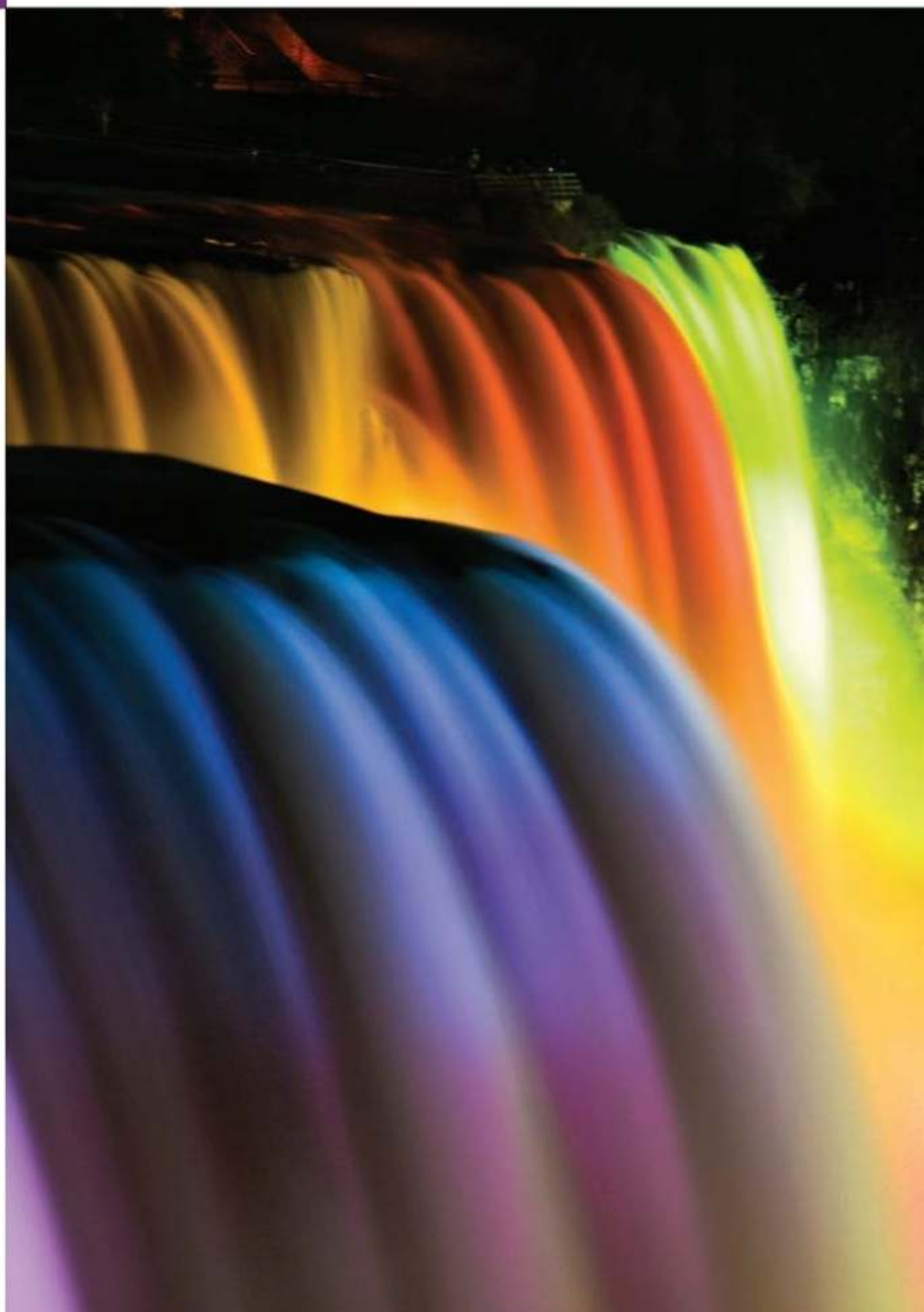
6.37 ***(Instrucción asistida por computadora: supervisión del rendimiento de los estudiantes)*** Los sistemas de instrucción asistida por computadora más sofisticados supervisan el rendimiento del estudiante durante cierto tiempo. La decisión de empezar un nuevo tema se basa a menudo en el éxito del estudiante con los temas anteriores. Modifique el programa del ejercicio 6.36 para contar el número de respuestas correctas e incorrectas introducidas por el estudiante. Una vez que el estudiante escriba 10 respuestas, su programa debe calcular el porcentaje de respuestas correctas. Si éste es menor del 75%, imprima `"Por favor pida ayuda adicional a su instructor"` y reinicie el programa, para que otro estudiante pueda probarlo. Si el porcentaje es del 75% o mayor, muestre el mensaje `"Felicidades, esta listo para pasar al siguiente nivel!"` y luego reinicie el programa, para que otro estudiante pueda probarlo.

6.38 ***(Instrucción asistida por computadora: niveles de dificultad)*** En los ejercicios 6.35 al 6.37 se desarrolló un programa de instrucción asistida por computadora para enseñar a un estudiante de escuela primara cómo multiplicar. Modifique el programa para que permita al usuario introducir un nivel de dificultad. Un nivel de 1 significa que el programa debe usar sólo números de un dígito en los problemas; un nivel 2 significa que el programa debe utilizar números de dos dígitos máximo, y así en lo sucesivo.

6.39 ***(Instrucción asistida por computadora: variación de los tipos de problemas)*** Modifique el programa del ejercicio 6.38 para permitir al usuario que elija el tipo de problemas aritméticos que desea estudiar. Una opción de `1` significa problemas de suma solamente, `2` problemas de resta, `3` problemas de multiplicación, `4` problemas de división y `5` significa una mezcla aleatoria de problemas de todos estos tipos.

7

Arreglos y objetos `ArrayList`



Comienza en el principio…
y continúa hasta que llegues
al final; entonces detente.
—Lewis Carroll

Ahora ve, escríbelo ante ellos en
una tabla, y regístralo en un libro.
—Isaías 30:8

Ir más allá es tan malo
como no llegar.
—Confucio

Objetivos

En este capítulo aprenderá a:

- Conocer qué son los arreglos.
- Utilizar arreglos para almacenar datos en, y obtenerlos de listas y tablas de valores.
- Declarar arreglos, inicializarlos y hacer referencia a elementos individuales de ellos.
- Iterar a través de los arreglos mediante la instrucción `for` mejorada.
- Pasar arreglos a los métodos.
- Declarar y manipular arreglos multidimensionales.
- Usar listas de argumentos de longitud variable.
- Leer los argumentos de línea de comandos en un programa.
- Realizar manipulaciones comunes de arreglos con los métodos de la clase `Arrays`.
- Usar la clase `ArrayList` para manipular una estructura de datos tipo arreglo, cuyo tamaño es ajustable.

Plan general

7.1 Introducción

En este capítulo presentamos las **estructuras de datos**: colecciones de elementos de datos relacionados. Los **arreglos** son estructuras de datos que consisten de elementos de datos relacionados, del mismo tipo. Conservan la misma longitud una vez creados, aunque puede reasignarse una variable tipo arreglo de tal forma que haga referencia a un nuevo arreglo de distinta longitud. En los capítulos 20 a 22 (en el sitio Web) estudiaremos con detalle las estructuras de datos.

Después de hablar acerca de cómo se declaran, crean e inicializan los arreglos, presentaremos una serie de ejemplos prácticos que demuestran varias manipulaciones comunes de los arreglos. Introduciremos el mecanismo de manejo de excepciones de Java y lo utilizaremos para permitir que un programa se siga ejecutando cuando intente acceder al elemento inexistente de un arreglo. También presentaremos un caso de estudio en el que se examina la forma en que los arreglos pueden ayudar a simular los procesos de barajar y repartir cartas en una aplicación que implementa un juego de cartas. Después presentaremos la instrucción `for` mejorada de java, la cual permite que un programa acceda a los datos en un arreglo con más facilidad que la instrucción `for` controlada por contador, que presentamos en la sección 5.3. Ampliaremos el caso de estudio de la clase `LibroCalificaciones` de los capítulos 3 a 5. En especial, utilizaremos los arreglos para permitir que la clase mantenga un conjunto de calificaciones *en memoria* y analizar las calificaciones que obtuvieron los estudiantes en distintos exámenes. Le mostraremos cómo usar listas de argumentos de longitud variable para crear métodos que se puedan llamar con números variables de argumentos, y demostraremos cómo procesar los argumentos de la línea de comandos en el método `main`. Más adelante, presentaremos algunas manipulaciones comunes de arreglos con métodos `static` de la clase `Arrays`, del paquete `java.util`.

Aunque se utilizan con frecuencia, los arreglos tienen capacidades limitadas. Por ejemplo, es necesario especificar el tamaño, y si en tiempo de ejecución desea modificarlo, debe hacerlo en forma manual mediante la creación de un nuevo arreglo. Al final de este capítulo le presentaremos una de las estructuras de datos preconstruidas de Java, proveniente de las clases de colecciones de la API de Java. Estas colecciones ofrecen mayores capacidades que los arreglos tradicionales y son reutilizables, confiables, poderosas y eficientes. Nos enfocaremos en la colección `ArrayList`. Los objetos `ArrayList` son similares a los arreglos, sólo que proporcionan una funcionalidad mejorada, como el **ajuste de tamaño dinámico**: aumentan su tamaño en forma dinámica, en tiempo de ejecución, para poder alojar elementos adicionales.

7.2 Arreglos

Un arreglo es un grupo de variables (llamadas **elementos** o **componentes**) que contienen valores, todos del mismo tipo. Los arreglos son *objetos*, por lo que se consideran como tipos de referencia. Como veremos pronto, lo que consideramos por lo general como un arreglo es en realidad una referencia a un objeto arreglo en memoria. Los *elementos* de un arreglo pueden ser tipos primitivos o de referencia (incluyendo arreglos, como veremos en la sección 7.9). Para hacer referencia a un elemento específico en un arreglo, debemos especificar el nombre de la referencia al arreglo y el *número de la posición* del elemento en el arreglo. El número de la posición del elemento se conoce formalmente como el **índice** o **subíndice** del elemento.

En la figura 7.1 se muestra una representación lógica de un arreglo de enteros, llamado c. Este arreglo contiene 12 elementos. Un programa puede hacer referencia a cualquiera de estos elementos mediante una **expresión de acceso a un arreglo** que contiene el nombre del arreglo, seguido por el índice del elemento específico encerrado entre **corchetes** ([]). El primer elemento en cualquier arreglo tiene el **índice cero**, y algunas veces se le denomina **elemento cero**. Por lo tanto, los elementos del arreglo c son c[0], c[1], c[2], y así en lo sucesivo. El mayor índice en el arreglo c es 11: 1 menos que 12, el número de elementos en el arreglo. Los nombres de los arreglos siguen las mismas convenciones que los demás nombres de variables.

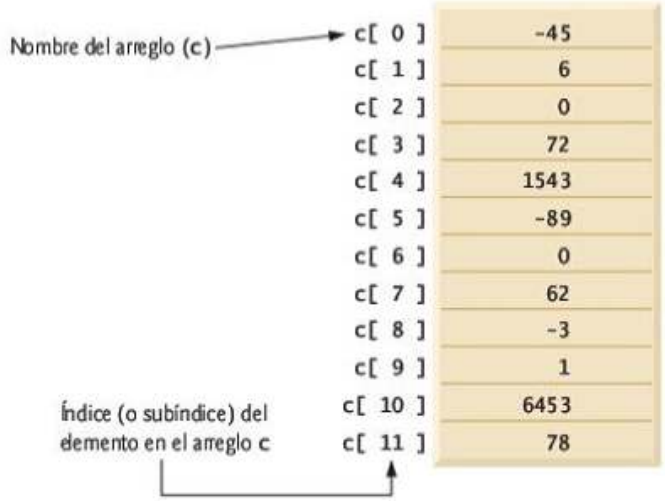


Fig. 7.1 | Un arreglo con 12 elementos.

Un índice debe ser un entero positivo. Un programa puede utilizar una expresión como índice. Por ejemplo, si suponemos que la variable a es 5 y que b es 6, entonces la instrucción

```
c[ a + b ] += 2;
```

suma 2 al elemento c[11] del arreglo. El nombre de un arreglo con subíndice es una expresión de acceso al arreglo, la cual puede utilizarse en el lado izquierdo de una asignación, para colocar un nuevo valor en un elemento del arreglo.

Error común de programación 7.1

Un índice debe ser un valor `int`*, o un valor de un tipo que pueda promoverse a* `int`*;* `byte`*,* `short` *o* `char`*, pero no* `long`*. De lo contrario, ocurre un error de compilación.*

Vamos a examinar el arreglo c de la figura 7.1 con más detalle. El **nombre** del arreglo es c. Cada objeto arreglo conoce su propia longitud y mantiene esta información en una **variable de instancia**

`length`. La expresión `c.length` accede al campo `length` del arreglo `c` para determinar la longitud del arreglo. Aun cuando la variable de instancia `length` de un arreglo es `public`, no puede cambiarse, ya que es una variable `final`. La manera en que se hace referencia a los 12 elementos de este arreglo es: `c[0]`, `c[1]`, `c[2]`, ..., `c[11]`. El valor de `c[0]` es –45, el valor de `c[1]` es 6, el de `c[2]` es 0, el de `c[7]` es 62 y el de `c[11]` es 78. Para calcular la suma de los valores contenidos en los primeros tres elementos del arreglo `c` y almacenar el resultado en la variable `suma`, escribiríamos lo siguiente:

```
suma = c[ 0 ] + c[ 1 ] + c[ 2 ];
```

Para dividir el valor de `c[6]` entre 2 y asignar el resultado a la variable `x`, escribiríamos lo siguiente:

```
x = c[ 6 ] / 2;
```

7.3 Declaración y creación de arreglos

Los objetos arreglo ocupan espacio en memoria. Al igual que los demás objetos, los arreglos se crean con la palabra clave `new`. Para crear un objeto arreglo, debemos especificar el tipo de cada elemento y el número de elementos que se requieren para el arreglo, como parte de una **expresión para crear un arreglo** que utiliza la palabra clave `new`. Dicha expresión devuelve una referencia que puede almacenarse en una variable tipo arreglo. La siguiente declaración y expresión crea un objeto arreglo, que contiene 12 elementos `int` y almacena la referencia del arreglo en la variable `c`:

```
int[] c = new int[ 12 ];
```

Esta expresión puede usarse para crear el arreglo que se muestra en la figura 7.1. Al crear un arreglo, cada uno de sus elementos recibe un valor predeterminado: cero para los elementos numéricos de tipos primitivos, `false` para los elementos `boolean` y `null` para las referencias. Como pronto veremos, podemos proporcionar valores iniciales para los elementos no predeterminados al crear un arreglo.

La creación del arreglo de la figura 7.1 también puede realizarse en dos pasos, como se muestra a continuación:

```
int[] c; // declara la variable arreglo
c = new int[ 12 ]; // crea el arreglo; lo asigna a la variable tipo arreglo
```

En la declaración, los corchetes que van después del tipo indican que `c` es una variable que hará referencia a un arreglo (es decir, la variable almacenará una referencia a un arreglo). En la instrucción de asignación, la variable arreglo `c` recibe la referencia a un nuevo objeto arreglo de 12 elementos `int`.

Error común de programación 7.2

En la declaración de un arreglo, si se especifica el número de elementos en los corchetes de la declaración (por ejemplo, `int[12] c;`), se produce un error de sintaxis.

Un programa puede crear varios arreglos en una sola declaración. La siguiente declaración reserva 100 elementos para `b` y 27 para `x`:

```
String[] b = new String[ 100 ], x = new String[ 27 ];
```

Cuando el tipo del arreglo y los corchetes se combinan al principio de la declaración, todos los identificadores en ésta son variables tipo arreglo. En este caso, las variables `b` y `x` hacen referencia a arreglos `String`. Por cuestión de legibilidad, es preferible declarar sólo una variable en cada declaración. La declaración anterior es equivalente a:

```
String[] b = new String[ 100 ]; // crea el arreglo b
String[] x = new String[ 27 ]; // crea el arreglo x
```

Buena práctica de programación 7.1

Por cuestión de legibilidad, declare sólo una variable en cada declaración. Mantenga cada declaración en una línea separada e introduzca un comentario que describa a la variable que está declarando.

Cuando sólo se expone una variable en cada declaración, los corchetes se pueden colocar después del tipo o del nombre de la variable, como en:

```
String b[] = new String[ 100 ]; // crea el arreglo b
String x[] = new String[ 27 ]; // crea el arreglo x
```

Error común de programación 7.3

Exponer múltiples variables tipo arreglo en una sola declaración puede provocar errores sutiles. Considere la declaración `int[] a, b, c;`. *Si* `a`, `b` *y* `c` *deben declararse como variables tipo arreglo, entonces esta declaración es correcta; al colocar corchetes directamente después del tipo, indicamos que* todos *los identificadores en la declaración son variables tipo arreglo. No obstante, si sólo* `a` *debe ser una variable tipo arreglo, y* `b` *y* `c` *deben ser variables* `int` *individuales, entonces esta declaración es incorrecta; la declaración* `int a[], b, c;` *lograría el resultado deseado.*

Un programa puede declarar arreglos de cualquier tipo. Cada elemento de un arreglo de tipo primitivo contiene un valor del tipo del elemento declarado del arreglo. De manera similar, en un arreglo de un tipo de referencia, cada elemento es una referencia a un objeto del tipo del elemento declarado del arreglo. Por ejemplo, cada elemento de un arreglo `int` es un valor `int`, y cada elemento de un arreglo `String` es una referencia a un objeto `String`.

7.4 Ejemplos acerca del uso de los arreglos

En esta sección presentaremos varios ejemplos que muestran cómo declarar, crear e inicializar arreglos, y cómo manipular sus elementos.

Cómo crear e inicializar un arreglo

En la aplicación de la figura 7.2 se utiliza la palabra clave `new` para crear un arreglo de 10 elementos `int`, los cuales en un principio tienen el valor cero (el valor predeterminado para las variables `int`). En la línea 8 se declara `arreglo`: una referencia capaz de referirse a un arreglo de elementos `int`. En la línea 10 se crea el objeto arreglo y se asigna su referencia a la variable `arreglo`. La línea 12 imprime los encabezados de las columnas. La primera columna representa el índice (0 a 9) para cada elemento del arreglo, y la segunda contiene el valor predeterminado (0) de cada elemento del arreglo.

```
1  // Fig. 7.2: InicArreglo.java
2  // Inicialización de los elementos de un arreglo con valores predeterminados de cero.
3
4  public class InicArreglo
5  {
6     public static void main( String[] args )
7     {
8        int[] arreglo; // declara un arreglo con el mismo nombre
9
10       arreglo = new int[ 10 ]; // crea el objeto arreglo
```

Fig. 7.2 | Inicialización de los elementos de un arreglo con valores predeterminados de cero (parte 1 de 2).

```
11
12          System.out.printf( "%s%8s\n", "Indice", "Valor" ); // encabezados de columnas
13
14          // imprime el valor de cada elemento del arreglo
15          for ( int contador = 0; contador < arreglo.length; contador++ )
16             System.out.printf( "%5d%8d\n", contador, arreglo[ contador ] );
17       } // fin de main
18    } // fin de la clase InicArreglo
```

```
Indice   Valor
     0       0
     1       0
     2       0
     3       0
     4       0
     5       0
     6       0
     7       0
     8       0
     9       0
```

Fig. 7.2 | Inicialización de los elementos de un arreglo con valores predeterminados de cero (parte 2 de 2).

La instrucción `for` en las líneas 15 y 16 imprime el número de índice (representado por `contador`) y el valor de cada elemento del arreglo (representado por `arreglo[contador]`). Al principio la variable de control del ciclo `contador` es 0 (los valores de los índices empiezan en 0, por lo que al utilizar un **conteo con base cero** se permite al ciclo acceder a todos los elementos del arreglo. La condición de continuación de ciclo de la instrucción `for` utiliza la expresión `arreglo.length` (línea 15) para determinar la longitud del arreglo. En este ejemplo la longitud del arreglo es de 10, por lo que el ciclo continúa ejecutándose mientras el valor de la variable de control `contador` sea menor que 10. El valor más alto para el subíndice de un arreglo de 10 elementos es 9, por lo que al utilizar el operador "menor que" en la condición de continuación de ciclo se garantiza que el ciclo no trate de acceder a un elemento *más allá* del final del arreglo (es decir, durante la iteración final del ciclo, `contador` es 9). Pronto veremos lo que hace Java cuando encuentra un *subíndice fuera de rango* en tiempo de ejecución.

Uso de un inicializador de arreglo

Usted puede crear un arreglo e inicializar sus elementos con un **inicializador de arreglo**: una lista de expresiones separadas por comas (la cual se conoce también como **lista inicializadora**) y encerrada entre llaves. En este caso, la longitud del arreglo se determina con base en el número de elementos en la lista inicializadora. Por ejemplo, la declaración,

```
int[] n = { 10, 20, 30, 40, 50 };
```

crea un arreglo de cinco elementos con los valores de índices 0 a 4. El elemento `n[0]` se inicializa con 10, `n[1]` se inicializa con 20, y así en lo sucesivo. Cuando el compilador encuentra la declaración de un arreglo que incluye una lista inicializadora, cuenta el número de inicializadores en la lista para determinar el tamaño del arreglo, y después establece la operación `new` apropiada "detrás de las cámaras".

La aplicación de la figura 7.3 inicializa un arreglo de enteros con 10 valores (línea 9) y muestra el arreglo en formato tabular. El código para mostrar los elementos del arreglo (líneas 14 y 15) es idéntico al de la figura 7.2 (líneas 15 y 16).

```
1  // Fig. 7.3: InicArreglo.java
2  // Inicialización de los elementos de un arreglo con un inicializador de arreglo.
3
4  public class InicArreglo
5  {
6     public static void main( String[] args )
7     {
8        // la lista inicializadora especifica el valor para cada elemento
9        int[] arreglo = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10
11       System.out.printf( "%s%8s\n", "Indice", "Valor" ); // encabezados de columnas
12
13       // imprime el valor del elemento de cada arreglo
14       for ( int contador = 0; contador < arreglo.length; contador++ )
15          System.out.printf( "%5d%8d\n", contador, arreglo[ contador ] );
16    } // fin de main
17 } // fin de la clase InicArreglo
```

```
Indice   Valor
     0      32
     1      27
     2      64
     3      18
     4      95
     5      14
     6      90
     7      70
     8      60
     9      37
```

Fig. 7.3 | Inicialización de los elementos de un arreglo con un inicializador de arreglo.

Cálculo de los valores a guardar en un arreglo

La aplicación de la figura 7.4 crea un arreglo de 10 elementos y asigna a cada elemento uno de los enteros pares del 2 al 20 (2, 4, 6, ..., 20). Después, la aplicación muestra el arreglo en formato tabular. La instrucción for en las líneas 12 y 13 calcula el valor de un elemento del arreglo, multiplicando el valor actual de la variable de control contador por 2, y después le suma 2.

```
1  // Fig. 7.4: InicArreglo.java
2  // Cálculo de los valores a colocar en los elementos de un arreglo.
3
4  public class InicArreglo
5  {
6     public static void main( String[] args )
7     {
8        final int LONGITUD_ARREGLO = 10; // declara la constante
9        int[] arreglo = new int[ LONGITUD_ARREGLO ]; // crea el arreglo
10
11       // calcula el valor para cada elemento del arreglo
12       for ( int contador = 0; contador < arreglo.length; contador++ )
13          arreglo[ contador ] = 2 + 2 * contador;
```

Fig. 7.4 | Cálculo de los valores a colocar en los elementos de un arreglo (parte 1 de 2).

```
14
15        System.out.printf( "%s%8s\n", "Indice", "Valor" ); // encabezados de columnas
16
17        // imprime el valor de cada elemento del arreglo
18        for ( int contador = 0; contador < arreglo.length; contador++ )
19           System.out.printf( "%5d%8d\n", contador, arreglo[ contador ] );
20     } // fin de main
21  } // fin de la clase InicArreglo
```

```
Indice   Valor
     0       2
     1       4
     2       6
     3       8
     4      10
     5      12
     6      14
     7      16
     8      18
     9      20
```

Fig. 7.4 | Cálculo de los valores a colocar en los elementos de un arreglo (parte 2 de 2).

La línea 8 utiliza el modificador `final` para declarar la variable constante `LONGITUD_ARREGLO` con el valor 10. Las variables constantes deben inicializarse antes de usarlas, y no pueden modificarse de ahí en adelante. Si trata de *modificar* una variable `final` después de inicializarla en su declaración, el compilador genera el siguiente mensaje de error:

```
cannot assign a value to final variable nombreVariable
```

Si tratamos de acceder al valor de una variable `final` antes de inicializarla, el compilador produce el siguiente mensaje de error:

```
variable nombreVariable might not have been initialized
```

Buena práctica de programación 7.2

Las variables constantes también se conocen como ***constantes con nombre****. Con frecuencia mejoran la legibilidad de un programa, en comparación con los programas que utilizan valores literales (por ejemplo, 10); una constante con nombre como* `LONGITUD_ARREGLO` *indica sin duda su propósito, mientras que un valor literal podría tener distintos significados, con base en su contexto..*

Error común de programación 7.4

Asignar un valor a una variable constante después de inicializarla es un error de compilación.

Error común de programación 7.5

Tratar de usar una constante antes de inicializarla es un error de compilación.

Suma de los elementos de un arreglo

A menudo, los elementos de un arreglo representan una serie de valores que se emplearán en un cálculo. Por ejemplo, si los elementos del arreglo representan las calificaciones de un examen, tal vez el profesor desee sumar el total de los elementos del arreglo y utilizar esa suma para calcular el promedio de la clase

para el examen. Los ejemplos que utilizan la clase `LibroCalificaciones` en las figuras 7.14 y 7.18 utilizan esta técnica.

La figura 7.5 suma los valores contenidos en el arreglo entero de 10 elementos. El programa declara, crea e inicializa el arreglo en la línea 8. La instrucción `for` realiza los cálculos. [*Nota:* los valores suministrados como inicializadores de arreglos generalmente se introducen en un programa, en vez de especificarse en una lista inicializadora. Por ejemplo, una aplicación podría recibir los valores del usuario o de un archivo en disco (como veremos en el capítulo 17, en el sitio Web del libro), Archivos, flujos y serialización de objetos). Al hacer que los datos se introduzcan como entrada en un programa (en vez de "codificarlos a mano" en el mismo) éste se hace más flexible, ya que puede utilizarse con distintos conjuntos de datos].

```
1  // Fig. 7.5: SumaArreglo.java
2  // Cálculo de la suma de los elementos de un arreglo.
3
4  public class SumaArreglo
5  {
6     public static void main( String[] args )
7     {
8        int[] arreglo = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // suma el valor de cada elemento al total
12       for ( int contador = 0; contador < arreglo.length; contador++ )
13          total += arreglo[ contador ];
14
15       System.out.printf( "Total de los elementos del arreglo: %d\n", total );
16    } // fin de main
17 } // fin de la clase SumaArreglo
```

```
Total de los elementos del arreglo: 849
```

Fig. 7.5 | Cálculo de la suma de los elementos de un arreglo.

Uso de gráficos de barra para mostrar los datos de un arreglo en forma gráfica

Muchos programas presentan datos a los usuarios en forma gráfica. Por ejemplo, con frecuencia los valores numéricos se muestran como barras en un gráfico de barras. En dicho gráfico, las barras más largas representan valores numéricos más grandes en forma proporcional. Una manera sencilla de mostrar los datos numéricos en forma gráfica es mediante un gráfico de barras que muestre cada valor numérico como una barra de asteriscos (*).

A los profesores les gusta examinar a menudo la distribución de las calificaciones en un examen. Un profesor podría graficar el número de calificaciones en cada una de varias categorías, para visualizar la distribución de las calificaciones. Suponga que las calificaciones en un examen fueron 87, 68, 94, 100, 83, 78, 85, 91, 76 y 87. Se incluye una calificación de 100, dos calificaciones en el rango de 90 a 99, cuatro calificaciones en el rango de 80 a 89, dos en el rango de 70 a 79, una en el rango de 60 a 69 y ninguna por debajo de 60. Nuestra siguiente aplicación (figura 7.6) almacena estos datos de distribución de las calificaciones en un arreglo de 11 elementos, cada uno de los cuales corresponde a una categoría de calificaciones. Por ejemplo, `arreglo[0]` indica el número de calificaciones en el rango de 0 a 9, `arreglo[7]` indica el número de calificaciones en el rango de 70 a 79 y `arreglo[10]` indica el número de calificaciones de 100. Las clases `LibroCalificaciones` que veremos más adelante en este capítulo (figuras 7.14 y 7.18) contienen código para calcular estas frecuencias de calificaciones, con base en un conjunto de calificaciones. Por ahora crearemos el arreglo en forma manual con las frecuencias de las calificaciones dadas.

```
1  // Fig. 7.6: GraficoBarras.java
2  // Programa para imprimir gráficos de barras.
3
4  public class GraficoBarras
5  {
6     public static void main( String[] args )
7     {
8        int[] arreglo = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
9
10       System.out.println( "Distribucion de calificaciones:" );
11
12       // para cada elemento del arreglo, imprime una barra del gráfico
13       for ( int contador = 0; contador < arreglo.length; contador++ )
14       {
15          // imprime etiqueta de la barra ( "00-09: ", ..., "90-99: ", "100: " )
16          if ( contador == 10 )
17             System.out.printf( "%5d: ", 100 );
18          else
19             System.out.printf( "%02d-%02d: ",
20                contador * 10, contador * 10 + 9  );
21
22          // imprime barra de asteriscos
23          for ( int estrellas = 0; estrellas < arreglo[ contador ]; estrellas++ )
24             System.out.print( "*" );
25
26          System.out.println(); // inicia una nueva línea de salida
27       } // fin de for externo
28    } // fin de main
29 } // fin de la clase GraficoBarras
```

```
Distribucion de calificaciones:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 7.6 | Programa para imprimir gráficos de barras.

La aplicación lee los números del arreglo y grafica la información en forma de un gráfico de barras. Muestra cada rango de calificaciones seguido de una barra de asteriscos, que indican el número de calificaciones en ese rango. Para etiquetar cada barra, las líneas 16 a 20 imprimen un rango de notas (por ejemplo, "70-79:") con base en el valor actual de `contador`. Cuando `contador` es 10, la línea 17 imprime 100 con una anchura de campo de 5, seguida de dos puntos y un espacio, para alinear la etiqueta "100:" con las otras etiquetas de las barras. La instrucción `for` anidada (líneas 23 y 24) imprime las barras en pantalla. Observe la condición de continuación de ciclo en la línea 23 (`estrellas < arreglo[contador]`). Cada vez que el programa llega al `for` interno, el ciclo cuenta desde 0 hasta `arreglo[contador]`, con lo cual utiliza un valor en `arreglo` para determinar el número de asteriscos a mostrar en pantalla. En este

ejemplo, ningún estudiante recibió una calificación menor de 60, por lo que los valores de `arreglo[0]` hasta `arreglo[5]` son ceros, y no se muestran asteriscos enseguida de los primeros seis rangos de calificaciones. En la línea 19, el especificador de formato `%02d` indica que se debe dar formato a un valor `int` como un campo de dos dígitos. La **bandera 0** en el especificador de formato muestra un `0` a la izquierda para los valores con menos dígitos que la anchura de campo (2).

Uso de los elementos de un arreglo como contadores

En ocasiones, los programas utilizan variables tipo contador para sintetizar datos, como los resultados de una encuesta. En la figura 6.7 utilizamos contadores separados en nuestro programa para tirar dados, para rastrear el número de veces que aparecía cada una de las caras de un dado con seis lados, al tiempo que la aplicación tiraba el dado 6,000,000 veces. En la figura 7.7 se muestra una versión de esta aplicación, sólo que ahora se utiliza un arreglo.

```
1  // Fig. 7.7: TirarDado.java
2  // Programa para tirar dados que utiliza arreglos en vez de switch.
3  import java.util.Random;
4
5  public class TirarDado
6  {
7     public static void main( String[] args )
8     {
9        Random numerosAleatorios = new Random(); // generador de números aleatorios
10       int[] frecuencia = new int[ 7 ]; // arreglo de contadores de frecuencia
11
12       // tira el dado 6,000,000 veces; usa el valor del dado como índice de frecuencia
13       for ( int tiro = 1; tiro <= 6000000; tiro++ )
14          ++frecuencia[ 1 + numerosAleatorios.nextInt( 6 ) ];
15
16       System.out.printf( "%s%10s\n", "Cara", "Frecuencia" );
17
18       // imprime el valor de cada elemento del arreglo
19       for ( int cara = 1; cara < frecuencia.length; cara++ )
20          System.out.printf( "%4d %10d\n", cara, frecuencia[ cara ] );
21    } // fin de main
22 } // fin de la clase TirarDado
```

```
Cara  Frecuencia
   1        1015
   2         999
   3         998
   4         996
   5        1044
   6         948
```

Fig. 7.7 | Programa para tirar dados que utiliza arreglos en vez de `switch`.

La figura 7.7 utiliza el arreglo `frecuencia` (línea 10) para contar las ocurrencias de cada lado del dado. *La instrucción individual en la línea 14 de este programa sustituye las líneas 23 a 46 de la figura 6.7.* La línea 14 utiliza el valor aleatorio para determinar qué elemento de `frecuencia` debe incrementar durante cada iteración del ciclo. El cálculo en la línea 14 produce números aleatorios del 1 al 6, por lo que el arreglo `frecuencia` debe ser lo bastante grande como para poder almacenar seis contadores. Sin embargo, utilizamos un arreglo de siete elementos, en el cual ignoramos `frecuencia[0]`; es más lógico que el valor de cara 1 incremente a `frecuencia[1]` que a `frecuencia[0]`. Por ende, cada valor de cara se utiliza como subíndice para el arreglo `frecuencia`. En la línea 14, se evalúa primero el cálculo dentro de

los corchetes para determinar qué elemento del arreglo se debe incrementar, y después el operador ++ suma uno a ese elemento. También sustituimos las líneas 50 a 52 de la figura 6.7 por un ciclo a través del arreglo `frecuencia` para imprimir los resultados en pantalla (líneas 19 a 20).

Uso de arreglos para analizar los resultados de una encuesta

Nuestro siguiente ejemplo utiliza arreglos para sintetizar los resultados de los datos recolectados en una encuesta:

> *Se pidió a veinte estudiantes que calificaran la calidad de la comida en la cafetería estudiantil en una escala del 1 al 5, en donde 1 significa "pésimo" y 5 significa "excelente". Coloque las 20 respuestas en un arreglo entero y determine la frecuencia de cada calificación.*

Ésta es una típica aplicación de procesamiento de arreglos (vea la figura 7.8). Deseamos resumir el número de respuestas de cada tipo (es decir, del 1 al 5). El arreglo `respuestas` (líneas 9 a 10) es un arreglo entero de 20 elementos que contiene las respuestas de los estudiantes a la encuesta. El último valor en el arreglo es, de manera intencional, una respuesta incorrecta (14). Cuando se ejecuta un programa en Java, se verifica la validez de los subíndices de los elementos del arreglo; todos los subíndices deben ser mayores o iguales a 0 y menores que la longitud del arreglo. Cualquier intento de acceder a un elemento fuera de ese rango de subíndices produce un error en tiempo de ejecución, el cual se conoce como `ArrayIndexOutOfBoundsException`. Al final de esta sección, hablaremos sobre el valor de respuesta inválido, demostraremos la **comprobación de límites** de un arreglo e introduciremos el mecanismo de manejo de excepciones de Java, el cual se puede utilizar para detectar y manejar una excepción `ArrayIndexOutOfBoundsException`.

```
1  // Fig. 7.8: EncuestaEstudiantes.java
2  // Programa de análisis de una encuesta.
3
4  public class EncuestaEstudiantes
5  {
6     public static void main( String[] args )
7     {
8        // arreglo de respuestas de estudiantes (lo más común es que se introduzcan en
            tiempo de ejecución)
9        int[] respuestas = { 1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3,
10       2, 3, 3, 2, 14 };
11       int[] frecuencia = new int[ 6 ]; // arreglo de contadores de frecuencia
12
13       // para cada respuesta, selecciona el elemento de respuestas y usa ese valor
14       // como índice de frecuencia para determinar el elemento a incrementar
15       for ( int respuesta = 0; respuesta < respuestas.length; respuesta++ )
16       {
17          try
18          {
19             ++frecuencia[ respuestas[ respuesta ] ];
20          } // fin de try
21          catch ( ArrayIndexOutOfBoundsException e )
22          {
23             System.out.println( e );
24             System.out.printf( "   respuestas[%d] = %d\n\n",
25                respuesta, respuestas[ respuesta ] );
26          } // fin de catch
27       } // fin de for
```

Fig. 7.8 | Programa de análisis de una encuesta (parte 1 de 2).

```
28
29          System.out.printf( "%s %10s\n", "Calificacion", "Frecuencia" );
30
31          // imprime el valor de cada elemento del arreglo
32          for ( int calificacion = 1; calificacion < frecuencia.length; calificacion++ )
33             System.out.printf( "%6d %10d\n", calificacion, frecuencia[ calificacion ] );
34       } // fin de main
35    } // fin de la clase EncuestaEstudiantes
```

```
java.lang.ArrayIndexOutOfBoundsException: 14
   respuestas[19] = 14

Calificacion     Frecuencia
           1              3
           2              4
           3              8
           4              2
           5              2
```

Fig. 7.8 | Programa de análisis de una encuesta (parte 2 de 2).

El arreglo `frecuencia`

Utilizamos el arreglo de *6 elementos* llamado `frecuencia` (línea 11) para contar el número de ocurrencias de cada respuesta. Cada elemento del arreglo se utiliza como un contador para uno de los posibles tipos de respuestas de la encuesta: `frecuencia[1]` cuenta el número de estudiantes que calificaron la comida como 1, `frecuencia[2]` cuenta el número de estudiantes que calificaron la comida como 2, y así en lo sucesivo.

Síntesis de los resultados

El ciclo `for` (líneas 15 a 27) recibe las respuestas del arreglo `respuestas` una a la vez, e incrementa uno de los contadores `frecuencia[1]` a `frecuencia[5]`; ignoramos `frecuencia[0]` ya que las respuestas de la encuesta se limitan al rango de 1 a 5. La instrucción clave en el ciclo aparece en la línea 19. Esta instrucción incrementa el contador de `frecuencia` apropiado, dependiendo del valor de `respuestas[respuesta]`.

Vamos a recorrer las primeras iteraciones de la instrucción `for`:

- Cuando el contador `respuesta` es 0, el valor de `respuestas[respuesta]` es el valor de `respuestas[0]` (es decir, 1; vea la línea 9). En este caso, `frecuencia[respuestas[respuesta]]` se interpreta como `frecuencia[1]`, y el contador `frecuencia[1]` se incrementa en uno. Para evaluar la expresión, empezamos con el valor en el conjunto más interno de corchetes (`respuesta`, actualmente 0). El valor de `respuesta` se inserta en la expresión y se evalúa el siguiente conjunto de corchetes (`respuestas[respuesta]`). Ese valor se utiliza como subíndice del arreglo `frecuencia`, para determinar cuál contador se incrementará (en este caso, `frecuencia[1]`).

- En la siguiente iteración del ciclo, `respuesta` es 1, por lo que `respuestas[respuesta]` es el valor de `respuestas[1]` (es decir, 2; vea la línea 9). Por lo tanto, `frecuencia[respuestas[respuesta]]` se interpreta como `frecuencia[2]`, lo cual provoca que se incremente `frecuencia[2]`.

- Cuando `respuesta` es 2, `respuestas[respuesta]` es el valor de `respuestas[2]` (es decir, 5; vea la línea 9). Por lo tanto, `frecuencia[respuestas[respuesta]]` se interpreta como `frecuencia[5]`, lo cual provoca que se incremente `frecuencia[5]`, y así en lo sucesivo.

Sin importar el número de respuestas procesadas en la encuesta, el programa sólo requiere un arreglo de seis elementos (en el cual se ignora el elemento cero) para resumir los resultados, ya que todos los valores de las respuestas se encuentran entre 1 y 5, y los valores de subíndice para un arreglo de seis elementos son del 0 al 5. En la salida del programa, la columna `Frecuencia` sintetiza sólo 19 de los 20 valores en el arreglo `respuestas`; el último elemento del arreglo respuestas contiene una respuesta incorrecta que no se contó.

Manejo de excepciones: procesamiento de la respuesta incorrecta

Una **excepción** indica un problema que ocurre mientras se ejecuta un programa. El nombre "excepción" sugiere que el problema no ocurre con frecuencia; si la "regla" es que una instrucción por lo general se ejecuta en forma correcta, entonces el problema representa la "excepción a la regla". El **manejo de excepciones** nos permite crear **programas tolerantes a fallas** que pueden resolver (o manejar) las excepciones. En muchos casos, esto permite a un programa continuar su ejecución como si no hubiera encontrado ningún problema. Por ejemplo, la aplicación `EncuestaEstudiantes` sigue mostrando resultados (figura 7.8), aun cuando una de las respuestas esté fuera del rango. Los problemas más severos podrían evitar que un programa continuara su ejecución normal, en vez de requerir que el programa notifique al usuario sobre el problema y luego termine. Cuando la JVM o un método detecta un problema, como un índice de arreglo inválido o el argumento de un método inválido, **lanza** una excepción; es decir, ocurre una excepción.

La instrucción `try`

Para manejar una excepción, hay que colocar el código que podría lanzar una excepción en una **instrucción `try`** (líneas 17 a 26). El **bloque `try`** (líneas 17 a 20) contiene el código que podría *lanzar* una excepción, y el **bloque `catch`** (líneas 21 a 26) contiene el código que *maneja* la excepción, si ocurre una. Puede tener muchos bloques `catch` para manejar distintos tipos de excepciones que podrían lanzarse en el bloque `try` correspondiente. Cuando la línea 19 incrementa en forma correcta un elemento del arreglo `frecuencia`, se ignoran las líneas 21 a 26. Se requieren las llaves que delimitan los cuerpos de los bloques `try` y `catch`.

Ejecución del bloque `catch`

Cuando el programa encuentra el valor 14 en el arreglo `respuestas`, intenta sumar 1 a `frecuencia[14]`, que está *fuera* de los límites del arreglo; `frecuencia` sólo tiene seis elementos. Como la comprobación de los límites de un arreglo se realiza en tiempo de ejecución, la JVM genera una excepción; en específico, la línea 19 lanza una excepción `ArrayIndexOutOfBoundsException` para notificar al programa sobre este problema. En este punto, el bloque `try` termina y el bloque `catch` comienza a ejecutarse; si usted declaró variables en el bloque `try`, ahora se encuentran fuera de alcance y no son accesibles en el bloque `catch`.

El bloque `catch` declara un tipo (`IndexOutOfRangeException`) y un parámetro de excepción (`e`). El bloque `catch` puede manejar excepciones del tipo especificado. Dentro del bloque `catch`, usted puede usar el identificador del parámetro para interactuar con un objeto excepción atrapada.

Tip para prevenir errores 7.1

Al escribir código para acceder al elemento de un arreglo, hay que asegurar que el subíndice del arreglo siempre sea mayor o igual a 0 y menor que la longitud del arreglo. Esto le ayudará a evitar excepciones del tipo `ArrayIndexOutOfBoundsException` *en su programa.*

Método `toString` *del parámetro de excepción*

Cuando las líneas 21 a 26 *atrapan* la excepción, el programa muestra un mensaje para indicar el problema que ocurrió. La línea 23 realiza una llamada implícita al método `toString` del objeto excepción para obtener el mensaje de error almacenado en el objeto excepción y mostrarlo. Una vez que se muestra el mensaje en este ejemplo, el programa considera que se manejó la excepción y continúa con la siguiente instrucción después de la llave de cierre del bloque `catch`. En este ejemplo se llega al fin de la instrucción `for` (línea 27), por lo que el programa continúa con el incremento de la variable de control en la línea 15. En el capítulo 8 usaremos el manejo de excepciones de nuevo, y en el capítulo 11 (en el sitio Web) presentaremos un análisis más detallado sobre el manejo de excepciones.

7.5 Caso de estudio: simulación para barajar y repartir cartas

Hasta ahora, en los ejemplos en este capítulo hemos utilizado arreglos que contienen elementos de tipos primitivos. En la sección 7.2 vimos que los elementos de un arreglo pueden ser de tipos primitivos o de tipos por referencia. En esta sección utilizaremos la generación de números aleatorios y un arreglo de elementos de tipo por referencia (objetos que representan cartas de juego) para desarrollar una clase que simule los procesos de barajar y repartir cartas. Después podremos utilizar esta clase para implementar aplicaciones que ejecuten juegos específicos de cartas. Los ejercicios al final del capítulo utilizan las clases que desarrollaremos aquí para crear una aplicación simple de póquer.

Primero desarrollaremos la clase `Carta` (figura 7.9), la cual representa una carta de juego que tiene una cara (`"As"`, `"Dos"`, `"Tres"`, ... `"Joto"`, `"Quina"`, `"Rey"`) y un palo (`"Corazones"`, `"Diamantes"`, `"Treboles"`, `"Espadas"`). Después desarrollaremos la clase `PaqueteDeCartas` (figura 7.10), la cual crea un paquete de 52 cartas en las que cada elemento es un objeto `Carta`. Luego construiremos una aplicación de prueba (figura 7.11) para demostrar las capacidades de barajar y repartir cartas de la clase `PaqueteDeCartas`.

La clase `Carta`

La clase `Carta` (figura 7.9) contiene dos variables de instancia `String` (`cara` y `palo`) que se utilizan para almacenar referencias al nombre de la cara y al del palo para una `Carta` específica. El constructor de la clase (líneas 10 a 14) recibe dos objetos `String` que utiliza para inicializar `cara` y `palo`. El método `toString` (líneas 17 a 20) crea un objeto `String` que consiste en la `cara` de la carta, el objeto `String` `"de"` y el `palo` de la carta. El método `toString` de `Carta` puede invocarse en forma explícita para obtener la representación de cadena de un objeto `Carta` (por ejemplo, `"As de Espadas"`). El método `toString` de un objeto se llama en forma *implícita* cuando el objeto se utiliza en donde se espera un objeto `String` (por ejemplo, cuando `printf` imprime en pantalla el objeto como un `String`, usando el especificador de formato `%s`, o cuando el objeto se concatena con un objeto `String` mediante el operador +). Para que ocurra este comportamiento, `toString` debe declararse con el encabezado que se muestra en la figura 7.9.

```
1  // Fig. 7.9: Carta.java
2  // La clase Carta representa una carta de juego.
3
4  public class Carta
5  {
6     private String cara; // cara de la carta ("As", "Dos", ...)
7     private String palo; // palo de la carta ("Corazones", "Diamantes", ...)
8
```

Fig. 7.9 | La clase `Carta` representa una carta de juego (parte 1 de 2).

```
9     // el constructor de dos argumentos inicializa la cara y el palo de la carta
10    public Carta( String caraCarta, String paloCarta )
11    {
12       cara = caraCarta; // inicializa la cara de la carta
13       palo = paloCarta; // inicializa el palo de la carta
14    } // fin del constructor de Carta con dos argumentos
15
16    // devuelve representación String de Carta
17    public String toString()
18    {
19       return cara + " de " + palo;
20    } // fin del método toString
21 } // fin de la clase Carta
```

Fig. 7.9 | La clase `Carta` representa una carta de juego (parte 2 de 2).

La clase `PaqueteDeCartas`

La clase `PaqueteDeCartas` (figura 7.10) declara como variable de instancia un arreglo `Carta` llamado `paquete` (línea 7). Un arreglo de tipo por referencia se declara igual que cualquier otro arreglo. La clase `PaqueteDeCartas` también declara la variable de instancia entera llamada `cartaActual` (línea 8), que representa la siguiente `Carta` a repartir del arreglo `paquete`, y la constante con nombre `NUMERO_DE_CARTAS` (línea 9), que indica el número de objetos `Carta` en el paquete (52).

```
1  // Fig. 7.10: PaqueteDeCartas.java
2  // La clase PaqueteDeCartas representa un paquete de cartas de juego.
3  import java.util.Random;
4
5  public class PaqueteDeCartas
6  {
7     private Carta[] paquete; // arreglo de objetos Carta
8     private int cartaActual; // subíndice de la siguiente Carta a repartir (0 a 51)
9     private static final int NUMERO_DE_CARTAS = 52; // número constante de Cartas
10    // generador de números aleatorios
11    private static final Random numerosAleatorios = new Random();
12
13    // el constructor llena el paquete de Cartas
14    public PaqueteDeCartas()
15    {
16       String[] caras = { "As", "Dos", "Tres", "Cuatro", "Cinco", "Seis",
17          "Siete", "Ocho", "Nueve", "Diez", "Joto", "Quina", "Rey" };
18       String[] palos = { "Corazones", "Diamantes", "Treboles", "Espadas" };
19
20       paquete = new Carta[ NUMERO_DE_CARTAS ]; // crea arreglo de objetos Carta
21       cartaActual = 0; // establece cartaActual para que la primera Carta repartida
                          sea paquete[ 0 ]
22
23       // llena el paquete con objetos Carta
24       for ( int cuenta = 0; cuenta < paquete.length; cuenta++ )
25          paquete[ cuenta ] =
26             new Carta( caras[ cuenta % 13 ], palos[ cuenta / 13 ] );
27    } // fin del constructor de PaqueteDeCartas
28
```

Fig. 7.10 | La clase `PaqueteDeCartas` representa un paquete de cartas de juego (parte 1 de 2).

```
29    // baraja el paquete de Cartas con algoritmo de una pasada
30    public void barajar()
31    {
32       // después de barajar, la repartición debe empezar en paquete[ 0 ] otra vez
33       cartaActual = 0; // reinicializa cartaActual
34
35       // para cada Carta, selecciona otra Carta aleatoria (0 a 51) y las intercambia
36       for ( int primera = 0; primera < paquete.length; primera++ )
37       {
38          // selecciona un número aleatorio entre 0 y 51
39          int segunda =  numerosAleatorios.nextInt( NUMERO_DE_CARTAS );
40
41          // intercambia Carta actual con la Carta seleccionada al azar
42          Carta temp = paquete[ primera ];
43          paquete[ primera ] = paquete[ segunda ];
44          paquete[ segunda ] = temp;
45       } // fin de for
46    } // fin de método barajar
47
48    // reparte una Carta
49    public Carta repartirCarta()
50    {
51       // determina si quedan Cartas por repartir
52       if ( cartaActual < paquete.length )
53          return paquete[ cartaActual++ ]; // devuelve la Carta actual en el arreglo
54       else
55          return null; // devuelve null para indicar que se repartieron todas las Cartas
56    } // fin del método repartirCarta
57 } // fin de la clase PaqueteDeCartas
```

Fig. 7.10 | La clase PaqueteDeCartas representa un paquete de cartas de juego (parte 2 de 2).

Constructor de PaqueteDeCartas

El constructor de la clase crea una instancia del arreglo paquete (línea 20) con un número de elementos igual a NUMERO_DE_CARTAS (52). Los elementos de paquete son null de manera predeterminada, por lo que el constructor utiliza una instrucción for (líneas 24 a 26) para llenar el arreglo paquete con objetos Carta. El ciclo inicializa la variable de control cuenta con 0 e itera mientras cuenta sea menor que paquete.length, lo cual hace que cuenta tome el valor de cada entero del 0 al 51 (los subíndices del arreglo paquete). Cada objeto Carta se instancia y se inicializa con dos objetos String: uno del arreglo caras (que contiene los objetos String del "As" hasta el "Rey") y uno del arreglo palos (que contiene los objetos String "Corazones", "Diamantes", "Treboles" y "Espadas"). El cálculo cuenta %13 siempre produce un valor de 0 a 12 (los 13 subíndices del arreglo caras en las líneas 16 y 17), y el cálculo cuenta / 13 siempre produce un valor de 0 a 3 (los cuatro subíndices del arreglo palos en la línea 18). Cuando se inicializa el arreglo paquete, contiene los objetos Carta con las caras del "As" al "Rey" en orden para cada palo ("Corazones", "Diamantes", "Treboles", "Espadas"). Usamos arreglos de objetos String para representar las caras y palos en este ejemplo. En el ejercicio 7.34 le pediremos que modifique este ejemplo para usar arreglos de constantes de enumeración, que representen las caras y los palos.

El método barajar de PaqueteDeCartas

El método barajar (líneas 30 a 46) baraja los objetos Carta en el paquete. El método itera a través de los 52 objetos Carta (subíndices 0 a 51 del arreglo). Para cada objeto Carta se elige al azar un número entre

0 y 51 para elegir otro objeto Carta. A continuación, el objeto Carta actual y el objeto Carta seleccionado al azar se intercambian en el arreglo. Este intercambio se realiza mediante las tres asignaciones en las líneas 42 a 44. La variable extra temp almacena en forma temporal uno de los dos objetos Carta que se van a intercambiar. El intercambio no se puede realizar sólo con las dos instrucciones

```
paquete[ primera ] = paquete[ segunda ];
paquete[ segunda ] = paquete[ primera ];
```

Si paquete[primera] es el "As" de "Espadas" y paquete[segunda] es la "Quina" de "Corazones", entonces después de la primera asignación, ambos elementos del arreglo contienen la "Quina" de "Corazones" y se pierde el "As" de "Espadas"; es por ello que se necesita la variable extra temp. Una vez que termina el ciclo for, los objetos Carta se ordenan al azar. Sólo se realizan 52 intercambios en una sola pasada del arreglo completo, ¡y el arreglo de objetos Carta se baraja!

[*Nota:* se recomienda utilizar lo que se conoce como algoritmo imparcial para barajar juegos reales de cartas. Dicho algoritmo asegura que todas las posibles secuencias de cartas barajadas tengan la misma probabilidad de ocurrir. Un algoritmo imparcial popular para barajar cartas es el algoritmo Fisher-Yates].

El método `repartirCarta` *de PaqueteDeCartas*

El método repartirCarta (líneas 49 a 56) reparte un objeto Carta en el arreglo. Recuerde que cartaActual indica el subíndice del siguiente objeto Carta que se repartirá (es decir, la Carta en la parte superior del paquete). Por ende, la línea 52 compara cartaActual con la longitud del arreglo paquete. Si el paquete no está vacío (es decir, si cartaActual es menor que 52), la línea 53 regresa el objeto Carta "superior" y postincrementa cartaActual para prepararse para la siguiente llamada a repartirCarta; en caso contrario, se devuelve null. En el capítulo 3 vimos que null representa una "referencia a nada".

Barajar y repartir cartas

La aplicación de la figura 7.11 demuestra la clase PaqueteDeCartas (figura 7.10). La línea 9 crea un objeto PaqueteDeCartas llamado miPaqueteDeCartas. El constructor de PaqueteDeCartas crea el paquete con los 52 objetos Carta, en orden por palo y por cara. La línea 10 invoca el método barajar de miPaqueteDeCartas para reordenar los objetos Carta. Las líneas 13 a 20 reparten los 52 objetos Carta y los imprimen en cuatro columnas, cada una con 13 objetos Carta. La líneas 16 reparten un objeto Carta mediante la invocación al método repartirCarta de miPaqueteDeCartas, y después muestra el objeto Carta justificado a la izquierda en un campo de 19 caracteres. Cuando una Carta se imprime como objeto String, el método toString de Carta (líneas 17 a 20 de la figura 7.9) se invoca en forma implícita. Las líneas 18 a 19 empiezan una nueva línea después de cada cuatro objetos Carta.

```
1  // Fig. 7.11: PruebaPaqueteDeCartas.java
2  // Aplicación para barajar y repartir cartas.
3
4  public class PruebaPaqueteDeCartas
5  {
6     // ejecuta la aplicación
7     public static void main( String[] args )
8     {
9        PaqueteDeCartas miPaqueteDeCartas = new PaqueteDeCartas();
10       miPaqueteDeCartas.barajar(); // coloca las Cartas en orden aleatorio
11
```

Fig. 7.11 | Aplicación para barajar y repartir cartas (parte 1 de 2).

```
12      // imprime las 52 Cartas en el orden en el que se reparten
13      for ( int i = 1; i <= 52; i++ )
14      {
15         // reparte e imprime una Carta
16         System.out.printf( "%-19s", miPaqueteDeCartas.repartirCarta() );
17
18         if ( i % 4 == 0 ) // imprime una nueva línea después de cada cuatro cartas
19               System.out.println();
20         } // fin de for
21      } // fin de main
22   } // fin de la clase PruebaPaqueteDeCartas
```

```
Nueve de Espadas      Joto de Corazones     Quina de Treboles     Siete de Treboles
Seis de Corazones     Seis de Treboles      Joto de Diamantes     Tres de Diamantes
Diez de Diamantes     Cinco de Diamantes    As de Treboles        Rey de Diamantes
Siete de Corazones    Cuatro de Corazones   Cuatro de Espadas     Nueve de Corazones
Dos de Espadas        Quina de Diamantes    Dos de Corazones      Quina de Corazones
As de Corazones       Cuatro de Treboles    Cinco de Espadas      Joto de Treboles
Cinco de Treboles     Siete de Diamantes    As de Espadas         Ocho de Espadas
Nueve de Treboles     Cuatro de Diamantes   Siete de Espadas      Rey de Corazones
Quina de Espadas      Dos de Diamantes      Rey de Treboles       Diez de Corazones
Cinco de Corazones    As de Diamantes       Rey de Espadas        Joto de Espadas
Ocho de Diamantes     Tres de Espadas       Ocho de Treboles      Seis de Diamantes
Nueve de Diamantes    Tres de Treboles      Diez de Treboles      Dos de Treboles
Tres de Corazones     Ocho de Corazones     Diez de Espadas       Seis de Espadas
```

Fig. 7.11 | Aplicación para barajar y repartir cartas (parte 2 de 2).

7.6 Instrucción for mejorada

La **instrucción for mejorada** itera a través de los elementos de un arreglo *sin* utilizar un contador, con lo cual evita la posibilidad de "salirse" del arreglo. En la sección 7.14 le mostraremos cómo usar la instrucción `for` mejorada con las estructuras de datos preconstruidas de la API de Java (conocidas como colecciones). La sintaxis de una instrucción `for` mejorada es:

```
for ( parámetro : nombreArreglo )
   instrucción
```

en donde *parámetro* tiene un tipo y un identificador (por ejemplo, `int numero`), y *nombreArreglo* es el arreglo a través del cual se iterará. El tipo del parámetro debe coincidir con el de los elementos en el arreglo. Como se muestra en el siguiente ejemplo, el identificador representa valores de elementos sucesivos en el arreglo, en iteraciones sucesivas del ciclo.

La figura 7.12 utiliza la instrucción `for` mejorada (líneas 12 y 13) para calcular la suma de los enteros en un arreglo de calificaciones de estudiantes. El parámetro del `for` mejorado es de tipo `int`, ya que arreglo contiene valores `int`: el ciclo selecciona un valor `int` del arreglo durante cada iteración. La instrucción `for` mejorada itera a través de valores sucesivos en el arreglo, uno por uno. El encabezado del `for` mejorado se puede leer como "para cada iteración, asignar el siguiente elemento de `arreglo` a la variable `int numero`, después ejecutar la siguiente instrucción". Por lo tanto, para cada iteración, el identificador `numero` representa un valor `int` en `arreglo`. Las líneas 12 y 13 de la figura 7.5 son equivalentes a la siguiente repetición controlada por un contador que se utiliza en las líneas 12 y 13 de la figura 7.5, para totalizar los enteros en el `arreglo`, excepto que no se puede acceder a `contador` en la instrucción `for` mejorada:

```
for ( int contador = 0; contador < array.length; contador++ )
   total += arreglo[ contador ];
```

```
1  // Fig. 7.12: PruebaForMejorado.java
2  // Uso de la instrucción for mejorada para sumar el total de enteros en un arreglo.
3
4  public class PruebaForMejorado
5  {
6     public static void main( String[] args )
7     {
8        int[] arreglo = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // suma el valor de cada elemento al total
12       for ( int numero : arreglo )
13          total += numero;
14
15       System.out.printf( "Total de elementos del arreglo: %d\n", total );
16    } // fin de main
17 } // fin de la clase PruebaForMejorado
```

```
Total de elementos del arreglo: 849
```

Fig. 7.12 | Uso de la instrucción `for` mejorada para sumar el total de los enteros en un arreglo.

La instrucción `for` mejorada simplifica el código para iterar a través de un arreglo. No obstante, observe que *la instrucción `for` mejorada sólo puede utilizarse para obtener elementos del arreglo; no para modificar los elementos*. Si su programa necesita modificar elementos, use la instrucción `for` tradicional, controlada por contador.

La instrucción `for` mejorada se puede utilizar en lugar de la instrucción `for` controlada por contador, cuando el código que itera a través de un arreglo no requiere acceso al contador que indica el subíndice del elemento actual del arreglo. Por ejemplo, para totalizar los enteros en un arreglo se requiere acceso sólo a los valores de los elementos; el subíndice de cada elemento es irrelevante. No obstante, si un programa debe utilizar un contador por alguna razón que no sea tan sólo iterar a través de un arreglo (por ejemplo, imprimir un número de subíndice al lado del valor de cada elemento del arreglo, como en los primeros ejemplos en este capítulo), use la instrucción `for` controlada por contador.

7.7 Paso de arreglos a los métodos

Esta sección demuestra cómo pasar arreglos y elementos individuales de un arreglo como argumentos para los métodos. Para pasar un argumento tipo arreglo a un método, se especifica el nombre del arreglo sin corchetes. Por ejemplo, si el arreglo `temperaturasPorHora` se declara como

```
double[] temperaturasPorHora = new double[ 24 ];
```

entonces la llamada al método

```
modificarArreglo( temperaturasPorHora );
```

pasa la referencia del arreglo `temperaturasPorHora` al método `modificarArreglo`. Todo objeto arreglo "conoce" su propia longitud (a través de su campo `length`). Por ende, cuando pasamos a un método la referencia a un objeto arreglo, no necesitamos pasar la longitud del arreglo como un argumento adicional.

Para que un método reciba una referencia a un arreglo a través de una llamada a un método, la lista de parámetros del método debe especificar un parámetro tipo arreglo. Por ejemplo, el encabezado para el método `modificarArreglo` podría escribirse así:

```
void modificarArreglo( double[] b )
```

lo cual indica que `modificarArreglo` recibe la referencia de un arreglo `double` en el parámetro `b`. La llamada a este método pasa la referencia al arreglo `temperaturaPorHoras`, de manera que cuando el método llamado utiliza la variable `b` tipo arreglo, hace *referencia* al mismo objeto arreglo como `temperaturasPorHora` en el método que hizo la llamada.

Cuando un argumento para un método es todo un arreglo, o un elemento individual de un arreglo de un tipo por referencia, el método llamado recibe una *copia* de la referencia. Sin embargo, cuando un argumento para un método es un elemento individual de un arreglo de un tipo primitivo, el método llamado recibe una copia del *valor* del elemento. Dichos valores primitivos se conocen como **escalares** o **cantidades escalares**. Para pasar un elemento individual de un arreglo a un método, use el nombre indexado del elemento del arreglo como argumento en la llamada al método.

La figura 7.13 demuestra la diferencia entre pasar a un método todo un arreglo y pasar un elemento de un arreglo de tipo primitivo. Observe que `main` invoca de manera directa a los métodos `static` `modificarArreglo` (línea 19) y `modificarElemento` (línea 30). En la sección 6.4 vimos que un método `static` de una clase puede invocar de manera directa a otros métodos `static` de la misma clase.

```
1  // Fig. 7.13: PasoArreglo.java
2  // Paso de arreglos y elementos individuales de un arreglo a los métodos.
3
4  public class PasoArreglo
5  {
6     // main crea el arreglo y llama a modificarArreglo y a modificarElemento
7     public static void main( String[] args )
8     {
9        int[] arreglo = { 1, 2, 3, 4, 5 };
10
11       System.out.println(
12          "Efectos de pasar una referencia a un arreglo completo:\n" +
13          "Los valores del arreglo original son:" );
14
15       // imprime los elementos originales del arreglo
16       for ( int valor : arreglo )
17          System.out.printf( "   %d", valor );
18
19       modificarArreglo( arreglo ); // pasa la referencia al arreglo
20       System.out.println( "\n\nLos valores del arreglo modificado son:" );
21
22       // imprime los elementos modificados del arreglo
23       for ( int valor : arreglo )
24          System.out.printf( "   %d", valor );
25
26       System.out.printf(
27          "\n\nEfectos de pasar el valor de un elemento del arreglo:\n" +
28          "arreglo[3] antes de modificarElemento: %d\n", arreglo[ 3 ] );
29
30       modificarElemento( arreglo[ 3 ] ); // intento por modificar arreglo[ 3 ]
31       System.out.printf(
32          "arreglo[3] despues de modificarElemento: %d\n", arreglo[ 3 ] );
33    } // fin de main
34
```

Fig. 7.13 | Paso de arreglos y elementos individuales de un arreglo a los métodos (parte 1 de 2).

```
35    // multiplica cada elemento de un arreglo por 2
36    public static void modificarArreglo( int[] arreglo2 )
37    {
38       for ( int contador = 0; contador < arreglo2.length; contador++ )
39          arreglo2[ contador ] *= 2;
40    } // fin del método modificarArreglo
41
42    // multiplica el argumento por 2
43    public static void modificarElemento( int elemento )
44    {
45       elemento *= 2;
46       System.out.printf(
47          "Valor del elemento en modificarElemento: %d\n", elemento );
48    } // fin del método modificarElemento
49 } // fin de la clase PasoArreglo
```

```
Efectos de pasar una referencia a un arreglo completo:
Los valores del arreglo original son:
   1   2   3   4   5

Los valores del arreglo modificado son:
   2   4   6   8   10

Efectos de pasar el valor de un elemento del arreglo:
arreglo[3] antes de modificarElemento: 8
Valor del elemento en modificarElemento: 16
arreglo[3] despues de modificarElemento: 8
```

Fig. 7.13 | Paso de arreglos y elementos individuales de un arreglo a los métodos (parte 2 de 2).

La instrucción for mejorada en las líneas 16 y 17 imprime en pantalla los cinco elementos `int` de `arreglo`. La línea 19 invoca al método `modificarArreglo` y le pasa `arreglo` como argumento. El método `modificarArreglo` (líneas 36 a 40) recibe una copia de la referencia a `arreglo` y utiliza esta referencia para multiplicar cada uno de los elementos de `arreglo` por 2. Para demostrar que se modificaron los elementos de `arreglo`, en las líneas 23 y 24 se imprimen en pantalla los cinco elementos de `arreglo` otra vez. Como se muestra en la salida, el método `modificarArreglo` duplicó el valor de cada elemento. No pudimos usar la instrucción for mejorada en las líneas 38 y 39, ya que estamos modificando los elementos del arreglo.

La figura 7.13 demuestra a continuación que, cuando se pasa una copia de un elemento individual de un arreglo de tipo primitivo a un método, si se modifica la *copia* en el método que se llamó, el valor original de ese elemento *no* se ve afectado en el arreglo del método que hizo la llamada. Las líneas 26 a 28 imprimen en pantalla el valor de `arreglo[3]` *antes* de invocar al método `modificarElemento`. Recuerde que el valor de este elemento ahora es 8, después de haberlo modificado en la llamada a `modificarArreglo`. La línea 30 llama al método `modificarElemento` y le pasa `arreglo[3]` como argumento. Recuerde que `arreglo[3]` es en realidad un valor `int` (8) en `arreglo`. Por lo tanto, el programa pasa una copia del valor de `arreglo[3]`. El método `modificarElemento` (líneas 43 a 48) multiplica el valor recibido como argumento por 2, almacena el resultado en su parámetro `elemento` y después imprime en pantalla el valor de `elemento` (16). Como los parámetros de los métodos, al igual que las variables locales, dejan de existir cuando el método en el que se declaran termina su ejecución, el parámetro `elemento` del método se destruye cuando termina el método `modificarElemento`. Cuando el programa devuelve el control a `main`, las líneas 31 y 32 imprimen en pantalla el valor de `arreglo[3]` que *no se modificó* (es decir, 8).

Notas acerca del paso de argumentos a los métodos

El ejemplo anterior demostró la forma en que se pasan los arreglos y los elementos de arreglos de tipos primitivos a los métodos. Ahora veremos con más detalle la forma en que se pasan los argumentos a los métodos en general. En muchos lenguajes de programación, dos formas de pasar argumentos en las llamadas a métodos son el **paso por valor** y el **paso por referencia** (también conocidas como **llamada por valor** y **llamada por referencia**). Cuando se pasa un argumento por valor, se pasa una copia del *valor* del argumento al método que se llamó. Este método trabaja exclusivamente con la copia. Las modificaciones a la copia del método que se llamó *no* afectan el valor de la variable original en el método que hizo la llamada.

Cuando se pasa un argumento por referencia, el método que se llamó puede acceder de manera directa al valor del argumento en el método que hizo la llamada, y puede modificar esos datos si es necesario. El paso por referencia mejora el rendimiento, al eliminar la necesidad de copiar cantidades de datos posiblemente extensas.

A diferencia de otros lenguajes, Java *no* permite a los programadores elegir el paso por valor o el paso por referencia; *todos los argumentos se pasan por valor*. Una llamada a un método puede pasar dos tipos de valores: copias de valores primitivos (como valores de tipo `int` y `double`) y copias de referencias a objetos. Los objetos en sí no pueden pasarse a los métodos. Cuando un método modifica un parámetro de tipo primitivo, las modificaciones a ese parámetro no tienen efecto en el valor original del argumento en el método que hizo la llamada. Por ejemplo, cuando la línea 30 en `main` de la figura 7.13 pasa `arreglo[3]` al método `modificarElemento`, la instrucción en la línea 45 que duplica el valor del parámetro elemento *no* tiene efecto sobre el valor de `arreglo[3]` en `main`. Esto también se aplica para los parámetros de tipo por referencia. Si usted modifica un parámetro de tipo por referencia de modo que haga referencia a otro objeto, sólo el parámetro hace referencia al nuevo objeto; la referencia almacenada en la variable del método que hizo la llamada sigue haciendo referencia al objeto original.

Aunque la referencia a un objeto se pasa por valor, un método puede de todas formas interactuar con el objeto al que se hace referencia, llamando a sus métodos `public` mediante el uso de la copia de la referencia al objeto. Como la referencia almacenada en el parámetro es una copia de la referencia que se pasó como argumento, el parámetro en el método que se llamó y el argumento en el método que hizo la llamada hacen referencia al mismo objeto en la memoria. Por ejemplo, en la figura 7.13, tanto el parámetro `arreglo2` en el método `modificarArreglo` como la variable arreglo en `main` hacen referencia al *mismo* objeto en la memoria. Cualquier modificación que se realice usando el parámetro `arreglo2` se lleva a cabo en el mismo objeto al que `arreglo` hace referencia en el método que hizo la llamada. En la figura 7.13, las modificaciones realizadas en `modificarArreglo` en las que se utiliza `arreglo2`, afectan al contenido del objeto arreglo al que hace referencia arreglo en `main`. De esta forma, con una referencia a un objeto, el método que se llamó puede manipular de manera directa el objeto del método que hizo la llamada.

Tip de rendimiento 7.1

Pasar arreglos por referencia tiene sentido por cuestiones de rendimiento. Si los arreglos se pasaran por valor, se pasaría una copia de cada elemento. En los arreglos grandes que se pasan con frecuencia, esto desperdiciaría tiempo y consumiría una cantidad considerable de almacenamiento para las copias de los arreglos.

7.8 Caso de estudio: la clase `LibroCalificaciones` que usa un arreglo para almacenar las calificaciones

Las versiones anteriores de la clase `LibroCalificaciones` procesan un conjunto de calificaciones introducidas por el usuario, pero no mantienen los valores de las calificaciones individuales en variables de instancia de la clase. Por ende, los cálculos repetidos requieren que el usuario vuelva a introducir las mismas calificaciones. Una manera de resolver este problema sería almacenar cada calificación introducida por el usuario en una instancia individual de la clase. Por ejemplo, podríamos crear las variables de instancia `calificacion1`, `calificacion2`, ..., `calificacion10` en la clase `LibroCalificaciones` para almacenar 10 calificaciones de estudiantes. No obstante, el código para totalizar las calificaciones y deter-

minar el promedio de la clase sería voluminoso, además de que la clase no podría procesar más de 10 calificaciones a la vez. Para resolver este problema, almacenamos las calificaciones en un arreglo.

Almacenar las calificaciones de los estudiantes en un arreglo en la clase `LibroCalificaciones`

La clase `LibroCalificaciones` (figura 7.14) utiliza un arreglo de valores `int` para almacenar las calificaciones de varios estudiantes en un solo examen. Esto elimina la necesidad de introducir varias veces el mismo conjunto de calificaciones. El arreglo `calificaciones` se declara como una variable de instancia (línea 7), por lo que cada objeto `LibroCalificaciones` mantiene su propio conjunto de calificaciones. El constructor de la clase (líneas 10 a 14) tiene dos parámetros: el nombre del curso y un arreglo de calificaciones. Cuando una aplicación (por ejemplo, la clase `PruebaLibroCalificaciones` en la figura 7.15) crea un objeto `LibroCalificaciones`, la aplicación pasa un arreglo `int` existente al constructor, el cual asigna la referencia del arreglo a la variable de instancia `calificaciones` (línea 13). El tamaño del arreglo `calificaciones` se determina mediante la longitud del arreglo que se pasa al constructor. Por ende, un objeto `LibroCalificaciones` puede procesar un número de calificaciones variable. Los valores de las calificaciones en el arreglo que se pasa podría introducirlos un usuario desde el teclado, o leerse desde un archivo en el disco (como veremos en el capítulo 17, en el sitio Web). En nuestra aplicación de prueba, inicializamos un arreglo con valores de calificaciones (figura 7.15, línea 10). Una vez que las calificaciones se almacenan en una variable de instancia llamada `calificaciones` de la clase `LibroCalificaciones`, todos los métodos de la clase pueden acceder a los elementos de `calificaciones` *según sea necesario*, para realizar varios cálculos.

El método `procesarCalificaciones` (líneas 37 a 51) contiene una serie de llamadas a métodos que produce un reporte en el que se resumen las calificaciones. La línea 40 llama al método `imprimirCalificaciones` para imprimir el contenido del arreglo `calificaciones`. Las líneas 134 a 136 en el método `imprimirCalificaciones` utilizan una instrucción `for` para imprimir las calificaciones de los estudiantes. En este caso se *debe* utilizar una instrucción `for` controlada por contador, ya que las líneas 135 y 136 utilizan el valor de la variable contador estudiante para imprimir cada calificación enseguida de un número de estudiante específico (vea la figura 7.15). Aunque los subíndices de los arreglos empiezan en 0, lo común es que el profesor enumere a los estudiantes empezando desde 1. Por ende, las líneas 135 y 136 imprimen `estudiante + 1` como el número de estudiante para producir las etiquetas `"Estudiante 1: "`, `"Estudiante 2: "`, y así en lo sucesivo.

```
1  // Fig. 7.14: LibroCalificaciones.java
2  // Libro de calificaciones que utiliza un arreglo para almacenar las calificaciones
   de una prueba.
3
4  public class LibroCalificaciones
5  {
6     private String nombreDelCurso; // nombre del curso que representa este
                                     LibroCalificaciones
7     private int[] calificaciones; // arreglo de calificaciones de estudiantes
8
9     // el constructor de dos argumentos inicializa nombreDelCurso y el arreglo
      calificaciones
10    public LibroCalificaciones( String nombre, int[] arregloCalif )
11    {
12       nombreDelCurso = nombre; // inicializa nombreDelCurso
13       calificaciones = arregloCalif; // almacena las calificaciones
14    } // fin del constructor de LibroCalificaciones con dos argumentos
15
16    // método para establecer el nombre del curso
17    public void establecerNombreDelCurso( String nombre )
18    {
19       nombreDelCurso = nombre; // almacena el nombre del curso
20    } // fin del método establecerNombreDelCurso
```

Fig. 7.14 | La clase `LibroCalificaciones` que usa un arreglo para almacenar las calificaciones de una prueba (parte 1 de 4).

```
21
22      // método para obtener el nombre del curso
23      public String obtenerNombreDelCurso()
24      {
25         return nombreDelCurso;
26      } // fin del método obtenerNombreDelCurso
27
28      // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
29      public void mostrarMensaje()
30      {
31         // obtenerNombreDelCurso obtiene el nombre del curso
32         System.out.printf( "Bienvenido al libro de calificaciones para\n%s!\n\n",
33            obtenerNombreDelCurso() );
34      } // fin del método mostrarMensaje
35
36      // realiza varias operaciones sobre los datos
37      public void procesarCalificaciones()
38      {
39         // imprime el arreglo de calificaciones
40         imprimirCalificaciones();
41
42         // llama al método obtenerPromedio para calcular la calificación promedio
43         System.out.printf( "\nEl promedio de la clase es %.2f\n", obtenerPromedio() );
44
45         // llama a los métodos obtenerMinima y obtenerMaxima
46         System.out.printf( "La calificacion mas baja es %d\nLa calificacion mas alta
           es %d\n\n",
47            obtenerMinima(), obtenerMaxima() );
48
49         // llama a imprimirGraficoBarras para imprimir el gráfico de distribución de
              calificaciones
50         imprimirGraficoBarras();
51      } // fin del método procesarCalificaciones
52
53      // busca la calificación más baja
54      public int obtenerMinima()
55      {
56         int califBaja = calificaciones[ 0 ]; // asume que calificaciones[ 0 ] es la
                                                 más baja
57
58         // itera a través del arreglo de calificaciones
59         for ( int calificacion : calificaciones )
60         {
61            // si calificación es menor que califBaja, se asigna a califBaja
62            if ( calificacion < califBaja )
63               califBaja = calificacion; // nueva calificación más baja
64         } // fin de for
65
66         return califBaja; // devuelve la calificación más baja
67      } // fin del método obtenerMinima
68
69      // busca la calificación más alta
70      public int obtenerMaxima()
71      {
72         int califAlta = calificaciones[ 0 ]; // asume que calificaciones[ 0 ] es la
                                                 más alta
73
```

Fig. 7.14 | La clase `LibroCalificaciones` que usa un arreglo para almacenar las calificaciones de una prueba (parte 2 de 4).

```
74          // itera a través del arreglo de calificaciones
75          for ( int calificacion : calificaciones )
76          {
77             // si calificacion es mayor que califAlta, se asigna a califAlta
78             if ( calificacion > califAlta )
79                califAlta = calificacion; // nueva calificación más alta
80          } // fin de for
81
82          return califAlta; // devuelve la calificación más alta
83       } // fin del método obtenerMaxima
84
85       // determina la calificación promedio de la prueba
86       public double obtenerPromedio()
87       {
88          int total = 0; // inicializa el total
89
90          // suma las calificaciones para un estudiante
91          for ( int calificacion : calificaciones )
92             total += calificacion;
93
94          // devuelve el promedio de las calificaciones
95          return (double) total / calificaciones.length;
96       } // fin del método obtenerPromedio
97
98       // imprime grafico de barras que muestra la distribución de las calificaciones
99       public void imprimirGraficoBarras()
100      {
101         System.out.println( "Distribucion de calificaciones:" );
102
103         // almacena la frecuencia de las calificaciones en cada rango de 10 calificaciones
104         int[] frecuencia = new int[ 11 ];
105
106         // para cada calificación, incrementa la frecuencia apropiada
107         for ( int calificacion : calificaciones )
108            ++frecuencia[ calificacion / 10 ];
109
110         // para cada frecuencia de calificación, imprime una barra en el gráfico
111         for ( int cuenta = 0; cuenta < frecuencia.length; cuenta++ )
112         {
113            // imprime etiqueta de barra ( "00-09: ", ..., "90-99: ", "100: " )
114            if ( cuenta == 10 )
115               System.out.printf( "%5d: ", 100 );
116            else
117               System.out.printf( "%02d-%02d: ",
118                  cuenta * 10, cuenta * 10 + 9  );
119
120            // imprime barra de asteriscos
121            for ( int estrellas = 0; estrellas < frecuencia[ cuenta ]; estrellas++ )
122               System.out.print( "*" );
123
124            System.out.println(); // inicia una nueva línea de salida
125         } // fin de for externo
126      } // fin del método imprimirGraficoBarras
```

Fig. 7.14 | La clase LibroCalificaciones que usa un arreglo para almacenar las calificaciones de una prueba (parte 3 de 4).

```
127
128    // imprime el contenido del arreglo de calificaciones
129    public void imprimirCalificaciones()
130    {
131       System.out.println( "Las calificaciones son:\n" );
132
133       // imprime la calificación de cada estudiante
134       for ( int estudiante = 0; estudiante < calificaciones.length; estudiante++ )
135          System.out.printf( "Estudiante %2d: %3d\n",
136             estudiante + 1, calificaciones[ estudiante ] );
137    } // fin del método imprimirCalificaciones
138 } // fin de la clase LibroCalificaciones
```

Fig. 7.14 | La clase `LibroCalificaciones` que usa un arreglo para almacenar las calificaciones de una prueba (parte 4 de 4).

A continuación, el método `procesarCalificaciones` llama al método `obtenerPromedio` (línea 43) para obtener el promedio de las calificaciones en el arreglo. El método `obtenerPromedio` (líneas 86 a 96) utiliza una instrucción `for` mejorada para totalizar los valores en el arreglo calificaciones antes de calcular el promedio. El parámetro en el encabezado de la instrucción `for` mejorada (por ejemplo, `int calificacion`) indica que para cada iteración, la variable `int calificacion` recibe un valor en el arreglo `calificaciones`. El cálculo del promedio en la línea 95 utiliza `calificaciones.length` para determinar el número de calificaciones que se van a promediar.

Las líneas 46 y 47 en el método `procesarCalificaciones` llaman a los métodos `obtenerMinima` y `obtenerMaxima` para determinar las calificaciones más baja y más alta de cualquier estudiante en el examen, en forma respectiva. Cada uno de estos métodos utiliza una instrucción `for` mejorada para iterar a través del arreglo `calificaciones`. Las líneas 59 a 64 en el método `obtenerMinima` iteran a través del arreglo. Las líneas 62 y 63 comparan cada calificación con `califBaja`; si una calificación es menor que `califBaja`, a `califBaja` se le asigna esa calificación. Cuando la línea 66 se ejecuta, `califBaja` contiene la calificación más baja en el arreglo. El método `obtenerMaxima` (líneas 70 a 83) funciona de manera similar al método `obtenerMinima`.

Por último, la línea 50 en el método `procesarCalificaciones` llama al método `imprimirGraficoBarras` para imprimir un gráfico de distribución de los datos de las calificaciones, mediante el uso de una técnica similar a la de la figura 7.6. En ese ejemplo, calculamos en forma manual el número de calificaciones en cada categoría (es decir, de 0 a 9, de 10 a 19, ..., de 90 a 99 y 100) con sólo analizar un conjunto de calificaciones. En este ejemplo, las líneas 107 y 108 utilizan una técnica similar a la de las figuras 7.7 y 7.8 para calcular la frecuencia de las calificaciones en cada categoría. La línea 104 declara y crea el arreglo `frecuencia` de 11 valores `int` para almacenar la frecuencia de las calificaciones en cada categoría de éstas. Para cada `calificacion` en el arreglo `calificaciones`, las líneas 107 y 108 incrementan el elemento apropiado del arreglo `frecuencia`. Para determinar qué elemento se debe incrementar, la línea 108 divide la `calificacion` actual entre 10, mediante la división entera. Por ejemplo, si `calificacion` es 85, la línea 108 incrementa `frecuencia[8]` para actualizar la cuenta de calificaciones en el rango 80-89. Las líneas 111 a 125 imprimen a continuación el gráfico de barras (vea la figura 7.15), con base en los valores en el arreglo `frecuencia`. Al igual que las líneas 23 y 24 de la figura 7.6, las líneas 121 y 122 de la figura 7.14 utilizan un valor en el arreglo `frecuencia` para determinar el número de asteriscos a imprimir en cada barra.

La clase `PruebaLibroCalificaciones` *para demostrar la clase* `LibroCalificaciones`

La aplicación de la figura 7.15 crea un objeto de la clase `LibroCalificaciones` (figura 7.14) mediante el uso del arreglo `int arregloCalif` (que se declara y se inicializa en la línea 10). Las líneas 12 y 13 pasan el nombre de un curso y `arregloCalif` al constructor de `LibroCalificaciones`. La línea 14 imprime un mensaje de bienvenida, y la línea 15 invoca el método `procesarCalificaciones` del objeto `LibroCalificaciones`. La salida muestra el resumen de las 10 calificaciones en `miLibroCalificaciones`.

Observación de ingeniería de software 7.1

Un arnés de prueba (o aplicación de prueba) es responsable de crear un objeto de la clase que se probará así como de proporcionale datos. Los cuales podrían provenir de cualquiera de varias fuentes. Los datos de prueba pueden colocarse directamente en un arreglo con un inicializador de arreglos, pueden provenir del usuario mediante el teclado, de un archivo (como veremos en el capítulo 17, en el sitio Web del libro) o de una red (capítulo 27, en inglés, también en el sitio Web del libro). Después de pasar estos datos al constructor de la clase para instanciar el objeto, este arnés de prueba debe llamar al objeto para probar sus métodos y manipular sus datos. La recopilación de datos en el arnés de prueba de esta forma permite a la clase manipular datos provenientes de varias fuentes.

```
1  // Fig. 7.15: PruebaLibroCalificaciones.java
2  // PruebaLibroCalificaciones crea un objeto LibroCalificaciones, usando un arreglo
   de calificaciones,
3  // y después invoca al método procesarCalificaciones para analizarlas.
4  public class PruebaLibroCalificaciones
5  {
6     // el método main comienza la ejecución del programa
7     public static void main( String[] args )
8     {
9        // arreglo de calificaciones de estudiantes
10       int[] arregloCalif = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11
12       LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
13          "CS101 Introduccion a la programacion en Java", arregloCalif );
14       miLibroCalificaciones.mostrarMensaje();
15       miLibroCalificaciones.procesarCalificaciones();
16    } // fin de main
17 } // fin de la clase PruebaLibroCalificaciones
```

```
Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en Java!

Las calificaciones son:

Estudiante  1:  87
Estudiante  2:  68
Estudiante  3:  94
Estudiante  4: 100
Estudiante  5:  83
Estudiante  6:  78
Estudiante  7:  85
Estudiante  8:  91
Estudiante  9:  76
Estudiante 10:  87

El promedio de la clase es 84.90
La calificacion mas baja es 68
La calificacion mas alta es 100
```

Fig. 7.15 | PruebaLibroCalificaciones crea un objeto LibroCalificaciones mediante un arreglo de calificaciones, y después invoca al método procesarCalificaciones para analizarlas (parte 1 de 2).

```
Distribucion de calificaciones:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 7.15 | `PruebaLibroCalificaciones` crea un objeto `LibroCalificaciones` mediante un arreglo de calificaciones, y después invoca al método `procesarCalificaciones` para analizarlas (parte 2 de 2).

7.9 Arreglos multidimensionales

Los arreglos multidimensionales de dos dimensiones se utilizan con frecuencia para representar *tablas* de valores, las cuales consisten en información ordenada en *filas* y *columnas*. Para identificar un elemento específico de una tabla, debemos especificar dos subíndices. *Por convención*, el primero identifica la fila del elemento y el segundo su columna. Los arreglos que requieren dos subíndices para identificar un elemento específico se llaman **arreglos bidimensionales** (los arreglos multidimensionales pueden tener más de dos dimensiones). Java no soporta los arreglos multidimensionales directamente, pero permite al programador especificar arreglos unidimensionales, cuyos elementos sean también arreglos unidimensionales, con lo cual se obtiene el mismo efecto. La figura 7.16 ilustra un arreglo bidimensional a, que contiene tres filas y cuatro columnas (es decir, un arreglo de tres por cuatro). En general, a un arreglo con *m* filas y *n* columnas se le llama **arreglo de *m* por *n***.

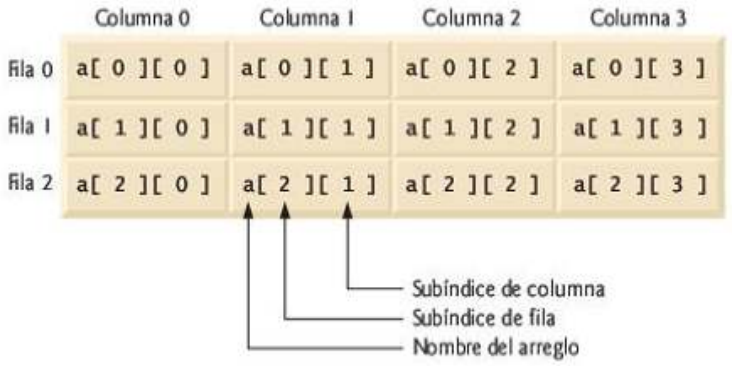


Fig. 7.16 | Arreglo bidimensional con tres filas y cuatro columnas.

Cada elemento en el arreglo a se identifica en la figura 7.16 mediante una *expresión de acceso a un arreglo* de la forma a [*fila*] [*columna*]; a es el nombre del arreglo, *fila* y *columna* son los subíndices que identifican en forma única a cada elemento en el arreglo a por número de fila y columna. Los nombres de los elementos en la *fila* 0 tienen todos un primer subíndice de 0, y los nombres de los elementos en la *columna* 3 tienen un segundo subíndice de 3.

Arreglos de arreglos unidimensionales

Al igual que los arreglos unidimensionales, multidimensionales pueden inicializarse mediante inicializadores de arreglos en las declaraciones. Un arreglo bidimensional b con dos filas y dos columnas podría declararse e inicializarse con **inicializadores de arreglos anidados**, como se muestra a continuación:

```
int[][] b = { { 1, 2 }, { 3, 4 } };
```

Los valores iniciales se agrupan por fila entre llaves. Así, 1 y 2 inicializan a b[0][0] y b[0][1]; 3 y 4 inicializan a b[1][0] y b[1][1], respectivamente. El compilador cuenta el número de inicializadores de arreglos anidados (representados por conjuntos de llaves dentro de las llaves externas) para determinar el número de filas en el arreglo b. El compilador cuenta los valores inicializadores en el inicializador de arreglos anidado de una fila, para determinar el número de columnas en esa fila. Como veremos en unos momentos, esto significa que *las filas pueden tener distintas longitudes*.

Los arreglos multidimensionales se mantienen como arreglos de arreglos unidimensionales. Por lo tanto, el arreglo b en la declaración anterior en realidad está compuesto de dos arreglos unidimensionales separados: uno que contiene los valores en la primera lista inicializadora anidada { 1, 2 } y otro que contiene los valores en la segunda lista inicializadora anidada { 3, 4 }. Así, el arreglo b en sí es un arreglo de dos elementos, cada uno de los cuales es un arreglo unidimensional de valores int.

Arreglos bidimensionales con filas de distintas longitudes

La forma en que se representan los arreglos multidimensionales los hace bastante flexibles. De hecho, las longitudes de las filas en el arreglo b *no* tienen que ser iguales. Por ejemplo,

```
int[][] b = { { 1, 2 }, { 3, 4, 5 } };
```

crea el arreglo entero b con dos elementos (los cuales se determinan según el número de inicializadores de arreglos anidados) que representan las filas del arreglo bidimensional. Cada elemento de b es una referencia a un arreglo unidimensional de variables int. El arreglo int de la fila 0 es un arreglo unidimensional con dos elementos (1 y 2), y el arreglo int de la fila 1 es un arreglo unidimensional con tres elementos (3, 4 y 5).

Creación de arreglos bidimensionales mediante expresiones de creación de arreglos

Un arreglo multidimensional con el mismo número de columnas en cada fila puede formarse mediante una expresión de creación de arreglos. Por ejemplo, en las siguientes líneas se declara el arreglo b y se le asigna una referencia a un arreglo de tres por cuatro:

```
int[][] b = new int[ 3 ][ 4 ];
```

En este caso, utilizamos los valores literales 3 y 4 para especificar el número de filas y columnas, respectivamente, pero esto no es obligatorio. Los programas también pueden utilizar variables para especificar las dimensiones de los arreglos, ya que *new crea arreglos en tiempo de ejecución, no en tiempo de compilación*. Al igual que con los arreglos unidimensionales, los elementos de un arreglo multidimensional se inicializan cuando se crea el objeto arreglo.

Un arreglo multidimensional, en el que cada fila tiene un número distinto de columnas, puede crearse de la siguiente manera:

```
int[][] b = new int[ 2 ][ ];   // crea 2 filas
b[ 0 ] = new int[ 5 ]; // crea 5 columnas para la fila 0
b[ 1 ] = new int[ 3 ]; // crea 3 columnas para la fila 1
```

Estas instrucciones crean un arreglo bidimensional con dos filas. La fila 0 tiene cinco columnas y la fila 1 tiene tres.

Ejemplo de arreglos bidimensionales: cómo mostrar los valores de los elementos

La figura 7.17 demuestra cómo inicializar arreglos bidimensionales con inicializadores de arreglos, y cómo utilizar ciclos `for` anidados para **recorrer** los arreglos (es decir, manipular cada uno de los elementos de cada arreglo). El método `main` de la clase `InicArreglo` declara dos arreglos. En la declaración de `arreglo1` (línea 9) se utilizan inicializadores de arreglos anidados de la *misma* longitud para inicializar la primera fila del arreglo con los valores 1, 2 y 3, y la segunda fila con los valores 4, 5 y 6. En la declaración de `arreglo2` (línea 10) se utilizan inicializadores anidados de *distintas* longitudes. En este caso, la primera fila se inicializa para tener dos elementos con los valores 1 y 2, respectivamente. La segunda fila se inicializa para tener un elemento con el valor 3. La tercera fila se inicializa para tener tres elementos con los valores 4, 5 y 6.

```
1  // Fig. 7.17: InicArreglo.java
2  // Inicialización de arreglos bidimensionales.
3
4  public class InicArreglo
5  {
6     // crea e imprime arreglos bidimensionales
7     public static void main( String[] args )
8     {
9        int[][] arreglo1 = { { 1, 2, 3 }, { 4, 5, 6 } };
10       int[][] arreglo2 = { { 1, 2 }, { 3 }, { 4, 5, 6 } };
11
12       System.out.println( "Los valores en arreglo1 por filas son" );
13       imprimirArreglo( arreglo1 ); // muestra arreglo1 por filas
14
15       System.out.println( "\nLos valores en arreglo2 por filas son" );
16       imprimirArreglo( arreglo2 ); // muestra arreglo2 por filas
17    } // fin de main
18
19    // imprime filas y columnas de un arreglo bidimensional
20    public static void imprimirArreglo( int[][] arreglo )
21    {
22       // itera a través de las filas del arreglo
23       for ( int fila = 0; fila < arreglo.length; fila++ )
24       {
25          // itera a través de las columnas de la fila actual
26          for ( int columna = 0; columna < arreglo[ fila ].length; columna++ )
27             System.out.printf( "%d  ", arreglo[ fila ][ columna ] );
28
29          System.out.println(); // inicia nueva línea de salida
30       } // fin de for externo
31    } // fin del método imprimirArreglo
32 } // fin de la clase InicArreglo
```

```
Los valores en arreglo1 por filas son
1  2  3
4  5  6

Los valores en arreglo2 por filas son
1  2
3
4  5  6
```

Fig. 7.17 | Inicialización de dos arreglos bidimensionales.

Las líneas 13 y 16 llaman al método imprimirArreglo (líneas 20 a 31) para imprimir los elementos de arreglo1 y arreglo2, respectivamente. El parámetro del método imprimirArreglo (int[][] arreglo) indica que el método recibe un arreglo bidimensional. La instrucción for (líneas 23 a 30) imprime las filas de un arreglo bidimensional. En la condición de continuación de ciclo de la instrucción for exterior, la expresión arreglo.length determina el número de filas en el arreglo. En la expresión for interior, la expresión arreglo[fila].length determina el número de columnas en la fila actual del arreglo. Esta condición permite al ciclo determinar el número exacto de columnas en cada fila.

Manipulaciones comunes en arreglos multidimensionales, realizadas mediante instrucciones for

En muchas manipulaciones comunes en arreglos se utilizan instrucciones for. Como ejemplo, la siguiente instrucción for asigna a todos los elementos en la fila 2 del arreglo a, en la figura 7.16, el valor de cero:

```
for ( int columna = 0; columna < a[ 2 ].length; columna++)
   a[ 2 ][ columna ] = 0;
```

Especificamos la fila 2; por lo tanto, sabemos que el primer subíndice siempre será 2 (0 es la primera fila y 1 es la segunda). Este ciclo for varía solamente el segundo índice (es decir, el índice de la columna). Si la fila 2 del arreglo a contiene cuatro elementos, entonces la instrucción for anterior es equivalente a las siguientes instrucciones de asignación:

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

La siguiente instrucción for anidada suma el total de los valores de todos los elementos del arreglo a:

```
int total = 0;
for ( int fila = 0; fila < a.length; fila++ )
{
   for ( int columna = 0; columna < a[ fila ].length; columna++ )
      total += a[ fila ][ columna ];
} // fin de for exterior
```

Estas instrucciones for anidadas suman el total de los elementos del arreglo, una fila a la vez. La instrucción for exterior empieza asignando 0 al índice fila, de manera que los elementos de la primera fila puedan totalizarse mediante la instrucción for interior. Después, la instrucción for exterior incrementa fila a 1, de manera que la segunda fila pueda totalizarse. Luego, la instrucción for exterior incrementa fila a 2, para que la tercera fila pueda totalizarse. La variable total puede mostrarse al terminar la instrucción for exterior. En el siguiente ejemplo le mostraremos cómo procesar un arreglo bidimensional de una manera similar, usando instrucciones for mejoradas anidadas.

7.10 Caso de estudio: la clase LibroCalificaciones que usa un arreglo bidimensional

En la sección 7.8 presentamos la clase LibroCalificaciones (figura 7.14), la cual utilizó un arreglo unidimensional para almacenar las calificaciones de los estudiantes en un solo examen. En la mayoría de los semestres, los estudiantes presentan varios exámenes. Es probable que los profesores quieran analizar las calificaciones a lo largo de todo el semestre, tanto para un solo estudiante como para la clase en general.

Cómo almacenar las calificaciones de los estudiantes en un arreglo bidimensional en la clase `LibroCalificaciones`

La figura 7.18 contiene una versión de la clase `LibroCalificaciones` que utiliza un arreglo bidimensional llamado `calificaciones`, para almacenar las calificaciones de un número de estudiantes en varios exámenes. Cada fila del arreglo representa las calificaciones de un solo estudiante durante todo el curso, y cada columna representa las calificaciones de todos los estudiantes que presentaron un examen específico. La clase `PruebaLibroCalificaciones` (figura 7.19) pasa el arreglo como argumento para el constructor de `LibroCalificaciones`. En este ejemplo, utilizamos un arreglo de diez por tres que contiene diez calificaciones de los estudiantes en tres exámenes. Cinco métodos realizan manipulaciones de arreglos para procesar las calificaciones. Cada método es similar a su contraparte en la versión anterior de la clase `LibroCalificaciones` con un arreglo unidimensional (figura 7.14). El método `obtenerMinima` (líneas 52 a 70) determina la calificación más baja de cualquier estudiante durante el semestre. El método `obtenerMaxima` (líneas 73 a 91) determina la calificación más alta de cualquier estudiante durante el semestre. El método `obtenerPromedio` (líneas 94 a 104) determina el promedio semestral de un estudiante específico. El método `imprimirGraficoBarras` (líneas 107 a 137) imprime un gráfico de barras de todas las calificaciones de los estudiantes durante el semestre. El método `imprimirCalificaciones` (líneas 140 a 164) imprime el arreglo bidimensional en formato tabular, junto con el promedio semestral de cada estudiante.

```
1  // Fig. 7.18: LibroCalificaciones.java
2  // Clase LibroCalificaciones que utiliza un arreglo bidimensional para almacenar
   calificaciones.
3
4  public class LibroCalificaciones
5  {
6     private String nombreDelCurso; // nombre del curso que representa este
                                     LibroCalificaciones
7     private int[][] calificaciones; // arreglo bidimensional de calificaciones de
                                     estudiantes
8
9     // el constructor con dos argumentos inicializa nombreDelCurso y el arreglo
         calificaciones
10    public LibroCalificaciones( String nombre, int[][] arregloCalif )
11    {
12       nombreDelCurso = nombre; // inicializa nombreDelCurso
13       calificaciones = arregloCalif; // almacena calificaciones
14    } // fin del constructor de LibroCalificaciones con dos argumentos
15
16    // método para establecer el nombre del curso
17    public void establecerNombreDelCurso( String nombre )
18    {
19       nombreDelCurso = nombre; // almacena el nombre del curso
20    } // fin del método establecerNombreDelCurso
21
22    // método para obtener el nombre del curso
23    public String obtenerNombreDelCurso()
24    {
25       return nombreDelCurso;
26    } // fin del método obtenerNombreDelCurso
27
28    // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
29    public void mostrarMensaje()
30    {
31       // obtenerNombreDelCurso obtiene el nombre del curso
32       System.out.printf( "Bienvenido al libro de calificaciones para\n%s!\n\n",
33          obtenerNombreDelCurso() );
34    } // fin del método mostrarMensaje
35
```

Fig. 7.18 | Clase `LibroCalificaciones` que utiliza un arreglo bidimensional para almacenar calificaciones (parte 1 de 4).

```
36      // realiza varias operaciones sobre los datos
37      public void procesarCalificaciones()
38      {
39         // imprime el arreglo de calificaciones
40         imprimirCalificaciones();
41
42         // llama a los métodos obtenerMinima y obtenerMaxima
43         System.out.printf( "\n%s %d\n%s %d\n\n",
44            "La calificacion mas baja en el libro de calificaciones es", obtenerMinima(),
45            "La calificacion mas alta en el libro de calificaciones es", obtenerMaxima() );
46
47         // imprime gráfico de distribución de todas las calificaciones para todas las
              pruebas
48         imprimirGraficoBarras();
49      } // fin del método procesarCalificaciones
50
51      // busca la calificación más baja
52      public int obtenerMinima()
53      {
54         // asume que el primer elemento del arreglo calificaciones es el más bajo
55         int califBaja = calificaciones[ 0 ][ 0 ];
56
57         // itera a través de las filas del arreglo calificaciones
58         for ( int[] califEstudiantes : calificaciones )
59         {
60            // itera a través de las columnas de la fila actual
61            for ( int calificacion : califEstudiantes )
62            {
63               // si la calificación es menor que califBaja, la asigna a califBaja
64               if ( calificacion < califBaja )
65                  califBaja = calificacion;
66            } // fin de for interior
67         } // fin de for exterior
68
69         return califBaja; // devuelve calificación más baja
70      } // fin del método obtenerMinima
71
72      // busca la calificación más alta
73      public int obtenerMaxima()
74      {
75         // asume que el primer elemento del arreglo calificaciones es el más alto
76         int califAlta = calificaciones[ 0 ][ 0 ];
77
78         // itera a través de las filas del arreglo calificaciones
79         for ( int[] califEstudiantes : calificaciones )
80         {
81            // itera a través de las columnas de la fila actual
82            for ( int calificacion : califEstudiantes )
83            {
84               // si la calificación es mayor que califAlta, la asigna a califAlta
85               if ( calificacion > califAlta )
86                  califAlta = calificacion;
87            } // fin de for interior
88         } // fin de for exterior
```

Fig. 7.18 | Clase `LibroCalificaciones` que utiliza un arreglo bidimensional para almacenar calificaciones (parte 2 de 4).

```
89
90          return califAlta; // devuelve la calificación más alta
91       } // fin del método obtenerMaxima
92
93       // determina la calificación promedio para un conjunto específico de calificaciones
94       public double obtenerPromedio( int[] conjuntoDeCalif )
95       {
96          int total = 0; // inicializa el total
97
98          // suma las calificaciones para un estudiante
99          for ( int calificacion : conjuntoDeCalif )
100            total += calificacion;
101
102         // devuelve el promedio de calificaciones
103         return (double) total / conjuntoDeCalif.length;
104      } // fin del método obtenerPromedio
105
106      // imprime gráfico de barras que muestra la distribución de calificaciones en
            general
107      public void imprimirGraficoBarras()
108      {
109         System.out.println( "Distribucion de calificaciones en general:" );
110
111         // almacena la frecuencia de las calificaciones en cada rango de 10 calificaciones
112         int[] frecuencia = new int[ 11 ];
113
114         // para cada calificación en LibroCalificaciones, incrementa la frecuencia
               apropiada
115         for ( int[] califEstudiantes : calificaciones )
116         {
117            for ( int calificacion : califEstudiantes )
118               ++frecuencia[ calificacion / 10 ];
119         } // fin de for exterior
120
121         // para cada frecuencia de calificaciones, imprime una barra en el gráfico
122         for ( int cuenta = 0; cuenta < frecuencia.length; cuenta++ )
123         {
124            // imprime etiquetas de las barras ( "00-09: ", ..., "90-99: ", "100: " )
125            if ( cuenta == 10 )
126               System.out.printf( "%5d: ", 100 );
127            else
128               System.out.printf( "%02d-%02d: ",
129                  cuenta * 10, cuenta * 10 + 9  );
130
131            // imprime barra de asteriscos
132            for ( int estrellas = 0; estrellas < frecuencia[ cuenta ]; estrellas++ )
133               System.out.print( "*" );
134
135            System.out.println(); // inicia una nueva línea de salida
136         } // fin de for exterior
137      } // fin del método imprimirGraficoBarras
138
139      // imprime el contenido del arreglo calificaciones
140      public void imprimirCalificaciones()
141      {
```

Fig. 7.18 | Clase `LibroCalificaciones` que utiliza un arreglo bidimensional para almacenar calificaciones (parte 3 de 4).

```
142         System.out.println( "Las calificaciones son:\n" );
143         System.out.print( "            " ); // alinea encabezados de columnas
144
145         // crea un encabezado de columna para cada una de las pruebas
146         for ( int prueba = 0; prueba < calificaciones[ 0 ].length; prueba++ )
147            System.out.printf( "Prueba %d  ", prueba + 1 );
148
149         System.out.println( "Promedio" ); // encabezado de columna de promedio de
                                             estudiantes
150
151         // crea filas/columnas de texto que representan el arreglo calificaciones
152         for ( int estudiante = 0; estudiante < calificaciones.length; estudiante++ )
153         {
154            System.out.printf( "Estudiante %2d", estudiante + 1 );
155
156            for ( int prueba : calificaciones[ estudiante ] ) // imprime calificaciones
                                                                 de estudiante
157               System.out.printf( "%8d", prueba );
158
159            // llama al método obtenerPromedio para calcular la calificación promedio
                  del estudiante;
160            // pasa fila de calificaciones como argumento para obtenerPromedio
161            double promedio = obtenerPromedio( calificaciones[ estudiante ] );
162            System.out.printf( "%9.2f\n", promedio );
163         } // fin de for exterior
164      } // fin del método imprimirCalificaciones
165   } // fin de la clase LibroCalificaciones
```

Fig. 7.18 | Clase `LibroCalificaciones` que utiliza un arreglo bidimensional para almacenar calificaciones (parte 4 de 4).

Los métodos `obtenerMinima` *y* `obtenerMaxima`

Cada uno de los métodos `obtenerMinima`, `obtenerMaxima`, `imprimirGraficoBarras` e `imprimirCalificaciones` itera a través del arreglo `calificaciones` mediante el uso de instrucciones `for` anidadas; por ejemplo, la instrucción `for` mejorada anidada de la declaración del método `obtenerMinima` (líneas 58 a 67). La instrucción `for` mejorada exterior itera a través del arreglo bidimensional `calificaciones`, asignando filas sucesivas al parámetro `califEstudiantes` en las iteraciones sucesivas. Los corchetes que van después del nombre del parámetro indican que `califEstudiantes` se refiere a un arreglo `int` unidimensional: una fila en el arreglo `calificaciones` que contiene las calificaciones de un estudiante. Para buscar la calificación más baja en general, la instrucción `for` interior compara los elementos del arreglo unidimensional actual `califEstudiantes` con la variable `califBaja`. Por ejemplo, en la primera iteración del `for` exterior, la fila 0 de `calificaciones` se asigna al parámetro `califEstudiantes`. Después, la instrucción `for` mejorada interior itera a través de `califEstudiantes` y compara cada valor de `calificacion` con `califBaja`. Si una calificación es menor que `califBaja`, a `califBaja` se le asigna esa calificación. En la segunda iteración de la instrucción `for` mejorada exterior, la fila 1 de `calificaciones` se asigna a `califEstudiantes`, y los elementos de esta fila se comparan con la variable `califBaja`. Esto se repite hasta que se hayan recorrido todas las filas de `calificaciones`. Cuando se completa la ejecución de la instrucción anidada, `califBaja` contiene la calificación más baja en el arreglo bidimensional. El método `obtenerMaxima` trabaja en forma similar al método `obtenerMinima`.

El método `imprimirGraficoBarras`

El método `imprimirGraficoBarras` en la figura 7.18 es casi idéntico al de la figura 7.14. Sin embargo, para imprimir la distribución de calificaciones en general durante todo un semestre, el método aquí utiliza instrucciones `for` mejoradas anidadas (líneas 115 a 119) para crear el arreglo unidimensional

`frecuencia`, con base en todas las calificaciones en el arreglo bidimensional. El resto del código en cada uno de los dos métodos `imprimirGraficoBarras` que muestran el gráfico es idéntico.

El método `imprimirCalificaciones`

El método `imprimirCalificaciones` (líneas 140 a 164) utiliza instrucciones `for` anidadas para imprimir valores del arreglo `calificaciones`, además del promedio semestral de cada estudiante. La salida (figura 7.19) muestra el resultado, el cual se asemeja al formato tabular del libro de calificaciones real de un profesor. Las líneas 146 y 147 imprimen los encabezados de columna para cada prueba. Aquí utilizamos una instrucción `for` controlada por contador, para poder identificar cada prueba con un número. De manera similar, la instrucción `for` en las líneas 152 a 163 imprime primero una etiqueta de fila mediante el uso de una variable contador para identificar a cada estudiante (línea 154). Aunque los subíndices de los arreglos empiezan en 0, las líneas 147 y 154 imprimen `prueba + 1` y `estudiante + 1` en forma respectiva, para producir números de prueba y de estudiante que empiecen en 1 (vea la figura 7.19). La instrucción `for` interna (líneas 156 y 157) utiliza la variable contador `estudiante` de la instrucción `for` exterior para iterar a través de una fila específica del arreglo `calificaciones`, e imprime la calificación de la prueba de cada estudiante. Observe que una instrucción `for` mejorada puede anidarse en una instrucción `for` controlada por contador, y viceversa. Por último, la línea 161 obtiene el promedio semestral de cada estudiante, para lo cual pasa la fila actual de `calificaciones` (es decir, `calificaciones[estudiante]`) al método `obtenerPromedio`.

El método `obtenerPromedio`

El método `obtenerPromedio` (líneas 94 a 104) recibe un argumento: un arreglo unidimensional de resultados de la prueba para un estudiante específico. Cuando la línea 161 llama a `obtenerPromedio`, el argumento es `calificaciones[estudiante]`, que especifica que debe pasarse una fila determinada del arreglo bidimensional `calificaciones` a `obtenerPromedio`. Por ejemplo, con base en el arreglo creado en la figura 7.19, el argumento `calificaciones[1]` representa los tres valores (un arreglo unidimensional de calificaciones) almacenados en la fila 1 del arreglo bidimensional `calificaciones`. Recuerde que un arreglo bidimensional es un arreglo cuyos elementos son arreglos unidimensionales. El método `obtenerPromedio` calcula la suma de los elementos del arreglo, divide el total entre el número de resultados de la prueba y devuelve el resultado de punto flotante como un valor `double` (línea 103).

La clase `PruebaLibroCalificaciones` *que demuestra la clase* `LibroCalificaciones`

La figura 7.19 crea un objeto de la clase `LibroCalificaciones` (figura 7.18) mediante el uso del arreglo bidimensional de valores `int` llamado `arregloCalif` (el cual se declara e inicializa en las líneas 10 a 19). Las líneas 21 y 22 pasan el nombre de un curso y `arregloCalif` al constructor de `LibroCalificaciones`. Después, las líneas 23 y 24 invocan a los métodos `mostrarMensaje` y `procesarCalificaciones` de `miLibroCalificaciones`, para mostrar un mensaje de bienvenida y obtener un informe que sintetice las calificaciones de los estudiantes para el semestre, en forma respectiva.

```
1  // Fig. 7.19: PruebaLibroCalificaciones.java
2  // PruebaLibroCalificaciones crea un objeto LibroCalificaciones, usando un arreglo
      bidimensional
3  // de calificaciones, y después invoca al método procesarCalificaciones para
      analizarlas
4  public class PruebaLibroCalificaciones
5  {
6     // el método main comienza la ejecución del programa
7     public static void main( String[] args )
8     {
```

Fig. 7.19 | `PruebaLibroCalificaciones` crea un objeto `LibroCalificaciones` mediante un arreglo bidimensional de calificaciones; después invoca al método `procesarCalificaciones` para analizarlas (parte 1 de 2).

```
9         // arreglo bidimensional de calificaciones de estudiantes
10        int[][] arregloCalif = { { 87, 96, 70 },
11                                 { 68, 87, 90 },
12                                 { 94, 100, 90 },
13                                 { 100, 81, 82 },
14                                 { 83, 65, 85 },
15                                 { 78, 87, 65 },
16                                 { 85, 75, 83 },
17                                 { 91, 94, 100 },
18                                 { 76, 72, 84 },
19                                 { 87, 93, 73 } };
20
21        LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
22           "CS101 Introduccion a la programacion en Java", arregloCalif );
23        miLibroCalificaciones.mostrarMensaje();
24        miLibroCalificaciones.procesarCalificaciones();
25     } // fin de main
26  } // fin de la clase PruebaLibroCalificaciones
```

```
Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en Java!

Las calificaciones son:

            Prueba 1   Prueba 2   Prueba 3   Promedio
Estudiante  1         87         96         70      84.33
Estudiante  2         68         87         90      81.67
Estudiante  3         94        100         90      94.67
Estudiante  4        100         81         82      87.67
Estudiante  5         83         65         85      77.67
Estudiante  6         78         87         65      76.67
Estudiante  7         85         75         83      81.00
Estudiante  8         91         94        100      95.00
Estudiante  9         76         72         84      77.33
Estudiante 10         87         93         73      84.33

La calificacion mas baja en el libro de calificaciones es 65
La calificacion mas alta en el libro de calificaciones es 100

Distribucion de calificaciones en general:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

Fig. 7.19 | PruebaLibroCalificaciones crea un objeto LibroCalificaciones mediante un arreglo bidimensional de calificaciones; después invoca al método procesarCalificaciones para analizarlas (parte 2 de 2).

7.11 Listas de argumentos de longitud variable

Con las **listas de argumentos de longitud variable** podemos crear métodos que reciben un número arbitrario de argumentos. Un tipo que va precedido por una **elipsis (...)** en la lista de parámetros de un método indica que éste recibe un número variable de argumentos de ese tipo específico. Este uso de la elipsis puede ocurrir sólo una vez en una lista de parámetros, y la elipsis, junto con su tipo, debe colocarse al final de la lista. Aunque podemos utilizar la sobrecarga de métodos y el paso de arreglos para realizar gran parte de lo que se logra con las listas de argumentos de longitud variable, es más conciso utilizar una elipsis en la lista de parámetros de un método.

La figura 7.20 demuestra el método promedio (líneas 7 a 16), el cual recibe una secuencia de longitud variable de valores `double`. Java trata a la lista de argumentos de longitud variable como un arreglo cuyos elementos son del mismo tipo. Así, el cuerpo del método puede manipular el parámetro `numeros` como un arreglo de valores `double`. Las líneas 12 y 13 utilizan el ciclo `for` mejorado para recorrer el arreglo y calcular el total de los valores `double` en el arreglo. La línea 15 accede a `numeros.length` para obtener el tamaño del arreglo `numeros` y usarlo en el cálculo del promedio. Las líneas 29, 31 y 33 en `main` llaman al método `promedio` con dos, tres y cuatro argumentos, respectivamente. El método `promedio` tiene una lista de argumentos de longitud variable (línea 7), por lo que puede promediar todos los argumentos `double` que le pase el método que hace la llamada. La salida revela que cada llamada al método `promedio` devuelve el valor correcto.

```
1  // Fig. 7.20: PruebaVarargs.java
2  // Uso de listas de argumentos de longitud variable.
3
4  public class PruebaVarargs
5  {
6     // calcula el promedio
7     public static double promedio( double... numeros )
8     {
9        double total = 0.0; // inicializa el total
10
11       // calcula el total usando la instrucción for mejorada
12       for ( double d : numeros )
13          total += d;
14
15       return total / numeros.length;
16    } // fin del método promedio
17
18    public static void main( String[] args )
19    {
20       double d1 = 10.0;
21       double d2 = 20.0;
22       double d3 = 30.0;
23       double d4 = 40.0;
24
25       System.out.printf( "d1 = %.1f\nd2 = %.1f\nd3 = %.1f\nd4 = %.1f\n\n",
26          d1, d2, d3, d4 );
27
28       System.out.printf( "El promedio de d1 y d2 es %.1f\n",
29          promedio( d1, d2 ) );
```

Fig. 7.20 | Uso de listas de argumentos de longitud variable (parte 1 de 2).

```
30         System.out.printf( "El promedio de d1, d2 y d3 es %.1f\n",
31            promedio( d1, d2, d3 ) );
32         System.out.printf( "El promedio de d1, d2, d3 y d4 es %.1f\n",
33            promedio( d1, d2, d3, d4 ) );
34      } // fin de main
35   } // fin de la clase PruebaVarargs
```

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

El promedio de d1 y d2 es 15.0
El promedio de d1, d2 y d3 es 20.0
El promedio de d1, d2, d3 y d4 es 25.0
```

Fig. 7.20 | Uso de listas de argumentos de longitud variable (parte 2 de 2).

Error común de programación 7.6

Colocar una elipsis en medio de una lista de parámetros de un método para indicar una lista de argumentos de longitud variable es un error de sintaxis. La elipsis sólo debe colocarse al final de la lista de parámetros.

7.12 Uso de argumentos de línea de comandos

En muchos sistemas, es posible pasar argumentos desde la línea de comandos (a éstos se les conoce como **argumentos de línea de comandos**) a una aplicación, para lo cual se incluye un parámetro de tipo `String[]` (es decir, un arreglo de objetos `String`) en la lista de parámetros de `main`, justo igual que como hemos hecho en todas las aplicaciones de este libro. Por convención, a este parámetro se le llama `args`. Cuando se ejecuta una aplicación con el comando `java`, Java pasa los argumentos de línea de comandos que aparecen después del nombre de la clase en el comando `java` al método `main` de la aplicación, en forma de objetos `String` en el arreglo `args`. El número de argumentos que se pasan desde la línea de comandos se obtiene accediendo al atributo `length` del arreglo. Los usos comunes de los argumentos de línea de comandos incluyen pasar opciones y nombres de archivos a las aplicaciones.

Nuestro siguiente ejemplo utiliza argumentos de línea de comandos para determinar el tamaño de un arreglo, el valor de su primer elemento y el incremento utilizado para calcular los valores de los elementos restantes del arreglo. El comando

```
java InicArreglo 5 0 4
```

pasa tres argumentos, `5`, `0` y `4`, a la aplicación `InicArreglo`. Los argumentos de línea de comandos se separan mediante espacio en blanco, sin comas. Cuando se ejecuta este comando, el método `main` de `InicArreglo` recibe el arreglo `args` de tres elementos (es decir, `args.length` es 3), en donde `args[0]` contiene el valor `String` `"5"`, `args[1]` contiene el valor `String` `"0"` y `args[2]` contiene el valor `String` `"4"`. El programa determina cómo usar estos argumentos; en la figura 7.21 convertimos los tres argumentos de la línea de comandos a valores `int` y los usamos para inicializar un arreglo. Cuando se ejecuta el programa, si `args.length` no es 3, el programa imprime un mensaje de error y termina (líneas 9 a 12). En cualquier otro caso, las líneas 14 a 32 inicializan y muestran el arreglo, con base en los valores de los argumentos de línea de comandos.

La línea 16 obtiene `args[0]` (un objeto `String` que especifica el tamaño del arreglo) y lo convierte en un valor `int`, que el programa utiliza para crear el arreglo en la línea 17. El método `static` `parseInt` de la clase `Integer` convierte su argumento `String` en un `int`.

```
1  // Fig. 7.21: InicArreglo.java
2  // Uso de los argumentos de línea de comandos para inicializar un arreglo.
3
4  public class InicArreglo
5  {
6     public static void main( String[] args )
7     {
8        // comprueba el número de argumentos de línea de comandos
9        if ( args.length != 3 )
10          System.out.println(
11             "Error: Vuelva a escribir el comando completo, incluyendo\n" +
12             "el tamanio del arreglo, el valor inicial y el incremento." );
13       else
14       {
15          // obtiene el tamaño del arreglo del primer argumento de línea de comandos
16          int longitudArreglo = Integer.parseInt( args[ 0 ] );
17          int[] arreglo = new int[ longitudArreglo ]; // crea el arreglo
18
19          // obtiene el valor inicial y el incremento de los argumentos de línea de
               comandos
20          int valorInicial = Integer.parseInt( args[ 1 ] );
21          int incremento = Integer.parseInt( args[ 2 ] );
22
23          // calcula el valor para cada elemento del arreglo
24          for ( int contador = 0; contador < arreglo.length; contador++ )
25             arreglo[ contador ] = valorInicial + incremento * contador;
26
27          System.out.printf( "%s%8s\n", "Indice", "Valor" );
28
29          // muestra el índice y el valor del arreglo
30          for ( int contador = 0; contador < arreglo.length; contador++ )
31             System.out.printf( "%5d%8d\n", contador, arreglo[ contador ] );
32       } // fin de else
33    } // fin de main
34 } // fin de la clase InicArreglo
```

```
java InicArreglo
Error: Vuelva a escribir el comando completo, incluyendo
el tamanio del arreglo, el valor inicial y el incremento.
```

```
java InicArreglo 5 0 4
Indice      Valor
     0          0
     1          4
     2          8
     3         12
     4         16
```

Fig. 7.21 | Inicialización de un arreglo mediante el uso de argumentos de línea de comandos (parte 1 de 2).

```
java InicArreglo 8 1 2
Indice      Valor
     0          1
     1          3
     2          5
     3          7
     4          9
     5         11
     6         13
     7         15
```

Fig. 7.21 | Inicialización de un arreglo mediante el uso de argumentos de línea de comandos (parte 2 de 2).

Las líneas 20 a 21 convierten los argumentos de línea de comandos `args[1]` y `args[2]` en valores `int`, y los almacenan en `valorInicial` e incremento, de manera respectiva. Las líneas 24 y 25 calculan el valor para cada elemento del arreglo.

Los resultados de la primera ejecución muestran que la aplicación recibió un número insuficiente de argumentos de línea de comandos. La segunda ejecución utiliza los argumentos de línea de comandos 5, 0 y 4 para especificar el tamaño del arreglo (5), el valor del primer elemento (0) y el incremento de cada valor en el arreglo (4), respectivamente. Los resultados correspondientes muestran que estos valores crean un arreglo que contiene los enteros 0, 4, 8, 12 y 16. Los resultados de la tercera ejecución muestran que los argumentos de línea de comandos 8, 1 y 2 producen un arreglo cuyos 8 elementos son los enteros impares positivos del 1 al 15.

7.13 La clase Arrays

Gracias a la clase **`Arrays`** no tenemos que reinventar la rueda, ya que proporciona métodos `static` para las manipulaciones comunes de arreglos. Estos métodos incluyen **`sort`** para ordenar un arreglo (acomodar los elementos en orden ascendente), **`binarySearch`** para buscar en un arreglo (es decir, determinar si un arreglo contiene un valor específico y, de ser así, en dónde se encuentra este valor), **`equals`** para comparar arreglos y **`fill`** para colocar valores en un arreglo. Estos métodos están sobrecargados para los arreglos de tipo primitivo y los arreglos de objetos. Nuestro enfoque en esta sección está en usar las herramientas integradas que proporciona la API de Java. En el capítulo 19 (en el sitio Web del libro), Búsqueda, ordenamiento y Big O, veremos cómo implementar nuestros propios algoritmos de búsqueda y ordenamiento, que son de gran interés para los investigadores y estudiantes de ciencias computacionales.

La figura 7.22 usa los métodos `sort`, `binarySearch`, `equals` y `fill` de la clase `Arrays`, y muestra cómo copiar arreglos con el **método `static` `arraycopy`** de la clase `System`. En `main`, la línea 11 ordena los elementos del arreglo `arregloDouble`. El método `static` `sort` de la clase `Arrays` ordena los elementos del arreglo en orden *ascendente*, de manera predeterminada. Más adelante en este capítulo veremos cómo ordenar elementos en forma *descendente*. Las versiones sobrecargadas de `sort` nos permiten ordenar un rango específico de elementos. Las líneas 12 a 15 imprimen en pantalla el arreglo ordenado.

```
1  // Fig. 7.22: ManipulacionesArreglos.java
2  // Métodos de la clase Arrays y System.arraycopy.
3  import java.util.Arrays;
4
```

Fig. 7.22 | Métodos de la clase `Arrays` (parte 1 de 3).

```
5  public class ManipulacionesArreglos
6  {
7     public static void main( String[] args )
8     {
9        // ordena arregloDouble en forma ascendente
10       double[] arregloDouble = { 8.4, 9.3, 0.2, 7.9, 3.4 };
11       Arrays.sort( arregloDouble );
12       System.out.printf( "\narregloDouble: " );
13
14       for ( double valor : arregloDouble )
15          System.out.printf( "%.1f ", valor );
16
17       // llena arreglo de 10 elementos con 7
18       int[] arregloIntLleno = new int[ 10 ];
19       Arrays.fill( arregloIntLleno, 7 );
20       mostrarArreglo( arregloIntLleno, "arregloIntLleno" );
21
22       // copia el arreglo arregloInt en el arreglo copiaArregloInt
23       int[] arregloInt = { 1, 2, 3, 4, 5, 6 };
24       int[] copiaArregloInt = new int[ arregloInt.length ];
25       System.arraycopy( arregloInt, 0, copiaArregloInt, 0, arregloInt.length );
26       mostrarArreglo( arregloInt, "arregloInt" );
27       mostrarArreglo( copiaArregloInt, "copiaArregloInt" );
28
29       // compara si arregloInt y copiaArregloInt son iguales
30       boolean b = Arrays.equals( arregloInt, copiaArregloInt );
31       System.out.printf( "\n\narregloInt %s copiaArregloInt\n",
32          ( b ? "==" : "!=" ) );
33
34       // compara si arregloInt y arregloIntLleno son iguales
35       b = Arrays.equals( arregloInt, arregloIntLleno );
36       System.out.printf( "arregloInt %s arregloIntLleno\n",
37          ( b ? "==" : "!=" ) );
38
39       // busca en arregloInt el valor 5
40       int ubicacion = Arrays.binarySearch( arregloInt, 5 );
41
42       if ( ubicacion >= 0 )
43          System.out.printf(
44             "Se encontro 5 en el elemento %d de arregloInt\n", ubicacion );
45       else
46          System.out.println( "No se encontro el 5 en arregloInt" );
47
48       // busca en arregloInt el valor 8763
49       ubicacion = Arrays.binarySearch( arregloInt, 8763 );
50
51       if ( ubicacion >= 0 )
52          System.out.printf(
53             "Se encontro el 8763 en el elemento %d de arregloInt\n", ubicacion );
54       else
55          System.out.println( "No se encontro el 8763 en arregloInt" );
56    } // fin de main
57
```

Fig. 7.22 | Métodos de la clase Arrays (parte 2 de 3).

```
58      // imprime los valores en cada arreglo
59      public static void mostrarArreglo( int[] arreglo, String descripcion )
60      {
61         System.out.printf( "\n%s: ", descripcion );
62
63         for ( int valor : arreglo )
64            System.out.printf( "%d ", valor );
65      } // fin del método mostrarArreglo
66   } // fin de la clase ManipulacionesArreglos
```

```
arregloDouble: 0.2 3.4 7.9 8.4 9.3
arregloIntLleno: 7 7 7 7 7 7 7 7 7 7
arregloInt: 1 2 3 4 5 6
copiaArregloInt: 1 2 3 4 5 6

arregloInt == copiaArregloInt
arregloInt != arregloIntLleno
Se encontro 5 en el elemento 4 de arregloInt
No se encontro el 8763 en arregloInt
```

Fig. 7.22 | Métodos de la clase `Arrays` (parte 3 de 3).

La línea 19 llama al método `static fill` de la clase `Arrays` para llenar los 10 elementos de `arregloIntLleno` con 7. Las versiones sobrecargadas de `fill` nos permiten llenar un rango específico de elementos con el mismo valor. La línea 20 llama al método `mostrarArreglo` de nuestra clase (declarado en las líneas 59 a 65) para imprimir en pantalla el contenido de `arregloIntLleno`.

La línea 25 copia los elementos de `arregloInt` en `copiaArregloInt`. El primer argumento (`arregloInt`) que se pasa al método `arraycopy` de `System` es el arreglo a partir del cual se van a copiar los elementos. El segundo argumento (0) es el índice que especifica el punto de inicio en el rango de elementos que se van a copiar del arreglo. Este valor puede ser cualquier índice de arreglo válido. El tercer argumento (`copiaArregloInt`) especifica el arreglo de destino que almacenará la copia. El cuarto argumento (0) especifica el índice en el arreglo de destino en donde deberá guardarse el primer elemento copiado. El último argumento especifica el número de elementos a copiar del arreglo en el primer argumento. En este caso, copiaremos todos los elementos en el arreglo.

En las líneas 30 y 35 se hace una llamada al método `static equals of class Arrays` para determinar si todos los elementos de los dos arreglos son equivalentes. Si los arreglos contienen los mismos elementos, en el mismo orden, el método regresa `true`; si no, regresa `false`.

En la líneas 40 y 49 se hace una llamada al método `static binarySearch` de la clase `Arrays` para realizar una búsqueda binaria en `arregloInt`, utilizando el segundo argumento (5 y 8763, como corresponde a cada uno) como la clave. Si se encuentra `valor`, `binarySearch` devuelve el índice del elemento; en caso contrario, `binarySearch` devuelve un valor negativo. El cual se basa en el punto de inserción de la clave de búsqueda: el índice en donde se insertaría la clave en el arreglo si se fuera a realizar una operación de inserción. Una vez que `binarySearch` determina el punto de inserción, cambia el signo de éste a negativo y le resta 1 para obtener el valor de retorno. Por ejemplo, en la figura 7.22, el punto de inserción para el valor 8763 es el elemento en el arreglo con el índice 6. El método `binarySearch` cambia el punto de inserción a -6, le resta 1 y devuelve el valor -7. Al restar 1 al punto de inserción se garantiza que el método `binarySearch` devuelva valores positivos (>= 0) sí, y sólo si se encuentra la clave. Este valor de retorno es útil para insertar elementos en un arreglo ordenado. En el capítulo 19 (en el sitio Web del libro) veremos la búsqueda binaria con detalle.

Error común de programación 7.7

Pasar un arreglo desordenado al método `binarySearch` *es un error lógico; el valor devuelto es indefinido.*

7.14 Introducción a las colecciones y la clase ArrayList

La API de Java provee varias estructuras de datos predefinidas, conocidas como **colecciones**, que se utilizan para almacenar grupos de objetos relacionados. Estas clases proveen métodos eficientes que organizan, almacenan y obtienen los datos sin necesidad de saber cómo se almacenan éstos. Gracias a esto, se reduce el tiempo de desarrollo de aplicaciones.

Usted ya utilizó arreglos para almacenar secuencias de objetos. Los arreglos no cambian de manera automática su tamaño en tiempo de ejecución para dar cabida a elementos adicionales. La clase de colección `ArrayList<T>` (del paquete `java.util`) provee una solución conveniente a este problema: puede cambiar su tamaño en forma *dinámica* para dar cabida a más elementos. La `T` (por convención) es un *receptáculo*: al declarar un nuevo objeto `ArrayList`, hay que reemplazarlo con el tipo de elementos que deseamos que contenga el objeto `ArrayList`. Esto es similar a especificar el tipo cuando declaramos un arreglo, pero *sólo se pueden usar tipos no primitivos con estas clases de colecciones*. Por ejemplo,

```
ArrayList< String > lista;
```

declara a lista como una colección `ArrayList` que sólo puede almacenar objetos `String`. Las clases con este tipo de receptáculo que se pueden usar con cualquier tipo se conocen como **clases genéricas**. En los capítulos 20 y 21 (en inglés, en el sitio Web del libro) hablaremos sobre más clases de colecciones genéricas y de genéricos. La figura 7.23 muestra algunos métodos comunes de la clase `ArrayList<T>`.

Método	Descripción
`add`	Agrega un elemento al final del objeto `ArrayList`.
`clear`	Elimina todos los elementos del objeto `ArrayList`.
`contains`	Devuelve `true` si el objeto `ArrayList` contiene el elemento especificado; en caso contrario, devuelve `false`.
`get`	Devuelve el elemento en el índice especificado.
`indexOf`	Devuelve el índice de la primera ocurrencia del elemento especificado en el objeto `ArrayList`.
`remove`	Sobrecargado. Elimina la primera ocurrencia del valor especificado o del elemento en el subíndice especificado.
`size`	Devuelve el número de elementos almacenados en el objeto `ArrayList`.
`trimToSize`	Recorta la capacidad del objeto `ArrayList` al número actual de elementos.

Fig. 7.23 | Algunos métodos y propiedades de la clase `ArrayList<T>`.

La figura 7.24 demuestra algunas capacidades comunes de `ArrayList`. La línea 10 crea un objeto `ArrayList` de objetos `String` vacío, con una capacidad inicial predeterminada de 10 elementos. Esta capacidad indica cuántos elementos puede contener el objeto `ArrayList` sin tener que crecer. El objeto `ArrayList` se implementa mediante el uso de un arreglo tras bambalinas. Cuando crece el objeto `ArrayList`, debe crear un arreglo interno más grande y copiar cada elemento al nuevo arreglo. Esta

operación consume mucho tiempo. Sería ineficiente que el objeto ArrayList creciera cada vez que se agregara un elemento. En cambio, sólo crece cuando se agrega un elemento *y* el número de elementos es igual que la capacidad; es decir, cuando no hay espacio para el nuevo elemento.

```
1  // Fig. 7.24: ColeccionArrayList.java
2  // Demostración de la colección de genéricos ArrayList.
3  import java.util.ArrayList;
4
5  public class ColeccionArrayList
6  {
7     public static void main( String[] args )
8     {
9        // crea  un nuevo objeto ArrayList de objetos String con una capacidad inicial de 10
10       ArrayList< String > elementos = new ArrayList< String >();
11
12       elementos.add( "rojo" ); // adjunta un elemento a la lista
13       elementos.add( 0, "amarillo" ); // inserta el valor en el subíndice 0
14
15       // encabezado
16       System.out.print(
17          "Mostrar contenido de lista con ciclo controlado por contador:" );
18
19       // muestra los colores en la lista
20       for ( int i = 0; i < elementos.size(); i++ )
21          System.out.printf( " %s", elementos.get( i ) );
22
23       // muestra los colores usando foreach en el método mostrar
24       mostrar( elementos,
25          "\nMostrar contenido de lista con instruccion for mejorada:" );
26
27       elementos.add( "verde" ); // agrega "verde" al final de la lista
28       elementos.add( "amarillo" ); // agrega "amarillo" al final de la lista
29       mostrar( elementos, "Lista con dos nuevos elementos:" );
30
31       elementos.remove( "amarillo" ); // elimina el primer "amarillo"
32       mostrar( elementos, "Eliminar primera instancia de amarillo:" );
33
34       elementos.remove( 1 ); // elimina elemento en subíndice 1
35       mostrar( elementos, "Eliminar segundo elemento de la lista (verde):" );
36
37       // verifica si hay un valor en la lista
38       System.out.printf( "\"rojo\" %sesta en la lista\n",
39          elementos.contains( "rojo" ) ? "": "no " );
40
41       // muestra el número de elementos en la lista
42       System.out.printf( "Tamanio: %s\n", elementos.size() );
43    } // fin de main
44
45    // muestra los elementos de ArrayList en la consola
46    public static void mostrar( ArrayList< String > elementos, String encabezado )
47    {
48       System.out.print( encabezado ); // mostrar encabezado
```

Fig. 7.24 | Demostración de la colección de genéricos ArrayList (parte 1 de 2).

```
49
50         // muestra cada elemento en elementos
51         for ( String elemento : elementos )
52            System.out.printf( " %s", elemento );
53
54         System.out.println(); // muestra fin de línea
55      } // fin del método mostrar
56   } // fin de la clase ColeccionArrayList
```

```
Mostrar contenido de lista con ciclo controlado por contador: amarillo rojo
Mostrar contenido de lista con instruccion for mejorada: amarillo rojo
Lista con dos nuevos elementos: amarillo rojo verde amarillo
Eliminar primera instancia de amarillo: rojo verde amarillo
Eliminar segundo elemento de la lista (verde): rojo amarillo
"rojo" esta en la lista
Tamanio: 2
```

Fig. 7.24 | Demostración de la colección de genéricos `ArrayList` (parte 2 de 2).

El método **add** agrega elementos al objeto `ArrayList` (líneas 12 y 13). El método `add` con un argumento agrega su argumento al final del objeto `ArrayList`. El método `add` con dos argumentos inserta un nuevo elemento en la posición especificada. El primer argumento es un subíndice. Al igual que en los arreglos, los subíndices de las colecciones empiezan en cero. El segundo argumento es el valor a insertar en ese subíndice. Los subíndices de todos los elementos subsiguientes se incrementan en uno. Por lo general, el proceso de insertar un elemento es más lento que agregar un elemento al final del objeto `ArrayList`.

Las líneas 20 y 21 muestran los elementos en el objeto `ArrayList`. El método **size** devuelve el número de elementos que se encuentran en ese momento en el objeto `ArrayList`. El método **get** de `ArrayList` (línea 21) obtiene el elemento en un subíndice especificado. Las líneas 24 y 25 muestran los elementos de nuevo, invocando al método mostrar (definido en las líneas 46 a 55). Las líneas 27 y 28 agregan dos elementos más al objeto `ArrayList`; después la línea 29 muestra los elementos de nuevo, para confirmar que se hayan agregado los dos elementos al final de la colección.

El método **remove** se utiliza para eliminar un elemento con un valor específico (línea 31). Sólo elimina el primer elemento que cumpla con esas características. Si no se encuentra dicho elemento en el objeto `ArrayList`, `remove` no hace nada. Una versión sobrecargada del método elimina el elemento en el subíndice especificado (línea 34). Cuando se elimina un elemento, se decrementan en uno los subíndices de todos los elementos que están después del elemento eliminado.

La línea 39 usa el método **contains** para verificar si un elemento está en el objeto `ArrayList`. El método `contains` devuelve `true` si el elemento se encuentra en el objeto `ArrayList`, y `false` en el caso contrario. Este método compara su argumento con cada elemento del objeto `ArrayList` en orden, por lo que puede ser ineficiente usar `contains` en un objeto `ArrayList` grande. La línea 42 muestra el tamaño del objeto `ArrayList`.

7.15 (Opcional) Caso de estudio de GUI y gráficos: dibujo de arcos

Mediante el uso de las herramientas para gráficos de Java, podemos crear dibujos complejos que, si los codificáramos línea por línea, sería un proceso tedioso. En las figuras 7.25 y 7.26 utilizamos arreglos e instrucciones de repetición para dibujar un arco iris, mediante el uso del método `fillArc` de `Graphics`. El proceso de dibujar arcos en Java es similar a dibujar óvalos; un arco es simplemente una sección de un óvalo.

La figura 7.25 empieza con las instrucciones `import` usuales para ciertos dibujos (líneas 3 a 5). Las líneas 10 y 11 declaran y crean dos nuevas constantes de colores: `VIOLETA` e `INDIGO`. Como tal vez lo sepa, los colores de un arco iris son rojo, naranja, amarillo, verde, azul, índigo y violeta. Java tiene constantes predefinidas sólo para los primeros cinco colores. Las líneas 15 a 17 inicializan un arreglo con los colores del arco iris, empezando con los arcos más interiores primero. El arreglo empieza con dos elementos `Color.WHITE`, que como veremos pronto, son para dibujar los arcos vacíos en el centro del arco iris. Las variables de instancia se pueden inicializar al momento de declararse, como se muestra en las líneas 10 a 17. El constructor (líneas 20 a 23) contiene una sola instrucción que llama al método **setBackground** (heredado de la clase `JPanel`) con el parámetro `Color.WHITE`. El método `setBackground` recibe un solo argumento `Color` y establece el color de fondo del componente a ese color.

```
1  // Fig. 7.25: DibujoArcoIris.java
2  // Demuestra el uso de colores en un arreglo.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class DibujoArcoIris extends JPanel
8  {
9     // Define los colores índigo y violeta
10    private final static Color VIOLETA = new Color( 128, 0, 128 );
11    private final static Color INDIGO = new Color( 75, 0, 130 );
12
13    // los colores a usar en el arco iris, empezando desde los más interiores
14    // Las dos entradas de color blanco producen un arco vacío en el centro
15    private Color[] colores =
16       { Color.WHITE, Color.WHITE, VIOLETA, INDIGO, Color.BLUE,
17         Color.GREEN, Color.YELLOW, Color.ORANGE, Color.RED };
18
19    // constructor
20    public DibujoArcoIris()
21    {
22       setBackground( Color.WHITE ); // establece el fondo al color blanco
23    } // fin del constructor de DibujoArcoIris
24
25    // dibuja un arco iris, usando círculos concéntricos
26    public void paintComponent( Graphics g )
27    {
28       super.paintComponent( g );
29
30       int radio = 20; // el radio de un arco
31
32       // dibuja el arco iris cerca de la parte central inferior
33       int centroX = getWidth() / 2;
34       int centroY = getHeight() - 10;
35
36       // dibuja arcos rellenos, empezando con el más exterior
37       for ( int contador = colores.length; contador > 0; contador-- )
38       {
```

Fig. 7.25 | Dibujo de un arco iris, usando arcos y un arreglo de colores (parte 1 de 2).

```
39          // establece el color para el arco actual
40          g.setColor( colores[ contador - 1 ] );
41
42          // rellena el arco desde 0 hasta 180 grados
43          g.fillArc( centroX - contador * radio,
44             centroY - contador * radio,
45             contador * radio * 2, contador * radio * 2, 0, 180 );
46       } // fin de for
47    } // fin del método paintComponent
48 } // fin de la clase DibujoArcoIris
```

Fig. 7.25 | Dibujo de un arco iris, usando arcos y un arreglo de colores (parte 2 de 2).

La línea 30 en `paintComponent` declara la variable local `radio`, que determina el radio de cada arco. Las variables locales `centroX` y `centroY` (líneas 33 y 34) determinan la ubicación del punto medio en la base del arco iris. El ciclo en las líneas 37 a 46 utiliza la variable de control `contador` para contar en forma regresiva, partiendo del final del arreglo, dibujando los arcos más grandes primero y colocando cada arco más pequeño encima del anterior. La línea 40 establece el color para dibujar el arco actual del arreglo. La razón por la que tenemos entradas `Color.WHITE` al principio del arreglo es para crear el arco vacío en el centro. De no ser así, el centro del arco iris sería tan sólo un semicírculo sólido color violeta. [*Nota:* puede cambiar los colores individuales y el número de entradas en el arreglo para crear nuevos diseños].

La llamada al método **`fillArc`** en las líneas 43 a 45 dibuja un semicírculo relleno. El método `fillArc` requiere seis parámetros. Los primeros cuatro representan el rectángulo delimitador en el cual se dibujará el arco. Los primeros dos de estos cuatro especifican las coordenadas para la esquina superior izquierda del rectángulo delimitador, y los siguientes dos especifican su anchura y su altura. El quinto parámetro es el ángulo inicial en el óvalo, y el sexto especifica el **barrido**, o la cantidad de arco que se cubrirá. El ángulo inicial y el barrido se miden en grados, en donde los cero grados apuntan a la derecha. Un barrido positivo dibuja el arco en sentido contrario a las manecillas del reloj, en tanto que un barrido negativo dibuja el arco en sentido de las manecillas del reloj. Un método similar a `fillArc` es **`drawArc`**; requiere los mismos parámetros que `fillArc`, pero dibuja el borde del arco, en vez de rellenarlo.

La clase `PruebaDibujoArcoIris` (figura 7.26) crea y establece un objeto `JFrame` para mostrar el arco iris en la pantalla. Una vez que el programa hace visible el objeto `JFrame`, el sistema llama al método `paintComponent` en la clase `DibujoArcoIris` para dibujar el arco iris en la pantalla.

```
1  // Fig. 7.26: PruebaDibujoArcoIris.java
2  // Aplicación de prueba para mostrar un arco iris.
3  import javax.swing.JFrame;
4
5  public class PruebaDibujoArcoIris
6  {
7     public static void main( String[] args )
8     {
9        DibujoArcoIris panel = new DibujoArcoIris();
10       JFrame aplicacion = new JFrame();
11
12       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```

Fig. 7.26 | Creación de un objeto `JFrame` para mostrar un arco iris (parte 1 de 2).

```
13   aplicacion.add( panel );
14   aplicacion.setSize( 400, 250 );
15   aplicacion.setVisible( true );
16   } // fin de main
17   } // fin de la clase PruebaDibujoArcoIris
```

Fig. 7.26 | Creación de un objeto `JFrame` para mostrar un arco iris (parte 2 de 2).

Ejercicio del caso de estudio de GUI y gráficos

7.1 (*Dibujo de espirales*) En este ejercicio, dibujará espirales con los métodos `drawLine` y `drawArc`.

a) Dibuje una espiral con forma cuadrada (como en la captura de pantalla izquierda de la figura 7.27), centrada en el panel, con el método `drawLine`. Una técnica es utilizar un ciclo que incremente la longitud de la línea después de dibujar cada segunda línea. La dirección en la cual se dibujará la siguiente línea debe ir después de un patrón distinto, como abajo, izquierda, arriba, derecha.

b) Dibuje una espiral circular (como en la captura de pantalla derecha de la figura 7.27), use el método `drawArc` para dibujar un semicírculo a la vez. Cada semicírculo sucesivo deberá tener un radio más grande (según lo especificado mediante la anchura del rectángulo delimitador) y debe seguir dibujando en donde terminó el semicírculo anterior.

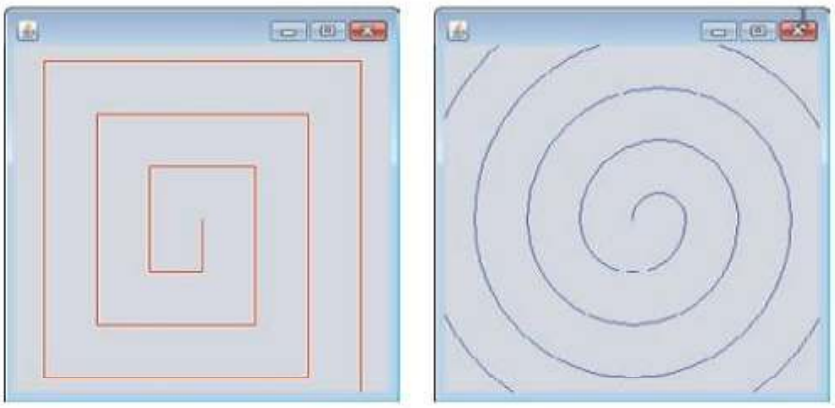


Fig. 7.27 | Dibujo de una espiral con `drawLine` (izquierda) y `drawArc` (derecha).

7.16 Conclusión

En este capítulo empezó nuestra introducción a las estructuras de datos, con la exploración del uso de los arreglos para almacenar datos y obtenerlos de listas y tablas de valores. Los ejemplos de este capítulo

demostraron cómo declarar un arreglo, inicializarlo y hacer referencia a los elementos individuales del mismo. Se introdujo la instrucción `for` mejorada para iterar a través de los arreglos. Utilizamos el manejo de excepciones para evaluar excepciones `ArrayIndexOutOfBoundsException` que ocurren cuando un programa trata de acceder al elemento de un arreglo que se encuentra fuera de sus límites. También le mostramos cómo pasar arreglos a los métodos, y cómo declarar y manipular arreglos multidimensionales. Por último, en este capítulo se demostró cómo escribir métodos que utilizan listas de argumentos de longitud variable, y cómo leer argumentos que se pasan a un programa desde la línea de comandos.

Presentamos la colección de genéricos `ArrayList<T>`, que provee toda la funcionalidad y el rendimiento de los arreglos, junto con otras herramientas útiles, tales como el ajuste de tamaño en forma dinámica. Utilizamos los métodos `add` para agregar nuevos elementos al final de un objeto `ArrayList` y para insertar elementos en un objeto `ArrayList`. Se utilizó el método `remove` para eliminar la primera ocurrencia de un elemento especificado, y se utilizó una versión sobrecargada de `remove` para eliminar un elemento en un subíndice especificado. Utilizamos el método `size` para obtener el número de elementos en el objeto `ArrayList`.

Continuaremos con nuestra cobertura de las estructuras de datos en el capítulo 20 (en inglés, en el sitio Web del libro). Este capítulo introduce el Java Collections Framework (Marco de trabajo de colecciones de Java), que utiliza los genéricos para permitir a los programadores especificar los tipos exactos de objetos que almacenará una estructura de datos específica. El capítulo 20 también presenta las otras estructuras de datos predefinidas de Java. La API Collections proporciona la clase `Arrays`, que contiene métodos utilitarios para la manipulación de arreglos. Ese capítulo 20 utiliza varios métodos `static` de la clase `Arrays` para realizar manipulaciones, como ordenar y buscar en los datos de un arreglo. Después de leer este capítulo podrá utilizar algunos de los métodos de `Arrays` que se describen en el capítulo 20, pero hay otros métodos de `Arrays` que requieren un conocimiento sobre los conceptos que presentaremos más adelante en este libro. El capítulo 21 (en inglés, en el sitio Web del libro) presenta el tema de los genéricos, que proveen los medios para crear modelos generales de métodos y clases que se pueden declarar una vez, pero se utilizan con muchos tipos de datos distintos. El capítulo 22 (también en inglés, en el sitio Web del libro) muestra cómo crear estructuras de datos dinámicas, como listas, colas, pilas y árboles, que pueden aumentar y reducir su tamaño a medida que se ejecutan los programas.

Ya le hemos presentado los conceptos básicos de las clases, los objetos, las instrucciones de control, los métodos, los arreglos y las colecciones. En el capítulo 8 analizaremos con más detalle las clases y los objetos.

Resumen

Sección 7.1 Introducción

- Los arreglos (pág. 241) son estructuras de datos de longitud fija que consisten en elementos de datos relacionados del mismo tipo.

Sección 7.2 Arreglos

- Un arreglo es un grupo de variables (llamadas elementos o componentes; pág. 242) que contienen valores, todos con el mismo tipo. Los arreglos son objetos, por lo cual se consideran como tipos por referencia.
- Un programa hace referencia a cualquiera de los elementos de un arreglo mediante una expresión de acceso a un arreglo (pág. 242), la cual incluye el nombre del arreglo, seguido del subíndice del elemento específico entre corchetes (`[ ]`; pág. 242).
- El primer elemento en cada arreglo tiene el subíndice cero (pág. 242), y algunas veces se le llama el elemento cero.
- Un subíndice debe ser un entero no negativo. Un programa puede utilizar una expresión como un subíndice.
- Un objeto tipo arreglo conoce su propia longitud, y almacena esta información en una variable de instancia `length` (pág. 242).

Sección 7.3 Declaración y creación de arreglos

- Para crear un objeto tipo arreglo, hay que especificar el tipo de los elementos del arreglo y el número de elementos como parte de una expresión de creación de arreglo (pág. 243), que utiliza la palabra clave `new`.
- Cuando se crea un arreglo, cada elemento recibe un valor predeterminado: cero para los elementos numéricos de tipo primitivo, `false` para los elementos booleanos y `null` para las referencias.
- En la declaración de un arreglo, su tipo y los corchetes pueden combinarse al principio de la declaración, para indicar que todos los identificadores en la declaración son variables tipo arreglo.
- Cada elemento de un arreglo de tipo primitivo contiene una variable del tipo declarado del arreglo. Cada elemento de un tipo por referencia es una alusión a un objeto del tipo declarado del arreglo.

Sección 7.4 Ejemplos acerca del uso de los arreglos

- Un programa puede crear un arreglo e inicializar sus elementos con un inicializador de arreglos (pág. 245).
- Las variables constantes (pág. 247) se declaran con la palabra clave `final`, deben inicializarse antes de utilizarlas, y no pueden modificarse de ahí en adelante.
- Cuando se ejecuta un programa en Java, la JVM comprueba los subíndices de los arreglos para asegurarse que sean mayores o iguales a 0 y menores que la longitud del arreglo. Si un programa utiliza un subíndice inválido, Java genera algo que se conoce como excepción (pág. 253), para indicar que ocurrió un error en el programa, en tiempo de ejecución.
- Cuando se ejecuta un programa, se comprueba la validez de los subíndices de los elementos de los arreglos; todos los subíndices deben ser mayores o iguales a 0 y menores que la longitud del arreglo. Si se produce un intento por utilizar un subíndice inválido para acceder a un elemento, ocurre una excepción `ArrayIndexOutOfRangeException` (pág. 253).
- Una excepción indica un problema que ocurre mientras se ejecuta un programa. El nombre "excepción" sugiere que el problema ocurre con poca frecuencia; si la "regla" es que por lo general una instrucción se ejecuta en forma correcta, entonces el problema representa la "excepción a la regla".
- El manejo de excepciones (pág. 253) nos permite crear programas tolerantes a fallas.
- Para manejar una excepción, hay que colocar cualquier código que podría lanzar una excepción (pág. 253) en una instrucción `try`.
- El bloque `try` (pág. 253) contiene el código que podría lanzar una excepción, y el bloque `catch` (pág. 253) contiene el código que maneja la excepción, en caso de que ocurra una.
- Es posible tener muchos bloques `catch` para manejar distintos tipos de excepciones que podrían lanzarse en el bloque `try` correspondiente.
- Cuando termina un bloque `try`, cualquier variable declarada en el bloque `try` queda fuera de alcance.
- Un bloque `catch` declara un tipo y un parámetro de excepción. Dentro del bloque `catch`, es posible usar el identificador del parámetro para interactuar con un objeto excepción atrapado.
- El método `toString` de un objeto excepción devuelve el mensaje de error de la excepción.

Sección 7.5 Caso de estudio: simulación para barajar y repartir cartas

- El método `toString` de un objeto se llama de manera implícita cuando el objeto se utiliza en donde se espera un objeto `String` (por ejemplo, cuando `printf` imprime el objeto como un valor `String` mediante el uso del especificador de formato `%s` o cuando el objeto se concatena con un `String` mediante el operador +).

Sección 7.6 Instrucción `for` *mejorada*

- La instrucción `for` mejorada (pág. 258) nos permite iterar a través de los elementos de un arreglo o de una colección, sin utilizar un contador. La sintaxis de una instrucción `for` mejorada es:

```
for (parámetro : nombreArreglo)
   instrucción
```

- en donde *parámetro* tiene un tipo y un identificador (por ejemplo, `int numero`), y *nombreArreglo* es el arreglo a través del cual se iterará.

- La instrucción for mejorada no puede usarse para modificar los elementos de un arreglo. Si un programa necesita modificar elementos, use la instrucción for tradicional, controlada por contador.

Sección 7.7 Paso de arreglos a los métodos

- Cuando un argumento se pasa por valor, se hace una copia del valor del argumento y se transfiere al método que se llamó. Este método trabaja exclusivamente con la copia.
- Cuando se pasa un argumento por referencia (pág. 262), el método al que se llamó puede acceder al valor del argumento en el método que lo llamó directamente, y es posible modificarlo.
- Todos los argumentos en Java se pasan por valor. Una llamada a un método puede transferir dos tipos de valores a un método: copias de valores primitivos y copias de referencias a objetos. Aunque la referencia a un objeto se pasa por valor (pág. 262), un método de todas formas puede interactuar con el objeto referenciado, llamando a sus métodos public mediante el uso de la copia de la referencia al objeto.
- Para pasar a un método una referencia a un objeto, sólo se especifica en la llamada al método el nombre de la variable que hace referencia al objeto.
- Cuando se pasa a un método un arreglo o un elemento individual del arreglo de un tipo por referencia, el método que se llamó recibe una copia del arreglo o referencia al elemento. Cuando se pasa un elemento individual de un tipo primitivo, el método que se llamó recibe una copia del valor del elemento.
- Para pasar un elemento individual de un arreglo a un método, use el nombre indexado del arreglo.

Sección 7.9 Arreglos multidimensionales

- Los arreglos multidimensionales con dos dimensiones se utilizan a menudo para representar tablas de valores, que consisten en información ordenada en filas y columnas.
- Un arreglo bidimensional (pág. 268) con *m* filas y *n* columnas se llama arreglo de *m* por *n*. Dicho arreglo puede inicializarse con un inicializador de arreglos, de la forma

  tipoArreglo[][] *nombreArreglo* = { { *inicializador fila 1* }, { *inicializador fila 2* }, ... };
- Los arreglos multidimensionales se mantienen como arreglos de arreglos unidimensionales separados. Como resultado, no es obligatorio que las longitudes de las filas en un arreglo bidimensional sean iguales.
- Un arreglo multidimensional con el mismo número de columnas en cada fila se puede crear mediante una expresión de creación de arreglos de la forma

  tipoArreglo[][] *nombreArreglo* = new *tipoArreglo*[*numFilas*][*numColumnas*];

Sección 7.11 Listas de argumentos de longitud variable

- Un tipo de argumento seguido por una elipsis (...; pág. 278) en la lista de parámetros de un método indica que éste recibe un número variable de argumentos de ese tipo específico. La elipsis puede ocurrir sólo una vez en la lista de parámetros de un método. Debe estar al final de la lista.
- Una lista de argumentos de longitud variable (pág. 278) se trata como un arreglo dentro del cuerpo del método. El número de argumentos en el arreglo se puede obtener mediante el campo length del arreglo.

Sección 7.12 Uso de argumentos de línea de comandos

- Para pasar argumentos a main (pág. 279) desde la línea de comandos, se incluye un parámetro de tipo String[] en la lista de parámetros de main. Por convención, el parámetro de main se llama args.
- Java pasa los argumentos de línea de comandos que aparecen después del nombre de la clase en el comando java al método main de la aplicación, en forma de objetos String en el arreglo args.

Sección 7.13 La clase Arrays

- La clase Arrays (pág. 281) provee métodos static que realizan manipulaciones comunes de arreglos, entre ellos sort para ordenar un arreglo, binarySearch para buscar en un arreglo ordenado, equals para comparar arreglos y fill para colocar elementos en un arreglo.

- El método `arraycopy` de la clase `System` (pág. 281) nos permite copiar los elementos de un arreglo en otro.

Sección 7.14 Introducción a las colecciones y la clase `ArrayList`

- Las clases de colecciones de la API de Java proveen métodos eficientes para organizar, almacenar y obtener datos sin tener que saber cómo se almacenan.
- Un `ArrayList<T>` (pág. 284) es similar a un arreglo, sólo que su tamaño se puede ajustar en forma dinámica.
- El método `add` (pág. 286) con un argumento adjunta un elemento al final de un objeto `ArrayList`.
- El método `add` con dos argumentos inserta un nuevo elemento en una posición especificada de un objeto `ArrayList`.
- El método `size` (pág. 286) devuelve el número actual de elementos que se encuentran en un objeto `ArrayList`.
- El método `remove`, con una referencia a un objeto como argumento, elimina el primer elemento que coincide con el valor del argumento.
- El método `remove`, con un argumento entero, elimina el elemento en el índice especificado, y todos los elementos arriba de ese subíndice se desplazan una posición hacia abajo.
- El método `contains` devuelve `true` si el elemento se encuentra en el objeto `ArrayList`, y `false` en caso contrario.

Ejercicios de autoevaluación

7.1 Complete las siguientes oraciones:

a) Las listas y tablas de valores pueden guardarse en _____ .
b) Un arreglo es un grupo de _____ (llamados elementos o componentes) que contiene valores, todos con el mismo _____ .
c) La _____ permite a los programadores iterar a través de los elementos en un arreglo, sin utilizar un contador.
d) El número utilizado para referirse a un elemento específico de un arreglo se conoce como el _____ de ese elemento.
e) Un arreglo que utiliza dos subíndices se conoce como un arreglo _____ .
f) Use la instrucción `for` mejorada _____ para recorrer el arreglo `double` llamado `numeros`.
g) Los argumentos de línea de comandos se almacenan en _____ .
h) Use la expresión _____ para recibir el número total de argumentos en una línea de comandos. Suponga que los argumentos de línea de comandos se almacenan en el objeto `String[] args`.
i) Dado el comando `java MiClase prueba`, el primer argumento de línea de comandos es _____ .
j) Un(a) _____ en la lista de parámetros de un método indica que el método puede recibir un número variable de argumentos.

7.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Un arreglo puede guardar muchos tipos distintos de valores.
b) Por lo general, el subíndice de un arreglo debe ser de tipo `float`.
c) Un elemento individual de un arreglo que se pasa a un método y se modifica ahí mismo, contendrá el valor modificado cuando el método llamado termine su ejecución.
d) Los argumentos de línea de comandos se separan por comas.

7.3 Realice las siguientes tareas para un arreglo llamado `fracciones`:

a) Declare una constante llamada `TAMANIO_ARREGLO` que se inicialice con 10.
b) Declare un arreglo con `TAMANIO_ARREGLO` elementos de tipo `double`, e inicialice los elementos con 0.
c) Haga referencia al elemento 4 del arreglo.
d) Asigne el valor `1.667` al elemento 9 del arreglo.
e) Asigne el valor `3.333` al elemento 6 del arreglo.
f) Sume todos los elementos del arreglo, utilizando una instrucción `for`. Declare la variable entera `x` como variable de control para el ciclo.

7.4 Realice las siguientes tareas para un arreglo llamado `tabla`:

a) Declare y cree el arreglo como un arreglo entero con tres filas y tres columnas. Suponga que se ha declarado la constante TAMANIO_ARREGLO con el valor de 3.
b) ¿Cuántos elementos contiene el arreglo?
c) Utilice una instrucción for para inicializar cada elemento del arreglo con la suma de sus índices. Suponga que se declaran las variables enteras x y y como variables de control.

7.5 Encuentre y corrija el error en cada uno de los siguientes fragmentos de programa:

a)
```
final int TAMANIO_ARREGLO = 5;
TAMANIO_ARREGLO = 10;
```
b) Suponga que
```
int[] b = new int[ 10 ];
for ( int i = 0; i <= b.length; i++ )
   b[ i ] = 1;
```
c) Suponga que
```
int[][] a = { { 1, 2 }, { 3, 4 } };
a[ 1, 1 ] = 5;
```

Respuestas a los ejercicios de autoevaluación

7.1 a) arreglos. b) variables, tipo. c) instrucción for mejorada. d) índice (o subíndice, o número de posición). e) bidimensional. f) `for ( double d : numeros )`. g) un arreglo de objetos `String`, llamado `args` por convención. h) `args.length`. i) `prueba`. j) elipsis (...).

7.2 a) Falso. Un arreglo sólo puede guardar valores del mismo tipo.
b) Falso. El subíndice de un arreglo debe ser un entero o una expresión entera.
c) Para los elementos individuales de tipo primitivo en un arreglo: falso. Un método al que se llama recibe y manipula una copia del valor de dicho elemento, por lo que las modificaciones no afectan el valor original. No obstante, si se pasa la referencia de un arreglo a un método, las modificaciones a los elementos del arreglo que se hicieron en el método al que se llamó se reflejan sin duda en el original. Para los elementos individuales de un tipo por referencia: verdadero. Un método al que se llama recibe una copia de la referencia de dicho elemento, y los cambios al objeto referenciado se reflejarán en el elemento del arreglo original.
d) Falso. Los argumentos de línea de comandos se separan por espacio en blanco.

7.3 a)
```
final int TAMANIO_ARREGLO = 10;
```
b)
```
double[] fracciones = new double[ TAMANIO_ARREGLO ];
```
c)
```
fracciones[ 4 ]
```
d)
```
fracciones[ 9 ] = 1.667;
```
e)
```
fracciones[ 6 ] = 3.333;
```
f)
```
double total = 0.0;
for ( int x = 0; x < fracciones.length; x++ )
   total += fracciones[ x ];
```

7.4 a)
```
int[][] tabla = new int[ TAMANIO_ARREGLO ][ TAMANIO_ARREGLO ];
```
b) Nueve.
c)
```
for ( int x = 0; x < tabla.length; x++ )
   for ( int y = 0; y < tabla[ x ].length; y++ )
      tabla[ x ][ y ] = x + y;
```

7.5 a) Error: asignar un valor a una constante después de inicializarla.
Corrección: asigne el valor correcto a la constante en una declaración `final int TAMANIO_ARREGLO`, o declare otra variable.
b) Error: se está haciendo referencia al elemento de un arreglo que está fuera de los límites del arreglo (`b[10]`).
Corrección: cambie el operador <= por <.
c) Error: la indexación del arreglo se está realizando en forma incorrecta.
Corrección: cambie la instrucción por `a[ 1 ][ 1 ] = 5;`.

Ejercicios

7.6 Complete las siguientes oraciones:

a) Un arreglo unidimensional `p` contiene cuatro elementos. Los nombres de esos elementos son _____, _____, _____ y _____.

b) Al proceso de nombrar un arreglo, declarar su tipo y especificar el número de dimensiones se le conoce como _____ el arreglo.

c) En un arreglo bidimensional, el primer índice identifica el(la) _____ de un elemento y el segundo identifica el(la) _____ de un elemento.

d) Un arreglo de *m* por *n* contiene _____ filas, _____ columnas y _____ elementos.

e) El nombre del elemento en la fila 3 y la columna 5 del arreglo `d` es _____.

7.7 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Para referirse a una ubicación o elemento específico dentro de un arreglo, especificamos el nombre del arreglo y el valor del elemento específico.

b) La declaración de un arreglo reserva espacio para el mismo.

c) Para indicar que deben reservarse 100 ubicaciones para el arreglo entero `p`, debe escribir la declaración

```
p[ 100 ];
```

d) Una aplicación que inicializa con cero los elementos de un arreglo con 15 elementos debe contener al menos una instrucción `for`.

e) Una aplicación que sume el total de los elementos de un arreglo bidimensional debe contener instrucciones `for` anidadas.

7.8 Escriba instrucciones en Java que realicen cada una de las siguientes tareas:

a) Mostrar el valor del elemento 6 del arreglo `f`.

b) Inicializar con 8 cada uno de los cinco elementos del arreglo entero unidimensional `g`.

c) Sumar el total de los 100 elementos del arreglo `c` de punto flotante.

d) Copiar el arreglo `a` de 11 elementos en la primera porción del arreglo `b`, el cual contiene 34 elementos.

e) Determinar e imprimir los valores menor y mayor contenidos en el arreglo `w` con 99 elementos de punto flotante.

7.9 Considere un arreglo entero `t` de dos por tres.

a) Escriba una instrucción que declare y cree a `t`.

b) ¿Cuántas filas tiene `t`?

c) ¿Cuántas columnas tiene `t`?

d) ¿Cuántos elementos tiene `t`?

e) Escriba expresiones de acceso para todos los elementos en la fila 1 de `t`.

f) Escriba expresiones de acceso para todos los elementos en la columna 2 de `t`.

g) Escriba una sola instrucción que asigne cero al elemento de `t` en la fila 0 y la columna 1.

h) Escriba instrucciones individuales para inicializar cada elemento de `t` con cero.

i) Escriba una instrucción `for` anidada que inicialice cada elemento de `t` con cero.

j) Escriba una instrucción `for` anidada que reciba como entrada del usuario los valores de los elementos de `t`.

k) Escriba una serie de instrucciones que determine e imprima el valor más pequeño en `t`.

l) Escriba una sola instrucción `printf` que muestre los elementos de la primera fila de `t`.

m) Escriba una instrucción que totalice los elementos de la tercera columna de `t`. No utilice repetición.

n) Escriba una serie de instrucciones para imprimir el contenido de `t` en formato tabular. Mencione los índices de columna como encabezados a lo largo de la parte superior, y enumere los índices de fila a la izquierda de cada fila.

7.10 ***(Comisión por ventas)*** Utilice un arreglo unidimensional para resolver el siguiente problema: una compañía paga a sus vendedores por comisión. Los vendedores reciben $200 por semana más el 9% de sus ventas totales de esa semana. Por ejemplo, un vendedor que acumule $5000 en ventas en una semana, recibirá $200 más el 9% de $5000, o un total de

$650. Escriba una aplicación (utilizando un arreglo de contadores) que determine cuántos vendedores recibieron salarios en cada uno de los siguientes rangos (suponga que el salario de cada vendedor se trunca a una cantidad entera):

a) $200–299
b) $300–399
c) $400–499
d) $500–599
e) $600–699
f) $700–799
g) $800–899
h) $900–999
i) $1000 en adelante

Sintetice los resultados en formato tabular.

7.11 Escriba instrucciones que realicen las siguientes operaciones con arreglos unidimensionales:

a) Asignar cero a los 10 elementos del arreglo `cuentas` de tipo entero.
b) Sumar uno a cada uno de los 15 elementos del arreglo `bono` de tipo entero.
c) Imprima los cinco valores del arreglo `mejoresPuntuaciones` de tipo entero en formato de columnas.

7.12 ***(Eliminación de duplicados)*** Use un arreglo unidimensional para resolver el siguiente problema: escriba una aplicación que reciba como entrada cinco números, cada uno de los cuales debe estar entre 10 y 100, inclusive. A medida que se lea cada número, muéstrelo solamente si no es un duplicado de un número que ya se haya leído. Prepárese para el "peor caso", en el que los cinco números son diferentes. Use el arreglo más pequeño que sea posible para resolver este problema. Muestre el conjunto completo de valores únicos introducidos, después de que el usuario introduzca cada nuevo valor.

7.13 Etiquete los elementos del arreglo bidimensional `ventas` de tres por cinco, para indicar el orden en el que se establecen en cero, mediante el siguiente fragmento de programa:

```
for ( int fila = 0; fila < ventas.length; fila++ )
{
   for ( int col = 0; col < ventas[ fila ].length; col++ )
   {
      ventas[ fila ][ col ] = 0;
   }
}
```

7.14 ***(Lista de argumentos de longitud variable)*** Escriba una aplicación que calcule el producto de una serie de enteros que se pasan al método `producto`, use una lista de argumentos de longitud variable. Pruebe su método con varias llamadas, cada una con un número distinto de argumentos.

7.15 ***(Argumentos de línea de comandos)*** Modifique la figura 7.2, de manera que el tamaño del arreglo se especifique mediante el primer argumento de línea de comandos. Si no se suministra un argumento de línea de comandos, use 10 como el valor predeterminado del arreglo.

7.16 ***(Uso de la instrucción for mejorada)*** Escriba una aplicación que utilice una instrucción `for` mejorada para sumar los valores `double` que se pasan mediante los argumentos de línea de comandos. [*Sugerencia:* use el método `static parseDouble` de la clase `Double` para convertir un `String` en un valor `double`].

7.17 ***(Tiro de dados)*** Escriba una aplicación para simular el tiro de dos dados. La aplicación debe utilizar un objeto de la clase `Random` una vez para tirar el primer dado, y de nuevo para tirar el segundo. Después debe calcularse la suma de los dos valores. Cada dado puede mostrar un valor entero del 1 al 6, por lo que la suma de los valores variará del 2 al 12, siendo 7 la suma más frecuente, mientras que 2 y 12 serán las sumas menos frecuentes. En la figura 7.28 se muestran las 36 posibles combinaciones de los dos dados. Su aplicación debe tirar los dados 36,000,000 veces. Utilice un arreglo unidimensional para registrar el número de veces que aparezca cada una de las posibles sumas. Muestre los resultados en formato tabular.

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

Fig. 7.28 | Las 36 posibles sumas de dos dados.

7.18 *(Juego de Craps)* Escriba una aplicación que ejecute 1,000,000 juegos de Craps (figura 6.8) y responda a las siguientes preguntas:

a) ¿Cuántos juegos se ganan en el primer tiro, en el segundo, …, en el vigésimo y después de éste?
b) ¿Cuántos juegos se pierden en el primer tiro, en el segundo, …, en el vigésimo y después de éste?
c) ¿Cuáles son las probabilidades de ganar en Craps? [*Nota:* debe descubrir que Craps es uno de los juegos de casino más justos. ¿Qué cree usted que significa esto?].
d) ¿Cuál es la duración promedio de un juego de Craps?
e) ¿Las probabilidades de ganar mejoran con la duración del juego?

7.19 *(Sistema de reservaciones de una aerolínea)* Una pequeña aerolínea acaba de comprar una computadora para su nuevo sistema de reservaciones automatizado. Se le ha pedido a usted que desarrolle el nuevo sistema. Usted escribirá una aplicación para asignar asientos en cada vuelo del único avión de la aerolínea (capacidad: 10 asientos).

Su aplicación debe mostrar las siguientes alternativas: `Por favor escriba 1 para Primera Clase` y `Por favor escriba 2 para Economico`. Si el usuario escribe `1`, su aplicación debe asignarle un asiento en la sección de primera clase (asientos 1 a 5). Si el usuario escribe `2`, su aplicación debe asignarle un asiento en la sección económica (asientos 6 a 10). Su aplicación deberá entonces imprimir un pase de abordar, indicando el número de asiento de la persona y si se encuentra en la sección de primera clase o clase económica.

Use un arreglo unidimensional del tipo primitivo `boolean` para representar la tabla de asientos del avión. Inicialice todos los elementos del arreglo con `false` para indicar que todos los asientos están vacíos. A medida que se asigne cada asiento, establezca el elemento correspondiente del arreglo en `true` para indicar que ese asiento ya no está disponible.

Su aplicación nunca deberá asignar un asiento que ya haya sido asignado. Cuando esté llena la sección económica, su programa deberá preguntar a la persona si acepta ser colocada en la sección de primera clase (y viceversa). Si la persona acepta, haga la asignación de asiento apropiada. Si no, imprima el mensaje `"El proximo vuelo sale en 3 horas"`.

7.20 *(Ventas totales)* Use un arreglo bidimensional para resolver el siguiente problema: una compañía tiene cuatro vendedores (1 a 4) que venden cinco productos distintos (1 a 5). Una vez al día, cada vendedor pasa una nota por cada tipo de producto vendido. Cada nota contiene lo siguiente:

a) El número del vendedor
b) El número del producto
c) El valor total en dólares de ese producto vendido en ese día

Así, cada vendedor pasa entre 0 y 5 notas de venta por día. Suponga que está disponible la información sobre todas las notas del mes pasado. Escriba una aplicación que lea toda esta información para las ventas del último mes y que resuma las ventas totales por vendedor, y por producto. Todos los totales deben guardarse en el arreglo bidimensional `ventas`. Después de procesar toda la información del mes pasado, muestre los resultados en formato tabular, en donde cada columna represente a un vendedor específico y cada fila simboliza un producto. Saque el total de cada fila para obtener las

ventas totales de cada producto durante el último mes. Calcule el total de cada columna para sacar las ventas totales de cada vendedor durante el último mes. Su impresión tabular debe incluir estos totales cruzados a la derecha de las filas totalizadas, y en la parte inferior de las columnas totalizadas.

7.21 ***(Gráficos de tortuga)*** El lenguaje Logo hizo famoso el concepto de los *gráficos de tortuga*. Imagine a una tortuga mecánica que camina por todo el cuarto, bajo el control de una aplicación en Java. La tortuga sostiene una pluma en una de dos posiciones, arriba o abajo. Mientras la pluma está abajo, el animalito va trazando figuras a medida que se va moviendo, y mientras la pluma está arriba, se mueve alrededor libremente, sin trazar nada. En este problema usted simulará la operación de la tortuga y creará un bloc de dibujo computarizado.

Utilice un arreglo de 20 por 20 llamado piso, que se inicialice con ceros. Lea los comandos de un arreglo que los contenga. Lleve el registro de la posición actual de la tortuga en todo momento, y si la pluma se encuentra arriba o abajo. Suponga que la tortuga siempre empieza en la posición (0, 0) del piso, con su pluma hacia arriba. El conjunto de comandos de la tortuga que su aplicación debe procesar se muestra en la figura 7.29.

Comando	Significado
1	Pluma arriba
2	Pluma abajo
3	Voltear a la derecha
4	Voltear a la izquierda
5,10	Avanzar hacia delante 10 espacios (reemplace el 10 por un número distinto de espacios)
6	Mostrar en pantalla el arreglo de 20 por 20
9	Fin de los datos (centinela)

Fig. 7.29 | Comandos de gráficos de tortuga.

Suponga que la tortuga se encuentra en algún lado cerca del centro del piso. El siguiente "programa" dibuja e imprime un cuadrado de 12 por 12, dejando la pluma en posición levantada:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

A medida que la tortuga se vaya desplazando con la pluma hacia abajo, asigne 1 a los elementos apropiados del arreglo piso. Cuando se dé el comando 6 (mostrar el arreglo en pantalla), siempre que haya un 1 en el arreglo, muestre un asterisco o cualquier carácter que usted elija. Siempre que haya un 0, muestre un carácter en blanco.

Escriba una aplicación para implementar las herramientas de gráficos de tortuga aquí descritas. Escriba varios programas de gráficos de tortuga para dibujar figuras interesantes. Agregue otros comandos para incrementar el poder de su lenguaje de gráficos de tortuga.

7.22 ***(Paseo del caballo)*** Un enigma interesante para los entusiastas del ajedrez es el problema del Paseo del caballo, propuesto originalmente por el matemático Euler. ¿Puede la pieza de ajedrez, conocida como caballo, moverse alrededor de un tablero de ajedrez vacío y tocar cada una de las 64 posiciones una y sólo una vez? A continuación estudiaremos este intrigante problema con detalle.

El caballo realiza solamente movimientos en forma de L (dos espacios en una dirección y uno en una dirección perpendicular). Por lo tanto, como se muestra en la figura 7.30, desde una posición cerca del centro de un tablero de ajedrez vacío, el caballo (etiquetado como C) puede hacer ocho movimientos distintos (numerados del 0 al 7).

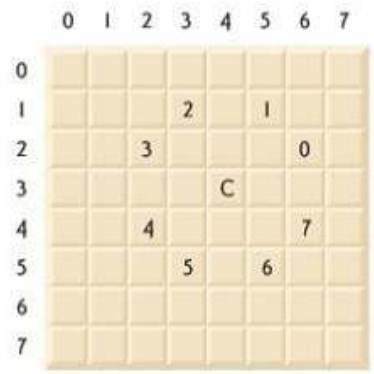


Fig. 7.30 | Los ocho posibles movimientos del caballo.

a) Dibuje un tablero de ajedrez de ocho por ocho en una hoja de papel, e intente realizar un Paseo del caballo en forma manual. Ponga un `1` en la posición inicial, un `2` en la segunda posición, un `3` en la tercera y así en lo sucesivo. Antes de empezar el paseo, estime qué tan lejos podrá avanzar, recuerde que un paseo completo consta de 64 movimientos. ¿Qué tan lejos llegó? ¿Estuvo esto cerca de su estimación?

b) Ahora desarrollaremos una aplicación para mover el caballo alrededor de un tablero de ajedrez. El tablero estará representado por un arreglo bidimensional de ocho por ocho, llamado tablero. Cada posición se inicializa con cero. Describiremos cada uno de los ocho posibles movimientos en términos de sus componentes horizontales y verticales. Por ejemplo, un movimiento de tipo 0, como se muestra en la figura 7.30, consiste en mover dos posiciones en forma horizontal a la derecha y una posición vertical hacia arriba. Un movimiento de tipo 2 consiste en mover una posición horizontalmente a la izquierda y dos posiciones verticales hacia arriba. Los movimientos horizontal a la izquierda y vertical hacia arriba se indican con números negativos. Los ocho movimientos pueden describirse mediante dos arreglos unidimensionales llamados `horizontal` y `vertical`, de la siguiente manera:

```
horizontal[ 0 ] = 2      vertical[ 0 ] = -1
horizontal[ 1 ] = 1      vertical[ 1 ] = -2
horizontal[ 2 ] = -1     vertical[ 2 ] = -2
horizontal[ 3 ] = -2     vertical[ 3 ] = -1
horizontal[ 4 ] = -2     vertical[ 4 ] = 1
horizontal[ 5 ] = -1     vertical[ 5 ] = 2
horizontal[ 6 ] = 1      vertical[ 6 ] = 2
horizontal[ 7 ] = 2      vertical[ 7 ] = 1
```

Deje que las variables `filaActual` y `columnaActual` indiquen la fila y columna, respectivamente, de la posición actual del caballo. Para hacer un movimiento de tipo `numeroMovimiento`, en donde `numeroMovimiento` puede estar entre 0 y 7, su programa debe utilizar las instrucciones

```
filaActual += vertical[ numeroMovimiento ];
columnaActual += horizontal[ numeroMovimiento];
```

Escriba una aplicación para mover el caballo alrededor del tablero de ajedrez. Utilice un contador que varíe de 1 a 64. Registre la última cuenta en cada posición a la que se mueva el caballo. Evalúe cada movimiento potencial para ver si el caballo ya visitó esa posición. Pruebe cada movimiento potencial para asegurarse que el caballo no se salga del tablero de ajedrez. Ejecute la aplicación. ¿Cuántos movimientos hizo el caballo?

c) Después de intentar escribir y ejecutar una aplicación de Paseo del caballo, probablemente haya desarrollado algunas ideas valiosas. Utilizaremos estas ideas para desarrollar una *heurística* (o regla empírica) para mover el caballo. La heurística no garantiza el triunfo, pero una heurística desarrollada con cuidado mejora de manera considerable la probabilidad de tener éxito. Tal vez usted ya observó que las posiciones externas son más difíciles que las cercanas al centro del tablero. De hecho, las posiciones más difíciles o inaccesibles son las cuatro esquinas.

La intuición sugiere que usted debe intentar mover primero el caballo a las posiciones más problemáticas y dejar pendientes aquellas a las que sea más fácil llegar, de manera que cuando el tablero se congestione cerca del final del paseo, habrá una mayor probabilidad de éxito.

Podríamos desarrollar una "heurística de accesibilidad" al clasificar cada una de las posiciones de acuerdo a qué tan accesibles son y luego mover siempre el caballo (usando los movimientos en L) a la posición más inaccesible. Etiquetaremos un arreglo bidimensional llamado `accesibilidad` con números que indiquen desde cuántas posiciones es accesible una posición determinada. En un tablero de ajedrez en blanco, cada una de las 16 posiciones más cercanas al centro se clasifican con `8`; cada posición en la esquina se clasifica con `2`; y las demás posiciones tienen números de accesibilidad `3`, `4` o `6`, de la siguiente manera:

```
2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 4 3
2 3 4 4 4 4 3 2
```

Escriba una nueva versión del Paseo del caballo, use la heurística de accesibilidad. El caballo deberá moverse siempre a la posición con el número de accesibilidad más bajo. En caso de un empate, el caballo podrá moverse a cualquiera de las posiciones empatadas. Por lo tanto, el paseo puede empezar en cualquiera de las cuatro esquinas. [*Nota:* al ir moviendo el caballo alrededor del tablero, su aplicación deberá reducir los números de accesibilidad a medida que se vayan ocupando más posiciones. De esta manera y en cualquier momento dado durante el paseo, el número de accesibilidad de cada una de las posiciones disponibles seguirá siendo igual al número preciso de posiciones desde las que se puede llegar a esa posición.] Ejecute esta versión de su aplicación. ¿Logró completar el paseo? Modifique el programa para realizar 64 paseos, en donde cada uno empiece desde una posición distinta en el tablero. ¿Cuántos paseos completos logró realizar?

d) Escriba una versión del programa del Paseo del caballo que, al encontrarse con un empate entre dos o más posiciones, decida qué posición elegir, más adelante busque aquellas posiciones que se puedan alcanzar desde las posiciones "empatadas". Su aplicación debe mover el caballo a la posición empatada para la cual el siguiente movimiento lo lleve a una posición con el número de accesibilidad más bajo.

7.23 ***(Paseo del caballo: métodos de fuerza bruta)*** En la parte (c) del ejercicio 7.22, desarrollamos una solución al problema del Paseo del caballo. El método utilizado, llamado "heurística de accesibilidad", genera muchas soluciones y se ejecuta con eficiencia.

A medida que se incremente de manera continua la potencia de las computadoras, seremos capaces de resolver más problemas con menos potencia y algoritmos relativamente menos sofisticados. A éste le podemos llamar el método de la "fuerza bruta" para resolver problemas.

a) Utilice la generación de números aleatorios para permitir que el caballo se desplace a lo largo del tablero (mediante sus movimientos legítimos en L) en forma aleatoria. Su aplicación debe ejecutar un paseo e imprimir el tablero final. ¿Qué tan lejos llegó el caballo?

b) La mayoría de las veces, la aplicación de la parte (a) produce un paseo relativamente corto. Ahora modifique su aplicación para intentar 1000 paseos. Use un arreglo unidimensional para llevar el registro del número de paseos de cada longitud. Cuando su programa termine de intentar los 1000 paseos, deberá imprimir esta información en un formato tabular ordenado. ¿Cuál fue el mejor resultado?

c) Es muy probable que la aplicación de la parte (b) le haya brindado algunos paseos "respetables", pero no completos. Ahora deje que su aplicación se ejecute hasta que produzca un paseo completo. [*Precaución:* esta versión del programa podría ejecutarse durante horas en una computadora poderosa.] Una vez más, mantenga una tabla del número de paseos de cada longitud e imprímala cuando se encuentre el primer paseo completo. ¿Cuántos paseos intentó su programa antes de producir uno completo? ¿Cuánto tiempo se tomó?

d) Compare la versión de la fuerza bruta del Paseo del caballo con la versión heurística de accesibilidad. ¿Cuál requirió un estudio más cuidadoso del problema? ¿Qué algoritmo fue más difícil de desarrollar? ¿Cuál necesitó más poder de cómputo? ¿Podríamos tener la certeza (por adelantado) de obtener un paseo completo mediante el método de la heurística de accesibilidad? ¿Podríamos tener la certeza (por adelantado) de obtener un paseo completo mediante el método de la fuerza bruta? Argumente las ventajas y desventajas de solucionar el problema mediante la fuerza bruta en general.

7.24 ***(Ocho reinas)*** Otro enigma para los entusiastas del ajedrez es el problema de las Ocho reinas, el cual pregunta lo siguiente: ¿es posible colocar ocho reinas en un tablero de ajedrez vacío, de tal manera que ninguna "ataque" a cualquier otra (es decir, que no haya dos reinas en la misma fila, en la misma columna o a lo largo de la misma diagonal)? Use la idea desarrollada en el ejercicio 7.22 para formular una heurística que resuelva el problema de las Ocho reinas. Ejecute su aplicación. [*Sugerencia:* es posible asignar un valor a cada una de las posiciones en el tablero de ajedrez, para indicar cuántas posiciones de un tablero vacío se "eliminan" si una reina se coloca en esa posición. A cada una de las esquinas se le asignaría el valor 22, como se demuestra en la figura 7.31. Una vez que estos "números de eliminación" se coloquen en las 64 posiciones, una heurística apropiada podría ser: coloque la siguiente reina en la posición con el número de eliminación más pequeño. ¿Por qué esta estrategia es intuitivamente atractiva?]

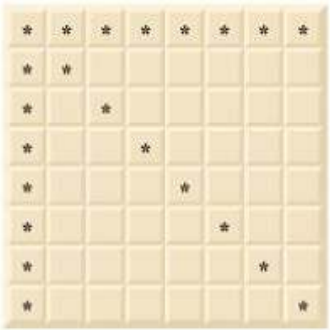


Fig. 7.31 | Las 22 posiciones eliminadas al colocar una reina en la esquina superior izquierda.

7.25 ***(Ocho reinas: métodos de fuerza bruta)*** En este ejercicio usted desarrollará varios métodos de fuerza bruta para resolver el problema de las Ocho reinas que presentamos en el ejercicio 7.24.

a) Utilice la técnica de la fuerza bruta aleatoria desarrollada en el ejercicio 7.23, para resolver el problema de las Ocho reinas.

b) Utilice una técnica exhaustiva (es decir, pruebe todas las combinaciones posibles de las ocho reinas en el tablero) para resolver el problema de las Ocho reinas.

c) ¿Por qué el método de la fuerza bruta exhaustiva podría no ser apropiado para resolver el problema del Paseo del caballo?

d) Compare y contraste el método de la fuerza bruta aleatoria con el de la fuerza bruta exhaustiva.

7.26 ***(Paseo del caballo: prueba del paseo cerrado)*** En el Paseo del caballo (ejercicio 7.22), se lleva a cabo un paseo completo cuando el caballo hace 64 movimientos, en los que toca cada esquina del tablero una sola vez. Un paseo cerrado ocurre cuando el movimiento 64 se encuentra a un movimiento de distancia de la posición en la que el caballo empezó el

paseo. Modifique la aplicación que escribió en el ejercicio 7.22 para probar si el paseo ha sido completo, y si se trató de un paseo cerrado.

7.27 ***(La criba de Eratóstenes)*** Un número primo es cualquier entero mayor que 1, divisible sólo por sí mismo y por el número 1. La Criba de Eratóstenes es un método para encontrar números primos, el cual opera de la siguiente manera:

a) Cree un arreglo del tipo primitivo `boolean`, con todos los elementos inicializados en `true`. Los elementos del arreglo con índices primos permanecerán como `true`. Cualquier otro elemento del arreglo con el tiempo cambiará a `false`.

b) Empiece con el índice 2 del arreglo y determine si un elemento dado es `true`. De ser así, itere a través del resto del arreglo y asigne `false` a todo elemento cuyo índice sea múltiplo del índice del elemento que tiene el valor `true`. Después continúe el proceso con el siguiente elemento que tenga el valor `true`. Para el índice 2 del arreglo, todos los elementos más allá del elemento 2 en el arreglo que tengan índices múltiplos de 2 (los índices 4, 6, 8, 10, etcétera) se establecerán en `false`; para el índice 3 del arreglo, todos los elementos más allá del elemento 3 en el arreglo que tengan índices múltiplos de 3 (los índices 6, 9, 12, 15, etcétera) se establecerán en `false`; y así en lo sucesivo.

Cuando este proceso termine, los elementos del arreglo que aún sean `true` indicarán que el índice es un número primo. Estos índices pueden mostrarse. Escriba una aplicación que utilice un arreglo de 1000 elementos para determinar e imprimir los números primos entre 2 y 999. Ignore los elementos 0 y 1 del arreglo.

7.28 ***(Simulación: La tortuga y la liebre)*** En este problema, usted recreará la clásica carrera de la tortuga y la liebre. Utilizará la generación de números aleatorios para desarrollar una simulación de este memorable suceso.

Nuestros competidores empezarán la carrera en la posición 1 de 70 posiciones. Cada una representa a una posible posición a lo largo del curso de la carrera. La línea de meta se encuentra en la 70. El primer competidor en llegar a la posición 70 o más allá recibirá una cubeta llena con zanahorias y lechuga frescas. El recorrido se abre paso hasta la cima de una resbalosa montaña, por lo que en ocasiones los competidores pierden terreno.

Un reloj hace tictac una vez por segundo. Con cada tic del reloj, su aplicación debe ajustar la posición de los animales de acuerdo con las reglas de la figura 7.32. Use variables para llevar el registro de las posiciones de los animales (los números son del 1 al 70). Empiece con cada animal en la posición 1 (la "puerta de inicio"). Si un animal se resbala hacia la izquierda antes de la posición 1, regréselo a la posición 1.

Animal	Tipo de movimiento	Porcentaje del tiempo	Movimiento actual
Tortuga	Paso pesado rápido	50%	3 posiciones a la derecha
	Resbalón	20%	6 posiciones a la izquierda
	Paso pesado lento	30%	1 posición a la derecha
Liebre	Dormir	20%	Ningún movimiento
	Gran salto	20%	9 posiciones a la derecha
	Gran resbalón	10%	12 posiciones a la izquierda
	Pequeño salto	30%	1 posición a la derecha
	Pequeño resbalón	20%	2 posiciones a la izquierda

Fig. 7.32 | Reglas para ajustar las posiciones de la tortuga y la liebre.

Genere los porcentajes en la figura 7.32 al producir un entero aleatorio i en el rango $1 \leq i \leq 10$. Para la tortuga, realice un "paso pesado rápido" cuando $1 \leq i \leq 5$, un "resbalón" cuando $6 \leq i \leq 7$ o un "paso pesado lento" cuando $8 \leq i \leq 10$. Utilice una técnica similar para mover a la liebre.

Empiece la carrera imprimiendo el mensaje

```
PUM !!!!!
Y ARRANCAN !!!!!
```

Luego, para cada tic del reloj (es decir, cada repetición de un ciclo) imprima una línea de 70 posiciones, mostrando la letra `T` en la posición de la tortuga y la letra `H` en la posición de la liebre. En ocasiones los competidores se encontrarán en la misma posición. En este caso, la tortuga muerde a la liebre y su aplicación debe imprimir `OUCH!!!` empezando en esa posición. Todas las posiciones de impresión distintas de la `T`, la `H` o el mensaje `OUCH!!!` (en caso de un empate) deben estar en blanco.

Después de imprimir cada línea, compruebe si uno de los animales ha llegado o se ha pasado de la posición 70. De ser así, imprima quién fue el ganador y termine la simulación. Si la tortuga gana, imprima `LA TORTUGA GANA!!! YAY!!!` Si la liebre gana, imprima `La liebre gana. Que mal.` Si ambos animales ganan en el mismo tic del reloj, tal vez usted quiera favorecer a la tortuga (la más débil) o tal vez quiera imprimir `Es un empate.` Si ninguno de los dos animales gana, ejecute el ciclo de nuevo para simular el siguiente tic del reloj. Cuando esté listo para ejecutar su aplicación, reúna a un grupo de aficionados para que vean la carrera. ¡Se sorprenderá al ver lo participativa que puede ser su audiencia!

Más adelante en el libro presentaremos una variedad de herramientas de Java, como gráficos, imágenes, animación, sonido y multihilos. Cuando estudie esas herramientas, tal vez disfrute al mejorar su simulación de la tortuga y la liebre.

7.29 ***(Serie de Fibonacci)*** La serie de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

empieza con los términos 0 y 1, y tiene la propiedad de que cada término sucesivo es la suma de los dos términos anteriores.

a) Escriba un método llamado `fibonacci(n)` que calcule el enésimo número de Fibonacci. Incorpore este método en una aplicación que permita al usuario introducir el valor de `n`.
b) Determine el número de Fibonacci más grande que puede imprimirse en su sistema.
c) Modifique la aplicación que escribió en la parte (a), de manera que utilice `double` en vez de `int` para calcular y devolver números de Fibonacci, y utilice esta aplicación modificada para repetir la parte (b).

Los ejercicios 7.30 a 7.33 son de una complejidad razonable. Una vez que haya resuelto estos problemas, obtendrá la capacidad de implementar la mayoría de los juegos populares de cartas con facilidad.

7.30 ***(Barajar y repartir cartas)*** Modifique la aplicación de la figura 7.11 para repartir una mano de póquer de cinco cartas. Después modifique la clase `PaqueteDeCartas` de la figura 7.10 para incluir métodos que determinen si una mano contiene

a) un par
b) dos pares
c) tres de un mismo tipo (como tres jotos)
d) cuatro de un mismo tipo (como cuatro ases)
e) una corrida (es decir, las cinco cartas del mismo palo)
f) una escalera (es decir, cinco cartas de valor consecutivo de la misma cara)
g) "full house" (es decir, dos cartas de un valor de la misma cara y tres cartas de otro valor de la misma cara)

[*Sugerencia:* agregue los métodos `obtenerCara` y `obtenerPalo` a la clase `Carta` de la figura 7.9.]

7.31 ***(Barajar y repartir cartas)*** Use los métodos desarrollados en el ejercicio 7.30 para escribir una aplicación que reparta dos manos de póquer de cinco cartas, que evalúe cada mano y determine cuál de las dos es mejor.

7.32 ***(Proyecto: barajar y repartir cartas)*** Modifique la aplicación desarrollada en el ejercicio 7.31, de manera que pueda simular el repartidor. La mano de cinco cartas del repartidor se reparte "cara abajo", por lo que el jugador no puede verla. A continuación, la aplicación debe evaluar la mano del repartidor y, con base en la calidad de ésta, debe sacar una, dos

o tres cartas más para reemplazar el número correspondiente de cartas que no necesita en la mano original. Después, la aplicación debe reevaluar la mano del repartidor. [Precaución: ¡éste es un problema difícil!]

7.33 ***(Proyecto: barajar y repartir cartas)*** Modifique la aplicación desarrollada en el ejercicio 7.32, de manera que pueda encargarse de la mano del repartidor en forma automática, pero debe permitir al jugador decidir cuáles cartas de su mano desea reemplazar. A continuación, la aplicación deberá evaluar ambas manos y determinar quién gana. Ahora utilice esta nueva aplicación para jugar 20 manos contra la computadora. ¿Quién gana más juegos, usted o la computadora? Haga que un amigo juegue 20 manos contra la computadora. ¿Quién gana más juegos? Con base en los resultados de estos juegos, refine su aplicación para jugar póquer. (Esto también es un problema difícil). Juegue 20 manos más. ¿Su aplicación modificada hace un mejor juego?

7.34 ***(Proyecto: barajar y repartir cartas)*** Modifique la aplicación de las figuras 7.9 a 7.11 para usar enumeraciones `Cara` y `Palo` que representen las caras y los palos de las cartas. Declare cada una de estas enumeraciones como un tipo `public` en su propio archivo de código fuente. Cada `Carta` debe tener las variables de instancia `Cara` y `Palo`. Éstas se deben inicializar mediante el constructor de `Carta`. En la clase `PaqueteDeCartas`, cree un arreglo de objetos `Cara` que se inicialice con los nombres de las constantes en la enumeración `Cara`, y un arreglo de objetos `Palo` que se inicialice con los nombres de las constantes en la enumeración `Palo`. [*Nota:* al imprimir en pantalla una constante de enumeración como un valor `String`, se muestra el nombre de la constante].

Sección especial: construya su propia computadora

En los siguientes problemas, nos desviaremos temporalmente del mundo de la programación en lenguajes de alto nivel, para "abrir de par en par" una computadora y ver su estructura interna. Presentaremos la programación en lenguaje máquina y escribiremos varios programas en este lenguaje. Para que ésta sea una experiencia valiosa, crearemos también una computadora (mediante la técnica de la *simulación* basada en software) en la que pueda ejecutar sus programas en lenguaje máquina.

7.35 ***(Programación en lenguaje máquina)*** Crearemos una computadora a la que llamaremos Simpletron. Como su nombre lo indica, es una máquina simple, pero poderosa. Simpletron sólo ejecuta programas escritos en el único lenguaje que entiende directamente: el lenguaje máquina de Simpletron (LMS).

Simpletron contiene un *acumulador*: un registro especial en el cual se coloca la información antes de que Simpletron la utilice en los cálculos, o que la analice de distintas maneras. Toda la información dentro de Simpletron se manipula en términos de *palabras*. Una palabra es un número decimal con signo de cuatro dígitos, como `+3364`, `-1293`, `+0007` y `-0001`. Simpletron está equipada con una memoria de 100 palabras, y se hace referencia a ellas mediante sus números de ubicación `00`, `01`, . . . , `99`.

Antes de ejecutar un programa LMS debemos *cargar*, o colocar, el programa en memoria. La primera instrucción de cada programa LMS se coloca siempre en la ubicación `00`. El simulador empezará a ejecutarse en esta ubicación.

Cada instrucción escrita en LMS ocupa una palabra de la memoria de Simpletron (y por lo tanto, las instrucciones son números decimales de cuatro dígitos con signo). Supondremos que el signo de una instrucción LMS siempre será positivo, pero el de una palabra de información puede ser positivo o negativo. Cada una de las ubicaciones en la memoria de Simpletron puede contener una instrucción, un valor de datos utilizado por un programa o un área no utilizada (y por lo tanto indefinida) de memoria. Los primeros dos dígitos de cada instrucción LMS son el código de operación que especifica la operación a realizar. Los códigos de operación de LMS se sintetizan en la figura 7.33.

Código de operación	Significado
Operaciones de entrada/salida:	
`final int LEE = 10;`	Lee una palabra desde el teclado y la introduce en una ubicación específica de memoria.

Fig. 7.33 | Códigos de operación del Lenguaje máquina Simpletron (LMS) (parte 1 de 2).

Código de operación	Significado
`final int ESCRIBE = 11;`	Escribe una palabra de una ubicación específica de memoria y la imprime en la pantalla.
Operaciones de carga/almacenamiento:	
`final int CARGA = 20;`	Carga una palabra de una ubicación específica de memoria y la coloca en el acumulador.
`final int ALMACENA = 21;`	Almacena una palabra del acumulador dentro de una ubicación específica de memoria.
Operaciones aritméticas:	
`final int SUMA = 30;`	Suma una palabra de una ubicación específica de memoria a la palabra en el acumulador (deja el resultado en el acumulador).
`final int RESTA = 31;`	Resta una palabra de una ubicación específica de memoria a la palabra en el acumulador (deja el resultado en el acumulador).
`final int DIVIDE = 32;`	Divide una palabra de una ubicación específica de memoria entre la palabra en el acumulador (deja el resultado en el acumulador).
`final int MULTIPLICA = 33;`	Multiplica una palabra de una ubicación específica de memoria por la palabra en el acumulador (deja el resultado en el acumulador).
Operaciones de transferencia de control:	
`final int BIFURCA = 40;`	Bifurca hacia una ubicación específica de memoria.
`final int BIFURCANEG = 41;`	Bifurca hacia una ubicación específica de memoria si el acumulador es negativo.
`final int BIFURCACERO = 42;`	Bifurca hacia una ubicación específica de memoria si el acumulador es cero.
`final int ALTO = 43;`	Alto. El programa completó su tarea.

Fig. 7.33 | Códigos de operación del Lenguaje máquina Simpletron (LMS) (parte 2 de 2).

Los últimos dos dígitos de una instrucción LMS son el *operando* (la dirección de la ubicación en memoria que contiene la palabra a la cual se aplica la operación). Consideremos varios programas simples en LMS.

El primer programa en LMS (figura 7.34) lee dos números del teclado, calcula e imprime su suma. La instrucción `+1007` lee el primer número del teclado y lo coloca en la ubicación `07` (que se ha inicializado con 0). Después, la instrucción `+1008` lee el siguiente número y lo coloca en la ubicación `08`. La instrucción *carga*, `+2007`, coloca el primer número en el acumulador y la instrucción *suma*, `+3008`, suma el segundo número al número en el acumulador. *Todas las instrucciones LMS aritméticas dejan sus resultados en el acumulador*. La instrucción *almacena*, `+2109`, coloca el resultado de vuelta en la ubicación de memoria `09`, desde la cual la instrucción *escribe*, `+1109`, toma el número y lo imprime (como un número decimal de cuatro dígitos con signo). La instrucción *alto*, `+4300`, termina la ejecución.

El segundo programa en LMS (figura 7.35) lee dos números desde el teclado, determina e imprime el valor más grande. Observe el uso de la instrucción `+4107` como una transferencia de control condicional, en forma muy similar a la instrucción `if` de Java.

Ahora escriba programas en LMS para realizar cada una de las siguientes tareas:

a) Usar un ciclo controlado por centinela para leer 10 números positivos. Calcular e imprimir la suma.

Ubicación	Número	Instrucción
00	+1007	(Lee A)
01	+1008	(Lee B)
02	+2007	(Carga A)
03	+3008	(Suma B)
04	+2109	(Almacena C)
05	+1109	(Escribe C)
06	+4300	(Alto)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Resultado C)

Fig. 7.34 | Programa en LMS que lee dos enteros y calcula la suma.

Ubicación	Número	Instrucción
00	+1009	(Lee A)
01	+1010	(Lee B)
02	+2009	(Carga A)
03	+3110	(Resta B)
04	+4107	(Bifurcación negativa a 07)
05	+1109	(Escribe A)
06	+4300	(Alto)
07	+1110	(Escribe B)
08	+4300	(Alto)
09	+0000	(Variable A)
10	+0000	(Variable B)

Fig. 7.35 | Programa en LMS que lee dos enteros y determina cuál de ellos es mayor.

b) Usar un ciclo controlado por contador para leer siete números, algunos positivos y otros negativos, y calcular e imprimir su promedio.
c) Leer una serie de números, determinar e imprimir el número más grande. El primer número leído indica cuántos números deben procesarse.

7.36 ***(Simulador de computadora)*** En este problema usted creará su propia computadora. No, no soldará componentes, sino que utilizará la poderosa técnica de la *simulación basada en software* para crear un *modelo de software* orientado a objetos de Simpletron, la computadora del ejercicio 7.35. Su simulador Simpletron convertirá la computadora que usted utiliza en Simpletron, y será capaz de ejecutar, probar y depurar los programas LMS que escribió en el ejercicio 7.35.

Cuando ejecute su simulador Simpletron, debe empezar y mostrar lo siguiente:

```
*** Bienvenido a Simpletron! ***
*** Por favor, introduzca en su programa una instruccion  ***
*** (o palabra de datos) a la vez. Yo le mostrare    ***
*** el numero de ubicacion y un signo de interrogacion (?) ***
```

```
*** Entonces usted escribira la palabra para esa ubicacion.  ***
*** Teclee -9999 para dejar de introducir su programa.  ***
```

Su aplicación debe simular la memoria del Simpletron con un arreglo unidimensional llamado `memoria`, que cuente con 100 elementos. Ahora suponga que el simulador se está ejecutando y examinaremos el diálogo a medida que introduzcamos el programa de la figura 7.35 (ejercicio 7.35):

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
```

Su programa debe mostrar la ubicación en memoria, seguida por un signo de interrogación. Cada uno de los valores a la derecha de un signo de interrogación es introducido por el usuario. Al introducir el valor centinela `-99999`, el programa debe mostrar lo siguiente:

```
*** Se completo la carga del programa ***
*** Empieza la ejecucion del programa  ***
```

Ahora el programa en LMS se ha colocado (o cargado) en el arreglo `memoria`. Simpletron debe a continuación ejecutar el programa en LMS. La ejecución comienza con la instrucción en la ubicación `00` y, como en Java, continúa en forma secuencial a menos que se lleve a otra parte del programa mediante una transferencia de control.

Use la variable `acumulador` para representar el registro acumulador. Use la variable `contadorDeInstrucciones` para llevar el registro de la ubicación en memoria que contiene la instrucción que se está ejecutando. Use la variable `codigoDeOperacion` para indicar la operación que se esté realizando actualmente (es decir, los dos dígitos a la izquierda en la palabra de instrucción). Use la variable `operando` para indicar la ubicación de memoria en la que operará la instrucción actual. Por lo tanto, `operando` está compuesta por los dos dígitos más a la derecha de la instrucción que se esté ejecutando en esos momentos. No ejecute las instrucciones directamente desde la memoria. En vez de eso, transfiera la siguiente instrucción a ejecutar desde la memoria hasta una variable llamada `registroDeInstruccion`. Luego "recoja" los dos dígitos a la izquierda y colóquelos en `codigoDeOperacion`, después "recoja" los dos dígitos a la derecha y colóquelos en `operando`. Cuando Simpletron comience con la ejecución, todos los registros especiales se deben inicializar con cero.

Ahora vamos a "dar un paseo" por la ejecución de la primera instrucción LMS, `+1009` en la ubicación de memoria `00`. A este procedimiento se le conoce como *ciclo de ejecución de una instrucción*.

El `contadorDeInstrucciones` nos indica la ubicación de la siguiente instrucción a ejecutar. Nosotros *buscamos* el contenido de esa ubicación de `memoria`, con la siguiente instrucción de Java:

```
registroDeInstruccion = memoria[ contadorDeInstrucciones ];
```

El código de operación y el operando se extraen del registro de instrucción, mediante las instrucciones

```
codigoDeOperacion = registroDeInstruccion / 100;
operando = registroDeInstruccion % 100;
```

Ahora, Simpletron debe determinar que el código de operación es en realidad un *lee* (en comparación con un *escribe*, *carga*, etcétera). Una instrucción `switch` establece la diferencia entre las 12 operaciones de LMS. En la instrucción `switch` se simula el comportamiento de varias instrucciones LMS, como se muestra en la figura 7.36. En breve hablaremos sobre las instrucciones de bifurcación y dejaremos las otras a usted.

Cuando el programa en LMS termine de ejecutarse, deberán mostrarse el nombre y contenido de cada registro, así como el contenido completo de la memoria. A este tipo de impresión se le denomina vaciado de la computadora (no, un vaciado de computadora no es un lugar al que van las computadoras viejas). Para ayudarlo a programar su método de vaciado, en la figura 7.37 se muestra un formato de vaciado de muestra. Un vaciado, después de la ejecución de un

Instrucción	Descripción
lee:	Mostrar el mensaje "Escriba un entero", después recibir como entrada el entero y almacenarlo en la ubicación memoria[operando].
carga:	`acumulador = memoria[ operando ];`
suma:	`acumulador += memoria[ operando ];`
alto:	Esta instrucción muestra el mensaje `*** Termino la ejecución de Simpletron ***`

Fig. 7.36 | Comportamiento de varias instrucciones de LMS en Simpletron.

```
REGISTROS:
acumulador                  +0000
contadorDeInstrucciones       00
registroDeInstruccion       +0000
codigoDeOperacion             00
operando                      00

MEMORIA:
      0     1     2     3     4     5     6     7     8     9
 0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Fig. 7.37 | Un vaciado de muestra.

programa de Simpletron, muestra los valores actuales de las instrucciones y los valores de los datos al momento en que se terminó la ejecución.

Procedamos ahora con la ejecución de la primera instrucción de nuestro programa: `+1009` en la ubicación `00`. Como lo hemos indicado, la instrucción `switch` simula esta tarea pidiendo al usuario que escriba un valor, leyendo el valor y almacenándolo en la ubicación de memoria `memoria[operando]`. A continuación, el valor se lee y se coloca en la ubicación `09`.

En este punto se ha completado la simulación de la primera instrucción. Todo lo que resta es preparar a Simpletron para que ejecute la siguiente instrucción. Como la instrucción que acaba de ejecutarse no es una transferencia de control, sólo necesitamos incrementar el registro contador de instrucciones de la siguiente manera:

```
++contadorDeInstrucciones;
```

Esta acción completa la ejecución simulada de la primera instrucción. Todo el proceso (es decir, el ciclo de ejecución de una instrucción) empieza de nuevo, con la búsqueda de la siguiente instrucción a ejecutar.

Ahora veremos cómo se simulan las instrucciones de bifurcación (las transferencias de control). Todo lo que necesitamos hacer es ajustar el valor en el contador de instrucciones de manera apropiada. Por lo tanto, la instrucción de bifurcación incondicional (`40`) se simula dentro de la instrucción `switch` como

```
contadorDeInstrucciones = operando;
```

La instrucción condicional "bifurcar si el acumulador es cero" se simula como

```
if ( acumulador == 0 )
   contadorDeInstrucciones = operando;
```

En este punto, usted debe implementar su simulador Simpletron y ejecutar cada uno de los programas que escribió en el ejercicio 7.35. Si lo desea, puede embellecer al LMS con características adicionales y ofrecerlas en su simulador.

Su simulador debe comprobar diversos tipos de errores. Por ejemplo, durante la fase de carga del programa, cada número que el usuario escribe en la `memoria` de Simpletron debe encontrarse dentro del rango de `-9999` a `+9999`. Su simulador debe probar que cada número introducido se encuentre dentro de este rango y, en caso contrario, seguir pidiendo al usuario que vuelva a introducir el número hasta que introduzca un número correcto.

Durante la fase de ejecución, su simulador debe comprobar varios errores graves, como los intentos de dividir entre cero, intentos de ejecutar códigos de operación inválidos, y desbordamientos del acumulador (es decir, las operaciones aritméticas que den como resultado valores mayores que `+9999` o menores que `-9999`). Dichos errores graves se conocen como *errores fatales*. Al detectar un error fatal, su simulador deberá imprimir un mensaje de error como

```
*** Intento de dividir entre cero ***
*** La ejecucion de Simpletron se termino en forma anormal ***
```

y deberá imprimir un vaciado de computadora completo en el formato que vimos antes. Este análisis ayudará al usuario a localizar el error en el programa.

7.37 ***(Modificaciones al simulador Simpletron)*** En el ejercicio 7.36 usted escribió una simulación de software de una computadora que ejecuta programas escritos en el Lenguaje máquina Simpletron (LMS). En este ejercicio proponemos varias modificaciones y mejoras al simulador Simpletron. En los ejercicios del capítulo 22 (en el sitio Web del libro), propondremos la creación de un compilador que convierta los programas escritos en un lenguaje de programación de alto nivel (una variación de Basic) a Lenguaje máquina Simpletron. Algunas de las siguientes modificaciones y mejoras pueden requerirse para ejecutar los programas producidos por el compilador:

a) Extienda la memoria del simulador Simpletron, de manera que contenga 1000 ubicaciones de memoria para permitir a Simpletron manejar programas más grandes.
b) Permita al simulador realizar cálculos de residuo. Esta modificación requiere de una instrucción adicional en LMS.
c) Permita al simulador realizar cálculos de exponenciación. Esta modificación requiere una instrucción adicional en LMS.
d) Modifique el simulador para que pueda utilizar valores hexadecimales, en vez de valores enteros para representar instrucciones en LMS.
e) Modifique el simulador para permitir la impresión de una nueva línea. Esta modificación requiere una instrucción adicional en LMS.
f) Modifique el simulador para procesar valores de punto flotante además de valores enteros.
g) Modifique el simulador para manejar la introducción de cadenas. [*Sugerencia:* cada palabra de Simpletron puede dividirse en dos grupos, cada una de las cuales guarda un entero de dos dígitos. Cada entero de dos dígitos representa el equivalente decimal de código ASCII (vea el apéndice B) de un carácter. Agregue una instrucción de lenguaje máquina que reciba como entrada una cadena y la almacene, empezando en una ubicación de memoria específica de Simpletron. La primera mitad de la palabra en esa ubicación será una cuenta del número de caracteres en la cadena (es decir, la longitud de la cadena). Cada media palabra subsiguiente contiene un carácter ASCII, expresado como dos dígitos decimales. La instrucción en lenguaje máquina convierte cada carácter en su equivalente ASCII y lo asigna a una media palabra].
h) Modifique el simulador para manejar la impresión de cadenas almacenadas en el formato de la parte (g). [*Sugerencia:* agregue una instrucción en lenguaje máquina que imprima una cadena, que empiece en cierta ubicación de memoria de Simpletron. La primera mitad de la palabra en esa ubicación es una cuenta del número de caracteres en la cadena (es decir, la longitud de la misma). Cada media palabra subsiguiente contiene un carácter ASCII expresado como dos dígitos decimales. La instrucción en lenguaje máquina comprueba la longitud e imprime la cadena, traduciendo cada número de dos dígitos en su carácter equivalente].

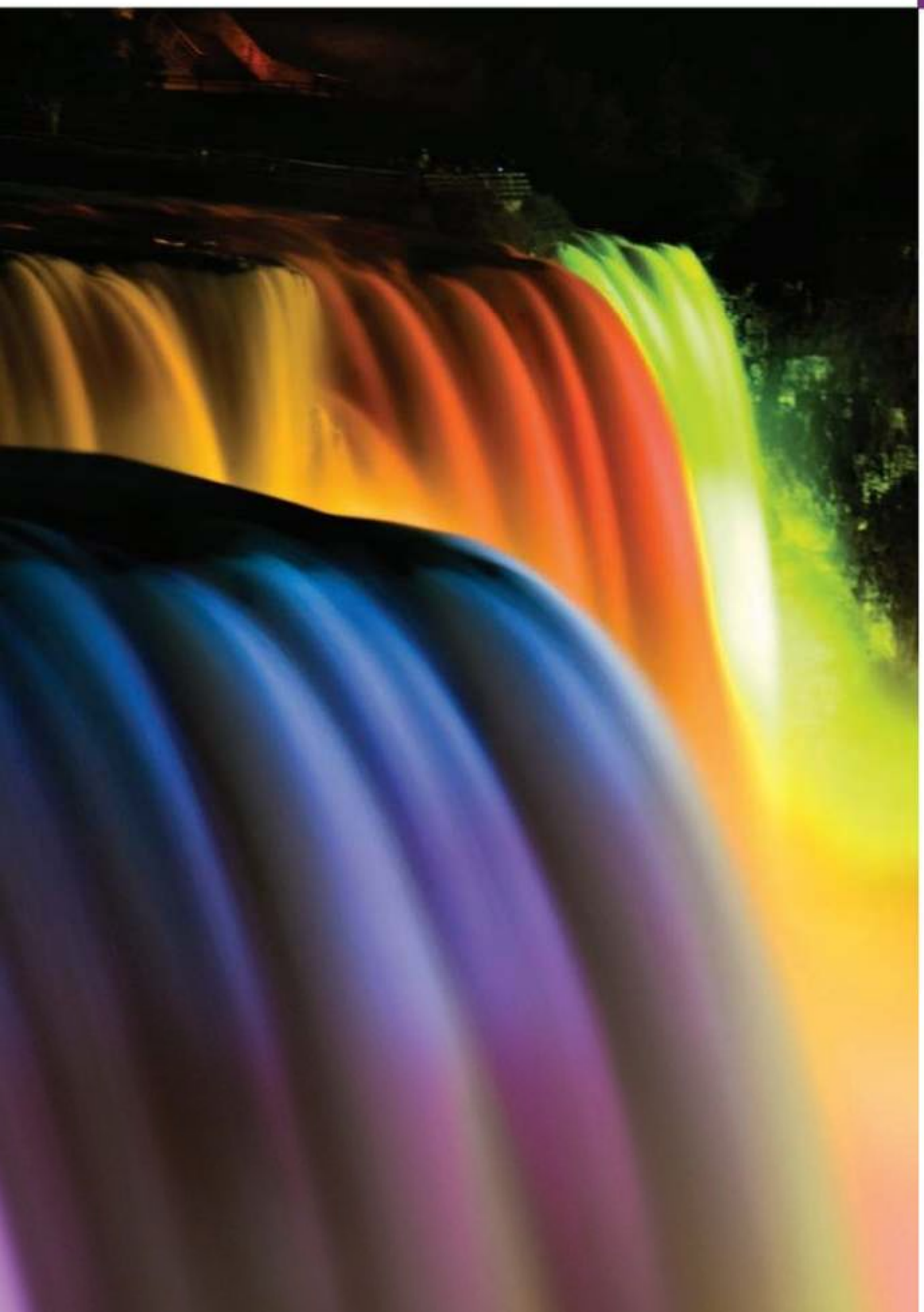
Marcar la diferencia

7.38 *(Votaciones)* Internet y Web permiten que cada vez haya más personas conectadas en red, unidas por una causa, expresen sus opiniones, etcétera. En 2008, los candidatos presidenciales usaron Internet en forma intensiva para expresar sus mensajes y recaudar dinero para sus campañas. En este ejercicio, escribirá un pequeño programa de votaciones que permite a los usuarios calificar cinco asuntos de conciencia social, desde 1 (menos importante) hasta 10 (más importante). Elija cinco causas que sean importantes para usted (por ejemplo, asuntos políticos, asuntos sobre el entorno global). Use un arreglo unidimensional llamado `temas` (de tipo `String`) para almacenar las cinco causas. Para sintetizar las respuestas de la encuesta, use un arreglo bidimensional de 5 filas y 10 columnas llamado `respuestas` (de tipo `int`), en donde cada fila corresponda a un elemento del arreglo `temas`. Cuando se ejecute el programa, debe pedir al usuario que califique cada asunto. Haga que sus amigos y familiares respondan a la encuesta. Después haga que el programa muestre un resumen de los resultados, incluya:

a) Un informe tabular con los cinco temas del lado izquierdo y las 10 calificaciones a lo largo de la parte superior, listando en cada columna el número de calificaciones recibidas para cada tema.
b) A la derecha de cada fila, muestre el promedio de las calificaciones para cada asunto específico.
c) ¿Qué asunto recibió la mayor puntuación total? Muestre ambos, el asunto y la puntuación.
d) ¿Cuál obtuvo la menor puntuación total? Muestre tanto el asunto como la puntuación total.

Clases y objetos: un análisis más detallado

8



En vez de esta absurda división entre sexos, deberían clasificar a las personas como estáticas y dinámicas.

—Evelyn Waugh

¿Es éste un mundo en el cual se deben ocultar las virtudes?

—William Shakespeare

¿Pero qué cosa, para servir a nuestros fines privados, olvida los engaños de nuestros amigos?

—Charles Churchill

Por encima de todo: hay que ser sinceros con nosotros mismos.

—William Shakespeare

No hay que ser "duros", sino simplemente sinceros.

—Oliver Wendell Holmes, Jr.

Objetivos

En este capítulo aprenderá a:

- Comprender el concepto de encapsulamiento y ocultamiento de datos.
- Usar la palabra clave `this`.
- Utilizar las variables y métodos `static`.
- Importar los miembros `static` de una clase.
- Utilizar el tipo `enum` para crear conjuntos de constantes con identificadores únicos.
- Declarar constantes `enum` con parámetros.
- Organizar las clases en paquetes para promover la reutilización.

Plan general

8.1 Introducción

Ahora analizaremos más de cerca la creación de clases, el control del acceso a los miembros de una clase y la creación de constructores. Hablaremos sobre la composición: una capacidad que permite a una clase tener referencias a objetos de otras clases como miembros. Analizaremos nuevamente el uso de los métodos *establecer* y *obtener*. Si lo recuerda, en la sección 6.10 presentamos el tipo básico `enum` para declarar un conjunto de constantes. En este capítulo, hablaremos sobre la relación entre los tipos `enum` y las clases para demostrar que, al igual que una clase, un `enum` se puede declarar en su propio archivo con constructores, métodos y campos. El capítulo también habla con detalle sobre los miembros de clase `static` y las variables de instancia `final`. Por último, explicaremos cómo organizar las clases en paquetes, para ayudar en la administración de aplicaciones extensas y promover la reutilización; después mostraremos una relación especial entre clases dentro del mismo paquete.

8.2 Caso de estudio de la clase `Tiempo`

Nuestro primer ejemplo consiste en dos clases: `Tiempo1` (figura 8.1) y `PruebaTiempo1` (figura 8.2). La clase `Tiempo1` representa la hora del día. La clase `PruebaTiempo1` es una clase de aplicación en la que el método `main` crea un objeto de la clase `Tiempo1` e invoca a sus métodos. Estas clases se deben declarar en filas *separadas* ya que ambas son de tipo `public`. El resultado de este programa aparece en la figura 8.2.

Declaración de la clase `Tiempo1`

Las variables de instancia `private int` llamadas `hora`, `minuto` y `segundo` de la clase `Tiempo1` (figura 8.1, líneas 6 a 8) representan la hora en formato de tiempo universal (formato de reloj de 24 horas, en el cual las horas se encuentran en el rango de 0 a 23). La clase `Tiempo1` contiene los métodos `public` `establecerTiempo` (líneas 12 a 25), `aStringUniversal` (líneas 28 a 31) y `toString` (líneas 34 a 39). A estos métodos también se les llama **servicios** `public` o la **interfaz** `public` que proporciona la clase a sus clientes.

El constructor predeterminado

En este ejemplo, la clase `Tiempo1` no declara un constructor, por lo que tiene un constructor predeterminado que le suministra el compilador. Cada variable de instancia recibe en forma implícita el valor pre-

```
1  // Fig. 8.1: Tiempo1.java
2  // La declaración de la clase Tiempo1 mantiene la hora en formato de 24 horas.
3
4  public class Tiempo1
5  {
6     private int hora;   // 0 - 23
7     private int minuto; // 0 - 59
8     private int segundo; // 0 - 59
9
10    // establece un nuevo valor de tiempo, usando la hora universal;
11    // lanza una excepción si la hora, minuto o segundo son inválidos
12    public void establecerTiempo( int h, int m, int s )
13    {
14       // valida la hora, el minuto y el segundo
15       if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
16          ( s >= 0 && s < 60 ))
17       {
18          hora = h;
19          minuto = m;
20          segundo = s;
21       } // fin de if
22       else
23          throw new IllegalArgumentException(
24             "hora, minuto y/o segundo estaban fuera de rango");
25    } // fin del método establecerTiempo
26
27    // convierte a objeto String en formato de hora universal (HH:MM:SS)
28    public String aStringUniversal()
29    {
30       return String.format( "%02d:%02d:%02d", hora, minuto, segundo );
31    } // fin del método aStringUniversal
32
33    // convierte a objeto String en formato de hora estándar (H:MM:SS AM o PM)
34    public String toString()
35    {
36       return String.format( "%d:%02d:%02d %s",
37          ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 ),
38          minuto, segundo, ( hora < 12 ? "AM" : "PM" ) );
39    } // fin del método toString
40 } // fin de la clase Tiempo1
```

Fig. 8.1 | La declaración de la clase `Tiempo1` mantiene la hora en formato de 24 horas.

determinado 0 para un `int`. Las variables de instancia también pueden inicializarse cuando se declaran en el cuerpo de la clase, usando la misma sintaxis de inicialización que la de una variable local.

El método `establecerTiempo` *y cómo lanzar excepciones*

El método `establecerTiempo` (líneas 12 a 25) es un método `public` que declara tres parámetros `int` y los utiliza para establecer la hora. La líneas 15 y 16 evalúan cada argumento, para determinar si el valor se encuentra en un rango especificado y, de ser así, las líneas 18 a 29 asignan los valores a las variables de instancia `hora`, `minuto` y `segundo`. El valor de hora debe ser mayor o igual que 0 y menor que 24, ya que el formato de hora universal representa las horas como enteros de 0 a 23 (por ejemplo, la 1 PM es la hora 13 y las 11 PM son la hora 23; medianoche es la hora 0 y mediodía es la hora 12). De manera similar,

los valores de `minuto` y `segundo` deben ser mayores o iguales que `0` y menores que `60`. Para los valores fuera de estos rangos, `establecerTiempo` **lanza una excepción** de tipo `IllegalArgumentException` (líneas 23 y 24), la cual notifica al código cliente que se pasó un argumento inválido al método. Como vimos en el capítulo 7, podemos usar `try...catch` para atrapar excepciones y tratar de recuperarnos de ellas, lo cual haremos en la figura 8.2. La **instrucción `throw`** (línea 23) crea un nuevo objeto de tipo `IllegalArgumentException`. Los paréntesis después del nombre de la clase indican una llamada al constructor de `IllegalArgumentException`. En este caso, llamamos al constructor que nos permite especificar un mensaje de error personalizado. Después de crear el objeto excepción, la instrucción `throw` termina de inmediato el método `establecerTiempo` y la excepción regresa al código que intentó establecer el tiempo.

El método `aStringUniversal`

El método `aStringUniversal` (líneas 28 a 31) no recibe argumentos y devuelve un objeto `String` en formato de hora universal, el cual consiste de dos dígitos para la hora, dos para los minutos y dos para los segundos. Por ejemplo, si la hora es 1:30:07 PM, el método `aStringUniversal` devuelve `13:30:07`. La línea 22 utiliza el método `static` **`format`** de la clase `String` para devolver un objeto `String` que contiene los valores con formato de `hora`, `minuto` y `segundo`, cada uno con dos dígitos y posiblemente, un `0` a la izquierda (el cual se especifica con la bandera `0`). El método `format` es similar al método `System.out.printf`, sólo que `format` devuelve un objeto `String` con formato, en vez de mostrarlo en una ventana de comandos. El método `aStringUniversal` devuelve el objeto `String` con formato.

El método `toString`

El método `toString` (líneas 34 a 39) no recibe argumentos y devuelve un objeto `String` en formato de hora estándar, el cual consiste en los valores de `hora`, `minuto` y `segundo` separados por signos de dos puntos (`:`), y seguidos de un indicador AM o PM (por ejemplo, `1:27:06 PM`). Al igual que el método `aStringUniversal`, el método `toString` utiliza el método `static String format` para dar formato a los valores de `minuto` y `segundo` como valores de dos dígitos, con ceros a la izquierda, en caso de ser necesario. La línea 29 utiliza un operador condicional (`?:`) para determinar el valor de hora en la cadena; si hora es `0` o `12` (AM o PM), aparece como 12; en cualquier otro caso, aparece como un valor de 1 a 11. El operador condicional en la línea 30 determina si se devolverá AM o PM como parte del objeto `String`.

En la sección 6.4 vimos que todos los objetos en Java tienen un método `toString` que devuelve una representación `String` del objeto. Optamos por devolver un objeto `String` que contiene la hora en formato estándar. El método `toString` se puede llamar en forma implícita cada vez que aparece un objeto `Tiempo1` en el código, en donde se necesita un `String`, como el valor para imprimir con un especificador de formato `%s` en una llamada a `System.out.printf`.

Uso de la clase `Tiempo1`

Como aprendió en el capítulo 3, cada clase que se declara representa un nuevo *tipo* en Java. Por lo tanto, después de declarar la clase `Tiempo1`, podemos utilizarla como un tipo en las declaraciones como

```
Tiempo1 puestasol; // puestasol puede guardar una referencia a un objeto Tiempo1
```

La clase de la aplicación `PruebaTiempo1` (figura 8.2) utiliza la clase `Tiempo1`. La línea 9 declara y crea un objeto `Tiempo1` y lo asigna a la variable local tiempo. El operador `new` invoca en forma implícita al constructor predeterminado de la clase `Tiempo1`, ya que `Tiempo1` no declara constructores. Las líneas 12 a 16 imprimen en pantalla la hora, primero en formato universal (mediante la invocación al método `aStringUniversal` de `tiempo` en la línea 13) y después en formato estándar (mediante la invocación explícita del método `toString` de `tiempo` en la línea 15) para confirmar que el objeto `Tiempo1` se haya inicializado en forma apropiada. La línea 19 invoca al método `establecerTiempo` del objeto tiempo para modificar

la hora. Las líneas 20 a 24 imprimen en pantalla la hora otra vez en ambos formatos, para confirmar que se haya ajustado en forma apropiada.

```
1  // Fig. 8.2: PruebaTiempo1.java
2  // Objeto Tiempo1 utilizado en una aplicación.
3
4  public class PruebaTiempo1
5  {
6     public static void main( String[] args )
7     {
8        // crea e inicializa un objeto Tiempo1
9        Tiempo1 tiempo = new Tiempo1(); // invoca el constructor de Tiempo1
10
11       // imprime representaciones de cadena del tiempo
12       System.out.print( "La hora universal inicial es: " );
13       System.out.println( tiempo.aStringUniversal() );
14       System.out.print( "La hora estandar inicial es: " );
15       System.out.println( tiempo.toString() );
16       System.out.println(); // imprime una linea en blanco
17
18       // modifica el tiempo e imprime el tiempo actualizado
19       tiempo.establecerTiempo( 13, 27, 6 );
20       System.out.print( "La hora universal despues de establecerTiempo es: " );
21       System.out.println( tiempo.aStringUniversal() );
22       System.out.print( "La hora estandar despues de establecerTiempo es: " );
23       System.out.println( tiempo.toString() );
24       System.out.println(); // imprime una linea en blanco
25
26       // intenta establecer el tiempo con valores inválidos;
27       try
28       {
29          tiempo.establecerTiempo( 99, 99, 99 ); // todos los valores fuera de rango
30       } // fin de try
31       catch (IllegalArgumentException e)
32       {
33          System.out.printf( "Excepcion: %s\n\n", e.getMessage() );
34       } // fin de catch
35
36       // muestra el tiempo después de tratar de establecer valores inválidos
37       System.out.println( "Despues de intentar ajustes invalidos:" );
38       System.out.print( "Hora universal: " );
39       System.out.println( tiempo.aStringUniversal() );
40       System.out.print( "Hora estandar: " );
41       System.out.println( tiempo.toString() );
42    } // fin de main
43 } // fin de la clase PruebaTiempo1
```

```
La hora universal inicial es: 00:00:00
La hora estandar inicial es: 12:00:00 AM

La hora universal despues de establecerTiempo es: 13:27:06
La hora estandar despues de establecerTiempo es: 1:27:06 PM
```

Fig. 8.2 | Objeto `Tiempo1` usado en una aplicación (parte 1 de 2).

```
Excepcion: hora, minuto y/o segundo estaban fuera de rango

Despues de intentar ajustes invalidos:
Hora universal: 13:27:06
Hora estandar: 1:27:06 PM
```

Fig. 8.2 | Objeto `Tiempo1` usado en una aplicación (parte 2 de 2).

Llamada al método `establecerTiempo` *de* `Tiempo1` *con valores inválidos*

Para ilustrar que el método `establecerTiempo` valida sus argumentos, la línea 29 llama al método `establecerTiempo` con los argumentos inválidos de 99 para `hora`, `minuto` y `segundo`. Esta instrucción se coloca en un bloque `try` (líneas 27 a 30) en caso de que `establecerTiempo` lance una excepción `IllegalArgumentException`, lo cual hará debido a que los argumentos son todos inválidos. Al ocurrir esto, la excepción se atrapa en las líneas 31 a 34, y la línea 33 muestra el mensaje de error de la excepción, llamando a su método `getMessage`. Las líneas 37 a 41 imprimen de nuevo la hora en ambos formatos, para confirmar que `establecerTiempo` no la haya cambiado cuando se suministraron argumentos inválidos.

Notas acerca de la declaración de la clase `Tiempo1`

Es necesario considerar diversas cuestiones sobre el diseño de clases, en relación con la clase `Tiempo1`. Las variables de instancia `hora`, `minuto` y `segundo` se declaran como `private`. La representación de datos que se utilice dentro de la clase no concierne a los clientes de la misma. Por ejemplo, sería perfectamente razonable que `Tiempo1` representara el tiempo en forma interna como el número de segundos transcurridos a partir de medianoche, o el número de minutos y segundos transcurridos a partir de medianoche. Los clientes podrían usar los mismos métodos `public` y obtener los mismos resultados, sin tener que preocuparse por lo anterior. (El ejercicio 8.5 le pide que represente la hora en la clase `Tiempo1` como el número de segundos transcurridos a partir de medianoche, y que muestre que, en definitiva, no hay cambios visibles para los clientes de la clase).

Observación de ingeniería de software 8.1

Las clases simplifican la programación, ya que el cliente sólo puede utilizar los métodos `public` *expuestos por la clase. Dichos miembros por lo general están orientados a los clientes, en vez de estar dirigidos a la implementación. Los clientes nunca se percatan de (ni se involucran en) la implementación de una clase. Por lo general se preocupan por lo que hace la clase, pero no cómo lo hace.*

Observación de ingeniería de software 8.2

Las interfaces cambian con menos frecuencia que las implementaciones. Cuando cambia una implementación, el código dependiente de ella debe cambiar de manera acorde, y el ocultamiento de ésta reduce la posibilidad de que otras partes del programa se vuelvan dependientes de los detalles de la implementación de la clase.

8.3 Control del acceso a los miembros

Los modificadores de acceso `public` y `private` controlan el acceso a las variables y los métodos de una clase. En el capítulo 9, presentaremos el modificador de acceso adicional `protected`. Como dijimos en la sección 8.2, el principal propósito de los métodos `public` es presentar a los clientes de la clase una vista de los servicios que proporciona (la interfaz `public` de la clase). Los clientes de la clase no necesitan preocuparse por la forma en que realiza sus tareas. Por esta razón, las variables y métodos `private` de una clase (es decir, los detalles de implementación de la clase) *no* son accesibles para sus clientes.

La figura 8.3 demuestra que los miembros de una clase private no son accesibles fuera de la clase. Las líneas 9 a 11 tratan de acceder en forma directa a las variables de instancia private hora, minuto y segundo del objeto tiempo de la clase Tiempo1. Al compilar este programa, el compilador genera mensajes de error que indican que estos miembros private no son accesibles. Este programa asume que se utiliza la clase Tiempo1 de la figura 8.1.

Error común de programación 8.1

Cuando un método que no es miembro de una clase trata de acceder a un miembro private de ésta, se produce un error de compilación.

```
1  // Fig. 8.3: PruebaAccesoMiembros.java
2  // Los miembros private de la clase Tiempo1 no son accesibles.
3  public class PruebaAccesoMiembros
4  {
5     public static void main( String[] args )
6     {
7        Tiempo1 tiempo = new Tiempo1(); // crea e inicializa un objeto Tiempo1
8
9        tiempo.hora = 7;    // error: hora tiene acceso privado en Tiempo1
10       tiempo.minuto = 15; // error: minuto tiene acceso privado en Tiempo1
11       tiempo.segundo = 30; // error: segundo tiene acceso privado en Tiempo1
12    } // fin de main
13 } // fin de la clase PruebaAccesoMiembros
```

```
PruebaAccesoMiembros.java:9: hora has private access in Tiempo1
      tiempo.hora = 7;    // error: hora tiene acceso privado en Tiempo1
             ^
PruebaAccesoMiembros.java:10: minuto has private access in Tiempo1
      tiempo.minuto = 15; // error: minuto tiene acceso privado en Tiempo1
             ^
PruebaAccesoMiembros.java:11: segundo has private access in Tiempo1
      tiempo.segundo = 30; // error: segundo tiene acceso privado en Tiempo1
             ^
3 errors
```

Fig. 8.3 | Los miembros privados de la clase Tiempo1 no son accesibles.

8.4 Referencias a los miembros del objeto actual mediante this

Cada objeto puede acceder a una referencia a sí mismo mediante la palabra clave this (también conocida como **referencia this**). Cuando se hace una llamada a un método no static para un objeto específico, el cuerpo del método utiliza en forma implícita la palabra clave this para hacer referencia a las variables de instancia y otros métodos. Esto permite al código de la clase saber qué objeto se debe manipular. Como verá en la figura 8.4, puede utilizar también la palabra clave this de manera explícita en el cuerpo de un método no static. La sección 8.5 muestra otro uso interesante de la palabra clave this. La sección 8.11 explica por qué no puede usarse la palabra clave this en un método static.

Ahora demostraremos el uso implícito y explícito de la referencia this (figura 8.4). Este ejemplo es el primero en el que declaramos *dos* clases en un archivo: la clase PruebaThis se declara en las líneas 4 a

11 y la clase `TiempoSimple` se declara en las líneas 14 a 47. Hicimos esto para demostrar que, al compilar un archivo `.java` que contiene más de una clase, el compilador produce un archivo de clase separado con la extensión `.class` para cada clase compilada. En este caso se produjeron dos archivos separados: `TiempoSimple.class` y `PruebaThis.class`. Cuando un archivo de código fuente (`.java`) contiene varias declaraciones de clases, el compilador coloca los archivos para esas clases en el mismo directorio. Observe además que sólo la clase `PruebaThis` se declara `public` en la figura 8.4. Un archivo de código fuente sólo puede contener una clase `public`; de lo contrario, se produce un error de compilación. Las clases que no son `public` sólo pueden ser usadas por otras en el mismo paquete. Por lo tanto, en este ejemplo, la clase `TiempoSimple` sólo puede ser utilizada por la clase `PruebaThis`.

```
1  // Fig. 8.4: PruebaThis.java
2  //Uso implícito y explícito de this para hacer referencia a los miembros de un objeto.
3
4  public class PruebaThis
5  {
6     public static void main( String[] args )
7     {
8        TiempoSimple tiempo = new TiempoSimple( 15, 30, 19 );
9        System.out.println( tiempo.crearString() );
10    } // fin de main
11 } // fin de la clase PruebaThis
12
13 // la clase TiempoSimple demuestra la referencia "this"
14 class TiempoSimple
15 {
16    private int hora;   // 0-23
17    private int minuto; // 0-59
18    private int segundo; // 0-59
19
20    // si el constructor utiliza nombres de parámetros idénticos a
21    // los nombres de las variables de instancia, se requiere la
22    // referencia "this" para diferenciar unos nombres de otros
23    public TiempoSimple( int hora, int minuto, int segundo )
24    {
25       this.hora = hora;      // establece la hora del objeto "this"
26       this.minuto = minuto;  // establece el minuto del objeto "this"
27       this.segundo = segundo;  // establece el segundo del objeto "this"
28    } // fin del constructor de TiempoSimple
29
30    // usa la referencia "this" explícita e implícita para llamar aStringUniversal
31    public String crearString()
32    {
33       return String.format( "%24s: %s\n%24s: %s",
34          "this.aStringUniversal()", this.aStringUniversal(),
35          "aStringUniversal()", aStringUniversal() );
36    } // fin del método crearString
37
38    // convierte a String en formato de hora universal (HH:MM:SS)
39    public String aStringUniversal()
40    {
```

Fig. 8.4 | Uso implícito y explícito de `this` para hacer referencia a los miembros de un objeto (parte 1 de 2).

```
41         // "this" no se requiere aquí para acceder a las variables de instancia,
42         // ya que el método no tiene variables locales con los mismos
43         // nombres que las variables de instancia
44         return String.format( "%02d:%02d:%02d",
45            this.hora, this.minuto, this.segundo );
46      } // fin del método aStringUniversal
47   } // fin de la clase TiempoSimple
```

```
this.aStringUniversal(): 15:30:19
     aStringUniversal(): 15:30:19
```

Fig. 8.4 | Uso implícito y explícito de `this` para hacer referencia a los miembros de un objeto (parte 2 de 2).

La clase `TiempoSimple` (líneas 14 a 47) declara tres variables de instancia `private`: `hora`, `minuto` y `segundo` (líneas 16 a 18). El constructor (líneas 23 a 28) recibe tres argumentos `int` para inicializar un objeto `TiempoSimple`. Para el constructor (línea 23) utilizamos nombres de parámetros idénticos a los nombres de las variables de instancia de la clase (líneas 16 a 18). No recomendamos esta práctica, pero lo hicimos aquí para ocultar las variables de instancia correspondientes y así poder ilustrar un caso en el que se requiere el uso *explícito* de la referencia `this`. Si un método contiene una variable local con el *mismo* nombre que el de un campo, hará referencia a la variable local en vez del campo. En este caso, la variable local oculta el campo en el alcance del método. No obstante, el método puede utilizar la referencia `this` para hacer referencia al campo oculto de manera explícita, como se muestra en el lado izquierdo de las asignaciones de las líneas 25 a 27 para las variables de instancia ocultas de `TiempoSimple`.

El método `crearString` (líneas 31 a 36) devuelve un objeto `String` creado por una instrucción que utiliza la referencia `this` en forma explícita e implícita. La línea 34 la utiliza en forma explícita para llamar al método `aStringUniversal`. La línea 35 la utiliza en forma implícita para llamar al mismo método. Observe que ambas líneas realizan la misma tarea. Por lo general, no es común utilizar la referencia `this` en forma explícita para hacer referencia a otros métodos en el objeto actual. Además, en la línea 45 del método `aStringUniversal` se utiliza en forma explícita la referencia `this` para acceder a cada variable de instancia. Esto *no* es necesario aquí, ya que el método *no* tiene variables locales que oculten las variables de instancia de la clase.

Error común de programación 8.2

A menudo se produce un error lógico cuando un método contiene un parámetro o variable local con el mismo nombre que un campo de la clase. En tal caso, use la referencia `this` si desea acceder al campo de la clase; de no ser así, se hará referencia al parámetro o variable local del método.

Tip para prevenir errores 8.1

Evite los nombres de los parámetros o variables locales que tengan conflicto con los nombres de los campos. Esto ayuda a evitar errores sutiles, difíciles de localizar.

Tip de rendimiento 8.1

Para conservar la memoria, Java mantiene sólo una copia de cada método por clase; todos los objetos de la clase invocan a este método. Por otro lado, cada objeto tiene su propia copia de las variables de instancia de la clase (es decir, las variables no `static`). Cada método de la clase utiliza en forma implícita la referencia `this` para determinar el objeto específico de la clase que se manipulará.

La clase de la aplicación `PruebaThis` (líneas 4 a 11) demuestra el uso de la clase `TiempoSimple`. La línea 8 crea una instancia de la clase `TiempoSimple` e invoca a su constructor. La línea 9 invoca al método `crearString` del objeto y después muestra los resultados en pantalla.

8.5 Caso de estudio de la clase `Tiempo`: constructores sobrecargados

Como sabe, puede declarar su propio constructor para especificar cómo deben inicializarse los objetos de una clase. A continuación demostraremos una clase con varios **constructores sobrecargados**, que permiten a los objetos de esa clase inicializarse de distintas formas. Para sobrecargar los constructores, sólo hay que proporcionar varias declaraciones del constructor con distintas firmas.

La clase `Tiempo2` con constructores sobrecargados

El constructor predeterminado de la clase `Tiempo1` (figura 8.1) inicializó `hora`, `minuto` y `segundo` con sus valores predeterminados de 0 (medianoche en formato de hora universal). El constructor predeterminado no permite que los clientes de la clase inicialicen la hora con valores específicos distintos de cero. La clase `Tiempo2` (figura 8.5) contiene cinco constructores sobrecargados que proporcionan formas convenientes para inicializar los objetos de la nueva clase `Tiempo2`. Cada constructor inicializa el objeto para que empiece en un estado consistente. En este programa, cuatro de los constructores invocan un quinto constructor, el cual a su vez llama al método `establecerTiempo` para asegurar que el valor suministrado para `hora` se encuentre en el rango de 0 a 23, y que los valores para `minuto` y `segundo` se encuentren cada uno en el rango de 0 a 59. Para invocar el constructor apropiado, el compilador relaciona el número, los tipos y el orden de los tipos de los argumentos determinados en la llamada al constructor con el número, los tipos y el orden de los tipos de los parámetros especificados en la declaración de cada constructor. La clase `Tiempo2` también proporciona métodos *establecer* y *obtener* para cada variable de instancia.

```
1  // Fig. 8.5: Tiempo2.java
2  // Declaración de la clase Tiempo2 con constructores sobrecargados.
3
4  public class Tiempo2
5  {
6     private int hora;   // 0 - 23
7     private int minuto; // 0 - 59
8     private int segundo; // 0 - 59
9
10    // Constructor de Tiempo2 sin argumentos:
11    // inicializa cada variable de instancia a cero
12    public Tiempo2()
13    {
14       this( 0, 0, 0 ); // invoca al constructor de Tiempo2 con tres argumentos
15    } // fin del constructor de Tiempo2 sin argumentos
16
17    // Constructor de Tiempo2: se suministra hora, minuto y segundo con valor
         predeterminado de 0
18    public Tiempo2( int h )
19    {
20       this( h, 0, 0 ); // invoca al constructor de Tiempo2 con tres argumentos
21    } // fin del constructor de Tiempo2 con un argumento
22
```

Fig. 8.5 | La clase `Tiempo2` con constructores sobrecargados (parte 1 de 3).

```
23    // Constructor de Tiempo2: se suministran hora y minuto, segundo con valor
         predeterminado de 0
24    public Tiempo2( int h, int m )
25    {
26       this( h, m, 0 ); // invoca al constructor de Tiempo2 con tres argumentos
27    } // fin del constructor de Tiempo2 con dos argumentos
28
29    // Constructor de Tiempo2: se suministran hora, minuto y segundo
30    public Tiempo2( int h, int m, int s )
31    {
32       establecerTiempo( h, m, s ); // invoca a establecerTiempo para validar el tiempo
33    } // fin del constructor de Tiempo2 con tres argumentos
34
35    // Constructor de Tiempo2: se suministra otro objeto Tiempo2
36    public Tiempo2( Tiempo2 tiempo )
37    {
38       // invoca al constructor de Tiempo2 con tres argumentos
39       this( tiempo.obtenerHora(), tiempo.obtenerMinuto(), tiempo.obtenerSegundo() );
40    } // fin del constructor de Tiempo2 con un objeto Tiempo2 como argumento
41
42    // Métodos "establecer"
43    // establece un nuevo valor de tiempo usando la hora universal;
44    // valida los datos
45    public void establecerTiempo( int h, int m, int s )
46    {
47       establecerHora( h );   // establece la hora
48       establecerMinuto( m ); // establece el minuto
49       establecerSegundo( s ); // establece el segundo
50    } // fin del método establecerTiempo
51
52    // valida y establece la hora
53    public void establecerHora( int h )
54    {
55       if ( h >= 0 && h < 24 )
56          hora = h;
57       else
58          throw new IllegalArgumentException( "hora debe ser de 0 a 23" );
59    } // fin del método establecerHora
60
61    // valida y establece el minuto
62    public void establecerMinuto( int m )
63    {
64       if ( m >= 0 && m < 60 )
65          minuto = m;
66       else
67          throw new IllegalArgumentException( "minuto debe ser de 0 a 59");
68    } // fin del método establecerMinuto
69
70    // valida y establece el segundo
71    public void establecerSegundo( int s )
72    {
73       if ( s >= 0 && s < 60 )
74          segundo = ( ( s >= 0 && s < 60 ) ? s : 0 );
```

Fig. 8.5 | La clase Tiempo2 con constructores sobrecargados (parte 2 de 3).

```
75          else
76             throw new IllegalArgumentException( "segundo debe ser de 0 a 59");
77       } // fin del método establecerSegundo
78
79       // Métodos "obtener"
80       // obtiene el valor de la hora
81       public int obtenerHora()
82       {
83          return hora;
84       } // fin del método obtenerHora
85
86       // obtiene el valor del minuto
87       public int obtenerMinuto()
88       {
89          return minuto;
90       } // fin del método obtenerMinuto
91
92       // obtiene el valor del segundo
93       public int obtenerSegundo()
94       {
95          return segundo;
96       } // fin del método obtenerSegundo
97
98       // convierte a String en formato de hora universal (HH:MM:SS)
99       public String aStringUniversal()
100      {
101         return String.format(
102            "%02d:%02d:%02d", obtenerHora(), obtenerMinuto(), obtenerSegundo() );
103      } // fin del método aStringUniversal
104
105      // convierte a String en formato de hora estándar (H:MM:SS AM o PM)
106      public String toString()
107      {
108         return String.format( "%d:%02d:%02d %s",
109            ( (obtenerHora() == 0 || obtenerHora() == 12) ? 12 : obtenerHora() % 12 ),
110            obtenerMinuto(), obtenerSegundo(), ( obtenerHora() < 12 ? "AM" : "PM" ) );
111      } // fin del método toString
112   } // fin de la clase Tiempo2
```

Fig. 8.5 | La clase `Tiempo2` con constructores sobrecargados (parte 3 de 3).

Constructores de la clase `Tiempo2`

Las líneas 12 a 15 declaran un **constructor sin argumentos** que, como su nombre lo indica, se invoca sin argumentos. Una vez que se declaran constructores en una clase, el compilador *no* proporciona un constructor predeterminado. Este constructor sin argumentos se asegura de que los clientes de la clase `Tiempo2` puedan crear objetos `Tiempo2` con valores predeterminados. Dicho constructor simplemente inicializa el objeto como se especifica en el cuerpo del constructor. En el cuerpo, presentamos un uso de la referencia `this` que se permite sólo como la *primera* instrucción en el cuerpo de un constructor. La línea 14 utiliza a `this` en la sintaxis de la llamada al método para invocar al constructor de `Tiempo2` que recibe tres parámetros (líneas 30 a 33) con valores de 0 para `hora`, `minuto` y `segundo`. El uso de la referencia `this` que se muestra aquí es una forma popular de reutilizar el código de inicialización que proporciona otro de los constructores de la clase, en vez de definir código similar en el cuerpo del constructor sin argumentos.

Utilizamos esta sintaxis en cuatro de los cinco constructores de Tiempo2 para que la clase sea más fácil de mantener y modificar. Si necesitamos cambiar la forma en que se inicializan los objetos de la clase Tiempo2, sólo hay que modificar el constructor al que necesitan llamar los demás constructores de la clase. Incluso hasta ese constructor podría no requerir de modificación en este ejemplo. Sólo llama al método establecerTiempo para realizar la verdadera inicialización, por lo que es posible que los cambios que pudiera requerir la clase se localicen en los métodos *establecer*.

Error común de programación 8.3

Es un error de sintaxis utilizar this *en el cuerpo de un constructor para llamar a otro de la misma clase, si esa llamada no es la primera instrucción en el constructor. También es un error de sintaxis cuando un método trata de invocar a un constructor directamente, mediante* this.

Error común de programación 8.4

Un constructor puede llamar a los métodos de la clase. Tenga en cuenta que tal vez las variables de instancia no estén aún inicializadas, ya que el constructor está en el proceso de inicializar el objeto. El uso de variables de instancia antes de inicializarlas en forma apropiada es un error lógico.

Las líneas 18 a 21 declaran un constructor de Tiempo2 con un solo parámetro int que representa la hora, que se pasa con 0 para minuto y segundo al constructor de las líneas 30 a 33. Las líneas 24 a 27 declaran un constructor de Tiempo2 que recibe dos parámetros int, los cuales representan la hora y el minuto, que se pasan con un 0 para segundo al constructor de las líneas 30 a 33. Al igual que el constructor sin argumentos, cada uno de estos constructores invoca al constructor en las líneas 30 a 33 para minimizar la duplicación de código. Las líneas 30 a 33 declaran el constructor Tiempo2 que recibe tres parámetros int, los cuales representan la hora, el minuto y el segundo. Este constructor llama a establecerTiempo para inicializar las variables de instancia.

Las líneas 36 a 40 declaran un constructor de Tiempo2 que recibe una referencia a otro objeto Tiempo2. En este caso, los valores del argumento Tiempo2 se pasan al constructor de tres argumentos en las líneas 30 a 33 para inicializar hora, minuto y segundo. La línea 39 podría haber accedido en forma directa a los valores hora, minuto y segundo del argumento tiempo del constructor con las expresiones tiempo.hora, tiempo.minuto y tiempo.segundo, aun cuando hora, minuto y segundo se declaran como variables private de la clase Tiempo2. Esto se debe a una relación especial entre los objetos de la misma clase. En un momento veremos por qué es preferible utilizar los métodos *obtener*.

Observación de ingeniería de software 8.3

Cuando un objeto de una clase tiene una referencia a otro objeto de la misma clase, el primer objeto puede acceder a todos los datos y métodos del segundo (incluyendo los que sean private*).*

El método establecerTiempo *de la clase* Tiempo2

El método establecerTiempo (líneas 45 a 50) invoca a los métodos establecerHora (líneas 53 a 59), establecerMinuto (líneas 62 a 68) y establecerSegundo (líneas 71 a 77), los cuales aseguran que el valor suministrado para hora esté en el rango de 0 a 23, y que los valores para minuto y segundo estén cada uno en el rango de 0 a 59. Si un valor está fuera de rango, cada uno de estos métodos lanza una excepción IllegalArgumentException (líneas 58, 67 y 76) para indicar cuál valor estaba fuera de rango.

Notas acerca de los métodos establecer *y* obtener, *y los constructores de la clase* Tiempo2

Los métodos *establecer* y *obtener* de Tiempo2 se llaman en el cuerpo de la clase. En especial, el método establecerTiempo llama a los métodos establecerHora, establecerMinuto y establecerSegundo en

las líneas 47 a 49, y los métodos `aStringUniversal` y `toString` llaman a los métodos `obtenerHora`, `obtenerMinuto` y `obtenerSegundo` en la línea 93 y en las líneas 100 y 101. En cada caso, estos métodos podrían haber accedido a los datos privados de la clase en forma directa, sin necesidad de llamar a los métodos *establecer* y *obtener*. Sin embargo, considere la acción de cambiar la representación del tiempo, de tres valores `int` (que requieren 12 bytes de memoria) a un solo valor `int` que represente el número total de segundos transcurridos a partir de medianoche (que requiere sólo 4 bytes de memoria). Si hacemos ese cambio, sólo tendrían que modificar los cuerpos de los métodos que acceden en forma directa a los datos `private`; en especial, los métodos *establecer* y *obtener* individuales para `hora`, `minuto` y `segundo`. No habría necesidad de modificar los cuerpos de los métodos `establecerTiempo`, `aStringUniversal` o `toString`, ya que no acceden directamente a los datos. Si se diseña la clase de esta forma, se reduce la probabilidad de que se produzcan errores de programación al momento de alterar la implementación de la clase.

De manera similar, cada constructor de `Tiempo2` podría incluir una copia de las instrucciones apropiadas de los métodos `establecerHora`, `establecerMinuto` y `establecerSegundo`. Esto sería un poco más eficiente, ya que se eliminan las llamadas extra al constructor y a `establecerTiempo`. No obstante, *duplicar* las instrucciones en varios métodos o constructores dificulta más el proceso de modificar la representación de datos interna de la clase. Si hacemos que los constructores de `Tiempo2` llamen al constructor con tres argumentos (o que incluso llamen a `establecerTiempo` directamente), cualquier modificación a la implementación de `establecerTiempo` sólo tendrá que hacerse una vez. Además, el compilador puede optimizar los programas al eliminar las llamadas a los métodos simples y reemplazarlas con el código expandido de sus declaraciones; una técnica conocida como **código en línea**, lo cual mejora el rendimiento del programa.

Observación de ingeniería de software 8.4

Al implementar un método de una clase, use los métodos establecer y obtener de la clase para acceder a sus datos `private`. Esto simplifica el mantenimiento del código y reduce la probabilidad de errores.

Uso de los constructores sobrecargados de la clase `Tiempo2`

La clase `PruebaTiempo2` (figura 8.6) invoca a los constructores sobrecargados de `Tiempo2` (líneas 8 a 12 y 40). La línea 8 invoca al constructor sin argumentos (figura 8.5, líneas 12 a 15). Las líneas 9 a 13 del programa demuestran el paso de argumentos a los demás constructores de `Tiempo2`. La línea 9 invoca al constructor de un solo argumento que recibe un valor `int` en las líneas 18 a 21 de la figura 8.5. La línea 10 invoca al constructor de dos argumentos en las líneas 24 a 27 de la figura 8.5. La línea 11 invoca al constructor de tres argumentos en las líneas 30 a 33 de la figura 8.5. La línea 12 invoca al constructor de un solo argumento que recibe un objeto `Time2` en las líneas 36 a 40 de la figura 8.5. A continuación, la aplicación muestra en pantalla las representaciones `String` de cada objeto `Tiempo2`, para confirmar que cada uno de ellos se haya inicializado en forma apropiada. La línea 40 intenta inicializar `t6` mediante la creación de un nuevo objeto `Tiempo2` y al pasar tres valores inválidos al constructor. Cuando el constructor intenta usar el valor de hora inválido para inicializar la hora del objeto, ocurre una excepción `IllegalArgumentException`. La cual atrapamos en la línea 42 y mostramos su mensaje de error, que se produce en la última línea de la salida.

```
1  // Fig. 8.6: PruebaTiempo2.java
2  // Uso de constructores sobrecargados para inicializar objetos Tiempo2.
3
4  public class PruebaTiempo2
5  {
```

Fig. 8.6 | Uso de constructores sobrecargados para inicializar objetos `Tiempo2` (parte 1 de 3).

```
6      public static void main( String[] args )
7      {
8         Tiempo2 t1 = new Tiempo2();  // 00:00:00
9         Tiempo2 t2 = new Tiempo2( 2 );  // 02:00:00
10        Tiempo2 t3 = new Tiempo2( 21, 34 );  // 21:34:00
11        Tiempo2 t4 = new Tiempo2( 12, 25, 42 ); // 12:25:42
12        Tiempo2 t5 = new Tiempo2( t4 ); // 12:25:42
13
14        System.out.println( "Se construyo con:" );
15        System.out.println( "t1: todos los argumentos predeterminados" );
16        System.out.printf( "   %s\n", t1.aStringUniversal() );
17        System.out.printf( "   %s\n", t1.toString() );
18
19        System.out.println(
20           "t2: se especifico hora; minuto y segundo predeterminados" );
21        System.out.printf( "   %s\n", t2.aStringUniversal() );
22        System.out.printf( "   %s\n", t2.toString() );
23
24        System.out.println(
25           "t3: se especificaron hora y minuto; segundo predeterminado" );
26        System.out.printf( "   %s\n", t3.aStringUniversal() );
27        System.out.printf( "   %s\n", t3.toString() );
28
29        System.out.println( "t4: se especificaron hora, minuto y segundo" );
30        System.out.printf( "   %s\n", t4.aStringUniversal() );
31        System.out.printf( "   %s\n", t4.toString() );
32
33        System.out.println( "t5: se especifico el objeto Tiempo2 llamado t4" );
34        System.out.printf( "   %s\n", t5.aStringUniversal() );
35        System.out.printf( "   %s\n", t5.toString() );
36
37        // intento de inicializar t6 con valores invalidos
38        try
39        {
40           Tiempo2 t6 = new Tiempo2( 27, 74, 99); // valores invalidos
41        } // fin de try
42        catch ( IllegalArgumentException e )
43        {
44           System.out.printf( "\nExcepcion al inicializar t6: %s\n",
45              e.getMessage() );
46        } // fin de catch
47     } // fin de main
48  } // fin de la clase PruebaTiempo2
```

```
Se construyo con:
t1: todos los argumentos predeterminados
   00:00:00
   12:00:00 AM
t2: se especifico hora; minuto y segundo predeterminados
   02:00:00
   2:00:00 AM
```

Fig. 8.6 | Uso de constructores sobrecargados para inicializar objetos `Tiempo2` (parte 2 de 3).

```
t3: se especificaron hora y minuto; segundo predeterminado
   21:34:00
   9:34:00 PM
t4: se especificaron hora, minuto y segundo
   12:25:42
   12:25:42 PM
t5: se especifico el objeto Tiempo2 llamado t4
   12:25:42
   12:25:42 PM

Excepcion al inicializar t6: hora debe ser de 0 a 23
```

Fig. 8.6 | Uso de constructores sobrecargados para inicializar objetos `Tiempo2` (parte 3 de 3).

8.6 Constructores predeterminados y sin argumentos

Toda clase debe tener cuando menos un constructor. Si no se proporcionan constructores en la declaración de una clase, el compilador crea un constructor predeterminado que no recibe argumentos cuando se le invoca. El constructor predeterminado inicializa las variables de instancia con los valores iniciales especificados en sus declaraciones, o con sus valores predeterminados (cero para los tipos primitivos numéricos, `false` para los valores `boolean` y `null` para las referencias). En la sección 9.4.1 aprenderá que el constructor predeterminado realiza otra tarea también.

Si su clase declara constructores, el compilador *no* creará uno predeterminado. En este caso, debe declarar un constructor sin argumentos si se requiere una inicialización predeterminada. Al igual que un constructor predeterminado, uno sin argumentos se invoca con paréntesis vacíos. El constructor sin argumentos de `Tiempo2` (líneas 12 a 15 de la figura 8.5) inicializa en forma explícita un objeto `Tiempo2`; para ello pasa un 0 a cada parámetro del constructor con tres argumentos. Como 0 es el valor predeterminado para las variables de instancia `int`, el constructor sin argumentos en este ejemplo podría declararse con un cuerpo vacío. En este caso, cada variable de instancia recibiría su valor predeterminado al momento de llamar al constructor sin argumentos. Si omitimos este constructor, los clientes de esta clase no podrían crear un objeto `Tiempo2` con la expresión `new Tiempo2()`.

Error común de programación 8.5

Si un programa intenta inicializar un objeto de una clase al pasar el número incorrecto de tipos de argumentos a su constructor, ocurre un error de compilación.

Tip para prevenir errores 8.2

Asegúrese de no incluir un tipo de valor de retorno en la definición de un constructor. Java permite que otros métodos de la clase, además de sus constructores, tengan el mismo nombre de la clase y especifiquen tipos de valores de retorno. Dichos métodos no son constructores, por lo que no se llaman cuando se crea una instancia de un objeto de la clase.

8.7 Observaciones acerca de los métodos *Establecer* y *Obtener*

Como sabe, los campos `private` de una clase pueden manipularse sólo mediante sus métodos. Una manipulación típica podría ser el ajuste del saldo bancario de un cliente (por ejemplo, una variable de instancia `private` de una clase llamada `CuentaBancaria`) mediante un método llamado `calcularInteres`. Las clases a menudo proporcionan métodos `public` para permitir a los clientes de la clase *establecer* (es decir, asignar valores a) u *obtener* (es decir, recibir los valores de) variables de instancia `private`.

Como ejemplo de nomenclatura, un método para establecer la variable de instancia `tasaInteres` se llamaría típicamente `establecerTasaInteres`, y un método para obtener la `tasaDeInteres` se llamaría `obtenerTasaInteres`. Los métodos *establecer* también se conocen por lo común como **métodos mutadores**, porque por lo general cambian el estado de un objeto; es decir, modificar los valores de las variables de instancia. Los métodos *obtener* también se conocen en general como **métodos de acceso** o **métodos de consulta**.

Comparación entre los métodos Establecer *y* Obtener, *y los datos* `public`

Parece ser que proporcionar herramientas para *establecer* y *obtener* es en esencia lo mismo que hacer las variables de instancia `public`. Ésta es una sutileza de Java que hace del lenguaje algo tan deseable para la ingeniería de software. Si una variable de instancia se declara como `public`, cualquier método que tenga una referencia a un objeto que contenga esta variable de instancia podrá leer o escribir en ella. Si una variable de instancia se declara como `private`, no hay duda de que un método *obtener* `public` permite a otros métodos el acceso a la variable, pero el método *obtener* puede *controlar* la manera en que el cliente puede tener acceso a ella. Por ejemplo, un método *obtener* podría controlar el formato de los datos que devuelve y, por ende, proteger el código cliente de la representación actual de los datos. Un método *establecer* `public` puede (y debe) escudriñar con cuidado los intentos por modificar el valor de la variable, y lanzar una excepción si es necesario. Por ejemplo, un intento por *establecer* el día del mes en una fecha 37 sería rechazado, un intento por *establecer* el peso de una persona en un valor negativo sería negado, y así en lo sucesivo. Entonces, aunque los métodos *establecer* y *obtener* proporcionan acceso a los datos `private`, el acceso se restringe mediante la implementación de los métodos. Esto ayuda a promover la buena ingeniería de software.

Comprobación de validez en los métodos Establecer

Los beneficios de la integridad de los datos no se dan de manera automática sólo porque las variables de instancia se declaren como `private`; el programador debe proporcionar la comprobación de su validez. Java nos permite diseñar mejores programas de una manera conveniente. Los métodos *establecer* de una clase pueden devolver valores que indiquen que hubo intentos de asignar datos inválidos a los objetos de la clase. Un cliente de la clase puede probar el valor de retorno de un método *establecer* para determinar si el intento del cliente por modificar el objeto tuvo éxito, y entonces tomar la acción apropiada. Sin embargo, por lo general los métodos *establecer* tienen un tipo de valor de retorno `void` y utilizan el manejo de excepciones para indicar los intentos de asignar datos inválidos. En el capítulo 11 veremos con detalle el manejo de excepciones.

Observación de ingeniería de software 8.5

Cuando sea apropiado, proporcione métodos `public` *para cambiar y obtener los valores de las variables de instancia* `private`*. Esta arquitectura ayuda a ocultar la implementación de una clase a sus clientes, lo cual mejora la capacidad de modificación de un programa.*

Tip para prevenir errores 8.3

Utilizar métodos establecer *y* obtener *nos ayuda a crear clases que sean más fáciles de depurar y mantener. Si sólo un método realiza una tarea específica, como establecer la hora en un objeto* `Tiempo2`*, es más fácil depurar y mantener esa clase. Si la hora no se establece en forma apropiada, el código que modifica la variable de instancia hora se localiza en el cuerpo de un método:* `establecerHora`*. Así, sus esfuerzos de depuración pueden enfocarse en el método* `establecerHora`*.*

Métodos predicados

Otro uso común de los métodos de acceso es para evaluar si una condición es verdadera o falsa; por lo general, a dichos métodos se les llama **métodos predicados**. Un ejemplo sería el método `isEmpty` de la clase `ArrayList`, el cual devuelve `true` si el objeto `ArrayList` está vacío. Un programa podría evaluar el método `estaVacio` antes de tratar de leer otro elemento de un objeto `ArrayList`.

8.8 Composición

Una clase puede tener referencias a objetos de otras clases como miembros. A dicha capacidad se le conoce como **composición** y algunas veces como **relación *tiene un***. Por ejemplo, un objeto de la clase `RelojAlarma` necesita saber la hora actual *y* la hora en la que se supone sonará su alarma, por lo que es razonable incluir *dos* referencias a objetos `Tiempo` como miembros del objeto `RelojAlarma`.

La clase `Fecha`

El siguiente ejemplo de composición contiene tres clases: `Fecha` (figura 8.7), `Empleado` (figura 8.8) y `PruebaEmpleado` (figura 8.9). La clase `Fecha` (figura 8.7) declara las variables de instancia `mes`, `dia` y `anio` (líneas 6 a 8) para representar una fecha. El constructor recibe tres parámetros `int`. La línea 17 invoca el método utilitario `comprobarMes` (líneas 26 a 32) para validar el mes; si el valor está fuera de rango, el método lanza una excepción. La línea 15 asume que el valor de `anio` es correcto y no lo valida. La línea 19 invoca al método utilitario `comprobarDia` (líneas 35 a 48) para validar el `dia` con base en el `mes` y `anio` actuales. La líneas 38 determina si el día es correcto, con base en el número de días en el `mes` específico. Si el día no es correcto, las líneas 42 y 43 determinan si el `mes` es Febrero, el día 29 y el `anio` un año bisiesto. Si el `dia` sigue siendo inválido, el método lanza una excepción. Las líneas 21 y 22 en el constructor muestran en pantalla la referencia `this` como un objeto `String`. Puesto que `this` es una referencia al objeto `Fecha` actual, se hace una llamada *implícita* al método `toString` (líneas 51 a 54) para obtener la representación `String` del objeto.

```
1  // Fig. 8.7: Fecha.java
2  // Declaración de la clase Fecha.
3
4  public class Fecha
5  {
6     private int mes; // 1-12
7     private int dia; // 1-31 con base en el mes
8     private int anio; // cualquier año
9
10    private static final int[] diasPorMes = // días en cada mes
11       { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
12
13    // constructor: llama a comprobarMes para confirmar el valor apropiado para el mes;
14    // llama a comprobarDia para confirmar el valor apropiado para el día
15    public Fecha( int elMes, int elDia, int elAnio )
16    {
17       mes = comprobarMes( elMes ); // valida el mes
18       anio = elAnio; // pudo validar el año
19       dia = comprobarDia( elDia ); // valida el día
20
21       System.out.printf(
22          "Constructor de objeto Fecha para la fecha %s\n", this );
23    } // fin del constructor de Fecha
24
25    // método utilitario para confirmar el valor apropiado del mes
26    private int comprobarMes( int mesPrueba )
27    {
28       if ( mesPrueba > 0 && mesPrueba <= 12 ) // valida el mes
29          return mesPrueba;
```

Fig. 8.7 | Declaración de la clase `Fecha` (parte 1 de 2).

```
30          else // mes es inválido
31             throw new IllegalArgumentException ( "el mes debe ser 1 a 12");
32       } // fin del método comprobarMes
33
34       // método utilitario para confirmar el valor apropiado del día, con base en el
            mes y el año
35       private int comprobarDia( int diaPrueba )
36       {
37       // comprueba si el día está dentro del rango para el mes
38       if ( diaPrueba > 0 && diaPrueba <= diasPorMes[ mes ] )
39          return diaPrueba;
40
41       // comprueba si es año bisiesto
42       if ( mes == 2 && diaPrueba == 29 && ( anio % 400 == 0 ||
43       ( anio % 4 == 0 && anio % 100 != 0 ) ) )
44          return diaPrueba;
45
46       throw new IllegalArgumentException(
47          "dia fuera de rango para el mes y anio especificados" );
48    } // fin del método comprobarDia
49
50    // devuelve un objeto String de la forma mes/dia/anio
51    public String toString()
52    {
53       return String.format( "%d/%d/%d", mes, dia, anio );
54    } // fin del método toString
55 } // fin de la clase Fecha
```

Fig. 8.7 | Declaración de la clase `Fecha` (parte 2 de 2).

La clase `Empleado`

La clase `Empleado` (figura 8.8) tiene las variables de instancia `primerNombre`, `apellidoPaterno`, `fechaNacimiento` y `fechaContratacion`. Los miembros `primerNombre` y `apellidoPaterno` (líneas 6 a 7) son referencias a objetos `String`. Los miembros `fechaNacimiento` y `fechaContratacion` (líneas 8 y 9) son referencias a objetos `Fecha`. Esto demuestra que una clase puede tener como variables de instancia referencias a objetos de otras clases. El constructor de `Empleado` (líneas 12 a 19) recibe cuatro parámetros: `nombre`, `apellido`, `fechaDeNacimiento` y `fechaDeContratacion`. Los objetos referenciados por los parámetros se asignan a las variables de instancia del objeto `Empleado`. Cuando se hace una llamada al método `toString` de la clase `Empleado`, éste devuelve un objeto `String` que contiene el nombre del empleado y las representaciones `String` de los dos objetos `Fecha`. Cada uno de estos objetos `String` se obtiene mediante una llamada *implícita* al método `toString` de la clase `Fecha`.

```
1  // Fig. 8.8: Empleado.java
2  // Clase Empleado con referencias a otros objetos.
3
4  public class Empleado
5  {
6     private String primerNombre;
7     private String apellidoPaterno;
8     private Fecha fechaNacimiento;
9     private Fecha fechaContratacion;
```

Fig. 8.8 | Clase `Empleado` con referencias a otros objetos (parte 1 de 2).

```
10
11    // constructor para inicializar nombre, fecha de nacimiento y fecha de contratación
12    public Empleado( String nombre, String apellido, Fecha fechaDeNacimiento,
13       Fecha fechaDeContratacion )
14    {
15       primerNombre = nombre;
16       apellidoPaterno = apellido;
17       fechaNacimiento = fechaDeNacimiento;
18       fechaContratacion = fechaDeContratacion;
19    } // fin del constructor de Empleado
20
21    // convierte Empleado a formato String
22    public String toString()
23    {
24       return String.format( "%s, %s  Contratado: %s  Cumpleanios: %s",
25          apellidoPaterno, primerNombre, fechaContratacion, fechaNacimiento );
26    } // fin del método toString
27 } // fin de la clase Empleado
```

Fig. 8.8 | Clase `Empleado` con referencias a otros objetos (parte 2 de 2).

La clase `PruebaEmpleado`

La clase `PruebaEmpleado` (figura 8.9) crea dos objetos `Fecha` (líneas 8 y 9) para representar la fecha de nacimiento y de contratación de un `Empleado`. La línea 10 crea un `Empleado` e inicializa sus variables de instancia, al pasar al constructor dos objetos `String` (que representan el nombre y el apellido del `Empleado`) y dos objetos `Fecha` (que representan la fecha de nacimiento y de contratación). La línea 12 invoca en forma implícita el método `toString` de `Empleado` para mostrar en pantalla los valores de sus variables de instancia y demostrar que el objeto se inicializó en forma apropiada.

```
1  // Fig. 8.9: PruebaEmpleado.java
2  // Demostración de la composición.
3
4  public class PruebaEmpleado
5  {
6     public static void main( String[] args )
7     {
8        Fecha nacimiento = new Fecha( 7, 24, 1949 );
9        Fecha contratacion = new Fecha( 3, 12, 1988 );
10       Empleado empleado = new Empleado( "Bob", "Blue", nacimiento, contratacion );
11
12       System.out.println( empleado );
13    } // fin de main
14 } // fin de la clase PruebaEmpleado
```

```
Constructor de objeto Fecha para la fecha 7/24/1949
Constructor de objeto Fecha para la fecha 3/12/1988
Blue, Bob  Contratado: 3/12/1988  Cumpleanios: 7/24/1949
```

Fig. 8.9 | Demostración de la composición.

8.9 Enumeraciones

En la figura 6.8 presentamos el tipo básico enum, que define a un conjunto de constantes que se representan como identificadores únicos. En ese programa, las constantes enum representaban el estado del juego. En esta sección, hablaremos sobre la relación entre los tipos enum y las clases. Al igual que las clases, todos los tipos enum son tipos por referencia. Un tipo enum se declara con una **declaración enum**, la cual es una lista separada por comas de constantes enum; la declaración puede incluir, de manera opcional, otros componentes de las clases tradicionales, como constructores, campos y métodos. Cada declaración enum declara a una clase enum con las siguientes restricciones:

1. Los tipos enum son implícitamente `final`, ya que declaran constantes que no deben modificarse.
2. Las constantes enum son implícitamente `static`.
3. Cualquier intento por crear un objeto de un tipo enum con el operador new produce un error de compilación.

Las constantes enum pueden emplearse en cualquier parte en donde sea posible utilizar las constantes, como en las etiquetas `case` de las instrucciones `switch`, y para controlar las instrucciones `for` mejoradas.

La figura 8.10 ilustra cómo declarar variables de instancia, un constructor y varios métodos en un tipo enum. La declaración enum (líneas 5 a 37) contiene dos partes: las constantes enum y los demás miembros del tipo enum. La primera parte (líneas 8 a 13) declara seis constantes enum. Cada una va seguida opcionalmente por argumentos que se pasan al **constructor de enum** (líneas 20 a 24). Al igual que los constructores que hemos visto en las clases, un constructor de enum puede especificar cualquier número de parámetros, y sobrecargarse. En este ejemplo, el constructor de enum tiene dos parámetros `String`. Para inicializar cada constante enum en forma apropiada, debe ir seguida de paréntesis que contienen dos argumentos `String`, los cuales se pasan al constructor de enum. La segunda parte (líneas 16 a 36) declara a los demás miembros del tipo enum: dos variables de instancia (líneas 16 y 17), un constructor (líneas 20 a 24) y dos métodos (líneas 27 a 30 y líneas 33 a 36).

```
1  // Fig. 8.10: Libro.java
2  // Declara un tipo enum con constructor y campos de instancia explícitos,
3  // junto con métodos de acceso para estos campos
4
5  public enum Libro
6  {
7     // declara constantes de tipo enum
8     JHTP( "Java How to Program", "2012" ),
9     CHTP( "C How to Program", "2007" ),
10    IW3HTP( "Internet & World Wide Web How to Program", "2008" ),
11    CPPHTP( "C++ How to Program", "2012" ),
12    VBHTP( "Visual Basic 2010 How to Program", "2011" ),
13    CSHARPHTP( "Visual C# 2010 How to Program", "2011" );
14
15    // campos de instancia
16    private final String titulo; // título del libro
17    private final String anioCopyright; // año de copyright
```

Fig. 8.10 | Declaración de un tipo enum con constructor, campos de instancia y métodos de acceso para estos campos (parte 1 de 2).

```
18
19    // constructor de enum
20    Libro( String tituloLibro, String anio )
21    {
22       titulo = tituloLibro;
23       anioCopyright = anio;
24    } // fin de constructor de enum Libro
25
26    // método de acceso para el campo titulo
27    public String obtenerTitulo()
28    {
29       return titulo;
30    } // fin del método obtenerTitulo
31
32    // método de acceso para el campo anioCopyright
33    public String obtenerAnioCopyright()
34    {
35       return anioCopyright;
36    } // fin del método obtenerAnioCopyright
37 } // fin de enum Libro
```

Fig. 8.10 | Declaración de un tipo `enum` con constructor, campos de instancia y métodos de acceso para estos campos (parte 2 de 2).

Las líneas 16 y 17 declaran las variables de instancia `titulo` y `anioCopyright`. Cada constante `enum` en `Libro` en realidad es un objeto de tipo `Libro` que tiene su propia copia de las variables de instancia `titulo` y `anioCopyright`. El constructor (líneas 20 a 24) recibe dos parámetros `String`, uno que especifica el título del libro y otro que determina su año de copyright. Las líneas 22 y 23 asignan estos parámetros a las variables de instancia. Las líneas 27 a 36 declaran dos métodos, que devuelven el título del libro y el año de copyright.

La figura 8.11 prueba el tipo `enum Libro` e ilustra cómo iterar a través de un rango de constantes `enum`. Para cada `enum`, el compilador genera el método `static values` (que se llama en la línea 12), el cual devuelve un arreglo de las constantes de `enum`, en el orden en el que se declararon. Las líneas 12 a 14 utilizan la instrucción `for` mejorada para mostrar todas las constantes declaradas en la `enum` llamada `Libro`. La línea 14 invoca los métodos `obtenerTitulo` y `obtenerAnioCopyright` de la `enum Libro` para obtener el título y el año de copyright asociados con la constante. Observe que cuando se convierte una constante `enum` en un objeto `String` (por ejemplo, `libro` en la línea 13), el identificador de la constante se utiliza como la representación `String` (por ejemplo, `JHTP` para la primera constante `enum`).

```
1  // Fig. 8.11: PruebaEnum.java
2  // Prueba del tipo enum Libro.
3  import java.util.EnumSet;
4
5  public class PruebaEnum
6  {
7     public static void main( String[] args )
8     {
9        System.out.println( "Todos los libros:\n" );
```

Fig. 8.11 | Prueba de un tipo `enum` (parte 1 de 2).

```
10
11         // imprime todos los libros en enum Libro
12         for ( Libro libro : Libro.values() )
13            System.out.printf( "%-10s%-45s%s\n", libro,
14               libro.obtenerTitulo(), libro.obtenerAnioCopyright() );
15
16         System.out.println( "\nMostrar un rango de constantes enum:\n" );
17
18         // imprime los primeros cuatro libros
19         for ( Libro libro : EnumSet.range( Libro.JHTP, Libro.CPPHTP ) )
20            System.out.printf( "%-10s%-45s%s\n", libro,
21               libro.obtenerTitulo(), libro.obtenerAnioCopyright() );
22      } // fin de main
23   } // fin de la clase PruebaEnum
```

```
Todos los libros:

JHTP       Java How to Program                           2012
CHTP       C How to Program                              2007
IW3HTP     Internet & World Wide Web How to Program      2008
CPPHTP     C++ How to Program                            2012
VBHTP      Visual Basic 2010 How to Program              2011
CSHARPHTP  Visual C# 2010 How to Program                 2011

Mostrar un rango de constantes enum:

JHTP       Java How to Program                           2012
CHTP       C How to Program                              2007
IW3HTP     Internet & World Wide Web How to Program      2008
CPPHTP     C++ How to Program                            2012
```

Fig. 8.11 | Prueba de un tipo enum (parte 2 de 2).

Las líneas 19 a 21 utilizan el método `static` **range** de la clase **EnumSet** (declarada en el paquete `java.util`) para mostrar un rango de las constantes de la enum `Libro`. El método `range` recibe dos parámetros (la primera y la última constantes enum en el rango) y devuelve un objeto `EnumSet` que contiene todas las constantes entre estas dos, ambas inclusive. Por ejemplo, la expresión `EnumSet.range( Libro.JHTP, Libro.CPPHTP )` devuelve un objeto `EnumSet` que contiene `Libro.JHTP`, `Libro.CHTP`, `Libro.IW3HTP` y `Libro.CPPHTP`. La instrucción `for` mejorada se puede utilizar con un objeto `EnumSet`, justo igual que como se utiliza con un arreglo, por lo que las líneas 12 a 14 la utilizan para mostrar el título y el año de copyright de cada libro en el objeto `EnumSet`. La clase `EnumSet` proporciona varios métodos `static` más para crear conjuntos de constantes enum a partir del mismo tipo de enum.

Error común de programación 8.6

En una declaración enum, *es un error de sintaxis declarar constantes* enum *después de los constructores, campos y métodos del tipo de* enum.

8.10 Recolección de basura y el método `finalize`

Toda clase en Java tiene los métodos de la clase `Object` (paquete `java.lang`), uno de los cuales es el método `finalize`. Este método se utiliza raras veces, debido a que puede provocar problemas de rendimiento y hay cierta incertidumbre en cuanto a si realmente se llamará o no. Sin embargo, y como `finalize`

forma parte de cada clase, hablaremos aquí sobre este método para que a usted se le facilite comprender su propósito planeado. Los detalles completos acerca del método `finalize` están más allá del alcance de este libro, además de que la mayoría de los programadores no deben usarlo; pronto veremos por qué. Aprenderá más acerca de la clase `Object` en el capítulo 9.

Todo objeto utiliza recursos del sistema, como la memoria. Necesitamos una manera disciplinada de regresarlos al sistema cuando ya no se necesitan; de lo contrario, podrían ocurrir "fugas de recursos" que impidan que nuestro programa, o tal vez hasta otros programas, los utilicen. La máquina virtual de Java (JVM) realiza la **recolección de basura** en forma automática para reclamar la memoria ocupada por los objetos que ya no se usan. Cuando ya no hay más referencias a un objeto, éste se convierte en candidato para la recolección de basura. Por lo general, esto ocurre cuando la JVM ejecuta su **recolector de basura**. Por lo tanto, las fugas de memoria que son comunes en otros lenguajes como C y C++ (debido a que en esos lenguajes, la memoria no se reclama de manera automática) son menos probables en Java, pero algunas pueden ocurrir de todas formas, aunque con menos magnitud. Pueden ocurrir otros tipos de fugas de recursos. Por ejemplo, una aplicación podría abrir un archivo en disco para modificar el contenido. Si la aplicación no cierra el archivo, ninguna otra aplicación puede utilizarlo sino hasta que termine la que lo abrió.

El recolector de basura llama al **método `finalize`** para realizar las **tareas de preparación para la terminación** sobre un objeto, justo antes de que el recolector de basura reclame la memoria de ese objeto. El método `finalize` no recibe parámetros y tiene el tipo de valor de retorno `void`. Un problema con el método `finalize` es que no se garantiza que el recolector de basura se ejecute en un tiempo especificado. De hecho, tal vez el recolector de basura nunca se ejecute antes de que termine un programa. Por ende, no queda claro si (o cuándo) se hará la llamada al método `finalize`. Por esta razón, la mayoría de los programadores deben evitar el uso del método `finalize`.

Observación de ingeniería de software 8.6

Una clase que utiliza recursos del sistema, como archivos en el disco, debe proporcionar un método que los programadores puedan llamar para liberar recursos cuando ya no se necesiten en un programa. Muchas clases de la API de Java proporcionan métodos `close` o `dispose` para este propósito. Por ejemplo, la clase `Scanner` tiene un método `close`. En la sección 11.13 hablaremos sobre las nuevas características de Java SE 7 relacionadas con este tema.

8.11 Miembros de clase `static`

Cada objeto tiene su propia copia de todas las variables de instancia de la clase. En ciertos casos, sólo debe *compartirse* una copia de cierta variable entre todos los objetos de una clase. En esos casos se utiliza un **campo `static`** (al cual se le conoce como una **variable de clase**). Una variable `static` representa **información en toda la clase**: todos los objetos de la clase comparten la *misma* pieza de datos. La declaración de una variable `static` comienza con la palabra clave `static`.

Veamos un ejemplo con datos `static`. Suponga que tenemos un videojuego con `Marcianos` y otras criaturas espaciales. Cada `Marciano` tiende a ser valiente y deseoso de atacar a otras criaturas espaciales cuando sabe que hay al menos otros cuatro `Marcianos` presentes. Si están presentes menos de cinco `Marcianos`, cada uno se vuelve cobarde. Por lo tanto, necesitan saber el valor de `cuentaMarcianos`. Podríamos dotar a la clase `Marciano` con la variable `cuentaMarcianos` como variable de instancia. Si hacemos esto, entonces cada `Marciano` tendrá una *copia separada* de la variable de instancia, y cada vez que creemos un nuevo `Marciano`, tendremos que actualizar la variable de instancia `cuentaMarcianos` en todos los objetos `Marciano`. Las copias redundantes desperdician espacio y tiempo en actualizar cada una de las copias de la variable, además de ser un proceso propenso a errores. En vez de ello, declaramos a `cuentaMarcianos` como `static`, lo cual convierte a `cuentaMarcianos` en datos disponibles en toda la clase. Cada objeto `Marciano` puede ver la `cuentaMarcianos` como si fuera una variable de instancia de

la clase Marciano, pero sólo se mantiene una copia de la variable static cuentaMarcianos. Esto nos ahorra espacio. Ahorramos tiempo al hacer que el constructor de Marciano incremente la variable static cuentaMarcianos; como sólo hay una copia, no tenemos que incrementar copias separadas para cada uno de los objetos Marciano.

Observación de ingeniería de software 8.7

Use una variable static *cuando todos los objetos de una clase tengan que utilizar la misma copia de la variable.*

Las variables static tienen alcance a nivel de clase. Los miembros public static de una clase pueden utilizarse a través de una referencia a cualquier objeto de esa clase, o calificando el nombre del miembro con el nombre de la clase y un punto (.), como en Math.random(). El código cliente puede acceder a los miembros private static de una clase solamente a través de los métodos de esa clase. En realidad, *los miembros* static *de una clase existen a pesar de que no existan objetos de esa clase*; están disponibles tan pronto como la clase se carga en memoria, en tiempo de ejecución. Para acceder a un miembro public static cuando no existen objetos de la clase (y aún cuando sí existen), se debe anteponer el nombre de la clase y un punto (.) al miembro static de la clase, como en Math.PI. Para acceder a un miembro private static cuando no existen objetos de la clase, debe proporcionarse un método public static, y para llamar a este método se debe calificar su nombre con el nombre de la clase y un punto.

Observación de ingeniería de software 8.8

Las variables de clase y los métodos static *existen, y pueden utilizarse, incluso aunque no se hayan instanciado objetos de esa clase.*

Un método static no puede acceder a los miembros no static de la clase, ya que a un método static se le puede invocar aún cuando no se hayan creado instancias de objetos de la clase. Por la misma razón, no es posible usar la referencia this en un método static. La referencia this debe referirse a un objeto específico de la clase, y cuando se hace la llamada a un método static, podría darse el caso de que no hubiera objetos de su clase en la memoria.

Error común de programación 8.7

Si un método static *llama a un método de instancia (no* static*) en la misma clase, sólo con el nombre del método, se produce un error de compilación. De manera similar, si un método* statictrata *de acceder a una variable de instancia en la misma clase, únicamente con el nombre de la variable, se produce un error de compilación.*

Error común de programación 8.8

Hacer referencia a this *en un método* static *es un error de compilación.*

Rastreo del número de objetos empleado creados

En nuestro siguiente programa declaramos dos clases: Empleado (figura 8.12) y PruebaEmpleado (figura 8.13). La clase Empleado declara la variable private static llamada cuenta (figura 8.12, línea 9), y el método public static llamado obtenerCuenta (líneas 36 a 39). La variable static cuenta se inicializa con cero en la línea 9. Si no se inicializa una variable static, el compilador asigna a esa variable un valor predeterminado (en este caso, 0, el valor predeterminado para el tipo int). La variable cuenta mantiene la cuenta del número de objetos de la clase Empleado que se han creado hasta ese momento.

Cuando existen objetos Empleado, la variable cuenta se puede utilizar en cualquier método de un objeto Empleado; este ejemplo incrementa cuenta en el constructor (línea 18). El método public static

```
1  // Fig. 8.12: Empleado.java
2  // Variable static que se utiliza para mantener una cuenta del
3  // número de objetos Empleado en la memoria.
4
5  public class Empleado
6  {
7     private String primerNombre;
8     private String apellidoPaterno;
9     private static int cuenta = 0; // número de objetos Empleado creados
10
11    // inicializa Empleado, suma 1 a la variable static cuenta e
12    // imprime objeto String que indica que se llamó al constructor
13    public Empleado( String nombre, String apellido )
14    {
15       primerNombre = nombre;
16       apellidoPaterno = apellido;
17
18       ++cuenta;  // incrementa la variable static cuenta de empleados
19       System.out.printf( "Constructor de Empleado: %s %s; cuenta = %d\n",
20          primerNombre, apellidoPaterno, cuenta );
21    } // fin de constructor de Empleado
22
23    // obtiene el primer nombre
24    public String obtenerPrimerNombre()
25    {
26       return primerNombre;
27    } // fin del método obtenerPrimerNombre
28
29    // obtiene el apellido paterno
30    public String obtenerApellidoPaterno()
31    {
32       return apellidoPaterno;
33    } // fin del método obtenerApellidoPaterno
34
35    // método static para obtener el valor de la variable static cuenta
36    public static int obtenerCuenta()
37    {
38       return cuenta;
39    } // fin del método obtenerCuenta
40 } // fin de la clase Empleado
```

Fig. 8.12 | Variable `static` que se utiliza para mantener la cuenta del número de objetos `Empleado` en memoria.

`obtenerCuenta` (líneas 36 a 39) devuelve el número de objetos `Empleado` que se han creado hasta ese momento. Cuando no existen objetos de la clase `Empleado`, el código cliente puede acceder a la variable cuenta mediante una llamada al método `obtenerCuenta` a través del nombre de la clase, como en `Empleado.obtenerCuenta()`. Cuando existen objetos, también se puede llamar el método `obtenerCuenta` a través de cualquier referencia a un objeto `Empleado`.

Buena práctica de programación 8.1

Para invocar a cualquier método `static`, utilice el nombre de la clase y un punto (.) para enfatizar que el método que se está llamando es un método `static`.

El método main de PruebaEmpleado (figura 8.13) crea instancias de dos objetos Empleado (líneas 13 y 14). Cuando se invoca el constructor de cada objeto Empleado, en las líneas 15 y 16 de la figura 8.12 se asigna el primer nombre y el apellido paterno del Empleado a las variables de instancia primerNombre y apellidoPaterno. Estas dos instrucciones *no* sacan copias de los argumentos String originales. En realidad, los objetos String en Java son **inmutables** (no pueden modificarse una vez que son creados). Por lo tanto, es seguro tener muchas referencias a un solo objeto String. Éste no es normalmente el caso para los objetos de la mayoría de las otras clases en Java. Si los objetos String son inmutables, tal vez se pregunte por qué podemos utilizar los operadores + y += para concatenar objetos String. En realidad, las operaciones de concatenación de objetos String producen un *nuevo* objeto String, el cual contiene los valores concatenados. Los objetos String originales no se modifican.

Cuando main ha terminado de usar los dos objetos Empleado, las referencias e1 y e2 se establecen en null, en las líneas 31 y 32. En este punto, las referencias e1 y e2 ya no hacen referencia a los objetos que se instanciaron en las líneas 13 y 14. Esto "marca a los objetos para la recolección de basura", ya que no existen más referencias a esos objetos en el programa.

```
1  // Fig. 8.13: PruebaEmpleado.java
2  // Demostración de miembros static.
3
4  public class PruebaEmpleado
5  {
6     public static void main( String[] args )
7     {
8        // muestra que la cuenta es 0 antes de crear Empleados
9        System.out.printf( "Empleados antes de instanciar: %d\n",
10          Empleado.obtenerCuenta() );
11
12       // crea dos Empleados; la cuenta debe ser 2
13       Empleado e1 = new Empleado( "Susan", "Baker" );
14       Empleado e2 = new Empleado( "Bob", "Blue" );
15
16       // muestra que la cuenta es 2 después de crear dos Empleados
17       System.out.println( "\nEmpleados despues de instanciar: " );
18       System.out.printf( "mediante e1.obtenerCuenta(): %d\n", e1.obtenerCuenta() );
19       System.out.printf( "mediante e2.obtenerCuenta(): %d\n", e2.obtenerCuenta() );
20       System.out.printf( "mediante Empleado.obtenerCuenta(): %d\n",
21          Empleado.obtenerCuenta() );
22
23       // obtiene los nombres de los Empleados
24       System.out.printf( "\nEmpleado 1: %s %s\nEmpleado 2: %s %s\n\n",
25          e1.obtenerPrimerNombre(), e1.obtenerApellidoPaterno(),
26          e2.obtenerPrimerNombre(), e2.obtenerApellidoPaterno() );
27
28       // en este ejemplo, sólo hay una referencia a cada Empleado,
29       // por lo que las siguientes dos instrucciones indican que estos objetos
30       // son candidatos para la recolección de basura
31       e1 = null;
32       e2 = null;
33    } // fin de main
34 } // fin de la clase PruebaEmpleado
```

Fig. 8.13 | Demostración de miembros static (parte 1 de 2).

```
Empleados antes de instanciar: 0
Constructor de Empleado: Susan Baker; cuenta = 1
Constructor de Empleado: Bob Blue; cuenta = 2

Empleados despues de instanciar:
mediante e1.obtenerCuenta(): 2
mediante e2.obtenerCuenta(): 2
mediante Empleado.obtenerCuenta(): 2

Empleado 1: Susan Baker
Empleado 2: Bob Blue
```

Fig. 8.13 | Demostración de miembros `static` (parte 2 de 2).

De un momento a otro, el recolector de basura podría reclamar la memoria para estos objetos (o el sistema operativo la reclama cuando el programa termina). La JVM no garantiza cuándo se va a ejecutar el recolector de basura (o si acaso se va a ejecutar). Cuando lo haga, es posible que no se recolecte ningún objeto, o que sólo se recolecte un subconjunto de los objetos candidatos.

8.12 Declaración `static import`

En la sección 6.3 aprendió acerca de los campos y métodos `static` de la clase `Math`. Para invocar a estos campos y métodos, anteponemos a cada uno de ellos el nombre de la clase `Math` y un punto (`.`). Una declaración **`static import`** nos permite importar los miembros `static` de una clase o interfaz, para poder acceder a ellos mediante sus nombres no calificados en nuestra clase; el nombre de la clase y el punto (`.`) no se requieren para usar un miembro `static` importado.

Una declaración `static` import tiene dos formas: una que importa un miembro `static` específico (que se conoce como declaración **`static import` individual**) y una que importa a *todos* los miembros `static` de una clase (que se conoce como declaración **`static import` sobre demanda**). La siguiente sintaxis importa un miembro `static` específico:

```
import static nombrePaquete.NombreClase.nombreMiembroEstático;
```

en donde *nombrePaquete* es el paquete de la clase (por ejemplo, `java.lang`), *NombreClase* es el nombre de la clase (por ejemplo, `Math`) y *nombreMiembroEstático* es el nombre del campo o método `static` (por ejemplo, `PI` o `abs`). La siguiente sintaxis importa todos los miembros `static` de una clase:

```
import static nombrePaquete.NombreClase.*;
```

El asterisco (`*`) indica que *todos* los miembros `static` de la clase especificada deben estar disponibles para usarlos en el archivo. Las declaraciones `static` import sólo importan miembros de clase `static`. Las instrucciones `import` regulares deben usarse para especificar las clases utilizadas en un programa.

La figura 8.14 demuestra una declaración `static` import. La línea 3 es una declaración `static` import, la cual importa todos los campos y métodos `static` de la clase `Math`, del paquete `java.lang`. Las líneas 9 a 12 acceden a los campos `static` `E` (línea 11) y `PI` (línea 12) de la clase `Math`, y los métodos `static` `sqrt` (línea 9) y `ceil` (línea 10) sin anteponer el nombre de la clase `Math` y un punto a los nombres del campo o los métodos.

Error común de programación 8.9

Si un programa trata de importar métodos `static` que tengan la misma firma, o campos `static` que tengan el mismo nombre, de dos o más clases, se produce un error de compilación.

```
1  // Fig. 8.14: PruebaStaticImport.java
2  // Uso de static import para importar métodos de la clase Math.
3  import static java.lang.Math.*;
4
5  public class PruebaStaticImport
6  {
7     public static void main( String[] args )
8     {
9        System.out.printf( "sqrt( 900.0 ) = %.1f\n", sqrt( 900.0 ) );
10       System.out.printf( "ceil( -9.8 ) = %.1f\n", ceil( -9.8 ) );
11       System.out.printf( "E = %f\n", E );
12       System.out.printf( "PI = %f\n", PI );
13    } // fin de main
14 } // fin de la clase PruebaStaticImport
```

```
sqrt( 900.0 ) = 30.0
ceil( -9.8 ) = -9.0
E = 2.718282
PI = 3.141593
```

Fig. 8.14 | Importación static de métodos de la clase `Math`.

8.13 Variables de instancia `final`

El **principio del menor privilegio** es fundamental para la buena ingeniería de software. En el contexto de una aplicación, el principio establece que al código sólo se le debe otorgar tanto privilegio y acceso como necesite para llevar a cabo su tarea designada, pero no más. Esto hace a sus programas más robustos, al evitar que el código modifique por accidente (o de manera intencional) los valores de las variables y haga llamadas a métodos que no deben ser accesibles.

Veamos ahora cómo se aplica este principio a las variables de instancia. Algunas de ellas necesitan modificarse, mientras que otras no. Usted puede utilizar la palabra clave `final` para especificar que una variable no puede modificarse (es decir, que sea una constante) y que cualquier intento por modificarla sería un error. Por ejemplo,

```
private final int INCREMENTO;
```

declara una variable de instancia `final` (constante) llamada `INCREMENTO`, de tipo `int`. Dichas variables se pueden inicializar al momento de declararse. De lo contrario, se *debe hacer* en cada uno de los constructores de la clase. Al inicializar las constantes en los constructores, cada objeto de la clase puede tener un valor distinto para la constante. Si una variable `final` no se inicializa en su declaración o en cada constructor, se produce un error de compilación.

Observación de ingeniería de software 8.9

Declarar una variable de instancia como `final` ayuda a hacer valer el principio del menor privilegio. Si una variable de instancia no debe modificarse, declárela como `final` para evitar su modificación.

Error común de programación 8.10

Tratar de modificar una variable de instancia `final` después de inicializarla es un error de compilación.

Tip para prevenir errores 8.4

Los intentos por modificar una variable de instancia `final` se atrapan en tiempo de compilación, en vez de producir errores en tiempo de ejecución. Siempre es preferible sacar los errores en tiempo de compilación, si es posible, en vez de permitir que se pasen hasta el tiempo de ejecución (en donde la experiencia nos ha demostrado que la reparación es, a menudo, mucho más costosa).

Observación de ingeniería de software 8.10

Un campo `final` también debe declararse como `static`, si se inicializa en su declaración con un valor que sea el mismo para todos los objetos de la clase. Después de la inicialización, su valor ya no puede cambiar. Por lo tanto, no es necesario tener una copia separada del campo para cada objeto de la clase. Al hacer a ese campo `static`, se permite que todos los objetos de la clase compartan el campo `final`.

8.14 Caso de estudio de la clase `Tiempo`: creación de paquetes

En casi todos los ejemplos de este libro hemos visto que las clases de bibliotecas preexistentes, como la API de Java, pueden importarse en un programa en Java. Cada clase en la API de Java pertenece a un paquete que contiene un grupo de clases relacionadas. Estos paquetes se definen una vez, pero se pueden importar en muchos programas. A medida que las aplicaciones se vuelven más complejas, los paquetes nos ayudan a administrar la complejidad de los componentes de una aplicación. Los paquetes también facilitan la reutilización de software, al permitir que los programas *importen* clases de otros paquetes (como lo hemos hecho en la mayoría de los ejemplos), en vez de *copiar* las clases en cada uno de los programas que las utiliza. Otro beneficio de los paquetes es que proporcionan una convención para los nombres de clases únicos, lo cual ayuda a evitar los conflictos de nombres de clases (que veremos más adelante en esta sección). En esta sección veremos cómo crear sus propios paquetes.

Pasos para declarar una clase reutilizable

Antes de poder importar una clase en varias aplicaciones, ésta debe colocarse en un paquete para que sea reutilizable. La figura 8.15 muestra cómo especificar el paquete en el que debe colocarse una clase. La figura 8.16 muestra cómo importar nuestra clase empaquetada, para poder usarla en una aplicación. Los pasos para crear una clase reutilizable son:

1. Declare una clase `public`. De lo contrario, sólo la podrán usar otras clases en el mismo paquete.
2. Seleccione un nombre único para el paquete y agregue una **declaración package** al archivo de código fuente para la declaración de la clase reutilizable. Sólo puede haber una declaración `package` en cada archivo de código fuente de Java, y debe ir antes que todas las demás declaraciones e instrucciones en el archivo. Los comentarios no son instrucciones, por lo que pueden colocarse antes de una instrucción `package` en un archivo. [*Nota:* si no se proporciona una instrucción `package`, la clase se coloca en lo que se conoce como paquete predeterminado y sólo es accesible para las demás clases en el paquete predeterminado que se encuentran en el mismo directorio. Todos los programas anteriores en este libro que tengan dos o más clases, han usado este paquete predeterminado].
3. Compile la clase de manera que se coloque en la estructura de directorio del paquete apropiada.
4. Importe la clase reutilizable en un programa, y utilícela.

Ahora veremos cada uno de estos pasos con detalle.

Pasos 1 y 2: crear una clase `public` *y agregar la instrucción* `package`

Para el *paso 1*, modificaremos la clase `public` `Tiempo1` que declaramos en la figura 8.1. La nueva versión se muestra en la figura 8.15. No se han hecho modificaciones a la implementación de la clase, por lo que no hablaremos otra vez aquí sobre sus detalles de implementación.

Para el *paso 2* agregamos una declaración `package` (línea 3), la cual declara a un paquete llamado `com.deitel.jhtp.cap08`. Al colocar una declaración `package` al principio de un archivo de código fuente de Java, indicamos que la clase declarada en el archivo forma parte del paquete especificado. Sólo las declaraciones `package`, las declaraciones `import` y los comentarios pueden aparecer fuera de las llaves de una declaración de clase. Un archivo de código fuente de Java debe tener el siguiente orden:

```
1  // Fig. 8.15: Tiempo1.java
2  // La declaración de la clase Tiempo1 mantiene la hora en formato de 24 horas.
3  package com.deitel.jhtp.cap08;
4
5  public class Tiempo1
6  {
7     private int hora; // 0 - 23
8     private int minuto; // 0 - 59
9     private int segundo; // 0 - 59
10
11    // establece un nuevo valor de tiempo, usando la hora universal; lanza una
12    // excepción si la hora, minuto o segundo son inválidos
13    public void establecerTiempo( int h, int m, int s )
14    {
15       // valida la hora, el minuto y el segundo
16       if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
17          ( s >= 0 && s < 60 ))
18       {
19          hora = h;
20          minuto = m;
21          segundo = s;
22       } // fin de if
23       else
24          throw new IllegalArgumentException(
25             "hora, minuto y/o segundo estaban fuera de rango");
26    } // fin del método establecerTiempo
27
28    // convierte a objeto String en formato de hora universal (HH:MM:SS)
29    public String aStringUniversal()
30    {
31       return String.format( "%02d:%02d:%02d", hora, minuto, segundo );
32    } // fin del método aStringUniversal
33
34    // convierte a objeto String en formato de hora estándar (H:MM:SS AM o PM)
35    public String toString()
36    {
37       return String.format( "%d:%02d:%02d %s",
38          ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 ),
39          minuto, segundo, ( hora < 12 ? "AM" : "PM" ) );
40    } // fin del método toString
41 } // fin de la clase Tiempo1
```

Fig. 8.15 | Empaquetamiento de la clase `Tiempo1` para reutilizarla.

1. una declaración `package` (si la hay),
2. declaraciones `import` (si las hay), y después
3. declaraciones de clases.

Sólo una de las declaraciones de las clases en un archivo específico pueden ser `public`. Las demás clases en el archivo se colocan en el paquete, y sólo las pueden utilizar las otras clases en el mismo paquete. Las clases que no son `public` están en un paquete para dar soporte a las clases reutilizables en éste.

Para proveer nombres de paquetes únicos, empiece cada uno con su nombre de dominio de Internet en orden inverso. Por ejemplo, nuestro nombre de dominio es `deitel.com`, por lo que los nombres de nuestros paquetes empiezan con `com.deitel`. Para el nombre de dominio *suescuela*`.edu`, el nombre del paquete debe empezar con `edu`.*suescuela*. Una vez que se invierte el nombre del dominio, podemos elegir cualquier otro nombre que deseemos para nuestro paquete. Si usted forma parte de una empresa con muchas divisiones, o de una universidad con muchas escuelas, tal vez sea conveniente que utilice el nombre de su división o escuela como el siguiente nombre en el paquete. Nosotros optamos por usar `jhtp` como el siguiente nombre en nuestro paquete, para indicar que esta clase es del libro en inglés *Java How To Program*. El último nombre en nuestro paquete especifica que es para el capítulo 8 (`cap08`).

Paso 3: compilar la clase empaquetada

El *paso 3* es compilar la clase, de manera que se almacene en el paquete apropiado. Cuando se compila un archivo de Java que contiene una declaración `package`, el archivo de clase resultante se coloca en el directorio especificado por la declaración. La declaración `package` en la figura 8.15 indica que la clase `Tiempo1` debe colocarse en el siguiente directorio:

```
com
   deitel
      jhtp
         ch08
```

Los nombres en la declaración `package` especifican la ubicación exacta de las clases en el paquete.

Al compilar una clase en un paquete, la opción `-d` de la línea de comandos de `javac` hace que el compilador `javac` cree los directorios apropiados, con base en la declaración `package` de la clase. Esta opción también especifica en dónde se deben almacenar los directorios. Por ejemplo, en una ventana de comandos utilizamos el siguiente comando de compilación

```
javac -d . Tiempo1.java
```

para especificar que el primer directorio en el nombre de nuestro paquete debe colocarse en el directorio actual. El punto (`.`) después de `-d` en el comando anterior representa el directorio actual en los sistemas operativos Windows, UNIX, Linux y Mac OS X (y en varios otros también). Después de ejecutar el comando de compilación, el directorio actual contiene un directorio llamado `com`, el cual contiene uno llamado `deitel`, que a su vez contiene uno llamado `jhtp`, y este último contiene un directorio llamado `cap08`. En el directorio `cap08` podemos encontrar el archivo `Tiempo1.class`. [*Nota:* si no utiliza la opción `-d`, entonces debe copiar o mover el archivo de clase al directorio del paquete apropiado después de compilarlo].

El nombre `package` forma parte del **nombre de clase completamente calificado**, por lo que el nombre de la clase `Tiempo1` es en realidad `com.deitel.jhtp.cap08.Tiempo1`. Puede utilizar este nombre completamente calificado en sus programas, o puede importar la clase y utilizar su **nombre simple** (el nombre de la clase por sí solo: `Tiempo1`) en el programa. Si otro paquete contiene también una clase `Tiempo1`, los nombres de clase completamente calificados pueden utilizarse para diferenciar una clase de otra en el programa, y evitar un **conflicto de nombres** (también conocido como **colisión de nombres**).

Paso 4: importar la clase reutilizable

Una vez que la clase se compila y se guarda en su paquete, se puede importar en los programas (*paso 4*). En la aplicación PruebaPaqueteTiempo1 de la figura 8.16, la línea 3 especifica que la clase Tiempo1 debe importarse para usarla en la clase PruebaPaqueteTiempo1. Esta clase se encuentra en el paquete predeterminado, ya que su archivo .java no contiene una declaración package. Como las dos clases se encuentran en distintos paquetes, se requiere la declaración import en la línea 3, de manera que la clase PruebaPaqueteTiempo1 pueda utilizar la clase Tiempo1.

```
1  // Fig. 8.16: PruebaPaqueteTiempo1.java
2  // Uso de un objeto Tiempo1 en una aplicación.
3  import com.deitel.jhtp.cap08.Tiempo1; // importa la clase Tiempo1
4
5  public class PruebaPaqueteTiempo1
6  {
7     public static void main( String[] args )
8     {
9        // crea e inicializa un objeto Tiempo1
10       Tiempo1 tiempo = new Tiempo1(); // invoca al constructor de Tiempo1
11
12       // imprime representaciones String de la hora
13       System.out.print( "La hora universal inicial es: " );
14       System.out.println( tiempo.aStringUniversal() );
15       System.out.print( "La hora estandar inicial es: " );
16       System.out.println( tiempo.toString() );
17       System.out.println(); // imprime una línea en blanco
18
19       // cambia la hora e imprime la hora actualizada
20       tiempo.establecerTiempo( 13, 27, 6 );
21       System.out.print( "La hora universal despues de establecerTiempo es: " );
22       System.out.println( tiempo.aStringUniversal() );
23       System.out.print( "La hora estandar despues de establecerTiempo es: " );
24       System.out.println( tiempo.toString() );
25       System.out.println(); // imprime una línea en blanco
26
27       // intenta establecer la hora con valores inválidos
28       try
29       {
30          tiempo.establecerTiempo( 99, 99, 99 ); // todos los valores fuera de rango
31       } // fin de try
32       catch (IllegalArgumentException e )
33       {
34          System.out.printf( "Excepcion: %s\n\n", e.getMessage() );
35       } // fin de catch
36
37       // muestra la hora después de tratar de establecer valores inválidos
38       System.out.println( "Despues de intentar ajustes invalidos:" );
39       System.out.print( "Hora universal: " );
40       System.out.println( tiempo.aStringUniversal() );
41       System.out.print( "Hora estandar: " );
42       System.out.println( tiempo.toString() );
43    } // fin de main
44 } // fin de la clase PruebaPaqueteTiempo1
```

Fig. 8.16 | Uso del objeto Tiempo1 en una aplicación (parte 1 de 2).

```
La hora universal inicial es: 00:00:00
La hora estandar inicial es: 12:00:00 AM

La hora universal despues de establecerTiempo es: 13:27:06
La hora estandar despues de establecerTiempo es: 1:27:06 PM

Excepcion: hora, minuto y/o segundo estaban fuera de rango

Despues de intentar ajustes invalidos:
Hora universal: 13:27:06
Hora estandar: 1:27:06 PM
```

Fig. 8.16 | Uso del objeto `Tiempo1` en una aplicación (parte 2 de 2).

La línea 3 se conoce como una **declaración import de tipo simple**; es decir, la declaración `import` especifica una clase que se va a importar. Cuando su programa utiliza varias clases del mismo paquete, puede importarlas con una sola declaración `import`. Por ejemplo,

```
import java.util.*; // importa las clases del paquete java.util
```

usa un asterisco (*) al final de la misma para informar al compilador que todas las clases `public` del paquete `java.util` están disponibles para usarlas en el programa. Esto se conoce como una **declaración import tipo sobre demanda**. La JVM sólo carga las clases del paquete `java.util` que se utilizan en el programa. La declaración `import` anterior nos permite utilizar el nombre simple de cualquier clase del paquete `java.util` en el programa. A lo largo de este libro, utilizaremos declaraciones `import` tipo simples, por claridad.

Error común de programación 8.11

Utilizar la declaración `import java.*;` produce un error de compilación. Se debe especificar el nombre exacto del paquete del que se desea importar clases.

Especificar la ruta de clases durante la compilación

Al compilar `PruebaPaqueteTiempo1`, `javac` debe localizar el archivo `.class` para `Tiempo1`, de forma que se asegure que la clase `PruebaPaqueteTiempo1` utilice a la clase `Tiempo1` en forma correcta. El compilador utiliza un objeto especial, llamado **cargador de clases**, para localizar las clases que necesita. El cargador de clases empieza buscando las clases estándar de Java que se incluyen con el JDK. Después busca los **paquetes opcionales**. Java cuenta con un **mecanismo de extensión** que permite agregar paquetes nuevos (opcionales), para fines de desarrollo y ejecución. Si la clase no se encuentra en las clases estándar de Java o en las clases de extensión, el cargador de clases busca en la **ruta de clases**, que contiene una lista de ubicaciones en la que se almacenan las clases. La ruta de clases consiste en una lista de directorios o **archivos de ficheros**, cada uno separado por un **separador de directorio**: un signo de punto y coma (;) en Windows o un signo de dos puntos (:) en UNIX/Linux/Mac OS X. Los archivos de ficheros son archivos individuales que contienen directorios de otros archivos, por lo general en formato comprimido. Por ejemplo, las clases estándar de Java que usted utiliza en sus programas están contenidas en el archivo de ficheros `rt.jar`, el cual se instala junto con el JDK. Los archivos de ficheros generalmente terminan con la extensión `.jar` o `.zip`. Los directorios y archivos de ficheros que se especifican en la ruta de clases contienen las clases que usted desea poner a disponibilidad del compilador y la máquina virtual de Java.

De manera predeterminada, la ruta de clases consiste sólo del directorio actual. Sin embargo, la ruta de clases puede modificarse de la siguiente manera:

1. proporcionando la opción **-classpath** al compilador `javac` o
2. estableciendo la **variable de entorno CLASSPATH** (una variable especial que usted define y el sistema operativo mantiene, de manera que las aplicaciones puedan buscar clases en las ubicaciones especificadas).

Para obtener más información sobre la ruta de clases, visite la página `download.oracle.com/javase/6/docs/technotes/tools/index.html#general`. La sección titulada "General Information" (información general) contiene información acerca de cómo establecer la ruta de clases para UNIX/Linux y Windows.

Error común de programación 8.12

Al especificar una ruta de clases explícita se elimina el directorio actual de la ruta de clases. Esto evita que las clases en el directorio actual (incluyendo los paquetes en ese directorio) se carguen correctamente. Si deben cargarse clases del directorio actual, hay que agregar un punto (.) en la ruta de clases para especificar el directorio actual.

Observación de ingeniería de software 8.11

En general, es una mejor práctica utilizar la opción `-classpath` del compilador, en vez de usar la variable de entorno CLASSPATH para especificar la ruta de clases para un programa. De esta manera, cada aplicación puede tener su propia ruta de clases.

Tip para prevenir errores 8.5

Al especificar la ruta de clases con la variable de entorno CLASSPATH se pueden producir errores sutiles y difíciles de localizar en los programas que utilicen versiones distintas del mismo paquete.

Para las figuras 8.15 y 8.16, no especificamos una ruta de clases explícita. Por lo tanto, para localizar las clases en el paquete `com.deitel.jhtp.cap08` de este ejemplo, el cargador de clases busca en el directorio actual el primer nombre en el paquete: `com`. A continuación, el cargador de clases navega por la estructura de directorios. El directorio `com` contiene al subdirectorio `deitel`; éste contiene al subdirectorio `jhtp`, y éste a su vez contiene al subdirectorio `cap08`. En este subdirectorio se encuentra el archivo `Tiempo1.class`, que se carga mediante el cargador de clases para asegurar que la clase se utilice de manera apropiada en nuestro programa.

Especificar la ruta de clases al ejecutar una aplicación

Al ejecutar una aplicación, la JVM debe poder localizar los archivos `.class` de las clases que se utilizan en esa aplicación. Al igual que el compilador, el comando `java` utiliza un cargador de clases que busca primero en las clases estándar y de extensión, y después en la ruta de clases (el directorio actual, de manera predeterminada). La ruta de clases puede especificarse en forma explícita, utilizando cualquiera de las técnicas descritas para el compilador. Al igual que con el compilador, es mejor especificar una ruta de clases individual para cada programa, mediante las opciones de la línea de comandos de la JVM. Usted puede especificar la ruta de clases en el comando `java` mediante las opciones de línea de comandos **`-classpath`** o **`-cp`**, seguidas de una lista de directorios o archivos de ficheros separados por signos de punto y coma (;) en Microsoft Windows, o signos de dos puntos (:) en UNIX/Linux/Mac OS X. De nuevo, si las clases deben cargarse del directorio actual, asegúrese de incluir un punto (.) en la ruta de clases para especificar el directorio actual.

8.15 Acceso a paquetes

Si no se especifica un modificador de acceso (`public`, `protected` o `private`; hablaremos sobre `protected` en el capítulo 9) para un método o variable al declararse en una clase, se considerará que tiene **acceso a nivel de paquete**. En un programa que consiste de una declaración de clase, esto no tiene un efecto

específico. No obstante, si un programa utiliza varias clases del mismo paquete (es decir, un grupo de clases relacionadas), éstas pueden acceder a los miembros con acceso a nivel de paquete de cada una de las otras clases directamente, a través de referencias a objetos de las clases apropiadas, o en el caso de los miembros `static`, a través del nombre de la clase. Raras veces se utiliza el acceso a nivel de paquete.

La aplicación de la figura 8.17 demuestra el acceso a los paquetes. La aplicación contiene dos clases en un archivo de código fuente: la clase de aplicación `PruebaDatosPaquete` (líneas 5 a 21) y la clase `DatosPaquete` (líneas 24 a 41). Al compilar este programa, el compilador produce dos archivos `.class` separados: `PruebaDatosPaquete.class` y `DatosPaquete.class`. El compilador coloca los dos archivos `.class` en el mismo directorio, por lo que las clases se consideran como parte del mismo paquete. En consecuencia, se permite a la clase `PruebaDatosPaquete` modificar los datos con acceso a nivel de paquete de los objetos `DatosPaquete`. También podemos colocar a `DatosPaquete` (líneas 24 a 41) en un archivo separado de código fuente. Mientras que ambas clases se compilen en el mismo directorio en el disco, la relación de acceso a nivel de paquete seguirá funcionando.

```
1  // Fig. 8.17: PruebaDatosPaquete.java
2  // Los miembros con acceso a nivel de paquete de una clase son accesibles
3  // para las demás clases en el mismo paquete.
4
5  public class PruebaDatosPaquete
6  {
7     public static void main( String[] args )
8     {
9        DatosPaquete datosPaquete = new DatosPaquete();
10
11       // imprime la representación String de datosPaquete
12       System.out.printf( "Despues de instanciar:\n%s\n", datosPaquete );
13
14       // modifica los datos con acceso a nivel de paquete en el objeto datosPaquete
15       datosPaquete.numero = 77;
16       datosPaquete.cadena = "Adios";
17
18       // imprime la representación String de datosPaquete
19       System.out.printf( "\nDespues de modificar valores:\n%s\n", datosPaquete );
20    } // fin de main
21 } // fin de la clase PruebaDatosPaquete
22
23 // clase con variables de instancia con acceso a nivel de paquete
24 class DatosPaquete
25 {
26    int numero; // variable de instancia con acceso a nivel de paquete
27    String cadena; // variable de instancia con acceso a nivel de paquete
28
29    // constructor
30    public DatosPaquete()
31    {
32       numero = 0;
33       cadena = "Hola";
34    } // fin del constructor de DatosPaquete
35
```

Fig. 8.17 | Los miembros con acceso a nivel de paquete de una clase son accesibles para las demás clases en el mismo paquete (parte 1 de 2).

```
36      // devuelve la representación String del objeto DatosPaquete
37      public String toString()
38      {
39         return String.format( "numero: %d; cadena: %s", numero, cadena );
40      } // fin del método toString
41   } // fin de la clase DatosPaquete
```

```
Despues de instanciar:
numero: 0; cadena: Hola

Despues de modificar valores:
numero: 77; cadena: Adios
```

Fig. 8.17 | Los miembros con acceso a nivel de paquete de una clase son accesibles para las demás clases en el mismo paquete (parte 2 de 2).

En la declaración de la clase `DatosPaquete`, las líneas 26 y 27 declaran las variables de instancia `numero` y `cadena` sin modificadores de acceso; por lo tanto, éstas son variables de instancia con acceso a nivel de paquete. El método `main` de la aplicación `PruebaDatosPaquete` crea una instancia de la clase `DatosPaquete` (línea 9) para demostrar la habilidad de modificar las variables de instancia de `DatosPaquete` directamente (como se muestra en las líneas 15 y 16). Los resultados de la modificación se pueden ver en la ventana de resultados.

8.16 (Opcional) Caso de estudio de GUI y gráficos: uso de objetos con gráficos

La mayoría de los gráficos que ha visto hasta este punto no varían cada vez que se ejecuta el programa. El ejercicio 6.2 en la sección 6.13 le pedía que creara un programa para generar figuras y colores al azar. En ese ejercicio, el dibujo cambiaba cada vez que el sistema llamaba a `paintComponent` para volver a dibujar el panel. Para crear un dibujo más consistente que permanezca sin cambios cada vez que se dibuja, debemos almacenar información acerca de las figuras mostradas, para que podamos reproducirlas en forma idéntica, cada vez que el sistema llame a `paintComponent`. Para ello, crearemos un conjunto de clases de figuras que almacenan información acerca de cada figura. Haremos a estas clases "inteligentes", al permitir que sus objetos se dibujen a sí mismos mediante el uso de un objeto `Graphics`.

La clase `MiLinea`

La figura 8.18 declara la clase `MiLinea`, que tiene todas estas capacidades. La clase `MiLinea` importa a `Color` y `Graphics` (líneas 3 y 4). Las líneas 8 a 11 declaran variables de instancia para las coordenadas necesarias para dibujar una línea, y la línea 12 declara la variable de instancia que almacena el color de la línea. El constructor en las líneas 15 a 22 recibe cinco parámetros, uno para cada variable de instancia que inicializa. El método `dibujar` en las líneas 25 a 29 requiere un objeto `Graphics` y lo utiliza para dibujar la línea en el color apropiado y en las coordenadas correctas.

```
1  // Fig. 8.18: MiLinea.java
2  // La clase MiLinea representa a una linea.
3  import java.awt.Color;
4  import java.awt.Graphics;
```

Fig. 8.18 | La clase `MiLinea` representa a una línea (parte 1 de 2).

```
5
6  public class MiLinea
7  {
8     private int x1; // coordenada x del primer punto final
9     private int y1; // coordenada y del primer punto final
10    private int x2; // coordenada x del segundo punto final
11    private int y2; // coordenada y del segundo punto final
12    private Color miColor; // el color de esta figura
13
14    // constructor con valores de entrada
15    public MiLinea( int x1, int y1, int x2, int y2, Color color )
16    {
17       this.x1 = x1; // establece la coordenada x del primer punto final
18       this.y1 = y1; // establece la coordenada y del primer punto final
19       this.x2 = x2; // establece la coordenada x del segundo punto final
20       this.y2 = y2; // establece la coordenada y del segundo punto final
21       miColor = color; // establece el color
22    } // fin del constructor de MiLinea
23
24    // Dibuja la línea en el color específico
25    public void dibujar( Graphics g )
26    {
27       g.setColor( miColor );
28       g.drawLine( x1, y1, x2, y2 );
29    } // fin del método dibujar
30 } // fin de la clase MiLinea
```

Fig. 8.18 | La clase `MiLinea` representa a una línea (parte 2 de 2).

La clase `PanelDibujo`

En la figura 8.19, declaramos la clase `PanelDibujo`, que generará objetos aleatorios de la clase `MiLinea`. La línea 12 declara un arreglo `MiLinea` para almacenar las líneas a dibujar. Dentro del constructor (líneas 15 a 37), la línea 17 establece el color de fondo a `Color.WHITE`. La línea 19 crea el arreglo con una longitud aleatoria entre 5 y 9. El ciclo en las líneas 22 a 36 crea un nuevo objeto `MiLinea` para cada elemento en el arreglo. Las líneas 25 a 28 generan coordenadas aleatorias para los puntos finales de cada línea, y las líneas 31 y 32 generan un color aleatorio para la línea. La línea 35 crea un nuevo objeto `MiLinea` con los valores generados al azar, y lo almacena en el arreglo.

El método `paintComponent` itera a través de los objetos `MiLinea` en el arreglo `lineas` usando una instrucción `for` mejorada (líneas 45 y 46). Cada iteración llama al método dibujar del objeto `MiLinea` actual, y le pasa el objeto `Graphics` para dibujar en el panel.

```
1  // Fig. 8.19: PanelDibujo.java
2  // Programa que utiliza la clase MiLinea
3  // para dibujar líneas al azar.
4  import java.awt.Color;
5  import java.awt.Graphics;
6  import java.util.Random;
7  import javax.swing.JPanel;
8
```

Fig. 8.19 | Creación de objetos `MiLinea` al azar (parte 1 de 2).

```
9  public class PanelDibujo extends JPanel
10 {
11    private Random numerosAleatorios = new Random();
12    private MiLinea[] lineas; // arreglo de lineas
13
14    // constructor, crea un panel con figuras al azar
15    public PanelDibujo()
16    {
17       setBackground( Color.WHITE );
18
19       lineas = new MiLinea[ 5 + numerosAleatorios.nextInt( 5 ) ];
20
21       // crea lineas
22       for ( int cuenta = 0; cuenta < lineas.length; cuenta++ )
23       {
24          // genera coordenadas aleatorias
25          int x1 = numerosAleatorios.nextInt( 300 );
26          int y1 = numerosAleatorios.nextInt( 300 );
27          int x2 = numerosAleatorios.nextInt( 300 );
28          int y2 = numerosAleatorios.nextInt( 300 );
29
30          // genera un color aleatorio
31          Color color = new Color( numerosAleatorios.nextInt( 256 ),
32             numerosAleatorios.nextInt( 256 ), numerosAleatorios.nextInt( 256 ) );
33
34          // agrega la linea a la lista de lineas a mostrar en pantalla
35          lineas[ cuenta ] = new MiLinea( x1, y1, x2, y2, color );
36       } // fin de for
37    } // fin del constructor de PanelDibujo
38
39    // para cada arreglo de figuras, dibuja las figuras individuales
40    public void paintComponent( Graphics g )
41    {
42       super.paintComponent( g );
43
44       // dibuja las lineas
45       for ( MiLinea linea : lineas )
46       linea.dibujar( g );
47    } // fin del método paintComponent
48 } // fin de la clase PanelDibujo
```

Fig. 8.19 | Creación de objetos `MiLinea` al azar (parte 2 de 2).

La clase `PruebaDibujo`

La clase `PruebaDibujo` en la figura 8.20 establece una nueva ventana para mostrar nuestro dibujo. Como estableceremos las coordenadas para las líneas sólo una vez en el constructor, el dibujo no cambia si se hace una llamada a `paintComponent` para actualizar el dibujo en la pantalla.

```
1  // Fig. 8.20: PruebaDibujo.java
2  // Creación de un objeto JFrame para mostrar un PanelDibujo en pantalla.
3  import javax.swing.JFrame;
```

Fig. 8.20 | Creación de un objeto `JFrame` para mostrar `PanelDibujo` (parte 1 de 2).

```
4
5  public class PruebaDibujo
6  {
7     public static void main( String[] args )
8     {
9        PanelDibujo panel = new PanelDibujo();
10       JFrame aplicacion = new JFrame();
11
12       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13       aplicacion.add( panel );
14       aplicacion.setSize( 300, 300 );
15       aplicacion.setVisible( true );
16    } // fin de main
17 } // fin de la clase PruebaDibujo
```

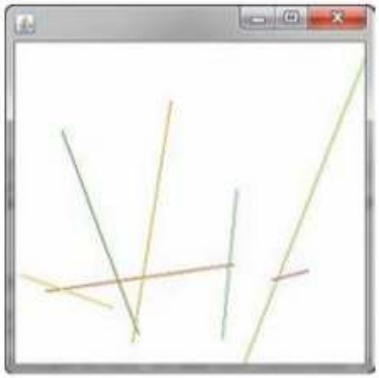


Fig. 8.20 | Creación de un objeto `JFrame` para mostrar `PanelDibujo` (parte 2 de 2).

Ejercicio del caso de estudio de GUI y gráficos

8.1 Extienda el programa de las figuras 8.18 a 8.20 para dibujar rectángulos y óvalos al azar. Cree las clases `MiRectangulo` y `MiOvalo`. Ambas deben incluir las coordenadas *x1*, *y1*, *x2*, *y2*, un color y una bandera `boolean` para determinar si la figura es rellena. Declare un constructor en cada clase con argumentos para inicializar todas las variables de instancia. Para ayudar a dibujar rectángulos y óvalos, cada clase debe proporcionar los métodos `obtenerXSupIzq`, `obtenerYSupIzq`, `obtenerAnchura` y `obtenerAltura`, que calculen la coordenada *x* superior izquierda, la coordenada *y* superior izquierda, la anchura y la altura, respectivamente. La coordenada *x* superior izquierda es el más pequeño de los dos valores de coordenada *x*, la coordenada y superior izquierda es el más pequeño de los dos valores de coordenada *y*, la anchura es el valor absoluto de la diferencia entre los dos valores de coordenada *x*, y la altura es el valor absoluto de la diferencia entre los dos valores de coordenada *y*.

La clase `PanelDibujo`, que extiende a `JPanel` y se encarga de la creación de las figuras, debe declarar tres arreglos, uno para cada tipo de figura. La longitud de cada arreglo debe ser un número aleatorio entre 1 y 5. El constructor de la clase `PanelDibujo` debe llenar cada uno de los arreglos con figuras de posición, tamaño, color y relleno aleatorios.

Además, modifique las tres clases de figuras para incluir lo siguiente:

a) Un constructor sin argumentos que establezca todas las coordenadas de la figura a `0`, el color de la figura a `Color.BLACK` y la propiedad de relleno a `false` (sólo en `MiRectangulo` y `MiOvalo`).

b) Métodos *establecer* para las variables de instancia en cada clase. Los métodos para establecer el valor de una coordenada deben verificar que el argumento sea mayor o igual a cero, antes de establecer la coordenada; si no es así, deben establecer la coordenada a cero. El constructor debe llamar a los métodos *establecer*, en vez de inicializar las variables locales directamente.

c) Métodos *obtener* para las variables de instancia en cada clase. El método `dibujar` debe hacer referencia a las coordenadas mediante los métodos *obtener*, en vez de acceder a ellas de manera directa.

8.17 Conclusión

En este capítulo presentamos conceptos adicionales de las clases. El ejemplo práctico de la clase `Tiempo` presentó una declaración de clase completa que consiste de datos `private`, constructores `public` sobrecargados para flexibilidad en la inicialización, métodos *establecer* y *obtener* para manipular los datos de la clase, y métodos que devuelven representaciones `String` de un objeto `Tiempo` en dos formatos distintos. Aprendió también que toda clase puede declarar un método `toString` que devuelva una representación `String` de un objeto de la clase, y que este método puede invocarse en forma implícita siempre que aparezca en el código un objeto de una clase, en donde se espera un `String`.

Aprendió que la referencia `this` se utiliza en forma implícita en los métodos no `static` de una clase para acceder a las variables de instancia de ésta y a otros métodos no `static`. Vio usos explícitos de la referencia `this` para acceder a los miembros de la clase (incluyendo los campos ocultos) y aprendió a utilizar la palabra clave `this` en un constructor para llamar a otro constructor de la clase.

Hablamos sobre las diferencias entre los constructores predeterminados que proporciona el compilador, y los constructores sin argumentos que proporciona el programador. Aprendió que una clase puede tener referencias a los objetos de otras clases como miembros; un concepto conocido como composición. Vio el tipo de clase `enum` y aprendió a usarlo para crear un conjunto de constantes para usarlas en un programa. Aprendió acerca de la capacidad de recolección de basura de Java y cómo reclama (de manera impredecible) la memoria de los objetos que ya no se utilizan. Explicamos la motivación para los campos `static` en una clase, y le demostramos cómo declarar y utilizar campos y métodos `static` en sus propias clases. También aprendió a declarar e inicializar variables `final`.

Aprendió a empaquetar sus propias clases para reutilizarlas, y cómo importarlas en una aplicación. Por último, aprendió que los campos que se declaran sin un modificador de acceso reciben un acceso a nivel de paquete, de manera predeterminada. Vio la relación entre las clases en el mismo paquete, que permite a cada una ingresar a los miembros de otras clases.

En el siguiente capítulo aprenderá acerca de un aspecto importante de la programación orientada a objetos en Java: la herencia. En ese capítulo verá que todas las clases en Java se relacionan en forma directa o indirecta con la clase llamada `Object`. También empezará a comprender cómo las relaciones entre las clases le permiten crear aplicaciones más poderosas.

Resumen

Sección 8.2 Caso de estudio de la clase `Tiempo`

- Los métodos `public` de una clase se conocen también como los servicios `public` de la clase, o su interfaz `public` (pág. 312).
- Los miembros `private` de una clase no son accesibles para sus clientes.
- El método `static format` de la clase `String` (pág. 314) es similar al método `System.out.printf`, excepto que `format` devuelve un objeto `String` con formato, en vez de mostrarlo en una ventana de comandos.
- Todos los objetos en Java tienen un método `toString`, que devuelve una representación `String` del objeto. El método `toString` se llama en forma implícita cuando aparece un objeto en el código en donde se requiere un `String`.

Sección 8.3 Control del acceso a los miembros

- Los modificadores de acceso `public` y `private` controlan el acceso a las variables y métodos de una clase.

- El principal propósito de los métodos `public` es presentar a los clientes de la clase una vista de los servicios que ésta provee. Los clientes no necesitan preocuparse por la forma en que la clase realiza sus tareas.
- Las variables y los métodos `private` de una clase (es decir, sus detalles de implementación) no son accesibles para sus clientes.

Sección 8.4 Referencias a los miembros del objeto actual mediante `this`

- Un método no `static` de un objeto utiliza en forma implícita la palabra clave `this` (pág. 317) para hacer referencia a las variables de instancia del objeto, y a los demás métodos. La palabra clave `this` también se puede utilizar en forma explícita.
- El compilador produce un archivo separado con la extensión `.class` para cada clase compilada.
- Si una variable local tiene el mismo nombre que el campo de una clase, la variable local oculta el campo. Usted puede usar la referencia `this` en un método para hacer referencia al campo oculto en forma explícita.

Sección 8.5 Caso de estudio de la clase `Tiempo`: constructores sobrecargados

- Los constructores sobrecargados permiten inicializar los objetos de una clase de varias formas distintas. El compilador diferencia a los constructores sobrecargados (pág. 320) con base en sus firmas.
- Para llamar a un constructor de una clase desde otro constructor de la misma clase, puede usar la palabra clave `this` seguida de paréntesis que contengan los argumentos del constructor. Dicha llamada al constructor debe aparecer como la primera instrucción en su cuerpo.

Sección 8.6 Constructores predeterminados y sin argumentos

- Si no se proporcionan constructores en una clase, el compilador crea uno predeterminado.
- Si una clase declara constructores, el compilador no crea un constructor predeterminado. En este caso, usted debe declarar un constructor sin argumentos (pág. 322) si se requiere la inicialización predeterminada.

Sección 8.7 Observaciones acerca de los métodos Establecer *y* Obtener

- Los métodos *establecer* se conocen comúnmente como métodos mutadores (pág. 327), ya que por lo general cambian un valor. Los métodos *obtener* se conocen comúnmente como métodos de acceso (pág. 327) o de consulta. Un método predicado (pág. 327) evalúa si una condición es verdadera o falsa.

Sección 8.8 Composición

- Una clase puede tener referencias a objetos de otras clases como miembros. A dicha capacidad se le conoce como composición (pág. 328), y algunas veces se le denomina relación *tiene un*.

Sección 8.9 Enumeraciones

- Todos los tipos `enum` (pág. 331) son tipos por referencia. Un tipo `enum` se declara con una declaración `enum`, que es una lista separada por comas de constantes `enum`. La declaración puede incluir, de manera opcional, otros componentes de las clases tradicionales, como: constructores, campos y métodos.
- Las constantes `enum` son implícitamente `final`, ya que declaran constantes que no deben modificarse.
- Las constantes `enum` son implícitamente `static`.
- Cualquier intento por crear un objeto de un tipo `enum` con el operador `new` produce un error de compilación.
- Las constantes `enum` se pueden utilizar en las etiquetas case de las instrucciones `switch` y para controlar las instrucciones `for` mejoradas.
- Cada constante `enum` en una declaración `enum` va seguida opcionalmente de argumentos que se pasan al constructor de la `enum`.
- Para cada `enum`, el compilador genera un método `static` llamado `values` (pág. 332), que devuelve un arreglo de las constantes de la `enum`, en el orden en el que se declararon.
- El método `static range` de `EnumSet` (pág. 333) recibe la primera y última constantes `enum` en un rango, y devuelve un objeto `EnumSet` que contiene todas las constantes entre estas dos constantes, ambas inclusive.

Sección 8.10 Recolección de basura y el método `finalize`

- Toda clase en Java tiene los métodos de la clase `Object`, uno de los cuales es `finalize`.
- La máquina virtual de Java (JVM) realiza la recolección automática de basura (pág. 334) para reclamar la memoria que ocupan los objetos que ya no se utilizan. Cuando ya no hay más referencias a un objeto, la JVM lo marca para la recolección de basura. La memoria para dicho objeto se puede reclamar cuando la JVM ejecuta su recolector de basura.
- El método `finalize` (pág. 334) es invocado por el recolector de basura, justo antes de que reclame la memoria del objeto. El método `finalize` no recibe parámetros y tiene el tipo de valor de retorno `void`.
- Tal vez el recolector de basura (pág. 334) nunca se ejecute antes de que un programa termine. Por lo tanto, no queda claro si se hará una llamada al método `finalize` (o cuándo se hará).

Sección 8.11 Miembros de clase `static`

- Una variable `static` (pág. 334) representa la información a nivel de clase que se comparte entre todos los objetos de la clase.
- Las variables `static` tienen alcance en toda la clase. Se puede tener acceso a los miembros `public static` de una clase a través de una referencia a cualquier objeto de la clase, o calificando el nombre del miembro con el nombre de la clase y un punto (.). El código cliente puede acceder a los miembros `private static` de una clase sólo a través de los métodos de la clase.
- Los miembros de clase `static` existen tan pronto como se carga la clase en memoria.
- Un método que se declara como `static` no puede acceder a los miembros de clase que no son `static`, ya que un método `static` puede llamarse incluso aunque no se hayan creado instancias de objetos de la clase.
- La referencia `this` no puede utilizarse en un método `static`.

Sección 8.12 Declaración `static import`

- Una declaración `static` import (pág. 338) permite a los programadores hacer referencia a los miembros `static` importados, sin tener que utilizar el nombre de la clase y un punto (.). Una declaración `static` import individual importa un miembro `static`, y una declaración `static` import sobre demanda importa a todos los miembros `static` de una clase.

Sección 8.13 Variables de instancia `final`

- En el contexto de una aplicación, el principio del menor privilegio (pág. 339) establece que al código se le debe otorgar sólo el nivel de privilegio y de acceso que necesita para realizar su tarea designada.
- La palabra clave `final` especifica que una variable no puede modificarse. Dichas variables deben inicializarse cuando se declaran, o por medio de cada uno de los constructores de una clase.

Sección 8.14 Caso de estudio de la clase `Tiempo`: creación de paquetes

- Cada clase en la API de Java pertenece a un paquete que contiene un grupo de clases relacionadas. Los paquetes ayudan a administrar la complejidad de los componentes de una aplicación, y facilitan la reutilización de software.
- Los paquetes proporcionan una convención para los nombres de clases únicos, que ayuda a evitar los conflictos de nombres de clases (pág. 342).
- Antes de poder importar una clase en varias aplicaciones, ésta debe colocarse en un paquete. Sólo puede haber una declaración `package` (pág. 340) en cada archivo de código fuente de Java, y debe ir antes de todas las demás declaraciones e instrucciones en el archivo.
- Cada nombre de paquete debe empezar con el nombre de dominio de Internet del programador, en orden inverso. Una vez que se invierte el nombre de dominio, podemos elegir cualquier otro nombre que deseemos para nuestro paquete.
- Al compilar una clase en un paquete, la opción `-d` de la línea de comandos de `javac` (pág. 342) especifica en dónde se debe almacenar el paquete, y hace que el compilador cree sus directorios, en caso de que no existan.
- El nombre `package` forma parte del nombre de clase completamente calificado (pág. 342). Esto ayuda a evitar los conflictos de nombres.

- Una declaración import de tipo simple (pág. 344) especifica una clase a importar. Una declaración import tipo sobre demanda (pág. 344) sólo importa las clases que el programa utilice de un paquete específico.
- El compilador utiliza un cargador de clases (pág. 344) para localizar las que necesita en la ruta de clases. La ruta de clases consiste en una lista de directorios o archivos de ficheros, cada uno diferenciado por un separador de directorio (pág. 344).
- La ruta de clases para el compilador y la JVM se puede especificar proporcionando la opción `-classpath` (pág. 345) al comando `javac` o `java`, o estableciendo la variable de entorno `CLASSPATH`. Si las clases deben cargarse del directorio actual, incluya un punto (`.`) en la ruta de clases.

Sección 8.15 Acceso a paquetes

- Si no se especifica un modificador de acceso para un método o variable al momento de su declaración en una clase, se considera que el método o variable tiene acceso a nivel de paquete (pág. 345).

Ejercicio de autoevaluación

8.1 Complete los siguientes enunciados:

a) Al compilar una clase en un paquete, la opción _____ de línea de comandos de `javac` especifica en dónde se debe almacenar el paquete, y hace que el compilador cree los directorios, en caso de que no existan.
b) El método `static` _____ de la clase `String` es similar al método `System.out.printf`, pero devuelve un objeto `String` con formato en vez de mostrar un objeto `String` en una ventana de comandos.
c) Si un método contiene una variable local con el mismo nombre que uno de los campos de su clase, la variable local _____ al campo en el alcance de ese método.
d) El recolector de basura llama al método _____ justo antes de reclamar la memoria de un objeto.
e) Una declaración _____ especifica una clase a importar.
f) Si una clase declara constructores, el compilador no creará un(a) _____ .
g) El método _____ de un objeto se llama en forma implícita cuando aparece un objeto en el código, en donde se necesita un `String`.
h) A los métodos establecer se les llama comúnmente _____ o _____ .
i) Un método _____ evalúa si una condición es verdadera o falsa.
j) Para cada `enum`, el compilador genera un método `static` llamado _____ , que devuelve un arreglo de las constantes de la `enum` en el orden en el que se declararon.
k) A la composición se le conoce algunas veces como relación _____ .
l) Una declaración _____ contiene una lista separada por comas de constantes.
m) Una variable _____ representa información a nivel de clase, que comparten todos los objetos de la clase.
n) Una declaración _____ importa un miembro `static`.
o) El _____ establece que al código se le debe otorgar sólo el nivel de privilegio y de acceso que necesita para realizar su tarea designada.
p) La palabra clave _____ especifica que una variable no se puede modificar.
q) Sólo puede haber un(a) _____ en un archivo de código fuente de Java, y debe ir antes de todas las demás declaraciones e instrucciones en el archivo.
r) Un(a) declaración _____ sólo importa las clases que utiliza el programa de un paquete específico.
s) El compilador utiliza un(a) _____ para localizar las clases que necesita en la ruta de clases.
t) La ruta de clases para el compilador y la JVM se puede especificar mediante la opción _____ para el comando `javac` o `java`, o estableciendo la variable de entorno _____ .
u) A los métodos *establecer* se les conoce comúnmente como _____ , ya que por lo general modifican un valor.
v) Un(a) _____ importa a todos los miembros `static` de una clase.
w) Los métodos `public` de una clase se conocen también como los _____ o _____ de la clase.

Respuesta al ejercicio de autoevaluación

8.1 a) -d. b) `format`. c) oculta. d) `finalize`. e) import de tipo simple. f) constructor predeterminado. g) `toString`. h) métodos de acceso, métodos de consulta. i) predicado. j) `values`. k) tiene un. l) `enum`. m) `static`. n) `static` import de tipo simple. o) principio de menor privilegio. p) `final`. q) declaración `package`. r) import tipo sobre demanda. s) cargador de clases. t) `-classpath`, `CLASSPATH`. u) métodos mutadores. v) declaración `static` import sobre demanda. w) servicios `public`, interfaz `public`.

Ejercicios

8.2 Explique la noción del acceso a nivel de paquete en Java. Explique los aspectos negativos del acceso a nivel de paquete.

8.3 ¿Qué ocurre cuando un tipo de valor de retorno, incluso `void`, se especifica para un constructor?

8.4 ***(Clase Rectangulo)*** Cree una clase llamada `Rectangulo` con los atributos longitud y anchura, cada uno con un valor predeterminado de 1. Debe tener métodos para calcular el perímetro y el área del rectángulo. Debe tener métodos *establecer* y *obtener* para `longitud` y `anchura`. Los métodos *establecer* deben verificar que `longitud` y `anchura` sean números de punto flotante mayores de 0.0, y menores de 20.0. Escriba un programa para probar la clase `Rectangulo`.

8.5 ***(Modificación de la representación de datos interna de una clase)*** Sería perfectamente razonable para la clase `Tiempo2` de la figura 8.5 representar la hora internamente como el número de segundos transcurridos desde medianoche, en vez de usar los tres valores enteros `hora`, `minuto` y `segundo`. Los clientes podrían usar los mismos métodos `public` y obtener los mismos resultados. Modifique la clase `Tiempo2` de la figura 8.5 para implementar un objeto `Tiempo2` como el número de segundos transcurridos desde medianoche, y mostrar que no hay cambios visibles para los clientes de la clase.

8.6 ***(Clase cuenta de ahorros)*** Cree una clase llamada `CuentaDeAhorros`. Use una variable `static` llamada `tasaInteresAnual` para almacenar la tasa de interés anual para todos los cuentahabientes. Cada objeto de la clase debe contener una variable de instancia `private` llamada `saldoAhorros`, que indique la cantidad que el ahorrador tiene actualmente en depósito. Proporcione el método `calcularInteresMensual` para calcular el interés mensual, multiplicando el `saldoAhorros` por la `tasaInteresAnual` dividida entre 12; este interés debe sumarse al `saldoAhorros`. Proporcione un método `static` llamado `modificarTasaInteres` para establecer la `tasaInteresAnual` en un nuevo valor. Escriba un programa para probar la clase `CuentaDeAhorros`. Cree dos instancias de objetos `CuentaDeAhorros`, `ahorrador1` y `ahorrador2`, con saldos de $2000.00 y $3000.00. Establezca la `tasaInteresAnual` en 4%, después calcule el interés mensual para cada uno de los 12 meses e imprima los nuevos saldos para ambos ahorradores. Luego establezca la `tasaInteresAnual` en 5%, calcule el interés del siguiente mes e imprima los nuevos saldos para ambos ahorradores.

8.7 ***(Mejora a la clase Tiempo2)*** Modifique la clase `Tiempo2` de la figura 8.5 para incluir un método `tictac`, que aumente el tiempo almacenado en un objeto `Tiempo2` por un segundo. Proporcione el método `incrementarMinuto` para incrementar en uno el minuto, y el método `incrementarHora` para adelantar en uno la hora. Escriba un programa para probar los métodos `tictac`, `incrementarMinuto` e `incrementarHora`, para asegurarse que funcionen de manera correcta. Asegúrese de probar los siguientes casos:

a) incrementar el minuto, de manera que cambie al siguiente minuto,
b) aumentar la hora, de manera que cambie a la siguiente hora, e
c) adelantar el tiempo de manera que cambie al siguiente día (por ejemplo, de 11:59:59 PM a 12:00:00 AM).

8.8 ***(Mejora a la clase Fecha)*** Modifique la clase `Fecha` de la figura 8.7 para realizar la comprobación de errores en los valores inicializadores para las variables de instancia `mes`, `dia` y `anio` (la versión actual sólo valida el mes y el día). Proporcione un método llamado `siguienteDia` para adelantar el `dia` en uno. Escriba un programa que evalúe el método `siguienteDia` en un ciclo que imprima la fecha durante cada iteración del ciclo, para mostrar que el método funciona de forma apropiada. Pruebe los siguientes casos:

a) Incrementar la fecha de manera que cambie al siguiente mes, y
b) Adelantar fecha de manera que cambie al siguiente año.

8.9 Vuelva a escribir el código de la figura 8.14, de manera que utilice una declaración `import` separada para cada miembro `static` de la clase `Math` que se utilice en el ejemplo.

8.10 Escriba un tipo `enum` llamado `LuzSemaforo`, cuyas constantes (`ROJO`, `VERDE`, `AMARILLO`) reciban un parámetro: la duración de la luz. Escriba un programa para probar la `enum` `LuzSemaforo`, de manera que muestre las constantes de la enum y sus duraciones.

8.11 *(Números complejos)* Cree una clase llamada `Complejo` para realizar operaciones aritméticas con números complejos. Estos números tienen la forma

parteReal + *parteImaginaria* * *i*

en donde *i* es

$\sqrt{-1}$

Escriba un programa para probar su clase. Use variables de punto flotante para representar los datos `private` de la clase. Proporcione un constructor que permita que un objeto de esta clase se inicialice al declararse. Proporcione un constructor sin argumentos con valores predeterminados, en caso de que no se proporcionen inicializadores. Ofrezca métodos `public` que realicen las siguientes operaciones:

a) Sumar dos números `Complejo`: las partes reales se suman entre sí y las partes imaginarias también.
b) Restar dos números `Complejo`: la parte real del operando derecho se resta de la parte real del operando izquierdo, y la parte imaginaria del operando derecho se resta de la parte imaginaria del operando izquierdo.
c) Imprimir números `Complejo` en la forma (*parteReal*, *parteImaginaria*).

8.12 *(Clase Fecha y Tiempo)* Cree una clase llamada `FechaYTiempo`, que combine la clase `Tiempo2` modificada del ejercicio 8.7 y la clase `Fecha` alterada del ejercicio 8.8. Cambie el método `incrementarHora` para llamar al método `siguienteDia` si el tiempo se incrementa hasta el siguiente día. Modifique los métodos `toString` y `aStringUniversal` para imprimir la fecha, junto con la hora. Escriba un programa para evaluar la nueva clase `FechaYTiempo`. En específico, pruebe incrementar la hora para que cambie al siguiente día.

8.13 *(Conjunto de enteros)* Cree la clase `ConjuntoEnteros`. Cada objeto `ConjuntoEnteros` puede almacenar enteros en el rango de 0 a 100. El conjunto se representa mediante un arreglo de valores `boolean`. El elemento del arreglo `a[i]` es `true` si el entero *i* se encuentra en el conjunto. El elemento del arreglo `a[j]` es `false` si el entero *j* no se encuentra dentro del conjunto. El constructor sin argumentos inicializa el arreglo de Java con el "conjunto vacío" (es decir, sólo valores `false`).

Proporcione los siguientes métodos: el método `static union` debe crear un tercer conjunto que sea la unión teórica de conjuntos para los dos conjuntos existentes (es decir, un elemento del nuevo arreglo se establece en `true` si ese elemento es `true` en cualquiera o en ambos de los conjuntos existentes; en caso contrario, el elemento del nuevo conjunto se establece en `false`). El método `static interseccion` debe crear un tercer conjunto que sea la intersección teórica de conjuntos para los dos conjuntos existentes (es decir, un elemento del arreglo del nuevo conjunto se establece en `false` si ese elemento es `false` en uno o ambos de los conjuntos existentes; en caso contrario, el elemento del nuevo conjunto se establece en `true`). El método `insertarElemento` debe insertar un nuevo entero *k* en un conjunto (estableciendo `a[k]` en `true`). El método `eliminarElemento` debe eliminar el entero *m* (estableciendo `a[m]` en `false`). El método `toString` debe devolver un `String` que contenga un conjunto como una lista de números separados por espacios. Incluya sólo aquellos elementos que estén presentes en el conjunto. Use `---` para representar un conjunto vacío. El método `esIgualA` debe determinar si dos conjuntos son iguales. Escriba un programa para probar la clase `ConjuntoEnteros`. Cree instancias de varios objetos `ConjuntoEnteros`. Pruebe que todos sus métodos funcionen de manera correcta.

8.14 *(Clase Fecha)* Cree la clase `Fecha` con las siguientes capacidades:

a) Imprimir la fecha en varios formatos, como

```
MM/DD/AAAA
Junio 14, 1992
DDD AAAA
```

b) Usar constructores sobrecargados para crear objetos `Fecha` inicializados con fechas de los formatos en la parte (a). En el primer caso, el constructor debe recibir tres valores enteros. En el segundo, debe recibir un objeto `String` y dos valores enteros. En el tercero debe recibir dos valores enteros, el primero de los

cuales representa el número de día en el año. [*Sugerencia:* para convertir la representación `String` del mes a un valor numérico, compare los objetos `String` usando el método `equals`. Por ejemplo, si `s1` y `s2` son cadenas, la llamada al método `s1.equals( s2 )` devuelve `true` si las cadenas son idénticas y devuelve `false` en cualquier otro caso].

8.15 *(Números racionales)* Cree una clase llamada `Racional` para realizar operaciones aritméticas con fracciones. Escriba un programa para probar su clase. Use variables enteras para representar las variables de instancia `private` de la clase: el `numerador` y el `denominador`. Proporcione un constructor que permita inicializarse a un objeto de esta clase al ser declarado. El constructor debe almacenar la fracción en forma reducida. La fracción

```
2/4
```

es equivalente a `1/2` y debe guardarse en el objeto como 1 en el `numerador` y 2 en el `denominador`. Proporcione un constructor sin argumentos con valores predeterminados, en caso de que no se proporcionen inicializadores. Proporcione métodos `public` que realicen cada una de las siguientes operaciones:

a) Sumar dos números `Racional`: el resultado de la suma debe almacenarse en forma reducida. Implemente esto como un método `static`.
b) Restar dos números `Racional`: el resultado de la resta debe almacenarse en forma reducida. Implemente esto como un método `static`.
c) Multiplicar dos números `Racional`: el resultado de la multiplicación debe almacenarse en forma reducida. Implemente esto como un método `static`.
d) Dividir dos números `Racional`: el resultado de la división debe almacenarse en forma reducida. Implemente esto como un método `static`.
e) Devolver una representación `String` de un número `Racional` en la forma `a/b`, en donde `a` es el `numerador` y `b` es el `denominador`.
f) Devolver una representación `String` de un número `Racional` en formato de punto flotante. (Considere proporcionar capacidades de formato, que permitan al usuario de la clase especificar el número de dígitos de precisión a la derecha del punto decimal).

8.16 *(Clase Entero Enorme)* Cree una clase llamada `EnteroEnorme` que utilice un arreglo de 40 elementos de dígitos, para guardar enteros de hasta 40 dígitos de longitud cada uno. Proporcione los métodos parse, `toString`, `sumar` y `restar`. El método `parse` debe recibir un `String`, extraer cada dígito mediante el método `charAt` y colocar el equivalente entero de cada dígito en el arreglo de enteros. Para comparar objetos `EnteroEnorme`, proporcione los siguientes métodos: `esIgualA`, `noEsIgualA`, `esMayorQue`, `esMenorQue`, `esMayorOIgualA`, y `esMenorOIgualA`. Cada uno de estos métodos deberá ser un método predicado que devuelva `true` si la relación se aplica entre los dos objetos `EnteroEnorme`, y `false` si no se aplica. Proporcione un método predicado llamado `esCero`. Si desea hacer algo más, proporcione también los métodos `multiplicar`, `dividir` y `residuo`. [*Nota:* los valores `boolean` primitivos pueden imprimirse como la palabra "true" o la palabra "false", con el especificador de formato `%b`].

8.17 *(Tres en raya)* Cree una clase llamada `TresEnRaya` que le permita escribir un programa para jugar al "tres en raya" (o "gato"). La clase debe contener un arreglo privado bidimensional de 3 por 3. Use una enumeración para representar el valor en cada celda del arreglo. Las constantes de la enumeración se deben llamar `X`, `O` y `VACIO` (para una posición que no contenga una `X` o un `O`). El constructor debe inicializar los elementos del tablero con `VACIO`. Permita dos jugadores humanos. Siempre que el primer jugador realice un movimiento, coloque una `X` en el cuadro especificado y coloque un `O` siempre que el segundo jugador realice un movimiento, el cual debe hacerse en un cuadro vacío. Luego en cada movimiento, determine si alguien ganó el juego o si hay un empate. Si desea hacer algo más, modifique su programa de manera que la compu-tadora realice los movimientos para uno de los jugadores. Además, permita que el jugador especifique si desea el primer o segundo turno. Si se siente todavía más motivado, desarrolle un programa que reproduzca un juego de "tres en raya" tridimensional, en un tablero de 4 por 4 por 4 [*Nota:* ¡Éste es un proyecto extremadamente retador!].

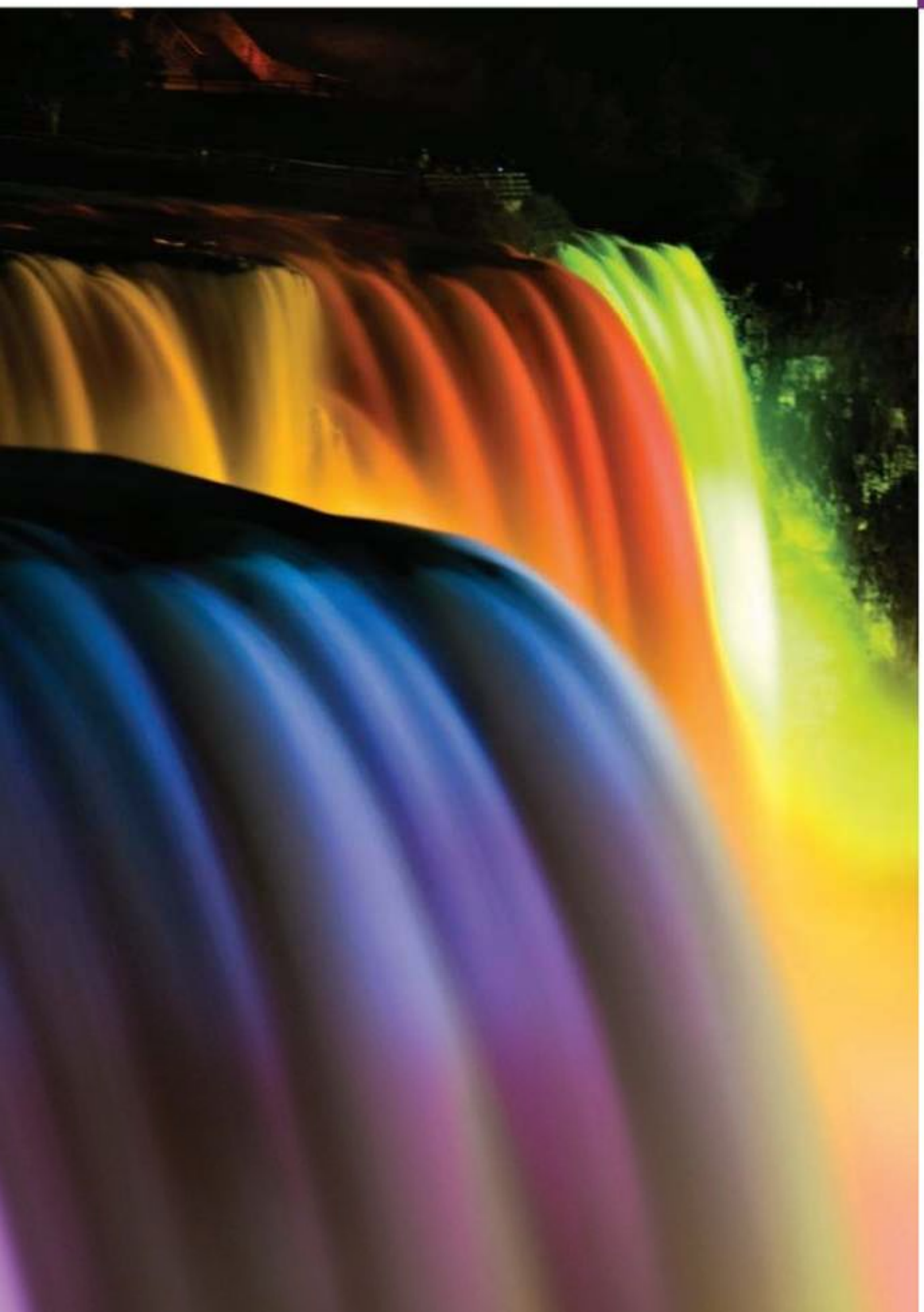
Marcar la diferencia

8.18 ***(Proyecto: Clase de respuesta de emergencia)*** El servicio de respuesta de emergencia estadounidense, *9-1-1*, conecta a los que llaman a un Punto de respuesta del servicio público (PSAP) *local*. Por tradición, el PSAP pide al que llama cierta información de identificación, incluyendo su dirección, número telefónico y la naturaleza de la emergencia. Después despacha los respondedores de emergencia apropiados (como la policía, una ambulancia o el departamento de bomberos). El *9-1-1 mejorado* (o *E9-1-1*) usa computadoras y bases de datos para determinar el domicilio del que llama, dirige la llamada al PSAP más cercano y muestra tanto el número de teléfono como la dirección del que llama a la persona que toma la llamada. El *9-1-1 mejorado inalámbrico* proporciona, a los encargados de tomar las llamadas, la información de identificación para las llamadas inalámbricas. Se desplegó en dos fases; durante la primera fase, las empresas de telecomunicaciones tuvieron que proporcionar el número telefónico inalámbrico y la ubicación del sitio de la estación base que transmitía la llamada. En la segunda fase, las empresas de telecomunicaciones tuvieron que proveer la ubicación del que hacía la llamada (mediante el uso de tecnologías como GPS). Para aprender más sobre el 9-1-1, visite `www.fcc.gov/pshs/services/911-services/Welcome.html` y `people.howstuffworks.com/9-1-1.htm`.

Una parte importante de crear una clase es determinar sus atributos (variables de instancia). Para este ejercicio de diseño de clases, investigue los servicios 9-1-1 en Internet. Después, diseñe una clase llamada `Emergencia` que podría utilizarse en un sistema de respuesta de emergencia 9-1-1 orientado a objetos. Liste los atributos que podría usar un objeto de esta clase para representar la emergencia. Por ejemplo, la clase podría contener información sobre quién reportó la emergencia (entre ésta su número telefónico), la ubicación de la emergencia, la hora del reporte, la naturaleza de la emergencia, el tipo de respuesta y el estado de la misma. Los atributos de la clase deben describir por completo la naturaleza del problema y lo que está ocurriendo para resolver ese problema.

Programación orientada a objetos: herencia

9



No digas que conoces a alguien por completo, hasta que tengas que dividir una herencia con él.
—Johann Kasper Lavater

Este método es para definirse como el número de la clase de todas las clases similares a la dada.
—Bertrand Rusell

Objetivos

En este capítulo aprenderá a:

- Comprender cómo la herencia fomenta la reutilización de software.
- Entender las nociones de las superclases y las subclases, y la relación entre ellas.
- Utilizar la palabra clave `extends` para crear una clase que herede los atributos y comportamientos de otra clase.
- Comprender el uso del modificador de acceso `protected` para dar a los métodos de la subclase acceso a los miembros de la superclase.
- Utilizar los miembros de superclases mediante `super`.
- Comprender cómo se utilizan los constructores en las jerarquías de herencia.
- Conocer los métodos de la clase `Object`, la superclase directa o indirecta de todas las clases en Java.

Plan general

9.1 Introducción

En este capítulo continuamos nuestra discusión acerca de la programación orientada a objetos (POO), introduciendo una de sus características principales: la **herencia**, que es una forma de reutilización de software en la que se crea una nueva clase al absorber los miembros de una existente, y se mejoran con nuevas capacidades, o con modificaciones en las capacidades ya existentes. Con la herencia, los programadores ahorran tiempo durante el desarrollo, al basar las nuevas clases en el software existente, probado y depurado, de alta calidad. Esto también aumenta la probabilidad de que un sistema se implemente y mantenga con efectividad.

Al crear una clase, en vez de declarar miembros completamente nuevos, el programador puede designar que la nueva clase herede los miembros de una existente. La cual se conoce como **superclase**, y la clase nueva como **subclase**. (El lenguaje de programación C++ se refiere a la superclase como la **clase base**, y a la subclase como **clase derivada**.) Cada subclase puede convertirse en la superclase de futuras subclases.

Una subclase puede agregar sus propios campos y métodos. Por lo tanto, una subclase es *más específica* que su superclase y representa a un grupo más especializado de objetos. La subclase exhibe los comportamientos de su superclase y puede modificarlos, de modo que operen en forma apropiada para la subclase. Es por ello que a la herencia se le conoce algunas veces como **especialización**.

La **superclase directa** es la superclase a partir de la cual la subclase hereda en forma explícita. Una **superclase indirecta** es cualquier clase arriba de la superclase directa en la **jerarquía de clases**, que define las relaciones de herencia entre las clases. En Java, la jerarquía de clases empieza con la clase `Object` (en el paquete `java.lang`), a partir de la cual se extienden (o "heredan") *todas* las clases en Java, ya sea en forma directa o indirecta. La sección 9.7 lista los métodos de la clase `Object`, de la cual heredan todas las demás clases. Java, sólo soporta la **herencia simple**, en donde cada clase se deriva sólo de *una* superclase directa. A diferencia de C++, Java *no* soporta la herencia múltiple (que ocurre cuando una clase se deriva de más de una superclase directa). En el capítulo 10, Programación orientada a objetos: polimorfismo, explicaremos cómo usar las interfaces de Java para obtener muchos de los beneficios de la herencia múltiple, lo que evita al mismo tiempo los problemas asociados.

Es necesario hacer una diferencia entre la **relación *es un*** y la **relación *tiene un***. La relación *es un* representa a la herencia. En este tipo de relación, *un objeto de una subclase puede tratarse también como un objeto de su superclase*. Por ejemplo, un auto *es un* vehículo. En contraste, la relación *tiene un* representa la composición (vea el capítulo 8). En este tipo de relación, *un objeto contiene referencias a objetos como miembros*. Por ejemplo, un auto *tiene un* volante de dirección (y un objeto auto tiene una referencia a un objeto volante de dirección).

Las clases nuevas pueden heredar de las clases en las **bibliotecas de clases**. Las organizaciones desarrollan sus propias bibliotecas de clases y pueden aprovechar las que ya están disponibles en todo el mundo. Es probable que algún día, la mayoría de software nuevo se construya a partir de **componentes reutilizables estandarizados**, como sucede actualmente con la mayoría de los automóviles y del hardware de computadora. Esto facilitará el desarrollo de software más poderoso, abundante y económico.

9.2 Superclases y subclases

A menudo, un objeto de una clase *es un* objeto de otra clase también. La figura 9.1 lista varios ejemplos simples de superclases y subclases; las superclases tienden a ser "más generales", y las subclases "más específicas". Por ejemplo, un `PrestamoAuto` *es un* `Prestamo`, así como `PrestamoMejoraCasa` y `PrestamoHipotecario`. Por ende, en Java se puede decir que la clase `PrestamoAuto` hereda de la clase `Prestamo`. En este contexto, dicha clase es una superclase y la clase `PrestamoAuto` es una subclase. Un `PrestamoAuto` *es un* tipo específico de `Prestamo`, pero es incorrecto afirmar que todo `Prestamo` *es un* `PrestamoAuto`; el `Prestamo` podría ser cualquier tipo de crédito.

Superclase	Subclases
`Estudiante`	`EstudianteGraduado, EstudianteNoGraduado`
`Figura`	`Circulo, Triangulo, Rectangulo, Esfera, Cubo`
`Prestamo`	`PrestamoAutomovil, PrestamoMejoraCasa, PrestamoHipotecario`
`Empleado`	`Docente, Administrativo`
`CuentaBancaria`	`CuentaDeCheques, CuentaDeAhorros`

Fig. 9.1 | Ejemplos de herencia.

Como todo objeto de una subclase *es un* objeto de su superclase, y como una superclase puede tener muchas subclases, el conjunto de objetos representados por una superclase es a menudo más grande que el de objetos representado por cualquiera de sus subclases. Por ejemplo, la superclase `Vehiculo` representa a todos los vehículos, entre ellos autos, camiones, barcos, bicicletas, etcétera. En contraste, la subclase `Auto` representa a un subconjunto más pequeño y específico de los vehículos.

Jerarquía de miembros de una comunidad universitaria

Las relaciones de herencia forman estructuras jerárquicas en forma de árbol. Una superclase existe en una relación jerárquica con sus subclases. Vamos a desarrollar una jerarquía de clases de ejemplo (figura 9.2), también conocida como **jerarquía de herencia**. Una comunidad universitaria tiene miles de miembros, compuestos por empleados, estudiantes y exalumnos. Los empleados pueden ser miembros del cuerpo docente o administrativo. Los miembros del cuerpo docente pueden ser administradores (como decanos y jefes de departamento) o maestros. La jerarquía podría contener muchas otras clases. Por ejemplo, los estudiantes pueden ser graduados o no graduados. Los no graduados pueden ser de primero, segundo, tercero o cuarto año.

Fig. 9.2 | Jerarquía de herencia para objetos `MiembroDeLaComunidad` universitaria.

Cada flecha en la jerarquía representa una relación *es un*. Por ejemplo, al seguir las flechas en esta jerarquía de clases podemos decir "un `Empleado` *es un* `MiembroDeLaComunidad`" y "un `Maestro` *es un* `miembro Docente`". `MiembroDeLaComunidad` es la superclase directa de `Empleado`, `Estudiante` y `Exalumno`, y es una superclase indirecta de todas las demás clases en el diagrama. Si comienza desde la parte inferior de él, podrá seguir las flechas y aplicar la relación *es un* hasta la superclase superior. Por ejemplo, un `Administrador` *es un* miembro `Docente`, *es un* `Empleado`, *es un* `MiembroDeLaComunidad` y, por supuesto, *es un* `Object`.

La jerarquía de Figura

Ahora considere la jerarquía de herencia de `Figura` en la figura 9.3. Esta jerarquía empieza con la superclase `Figura`, la cual se extiende mediante las subclases `FiguraBidimensional` y `FiguraTridimensional`; las `Figuras` son del tipo `FiguraBidimensional` o `FiguraTridimensional`. El tercer nivel de esta jerarquía contiene algunos tipos más específicos de figuras tipo `FiguraBidimensional` y `FiguraTridimensional`. Al igual que en la figura 9.2, podemos seguir las flechas desde la parte inferior del diagrama, hasta la superclase de más arriba en esta jerarquía de clases, para identificar varias relaciones *es un*. Por ejemplo, un `Triangulo` *es un* objeto `FiguraBidimensional` y *es una* `Figura`, mientras que una `Esfera` *es una* `FiguraTridimensional` y *es una* `Figura`. Esta jerarquía podría contener muchas otras clases. Por ejemplo, las elipses y los trapezoides son del tipo `FiguraBidimensional`.

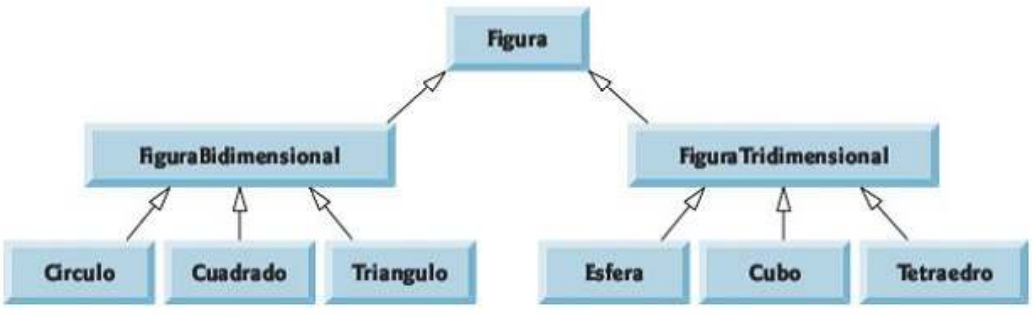


Fig. 9.3 | Jerarquía de herencia para `Figuras`.

No todas las relaciones de clases son una relación de herencia. En el capítulo 8 hablamos sobre la relación *tiene un*, en la que las clases tienen miembros que hacen referencia a los objetos de otras clases. Tales relaciones crean clases mediante la composición de clases existentes. Por ejemplo, dadas las clases `Empleado`, `FechaDeNacimiento` y `NumeroTelefonico`, no es apropiado decir que un `Empleado` *es una* `FechaDeNacimiento` o que un `Empleado` *es un* `NumeroTelefonico`. Sin embargo, un `Empleado` *tiene una* `FechaDeNacimiento` y también *tiene un* `NumeroTelefonico`.

Es posible tratar a los objetos de superclases y a los de subclases de manera similar; sus similitudes se expresan en los miembros de la superclase. Los objetos de todas las clases que extienden a una superclase común pueden tratarse como objetos de esa superclase; dichos objetos tienen una relación *es un* con la superclase. Más adelante en este capítulo y en el 10, consideraremos muchos ejemplos que aprovechan la relación *es un*.

Una subclase puede personalizar los métodos que hereda de su superclase. Para hacer esto, la subclase **sobrescribe** (redefine) el método de la superclase con una implementación apropiada, como veremos a menudo en los ejemplos de código de este capítulo.

9.3 Miembros protected

En el capítulo 8 hablamos sobre los modificadores de acceso `public` y `private`. Los miembros `public` de una clase son accesibles en cualquier parte en donde el programa tenga una referencia a un objeto de esa clase, o una de sus subclases. Los miembros `private` de una clase son accesibles sólo dentro de la misma clase. En esta sección presentaremos el modificador de acceso **protected**. El uso del acceso `protected` ofrece un nivel intermedio de acceso entre `public` y `private`. Los miembros `protected` de una superclase pueden ser utilizados por los miembros de esa superclase, por los de sus subclases y por los de otras clases en el mismo paquete; los miembros `protected` también tienen acceso a nivel de paquete.

Todos los miembros `public` y `protected` de una superclase retienen su modificador de acceso original cuando se convierten en miembros de la subclase (por ejemplo, los miembros `public` de la superclase se convierten en miembros `public` de la subclase, y los miembros `protected` de la superclase se vuelven en miembros `protected` de la subclase). Los miembros `private` de una superclase no pueden utilizarse fuera de la propia clase. En cambio, están *ocultos* en sus subclases y se pueden utilizar sólo a través de los métodos `public` o `protected` heredados de la superclase.

Los métodos de una subclase pueden referirse a los miembros `public` y `protected` que se hereden de la superclase con sólo utilizar los nombres de los miembros. Cuando un método de la subclase sobrescribe al método heredado de la *superclase*, éste último puede utilizarse desde la *subclase* si se antepone a su nombre la palabra clave **super** y un punto (.) separador. En la sección 9.4 hablaremos sobre el acceso a los miembros sobrescritos de la superclase.

Observación de ingeniería de software 9.1

Los métodos de una subclase no pueden tener acceso directo a los miembros `private` de su superclase. Una subclase puede modificar el estado de las variables de instancia `private` de la superclase sólo a través de los métodos que no sean `private`, que se proporcionan en la superclase y son heredados por la subclase.

Observación de ingeniería de software 9.2

Declarar variables de instancia `private` ayuda a los programadores a probar, depurar y modificar correctamente los sistemas. Si una subclase puede acceder a las variables de instancia `private` de su superclase, las clases que hereden de esa subclase podrían acceder a las variables de instancia también. Esto propagaría el acceso a las que deberían ser variables de instancia `private`, y se perderían los beneficios del ocultamiento de la información.

9.4 Relación entre las superclases y las subclases

Ahora vamos a usar una jerarquía de herencia que contiene tipos de empleados en la aplicación de nómina de una compañía, para hablar sobre la relación entre una superclase y su subclase. En esta compañía, a los empleados por comisión (que se representarán como objetos de una superclase) se les paga un porcentaje de sus ventas, en tanto que los empleados por comisión con salario base (que se representarán como objetos de una subclase) reciben un salario base, *más* un porcentaje de sus ventas.

Dividiremos nuestra discusión sobre la relación entre estas clases en cinco ejemplos. El primero declara la clase `EmpleadoPorComision`, la cual hereda directamente de la clase `Object` y declara como variables de instancia `private` el primer nombre, el apellido paterno, el número de seguro social, la tarifa de comisión y el monto de ventas en bruto (es decir, total).

El segundo ejemplo declara la clase `EmpleadoBaseMasComision`, la cual también hereda de manera directa de la clase `Object` y declara como variables de instancia `private` el primer nombre, el apellido paterno, el número de seguro social, la tarifa de comisión, el monto de ventas en bruto *y* el salario base. Para crear esta última clase, *escribiremos cada línea de código* que ésta requiera; pronto veremos que es mucho más eficiente crear esta clase haciendo que herede de la clase `EmpleadoPorComision`.

El tercer ejemplo declara una nueva clase `EmpleadoBaseMasComision`, la cual *extiende* a la clase `EmpleadoPorComision` (es decir, un `EmpleadoBaseMasComision` *es un* `EmpleadoPorComision` que también tiene un salario base). Esta *reutilización de software nos permite escribir menos código* al desarrollar la nueva subclase. En este ejemplo, `EmpleadoBaseMasComision` trata de acceder a los miembros `private` de la clase `EmpleadoPorComision`; esto produce errores de compilación, ya que la subclase no puede acceder a las variables de instancia `private` de la superclase.

El cuarto ejemplo muestra que si las variables de instancia de `EmpleadoPorComision` se declaran como `protected`, la subclase `EmpleadoBaseMasComision` puede acceder a esos datos de manera directa. Ambas clases `EmpleadoBaseMasComision` contienen una funcionalidad idéntica, pero le mostraremos que la versión heredada es más fácil de crear y de manipular.

Una vez que hablemos sobre la conveniencia de utilizar variables de instancia `protected`, crearemos el quinto ejemplo, el cual establece las variables de instancia de `EmpleadoPorComision` de vuelta a `private` para hacer cumplir la buena ingeniería de software. Después le mostraremos cómo es que la subclase `EmpleadoBaseMasComision` puede utilizar los métodos `public` de `EmpleadoPorComision` para manipular (de una forma controlada) las variables de instancia `private` heredadas de `EmpleadoPorComision`.

9.4.1 Creación y uso de una clase `EmpleadoPorComision`

Comenzaremos por declarar la clase `EmpleadoPorComision` (figura 9.4). La línea 4 empieza la declaración de la clase, e indica que la clase `EmpleadoPorComision` **`extiende`** (extends) (es decir, hereda de) la clase **`Object`** (del paquete `java.lang`). Esto hace que la clase `EmpleadoPorComision` herede los métodos de la clase `Object`; la clase `Object` no tiene campos. Si no especificamos de manera explícita a qué clase extiende la nueva clase, ésta hereda de `Object` en forma implícita. Por esta razón, es común que los programadores no incluyan "`extends Object`" en su código; en nuestro ejemplo lo haremos sólo por fines demostrativos.

Generalidades sobre los métodos y variables de instancia de la clase `EmpleadoPorComision`

Los servicios `public` de la clase `EmpleadoPorComision` incluyen un constructor (líneas 13 a 22), y los métodos `ingresos` (líneas 93 a 96) y `toString` (líneas 99 a 107). Las líneas 25 a 90 declaran métodos *establecer* y *obtener* `public` para las variables de instancia `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de la clase (las cuales se declaran en las líneas 6 a 10). La clase declara cada una de sus variables de instancia como `private`, por lo que los objetos de otras clases no pueden acceder directamente a estas variables. Declarar las variables de instancia como `private` y proporcionar métodos *establecer* y *obtener* para manipular y validar

las variables de instancia ayuda a cumplir con la buena ingeniería de software. Por ejemplo, los métodos `establecerVentasBrutas` y `establecerTarifaComision` validan sus argumentos antes de asignar los valores a las variables de instancia `ventasBrutas` y `tarifaComision`. En una aplicación real crítica para los negocios, también realizaríamos funciones de validación en los otros métodos *establecer* de la clase.

```
1  // Fig. 9.4: EmpleadoPorComision.java
2  // La clase EmpleadoPorComision representa a un empleado que
3  // recibe un porcentaje de las ventas brutas.
4  public class EmpleadoPorComision extends Object
5  {
6     private String primerNombre;
7     private String apellidoPaterno;
8     private String numeroSeguroSocial;
9     private double ventasBrutas; // ventas semanales totales
10    private double tarifaComision; // porcentaje de comisión
11
12    // constructor con cinco argumentos
13    public EmpleadoPorComision( String nombre, String apellido, String nss,
14       double ventas, double tarifa )
15    {
16       // la llamada implícita al constructor de Object ocurre aquí
17       primerNombre = nombre;
18       apellidoPaterno = apellido;
19       numeroSeguroSocial = nss;
20       establecerVentasBrutas( ventas ); // valida y almacena las ventas brutas
21       establecerTarifaComision( tarifa ); // valida y almacena la tarifa de comisión
22    } // fin del constructor de EmpleadoPorComision con cinco argumentos
23
24    // establece el primer nombre
25    public void establecerPrimerNombre( String nombre )
26    {
27       primerNombre = nombre;  // debería validar
28    } // fin del método establecerPrimerNombre
29
30    // devuelve el primer nombre
31    public String obtenerPrimerNombre()
32    {
33       return primerNombre;
34    } // fin del método obtenerPrimerNombre
35
36    // establece el apellido paterno
37    public void establecerApellidoPaterno( String apellido )
38    {
39       apellidoPaterno = apellido;  // debería validar
40    } // fin del método establecerApellidoPaterno
41
42    // devuelve el apellido paterno
43    public String obtenerApellidoPaterno()
44    {
```

Fig. 9.4 | La clase `EmpleadoPorComision` representa a un empleado que recibe como sueldo un porcentaje de las ventas brutas (parte 1 de 3).

```
45          return apellidoPaterno;
46       } // fin del método obtenerApellidoPaterno
47
48       // establece el número de seguro social
49       public void establecerNumeroSeguroSocial( String nss )
50       {
51          numeroSeguroSocial = nss; // debería validar
52       } // fin del método establecerNumeroSeguroSocial
53
54       // devuelve el número de seguro social
55       public String obtenerNumeroSeguroSocial()
56       {
57          return numeroSeguroSocial;
58       } // fin del método obtenerNumeroSeguroSocial
59
60       // establece el monto de ventas brutas
61       public void establecerVentasBrutas( double ventas )
62       {
63          if ( ventas >= 0.0 )
64             ventasBrutas = ventas;
65          else
66             throw new IllegalArgumentException(
67                "Las ventas brutas deben ser >= 0.0" );
68       } // fin del método establecerVentasBrutas
69
70       // devuelve el monto de ventas brutas
71       public double obtenerVentasBrutas()
72       {
73          return ventasBrutas;
74       } // fin del método obtenerVentasBrutas
75
76       // establece la tarifa de comisión
77       public void establecerTarifaComision( double tarifa )
78       {
79          if (tarifa > 0.0 && tarifa < 1.0 )
80             tarifaComision = tarifa;
81          else
82             throw new IllegalArgumentException(
83                "La tarifa de comisión debe ser > 0.0 y < 1.0" );
84       } // fin del método establecerTarifaComision
85
86       // devuelve la tarifa de comisión
87       public double obtenerTarifaComision()
88       {
89          return tarifaComision;
90       } // fin del método obtenerTarifaComision
91
92       // calcula los ingresos
93       public double ingresos()
94       {
95          return tarifaComision * ventasBrutas;
96       } // fin del método ingresos
```

Fig. 9.4 | La clase `EmpleadoPorComision` representa a un empleado que recibe como sueldo un porcentaje de las ventas brutas (parte 2 de 3).

```
97
98      // devuelve representación String del objeto EmpleadoPorComision
99      @Override // indica que este método sobrescribe el método de una superclase
100     public String toString()
101     {
102        return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
103           "empleado por comision", primerNombre, apellidoPaterno,
104           "numero de seguro social", numeroSeguroSocial,
105           "ventas brutas", ventasBrutas,
106           "tarifa de comision", tarifaComision );
107     } // fin del método toString
108  } // fin de la clase EmpleadoPorComision
```

Fig. 9.4 | La clase `EmpleadoPorComision` representa a un empleado que recibe como sueldo un porcentaje de las ventas brutas (parte 3 de 3).

El constructor de la clase `EmpleadoPorComision`

Los constructores no se heredan, por lo que la clase `EmpleadoPorComision` no hereda el constructor de la clase `Object`. Sin embargo, los constructores de una superclase de todas formas están disponibles para las subclases. De hecho, *la primera tarea del constructor de cualquier subclase es llamar al constructor de su superclase directa*, ya sea en forma explícita o implícita (si no se especifica una llamada al constructor), para asegurar que las variables de instancia heredadas de la superclase se inicialicen en forma apropiada. En este ejemplo, el constructor de la clase `EmpleadoPorComision` llama al constructor de la clase `Object` en forma implícita. En la sección 9.4.3 hablaremos sobre la sintaxis para llamar al constructor de una superclase en forma explícita. Si el código no incluye una llamada explícita al constructor de la superclase, Java genera una llamada *implícita* al constructor predeterminado o sin argumentos de la superclase. El comentario en la línea 16 de la figura 9.4 indica en dónde se hace la llamada implícita al constructor predeterminado de la superclase `Object` (el programador no necesita escribir el código para esta llamada). El constructor predeterminado (vacío) de la clase `Object` no hace nada. Observe que aun si una clase no tiene constructores, el constructor predeterminado que declara el compilador de manera implícita para la clase llamará al constructor predeterminado o sin argumentos de la superclase.

Una vez que se realiza la llamada implícita al constructor de `Object`, las líneas 17 a 21 del constructor de `EmpleadoPorComision` asignan valores a las variables de instancia de la clase. No validamos los valores de los argumentos `nombre`, `apellido` y `nss` antes de asignarlos a las variables de instancia correspondientes. Podríamos validar el nombre y el apellido; tal vez asegurarnos de que tengan una longitud razonable. De manera similar, podría validarse un número de seguro social mediante el uso de expresiones regulares (sección 16.7), para asegurar que contenga nueve dígitos, con o sin guiones cortos (por ejemplo, `123-45-6789` o `123456789`).

El método ingresos de la clase `EmpleadoPorComision`

El método `ingresos` (líneas 93 a 96) calcula los ingresos de un `EmpleadoPorComision`. La línea 95 multiplica la `tarifaComision` por las `ventasBrutas` y devuelve el resultado.

El método `toString` *y de la clase* `EmpleadoPorComision` *y la anotación* `@Override`

El método `toString` (líneas 99 a 107) es especial: es uno de los métodos que hereda cualquier clase de manera directa o indirecta de la clase `Object` (sintetizada en la sección 9.7). El método `toString` devuelve un `String` que representa a un objeto. Se llama de manera implícita cada vez que un objeto debe convertirse en una representación `String`, como cuando se imprime un objeto mediante `printf` o el método `format` de `String`, usando el especificador de formato `%s`. El método `toString` de la clase `Object` devuelve un `String` que incluye el nombre de la clase del objeto. En esencia, es un receptáculo que puede sobrescribirse por una subclase para especificar una representación `String`

apropiada de los datos en un objeto de la subclase. El método `toString` de la clase `EmpleadoPorComision` sobrescribe (redefine) al método `toString` de la clase `Object`. Al invocarse, el método `toString` de `EmpleadoPorComision` usa el método `String` llamado `format` para devolver un `String` que contiene información acerca del `EmpleadoPorComision`. Para sobrescribir a un método de una superclase, una subclase debe declarar un método con la misma firma (nombre del método, número de parámetros, tipos de los parámetros y orden de los tipos de los parámetros) que el método de la superclase; el método `toString` de `Object` no recibe parámetros, por lo que `EmpleadoPorComision` declara a `toString` sin parámetros.

La línea 99 usa la **anotación `@Override`** para indicar que el método `toString` debe sobrescribir un método de la superclase. Las anotaciones tienen varios propósitos. Por ejemplo, cuando intentamos sobrescribir el método de una superclase, los errores comunes son: asignar un nombre incorrecto al método de la subclase o utilizar el número o tipos incorrectos de los parámetros en la lista de parámetros. Cada uno de estos problemas crea una *sobrecarga no intencional* del método de la superclase. Si después tratamos de llamar al método en un objeto de la subclase, se invoca la versión de la superclase y se ignora la versión de la subclase; esto puede provocar ligeros errores lógicos. Cuando el compilador encuentra un método declarado con `@Override`, compara la firma del método con las firmas del método de la superclase. Si no hay una coincidencia exacta, el compilador emite un mensaje de error, como "el método no sobrescribe o implementa a un método de un supertipo". Esto indica que hemos sobrecargado de manera accidental un método de la superclase. Lo que sigue es corregir la firma del método, para que coincida con la del método de la superclase.

Como verá (cuando hablemos sobre las aplicaciones y los servicios Web en los capítulos 29 a 31, que se encuentran en inglés en la página Web del libro), las anotaciones también pueden agregar código de soporte complejo a nuestras clases para simplificar el proceso de desarrollo, y los servidores pueden utilizarlas para configurar ciertos aspectos de las aplicaciones Web.

Error común de programación 9.1

Usar una firma de método incorrecta al tratar de sobrescribir el método de una superclase produce una sobrecarga no intencional del método, la cual puede provocar ligeros errores lógicos.

Tip para prevenir errores 9.1

Declare los métodos sobrescritos con la anotación `@Override` para asegurar en tiempo de compilación que haya definido sus firmas correctamente. Siempre es mejor encontrar errores en tiempo de compilación que en tiempo de ejecución.

Error común de programación 9.2

Es un error de compilación sobrescribir un método con un modificador de acceso más restringido; un método `public` de la superclase no puede convertirse en un método `protected` o `private` en la subclase; un método `protected` de la superclase no puede convertirse en un método `private` en la subclase. Hacer esto sería quebrantar la relación es un, *en la que se requiere que todos los objetos de la subclase puedan responder a las llamadas a métodos que se hagan a los métodos `public` declarados en la superclase. Si un método `public` pudiera sobrescribirse como `protected` o `private`, los objetos de la subclase no podrían responder a las mismas llamadas a métodos que los objetos de la superclase. Una vez que se declara un método como `public` en una superclase, el método sigue siendo `public` para todas las subclases directas e indirectas de esa clase.*

La clase `PruebaEmpleadoPorComision`

La figura 9.5 prueba la clase `EmpleadoPorComision`. Las líneas 9 a 10 crean una instancia de un objeto `EmpleadoPorComision` e invocan a su constructor (líneas 13 a 22 de la figura 9.4) para inicializarlo con

"Sue" como el primer nombre, "Jones" como el apellido, "222-22-2222" como el número de seguro social, 10000 como el monto de ventas brutas y .06 como la tarifa de comisión. Las líneas 15 a 24 utilizan los métodos *obtener* de `EmpleadoPorComision` para obtener los valores de las variables de instancia del objeto e imprimirlas en pantalla. Las líneas 26 y 27 invocan a los métodos `establecerVentasBrutas` y `establecerTarifaComision` del objeto para modificar los valores de las variables de instancia `ventasBrutas` y `tarifaComision`. Las líneas 29 y 30 imprimen en pantalla la representación `String` del `EmpleadoPorComision` actualizado. Cuando se imprime un objeto en pantalla con el especificador de formato `%s`, se invoca de manera implícita el método `toString` del objeto para obtener su representación `String`. [*Nota:* en este capítulo no utilizaremos los métodos ingresos de nuestras clases; los usaremos mucho en el capítulo 10].

```
1  // Fig. 9.5: PruebaEmpleadoPorComision.java
2  // Programa de prueba de la clase EmpleadoPorComision.
3
4  public class PruebaEmpleadoPorComision
5  {
6     public static void main( String[] args )
7     {
8        // crea instancia de objeto EmpleadoPorComision
9        EmpleadoPorComision empleado = new EmpleadoPorComision(
10          "Sue", "Jones", "222-22-2222", 10000, .06 );
11
12       // obtiene datos del empleado por comisión
13       System.out.println(
14          "Informacion del empleado obtenida por los metodos establecer: \n" );
15       System.out.printf( "%s %s\n", "El primer nombre es",
16          empleado.obtenerPrimerNombre() );
17       System.out.printf( "%s %s\n", "El apellido paterno es",
18          empleado.obtenerApellidoPaterno() );
19       System.out.printf( "%s %s\n", "El numero de seguro social es",
20          empleado.obtenerNumeroSeguroSocial() );
21       System.out.printf( "%s %.2f\n", "Las ventas brutas son",
22          empleado.obtenerVentasBrutas() );
23       System.out.printf( "%s %.2f\n", "La tarifa de comision es",
24          empleado.obtenerTarifaComision() );
25
26       empleado.establecerVentasBrutas( 500 ); // establece las ventas brutas
27       empleado.establecerTarifaComision( .1 ); // establece la tarifa de comisión
28
29       System.out.printf( "\n%s:\n\n%s\n",
30          "Informacion actualizada del empleado, obtenida mediante toString", empleado );
31    } // fin de main
32 } // fin de la clase PruebaEmpleadoPorComision
```

```
Informacion del empleado obtenida por los metodos establecer:

El primer nombre es Sue
El apellido paterno es Jones
El numero de seguro social es 222-22-2222
Las ventas brutas son 10000.00
La tarifa de comision es 0.06
```

Fig. 9.5 | Programa de prueba de la clase `EmpleadoPorComision` (parte 1 de 2).

```
Informacion actualizada del empleado, obtenida mediante toString:

empleado por comision: Sue Jones
numero de seguro social: 222-22-2222
ventas brutas: 500.00
tarifa de comision: 0.10
```

Fig. 9.5 | Programa de prueba de la clase `EmpleadoPorComision` (parte 2 de 2).

9.4.2 Creación y uso de una clase `EmpleadoBaseMasComision`

Ahora hablaremos sobre la segunda parte de nuestra introducción a la herencia, mediante la declaración y prueba de la clase (completamente nueva e independiente) `EmpleadoBaseMasComision` (figura 9.6), la cual contiene los siguientes datos: primer nombre, apellido paterno, número de seguro social, monto de ventas brutas, tarifa de comisión *y* salario base. Los servicios `public` de la clase `EmpleadoBaseMasComision` incluyen un constructor `EmpleadoBaseMasComision` (líneas 15 a 25), y los métodos `ingresos` (líneas 112 a 115) y `toString` (líneas 118 a 127). Las líneas 28 a 109 declaran métodos *establecer* y *obtener* `public` para las variables de instancia `private` `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas`, `tarifaComision` *y* `salarioBase` de la clase (las cuales se declaran en las líneas 7 a 12). Estas variables y métodos encapsulan todas las características necesarias de un empleado por comisión con sueldo base. Observe la *similitud* entre esta clase y la clase `EmpleadoPorComision` (figura 9.4); en este ejemplo, no explotaremos todavía esa similitud.

```
1  // Fig. 9.6: EmpleadoBaseMasComision.java
2  // La clase EmpleadoBaseMasComision representa a un empleado que recibe
3  // un salario base, además de la comisión.
4
5  public class EmpleadoBaseMasComision
6  {
7     private String primerNombre;
8     private String apellidoPaterno;
9     private String numeroSeguroSocial;
10    private double ventasBrutas; // ventas totales por semana
11    private double tarifaComision; // porcentaje de comisión
12    private double salarioBase; // salario base por semana
13
14    // constructor con seis argumentos
15    public EmpleadoBaseMasComision( String nombre, String apellido,
16       String nss, double ventas, double tarifa, double salario )
17    {
18       // la llamada implícita al constructor de Object ocurre aquí
19       primerNombre = nombre;
20       apellidoPaterno = apellido;
21       numeroSeguroSocial = nss;
22       establecerVentasBrutas( ventas ); // valida y almacena las ventas brutas
23       establecerTarifaComision( tarifa ); // valida y almacena la tarifa de comisión
24       establecerSalarioBase( salario ); // valida y almacena el salario base
25    } // fin del constructor de EmpleadoBaseMasComision con seis argumentos
```

Fig. 9.6 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un sueldo base, además de una comisión (parte 1 de 3).

```
26
27    // establece el primer nombre
28    public void establecerPrimerNombre( String nombre )
29    {
30       primerNombre = nombre;
31    } // fin del método establecerPrimerNombre
32
33    // devuelve el primer nombre
34    public String obtenerPrimerNombre()
35    {
36       return primerNombre;
37    } // fin del método obtenerPrimerNombre
38
39    // establece el apellido paterno
40    public void establecerApellidoPaterno( String apellido )
41    {
42       apellidoPaterno = apellido;  //debe validar
43    } // fin del método establecerApellidoPaterno
44
45    // devuelve el apellido paterno
46    public String obtenerApellidoPaterno()
47    {
48       return apellidoPaterno;
49    } // fin del método obtenerApellidoPaterno
50
51    // establece el número de seguro social
52    public void establecerNumeroSeguroSocial( String nss )
53    {
54       numeroSeguroSocial = nss; // debe validar
55    } // fin del método establecerNumeroSeguroSocial
56
57    // devuelve el número de seguro social
58    public String obtenerNumeroSeguroSocial()
59    {
60       return numeroSeguroSocial;
61    } // fin del método obtenerNumeroSeguroSocial
62
63    // establece el monto de ventas brutas
64    public void establecerVentasBrutas( double ventas )
65    {
66       if ( ventas >= 0.0 )
67          ventasBrutas ) ventas;
68       else
69          throw new IllegalArgumentException(
70             "Las ventas brutas deben ser >= 0.0" );
71    } // fin del método establecerVentasBrutas
72
73    // devuelve el monto de ventas brutas
74    public double obtenerVentasBrutas()
75    {
76       return ventasBrutas;
77    } // fin del método obtenerVentasBrutas
```

Fig. 9.6 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un sueldo base, además de una comisión (parte 2 de 3).

```
78
79      // establece la tarifa de comisión
80      public void establecerTarifaComision( double tarifa )
81      {
82         if ( tarifa > 0.0 && tarifa < 1.0 )
83            tarifaComision = tarifa;
84         else
85            throw new IllegalArgumentException(
86               "La tarifa de comisión debe ser > 0.0 y < 1.0" );
87      } // fin del método establecerTarifaComision
88
89      // devuelve la tarifa de comisión
90      public double obtenerTarifaComision()
91      {
92         return tarifaComision;
93      } // fin del método obtenerTarifaComision
94
95      // establece el salario base
96      public void establecerSalarioBase( double salario )
97      {
98         if ( salario >= 0.0 )
99            salarioBase = salario;
100        else
101           throw new IllegalArgumentException(
102              "El salario base debe ser >= 0.0" );
103     } // fin del método establecerSalarioBase
104
105     // devuelve el salario base
106     public double obtenerSalarioBase()
107     {
108        return salarioBase;
109     } // fin del método obtenerSalarioBase
110
111     // calcula los ingresos
112     public double ingresos()
113     {
114        return salarioBase + ( tarifaComision * ventasBrutas );
115     } //fin del método ingresos
116
117     // devuelve representación String de EmpleadoBaseMasComision
118     @Override  // indica que este método sobrescribe el método de la superclase
119     public String toString()
120     {
121        return String.format(
122           "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
123           "empleado por comision con sueldo base", primerNombre, apellidoPaterno,
124           "numero de seguro social", numeroSeguroSocial,
125           "ventas brutas", ventasBrutas, "tarifa de comision", tarifaComision,
126           "salario base", salarioBase );
127     } // fin del método toString
128  } // fin de la clase EmpleadoBaseMasComision
```

Fig. 9.6 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un sueldo base, además de una comisión (parte 3 de 3).

Puesto que la clase `EmpleadoBaseMasComision` no especifica "`extends Object`" en la línea 5, entonces la clase extiende a `Object` en forma implícita. Además, de la misma manera que el constructor de la clase `EmpleadoPorComision` (líneas 13 a 22 de la figura 9.4), el constructor de la clase `EmpleadoBaseMasComision` invoca al constructor predeterminado de la clase `Object` en forma implícita, como se indica en el comentario de la línea 18.

El método ingresos de la clase `EmpleadoBaseMasComision` (líneas 112 a 115) devuelve el resultado de sumar el salario base del `EmpleadoBaseMasComision` al producto de multiplicar la tarifa de comisión por las ventas brutas del empleado.

La clase `EmpleadoBaseMasComision` sobrescribe al método `toString` de `Object` para que devuelva un objeto `String` que contiene la información del `EmpleadoBaseMasComision`. Una vez más, utilizamos el especificador de formato `%.2f` para dar formato a las ventas brutas, la tarifa de comisión y el salario base con dos dígitos de precisión a la derecha del punto decimal (línea 122).

Prueba de la clase `EmpleadoBaseMasComision`

La figura 9.7 prueba la clase `EmpleadoBaseMasComision`. Las líneas 9 a 11 crean un objeto `EmpleadoBaseMasComision` y pasan los argumentos "`Bob`", "`Lewis`", "`333-33-3333`", `5000`, `.04` y `300` al constructor como el primer nombre, apellido paterno, número de seguro social, ventas brutas, tarifa de comisión y salario base, respectivamente. Las líneas 16 a 27 utilizan los métodos *obtener* de `EmpleadoBaseMasComision` para obtener los valores de las variables de instancia del objeto e imprimirlos en pantalla. La línea 29 invoca al método `establecerSalarioBase` del objeto para modificar el salario base. El método `establecerSalarioBase` (figura 9.6, líneas 88 a 91) asegura que no se le asigne a la variable `salarioBase` un valor negativo. Las líneas 31 a 33 de la figura 9.7 invocan en forma implícita al método `toString` del objeto, para obtener su representación `String`.

```
1  // Fig. 9.7: PruebaEmpleadoBaseMasComision.java
2  // Programa de prueba de EmpleadoBaseMasComision.
3
4  public class PruebaEmpleadoBaseMasComision
5  {
6     public static void main( String[] args )
7     {
8        // crea instancia de objeto EmpleadoBaseMasComision
9        EmpleadoBaseMasComision empleado =
10          new EmpleadoBaseMasComision(
11          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
12
13       // obtiene datos del empleado por comisión con sueldo base
14       System.out.println(
15          "Informacion del empleado obtenida por metodos establecer: \n" );
16       System.out.printf( "%s %s\n", "El primer nombre es",
17          empleado.obtenerPrimerNombre() );
18       System.out.printf( "%s %s\n", "El apellido es",
19          empleado.obtenerApellidoPaterno() );
20       System.out.printf( "%s %s\n", "El numero de seguro social es",
21          empleado.obtenerNumeroSeguroSocial() );
22       System.out.printf( "%s %.2f\n", "Las ventas brutas son",
23          empleado.obtenerVentasBrutas() );
24       System.out.printf( "%s %.2f\n", "La tarifa de comision es",
25          empleado.obtenerTarifaComision() );
26       System.out.printf( "%s %.2f\n", "El salario base es",
27          empleado.obtenerSalarioBase() );
```

Fig. 9.7 | Programa de prueba de `EmpleadoBaseMasComision` (parte 1 de 2).

```
28
29        empleado.establecerSalarioBase( 1000 ); // establece el salario base
30
31        System.out.printf( "\n%s:\n\n%s\n",
32           "Informacion actualizada del empleado, obtenida por toString",
33           empleado.toString() );
34     } // fin de main
35  } // fin de la clase PruebaEmpleadoBaseMasComision
```

```
Informacion del empleado obtenida por metodos establecer:

El primer nombre es Bob
El apellido es Lewis
El numero de seguro social es 333-33-3333
Las ventas brutas son 5000.00
La tarifa de comision es 0.04
El salario base es 300.00

Informacion actualizada del empleado, obtenida por toString:

empleado por comision con sueldo base: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
salario base: 1000.00
```

Fig. 9.7 | Programa de prueba de `EmpleadoBaseMasComision` (parte 2 de 2).

Notas sobre la clase `EmpleadoBaseMasComision`

La mayor parte del código de la clase `EmpleadoBaseMasComision` (figura 9.6) es similar, si no es que idéntico, al código de la clase `EmpleadoPorComision` (figura 9.4). Por ejemplo, las variables de instancia `private` `primerNombre` y `apellidoPaterno`, y los métodos `establecerPrimerNombre`, `obtenerPrimerNombre`, `establecerApellidoPaterno` y `obtenerApellidoPaterno` son idénticos a los de la clase `EmpleadoPorComision`. Ambas clases también contienen las variables de instancia `private` `numeroSeguroSocial`, `tarifaComision` y `ventasBrutas`, así como los correspondientes métodos *obtener* y *establecer*. Además, el constructor de `EmpleadoBaseMasComision` es casi idéntico al de la clase `EmpleadoPorComision`, sólo que el constructor de `EmpleadoBaseMasComision` también establece el `salarioBase`. Las demás adiciones a la clase `EmpleadoBaseMasComision` son la variable de instancia `private` `salarioBase`, y los métodos `establecerSalarioBase` y `obtenerSalarioBase`. El método `toString` de la clase `EmpleadoBaseMasComision` es casi idéntico al de la clase `EmpleadoPorComision`, excepto que también imprime la variable de instancia `salarioBase` con dos dígitos de precisión a la derecha del punto decimal.

Literalmente hablando, *copiamos* el código de la clase `EmpleadoPorComision` y lo *pegamos* en la clase `EmpleadoBaseMasComision`, después modificamos esta clase para incluir un salario base y los métodos que lo manipulan. A menudo, este método de *"copiar y pegar"* está propenso a errores y consume mucho tiempo. Peor aún, se pueden esparcir muchas copias físicas del mismo código a lo largo de un sistema, con lo que el mantenimiento del código se convierte en una pesadilla. ¿Existe alguna manera de "absorber" las variables de instancia y los métodos de una clase, de manera que formen parte de otras clases *sin tener que copiar el código*? En los siguientes ejemplos responderemos a esta pregunta, utilizando un método más elegante para crear clases, que enfatiza los beneficios de la herencia.

Observación de ingeniería de software 9.3

Con la herencia, las variables de instancia y los métodos comunes *de todas las clases en la jerarquía se declaran en una superclase. Cuando se realizan modificaciones para estas características comunes en la superclase, entonces las subclases heredan los cambios. Sin la herencia, habría que modificar todos los archivos de código fuente que contengan una* copia *del código en cuestión.*

9.4.3 Creación de una jerarquía de herencia EmpleadoPorComision-EmpleadoBaseMasComision

Ahora declararemos la clase `EmpleadoBaseMasComision` (figura 9.8), que *extiende* a la clase `EmpleadoPorComision` (figura 9.4). Un objeto `EmpleadoBaseMasComision` *es un* `EmpleadoPorComision`, ya que la herencia traspasa las capacidades de la clase `EmpleadoPorComision`. La clase `EmpleadoBaseMasComision` también tiene la variable de instancia `salarioBase` (figura 9.8, línea 6). La palabra clave `extends` (línea 4) indica la herencia. `EmpleadoBaseMasComision` *hereda* las variables de instancia y los métodos de la clase `EmpleadoPorComision`, pero sólo se puede acceder de manera directa a los miembros `public` y `protected` de la superclase. El constructor de `EmpleadoPorComision` *no* se hereda. Por lo tanto, los servicios `public` de `EmpleadoBaseMasComision` incluyen su constructor (líneas 9 a 16), los métodos `public` heredados de `EmpleadoPorComision`, y los métodos `establecerSalarioBase` (líneas 19 a 26), `obtenerSalarioBase` (líneas 29 a 32), `ingresos` (líneas 35 a 40) y `toString` (líneas 43 a 53). Los métodos `ingresos` y `toString` *sobrescriben* los correspondientes métodos en la clase `EmpleadoPorComision`, ya que las versiones de su superclase no calculan de manera correcta los ingresos de un `EmpleadoBaseMasComision`, ni devuelven una representación `String` apropiada.

```
1  // Fig. 9.8: EmpleadoBaseMasComision.java
2  // Los miembros private de la superclase no se pueden utilizar en una subclase.
3
4  public class EmpleadoBaseMasComision extends EmpleadoPorComision
5  {
6     private double salarioBase; // salario base por semana
7
8     // constructor con seis argumentos
9     public EmpleadoBaseMasComision( String nombre, String apellido,
10       String nss, double ventas, double tarifa, double salario )
11    {
12       // llamada explícita al constructor de la superclase EmpleadoPorComision
13       super( nombre, apellido, nss, ventas, tarifa );
14
15       establecerSalarioBase( salario ); // valida y almacena el salario base
16    } // fin del constructor de EmpleadoBaseMasComision con seis argumentos
17
18    // establece el salario base
19    public void establecerSalarioBase( double salario )
20    {
21       if ( salario >= 0.0 )
22          salarioBase = salario;
23       else
24          throw new IllegalArgumentException(
25             "El salario base debe ser >= 0.0" );
26    } // fin del método establecerSalarioBase
```

Fig. 9.8 | Los miembros `private` de una superclase no se pueden utilizar en una subclase (parte 1 de 3).

```
27
28      // devuelve el salario base
29      public double obtenerSalarioBase()
30      {
31         return salarioBase;
32      } // fin del método obtenerSalarioBase
33
34      // calcula los ingresos
35      @Override  // indica que este método sobrescribe al método de la superclase
36      public double ingresos()
37      {
38         // no está permitido: tarifaComision y ventasBrutas son private en la superclase
39         return salarioBase + ( tarifaComision * ventasBrutas );
40      } // fin del método ingresos
41
42      // devuelve representación String de EmpleadoBaseMasComision
43      @Overrride // indica que este método sobrescribe al método de la superclase
44      public String toString()
45      {
46         // no está permitido: intentos por acceder a los miembros private de la superclase
47         return String.format(
48            "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
49            "empleado por comision con sueldo base", primerNombre, apellidoPaterno,
50            "numero de seguro social", numeroSeguroSocial,
51            "ventas brutas", ventasBrutas, "tarifa de comision", tarifaComision,
52            "salario base", salarioBase );
53      } // fin del método toString
54   } // fin de la clase EmpleadoBaseMasComision
```

```
EmpleadoBaseMasComision.java:39: error: tarifaComision has private access in Empleado-
PorComision
      return salarioBase + ( tarifaComision * ventasBrutas );
                             ^
EmpleadoBaseMasComision.java:39: error: ventasBrutas has private access in Empleado-
PorComision
      return salarioBase + ( tarifaComision * ventasBrutas );
                                              ^
EmpleadoBaseMasComision.java:49: error: primerNombre has private access in EmpleadoPor-
Comision
         "empleado por comision con sueldo base", primerNombre, apellidoPaterno,
                                                  ^

EmpleadoBaseMasComision.java:49: error: apellidoPaterno has private access in Empleado-
PorComision
         "empleado por comision con sueldo base", primerNombre, apellidoPaterno,
                                                                ^

EmpleadoBaseMasComision.java:50: error: numeroSeguroSocial has private access in
EmpleadoPorComision
         "numero de seguro social", numeroSeguroSocial,
                                    ^
EmpleadoBaseMasComision.java:51: error: ventasBrutas has private access in Empleado-
PorComision
         "ventas brutas", ventasBrutas, "tarifa de comision", tarifaComision,
                          ^
```

Fig. 9.8 | Los miembros `private` de una superclase no se pueden utilizar en una subclase (parte 2 de 3).

```
EmpleadoBaseMasComision.java:51: error: tarifaComision has private access in Empleado-
PorComision
        "ventas brutas", ventasBrutas, "tarifa de comision", tarifaComision,
                                                             ^
7 errors
```

Fig. 9.8 | Los miembros `private` de una superclase no se pueden utilizar en una subclase (parte 3 de 3).

El constructor de una subclase debe llamar al constructor de su superclase

El constructor de cada subclase debe llamar en forma implícita o explícita al constructor de su superclase para inicializar las variables de instancia heredadas de la superclase. La línea 13 en el constructor de `EmpleadoBaseMasComision` con seis argumentos (líneas 9 a 16) llama en forma explícita al constructor de la clase `EmpleadoPorComision` con cinco argumentos (declarado en las líneas 13 a 22 de la figura 9.4), para inicializar la porción correspondiente a la superclase de un objeto `EmpleadoBaseMasComision` (es decir, las variables `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision`). Para ello utilizamos la **sintaxis de llamada al constructor de la superclase**: la palabra clave `super`, seguida de un conjunto de paréntesis que contienen los argumentos del constructor de la superclase. Los argumentos `nombre`, `apellido`, `nss`, `ventas` y `tarifa` se utilizan para inicializar a los miembros `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de la superclase. Si el constructor de `EmpleadoBaseMasComision` no invocara al constructor de la superclase en forma explícita, Java trataría de invocar al constructor predeterminado o sin argumentos de la superclase. Como la clase `EmpleadoPorComision` no tiene un constructor así, el compilador generaría un error. La llamada explícita al constructor de la superclase en la línea 13 de la figura 9.8 debe ser la *primera* instrucción en el cuerpo del constructor de la subclase. Cuando una superclase contiene un constructor sin argumentos, puede usar a `super()` para llamar a ese constructor en forma explícita, pero esto se hace raras veces.

El método ingresos de `EmpleadoBaseMasComision`

El compilador genera errores para la línea 39 debido a que las variables de instancia `tarifaComision` y `ventasBrutas` de la superclase `EmpleadoPorComision` son `private`; no se permite a los métodos de la subclase `EmpleadoBaseMasComision` acceder a las variables de instancia `private` de la superclase `EmpleadoPorComision`. Utilizamos texto en gris en la figura 9.8 para indicar que el código es erróneo. El compilador genera errores adicionales en las líneas 49 a 51 del método `toString` de `EmpleadoBaseMasComision` por la misma razón. Se hubieran podido prevenir los errores en `EmpleadoBaseMasComision` al utilizar los métodos *obtener* heredados de la clase `EmpleadoPorComision`. Por ejemplo, la línea 39 podría haber utilizado `obtenerTarifaComision` y `obtenerVentasBrutas` para acceder a las variables de instancia `private` `tarifaComision` y `ventasBrutas` de `EmpleadoPorComision`. Las líneas 49 a 51 también podrían haber utilizado métodos establecer apropiados para *obtener* los valores de las variables de instancia de la superclase.

9.4.4 La jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` mediante el uso de variables de instancia `protected`

Para permitir que la clase `EmpleadoBaseMasComision` acceda en forma directa a las variables de instancia `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de la superclase, podemos declarar esos miembros como `protected` en la superclase. Como vimos en la sección 9.3, los miembros `protected` de una superclase se heredan por todas las subclases de esa superclase. En la nueva clase `EmpleadoPorComision`, modificamos sólo las líneas 6 a 10 de la figura 9.4 para declarar las variables de instancia con el modificador de acceso `protected`, como se muestra a continuación:

```
protected String primerNombre;
protected String apellidoPaterno;
protected String numeroSeguroSocial;
protected double ventasBrutas; // ventas totales por semana
protected double tarifaComision; // porcentaje de comisión
```

El resto de la declaración de la clase (que no mostramos aquí) es idéntico al de la figura 9.4.

Podríamos haber declarado las variables de instancia de `EmpleadoPorComision` como `public`, para permitir que la subclase `EmpleadoBaseMasComision` pueda acceder a ellas. No obstante, declarar variables de instancia `public` es una mala ingeniería de software, ya que permite el acceso sin restricciones a las variables de instancia, lo cual incrementa de manera considerable la probabilidad de errores. Con las variables de instancia `protected`, la subclase obtiene acceso a las variables de instancia, pero las clases que no son subclases y las clases que no están en el mismo paquete no pueden acceder a estas variables en forma directa; recuerde que los miembros de clase `protected` son también visibles para las otras clases en el mismo paquete.

La clase `EmpleadoBaseMasComision`

La clase `EmpleadoBaseMasComision` (figura 9.9) extiende la nueva versión de `EmpleadoPorComision` con variables de instancia `protected`. Los objetos de `EmpleadoBaseMasComision` heredan las variables de instancia `protected` `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de `EmpleadoPorComision`; ahora todas estas variables son miembros `protected` de `EmpleadoBaseMasComision`. Como resultado, el compilador no genera errores al compilar la línea 37 del método ingresos y las líneas 46 a 48 del método `toString`. Si otra clase extiende esta versión de la clase `EmpleadoBaseMasComision`, la nueva subclase también puede acceder a los miembros `protected`.

Cuando creamos un objeto `EmpleadoBaseMasComision`, éste contiene todas las variables de instancia declaradas en la jerarquía de clases hasta ese punto; es decir, las que pertenecen a las clases `Object`, `EmpleadoPorComision` y `EmpleadoBaseMasComision`. La clase `EmpleadoBaseMasComision` no hereda el constructor de la clase `EmpleadoPorComision`. Sin embargo, el constructor de la clase `EmpleadoBaseMasComision` con seis argumentos (líneas 10 a 15) llama al constructor de la clase `EmpleadoPorComision` con cinco argumentos en forma *explícita* para inicializar las variables de instancia que `EmpleadoBaseMasComision` heredó de la clase `EmpleadoPorComision`. De igual forma, el constructor de `EmpleadoPorComision` llama en forma *implícita* al constructor de la clase `Object`. El constructor de `EmpleadoBaseMasComision` debe hacer esto en forma *explícita*, debido a que `EmpleadoPorComision` no proporciona un constructor sin argumentos que pueda invocarse en forma implícita.

```
1  // Fig. 9.9: EmpleadoBaseMasComision.java
2  // EmpleadoBaseMasComision hereda las variables de instancia
3  // protected de EmpleadoPorComision.
4
5  public class EmpleadoBaseMasComision extends EmpleadoPorComision
6  {
7     private double salarioBase; // salario base por semana
8
9     // constructor con seis argumentos
10    public EmpleadoBaseMasComision( String nombre, String apellido,
11       String nss, double ventas, double tarifa, double salario )
12    {
```

Fig. 9.9 | `EmpleadoBaseMasComision` hereda las variables de instancia `protected` de `EmpleadoPorComision` (parte 1 de 2).

```
13          super( nombre, apellido, nss, ventas, tarifa );
14          establecerSalarioBase( salario ); // valida y almacena el salario base
15       } // fin del constructor de EmpleadoBaseMasComision con seis argumentos
16
17       // establece el salario base
18       public void establecerSalarioBase( double salario )
19       {
20          if ( salario >= 0.0 )
21             salarioBase = salario;
22          else
23             throw new IllegalArgumentException(
24                "El salario base debe ser >= 0.0" );
25       } // fin del método establecerSalarioBase
26
27       // devuelve el salario base
28       public double obtenerSalarioBase()
29       {
30          return salarioBase;
31       } // fin del método obtenerSalarioBase
32
33       // calcula los ingresos
34       @Override // indica que este método sobrescribe al método de la superclase
35       public double ingresos()
36       {
37          return salarioBase + ( tarifaComision * ventasBrutas );
38       } // fin del método ingresos
39
40       // devuelve representación String de EmpleadoBaseMasComision
41       @Override // indica que este método sobrescribe al método de la superclase
42       public String toString()
43       {
44          return String.format(
45             "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
46             "empleado por comision con salario base", primerNombre, apellidoPaterno,
47             "numero de seguro social", numeroSeguroSocial,
48             "ventas brutas", ventasBrutas, "tarifa comision", tarifaComision,
49             "salario base", salarioBase );
50       } // fin del método toString
51    } // fin de la clase EmpleadoBaseMasComision
```

Fig. 9.9 | `EmpleadoBaseMasComision` hereda las variables de instancia `protected` de `EmpleadoPorComision` (parte 2 de 2).

Prueba de la clase `EmpleadoBaseMasComision`

La clase `PruebaEmpleadoBaseMasComision` para este ejemplo es idéntica a la de la figura 9.7 y produce el mismo resultado, por lo que no lo mostraremos aquí. Aunque la versión de la clase `EmpleadoBaseMasComision` en la figura 9.6 no utiliza la herencia y la versión en la figura 9.9 sí, *ambas clases proveen la misma funcionalidad*. El código fuente en la figura 9.9 (47 líneas) es mucho más corto que el de la figura 9.6 (116 líneas), debido a que la mayor parte de la funcionalidad de `EmpleadoBaseMasComision` se hereda de `EmpleadoPorComision`; ahora sólo hay una copia de la funcionalidad de `EmpleadoPorComision`. Esto hace que el código sea más fácil de mantener, modificar y depurar, puesto que el código relacionado con un empleado por comisión sólo existe en la clase `EmpleadoPorComision`.

Notas sobre el uso de variable de instancia `protected`

En este ejemplo declaramos las variables de instancia de la superclase como `protected`, para que las subclases pudieran acceder a ellas. Al heredar variables de instancia `protected` se incrementa un poco el rendimiento, ya que podemos acceder de modo directo a las variables en la subclase, sin incurrir en la sobrecarga de una llamada a un método *establecer* u *obtener*. No obstante, en la mayoría de los casos es mejor utilizar variables de instancia `private`, para cumplir con la ingeniería de software apropiada, y dejar al compilador las cuestiones relacionadas con la optimización de código. Su código será más fácil de mantener, modificar y depurar.

El uso de variables de instancia `protected` crea varios problemas potenciales. En primer lugar, el objeto de la subclase puede establecer de manera directa el valor de una variable heredada, sin utilizar un método *establecer*. Por lo tanto, un objeto de la subclase puede asignar un valor inválido a la variable, con lo cual el objeto puede quedar en un estado inconsistente. Por ejemplo, si declaramos la variable de instancia `ventasBrutas` de `EmpleadoPorComision` como `protected`, un objeto de una subclase (por ejemplo, `EmpleadoBaseMasComision`) podría entonces asignar un valor negativo a `ventasBrutas`. Otro problema con el uso de variables de instancia `protected` es que hay más probabilidad de que los métodos de la subclase se escriban de manera que dependan de la implementación de datos de la superclase. En la práctica, las subclases sólo deben depender de los servicios de la superclase (es decir, métodos que no sean `private`) y no en la implementación de datos de la superclase. Si hay variables de instancia `protected` en la superclase, tal vez necesitemos modificar todas las subclases de esa superclase, si cambia la implementación de ésta. Por ejemplo, si por alguna razón tuviéramos que cambiar los nombres de las variables de instancia `primerNombre` y `apellidoPaterno` por nombre y apellido, entonces tendríamos que hacerlo para todas las ocurrencias en las que una subclase haga referencia directa a las variables de instancia `primerNombre` y `apellidoPaterno` de la superclase. En tal caso, se dice que el software es **frágil** o **quebradizo**, ya que un pequeño cambio en la superclase puede "quebrar" la implementación de la subclase. Es conveniente que el programador pueda modificar la implementación de la superclase sin dejar de proporcionar los mismos servicios a las subclases. Desde luego que, si cambian los servicios de la superclase, debemos reimplementar nuestras subclases. Un tercer problema es que los miembros `protected` de una clase son visibles para todas las clases que se encuentren en el mismo paquete que la clase que contiene los miembros `protected`; esto no siempre es conveniente.

Observación de ingeniería de software 9.4

Use el modificador de acceso `protected` cuando una superclase deba proporcionar un método sólo a sus subclases y a otras clases en el mismo paquete, pero no a otros clientes.

Observación de ingeniería de software 9.5

Al declarar variables de instancia `private` (a diferencia de `protected`) en la superclase, se permite que la implementación de la superclase para estas variables de instancia cambie sin afectar las implementaciones de las subclases.

Tip para prevenir errores 9.2

Evite usar variables de instancia `protected` en una superclase. En vez de ello, incluya métodos no `private` que accedan a las variables de instancia `private`. Esto asegurará que los objetos de la clase mantengan estados consistentes.

9.4.5 La jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` mediante el uso de variables de instancia `private`

Ahora reexaminaremos nuestra jerarquía una vez más, pero esta vez utilizaremos las mejores prácticas de ingeniería de software. La clase `EmpleadoPorComision` (figura 9.10) declara las variables de instancia `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` como

`private` (líneas 6 a 10), y proporciona los métodos `public` `establecerPrimerNombre`, `obtenerPrimerNombre`, `establecerApellidoPaterno`, `obtenerApellidoPaterno`, `establecerNumeroSeguroSocial`, `obtenerNumeroSeguroSocial`, `establecerVentasBrutas`, `obtenerVentasBrutas`, `establecerTarifaComision`, `obtenerTarifaComision`, `ingresos` y `toString` para manipular estos valores. Los métodos `ingresos` (líneas 93 a 96) y `toString` (líneas 99 a 107) utilizan los métodos obtener de la clase para sacar los valores de sus variables de instancia. Si decidimos modificar los nombres de las variables de instancia, no habrá que modificar las declaraciones de ingresos y de `toString`; sólo habrá que cambiar los cuerpos de los métodos *obtener* y *establecer* que manipulan directamente estas variables de instancia. Estos cambios ocurren sólo dentro de la superclase; no se necesitan cambios en la subclase. La localización de los efectos de los cambios como éste es una buena práctica de ingeniería de software.

```
1  // Fig. 9.10: EmpleadoPorComision.java
2  // EmpleadoPorComision usa los métodos para manipular sus
3  // variables de instancia private.
4  public class EmpleadoPorComision
5  {
6     private String primerNombre;
7     private String apellidoPaterno;
8     private String numeroSeguroSocial;
9     private double ventasBrutas; // ventas totales por semana
10    private double tarifaComision; // porcentaje de comisión
11
12    // constructor con cinco argumentos
13    public EmpleadoPorComision( String nombre, String apellido, String nss,
14       double ventas, double tarifa )
15    {
16       // la llamada implícita al constructor de Object ocurre aquí
17       primerNombre = nombre;
18       apellidoPaterno = apellido;
19       numeroSeguroSocial = nss;
20       establecerVentasBrutas( ventas ); // valida y almacena las ventas brutas
21       establecerTarifaComision( tarifa ); // valida y almacena la tarifa de comisión
22    } // fin del constructor de EmpleadoPorComision con cinco argumentos
23
24    // establece el primer nombre
25    public void establecerPrimerNombre( String nombre )
26    {
27       primerNombre = nombre;  // debería validar
28    } // fin del método establecerPrimerNombre
29
30    // devuelve el primer nombre
31    public String obtenerPrimerNombre()
32    {
33       return primerNombre;
34    } // fin del método obtenerPrimerNombre
35
36    // establece el apellido paterno
37    public void establecerApellidoPaterno( String apellido )
38    {
```

Fig. 9.10 | La clase `EmpleadoPorComision` utiliza métodos para manipular sus variables de instancia `private` (parte 1 de 3).

```
39          apellidoPaterno = apellido; // debería validar
40       } // fin del método establecerApellidoPaterno
41
42       // devuelve el apellido paterno
43       public String obtenerApellidoPaterno()
44       {
45          return apellidoPaterno;
46       } // fin del método obtenerApellidoPaterno
47
48       // establece el número de seguro social
49       public void establecerNumeroSeguroSocial( String nss )
50       {
51          numeroSeguroSocial = nss; // debería validar
52       } // fin del método establecerNumeroSeguroSocial
53
54       // devuelve el número de seguro social
55       public String obtenerNumeroSeguroSocial()
56       {
57          return numeroSeguroSocial;
58       } // fin del método obtenerNumeroSeguroSocial
59
60       // establece el monto de ventas brutas
61       public void establecerVentasBrutas( double ventas )
62       {
63          if ( ventas >= 0.0 )
64             ventasBrutas = ventas;
65          else
66             throw new IllegalArgumentException(
67                "Las ventas brutas deben ser >= 0.0" );
68       } // fin del método establecerVentasBrutas
69
70       // devuelve el monto de ventas brutas
71       public double obtenerVentasBrutas()
72       {
73          return ventasBrutas;
74       } // fin del método obtenerVentasBrutas
75
76       // establece la tarifa de comisión
77       public void establecerTarifaComision( double tarifa )
78       {
79          if (tarifa > 0.0 && tarifa < 1.0 )
80             tarifaComision = tarifa;
81          else
82             throw new IllegalArgumentException(
83                "La tarifa de comisión debe ser > 0.0 y < 1.0" );
84       } // fin del método establecerTarifaComision
85
86       // devuelve la tarifa de comisión
87       public double obtenerTarifaComision()
88       {
89          return tarifaComision;
90       } // fin del método obtenerTarifaComision
```

Fig. 9.10 | La clase `EmpleadoPorComision` utiliza métodos para manipular sus variables de instancia `private` (parte 2 de 3).

```
91
92     // calcula los ingresos
93     public double ingresos()
94     {
95        return obtenerTarifaComision() * obtenerVentasBrutas();
96     } // fin del método ingresos
97
98     // devuelve representación String del objeto EmpleadoPorComision
99     @Override // indica que este método sobrescribe el método de la superclase
100    public String toString()
101    {
102       return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
103          "empleado por comision", obtenerPrimerNombre(), obtenerApellidoPaterno(),
104          "numero de seguro social", obtenerNumeroSeguroSocial(),
105          "ventas brutas", obtenerVentasBrutas(),
106          "tarifa de comision", obtenerTarifaComision() );
107    } // fin del método toString
108 } // fin de la clase EmpleadoPorComision
```

Fig. 9.10 | La clase `EmpleadoPorComision` utiliza métodos para manipular sus variables de instancia `private` (parte 3 de 3).

La subclase `EmpleadoBaseMasComision` (figura 9.11) hereda los miembros no `private` de `EmpleadoPorComision`3 y puede acceder a los miembros `private` de su superclase, a través de esos métodos. La clase `EmpleadoBaseMasComision` tiene varios cambios que la diferencian de la figura 9.9. Los métodos `ingresos` (líneas 35 a 39) y `toString` (líneas 42 a 47) invocan cada uno al método `obtenerSalarioBase` para conseguir el valor del salario base, en vez de acceder en forma directa a `salarioBase`. Si decidimos cambiar el nombre de la variable de instancia `salarioBase`, sólo habrá que modificar los cuerpos de los métodos `establecerSalarioBase` y `obtenerSalarioBase`.

```
1  // Fig. 9.11: EmpleadoBaseMasComision.java
2  // La clase EmpleadoBaseMasComision hereda de EmpleadoPorComision y
3  // accede a los datos private de la superclase a través de los
4  // métodos public heredados.
5
6  public class EmpleadoBaseMasComision extends EmpleadoPorComision
7  {
8     private double salarioBase; // salario base por semana
9
10    // constructor con seis argumentos
11    public EmpleadoBaseMasComision( String nombre, String apellido,
12       String nss, double ventas, double tarifa, double salario )
13    {
14       super( nombre, apellido, nss, ventas, tarifa );
15       establecerSalarioBase( salario ); // valida y almacena el salario base
16    } // fin del constructor de EmpleadoBaseMasComision con seis argumentos
17
```

Fig. 9.11 | La clase `EmpleadoBaseMasComision` hereda de `EmpleadoPorComision` y accede a los datos `private` de la superclase a través de métodos `public` heredados (parte 1 de 2).

```
18    // establece el salario base
19    public void establecerSalarioBase( double salario )
20    {
21       if ( salario >= 0.0 )
22          salarioBase = salario;
23       else
24          throw new IllegalArgumentException(
25             "El salario base debe ser >= 0.0" );
26    } // fin del método establecerSalarioBase
27
28    // devuelve el salario base
29    public double obtenerSalarioBase()
30    {
31       return salarioBase;
32    } // fin del método obtenerSalarioBase
33
34    // calcula los ingresos
35    @Override // indica que este método sobrescribe el método de la superclase
36    public double ingresos()
37    {
38       return obtenerSalarioBase() + super.ingresos();
39    } // fin del método ingresos
40
41    // devuelve representación String de EmpleadoBaseMasComision
42    @Override // indica que este método sobrescribe el método de la superclase
43    public String toString()
44    {
45       return String.format( "%s %s\n%s: %.2f", "con sueldo base",
46          super.toString(), "sueldo base", obtenerSalarioBase() );
47    } // fin del método toString
48 } // fin de la clase EmpleadoBaseMasComision
```

Fig. 9.11 | La clase `EmpleadoBaseMasComision` hereda de `EmpleadoPorComision` y accede a los datos `private` de la superclase a través de métodos `public` heredados (parte 2 de 2).

El método `ingresos` *de la clase* `EmpleadoBaseMasComision`

El método `ingresos` (líneas 35 a 39) sobrescribe el método ingresos de `EmpleadoPorComision` (figura 9.10, líneas 93 a 96) para calcular los ingresos de un empleado por comisión con sueldo base. La nueva versión obtiene la porción de los ingresos del empleado, con base en la comisión solamente, mediante una llamada al método `ingresos` de `EmpleadoPorComision` con la expresión `super.ingresos()` (línea 38), y después suma el salario base a este valor para calcular los ingresos totales del empleado. Observe la sintaxis utilizada para invocar un método sobrescrito de la superclase desde una subclase: coloque la palabra clave `super` y un separador punto (`.`) antes del nombre del método de la superclase. Esta forma de invocar métodos es una buena práctica de ingeniería de software: si un método realiza todas o algunas de las acciones que necesita otro método, se hace una llamada a éste en vez de duplicar su código. Al hacer que el método `ingresos` de `EmpleadoBaseMasComision` invoque al método `ingresos` de `EmpleadoPorComision` para calcular parte de los ingresos del objeto `EmpleadoBaseMasComision`, *evitamos duplicar el código y se reducen los problemas de mantenimiento del mismo*. Si no utilizáramos "`super.`", entonces el método de `EmpleadoBaseMasComision` se *llamaría a sí mismo*, en vez de llamar a la versión de la superclase. Esto produciría un fenómeno que estudiaremos en el capítulo 18 (en la página Web del libro), conocido como *recursividad infinita*, que en un momento dado provocaría un desbordamiento de la pila de llamadas a métodos: un error fatal en tiempo de ejecución.

El método `toString` de `EmpleadoBaseMasComision`

De manera similar, el método `toString` de `EmpleadoBaseMasComision` (figura 9.11, líneas 38 a 43) sobrescribe el método `toString` de la clase `EmpleadoPorComision` (figura 9.10, líneas 91 a 99) para devolver una representación `String` apropiada para un empleado por comisión con salario base. La nueva versión crea parte de la representación `String` de un objeto `EmpleadoBaseMasComision` (es decir, el objeto `String` `"empleado por comision"` y los valores de las variables de instancia `private` de la clase `EmpleadoPorComision`), mediante una llamada al método `toString` de `EmpleadoPorComision` con la expresión `super.toString()` (figura 9.11, línea 46). Después, el método `toString` de `EmpleadoBaseMasComision` imprime en pantalla el resto de la representación `String` de un objeto `EmpleadoBaseMasComision` (es decir, el valor del salario base de la clase `EmpleadoBaseMasComision`).

Error común de programación 9.3

Cuando se sobrescribe un método de la superclase en una subclase, por lo general la versión correspondiente a la subclase llama a la versión de la superclase para que realice una parte del trabajo. Si no se antepone al nombre del método de la superclase la palabra clave `super` y el separador punto (.) al momento de llamar al método de la superclase, el método de la subclase se llama a sí mismo, creando potencialmente un error conocido como recursividad infinita. La recursividad, si se utiliza en forma correcta, es una poderosa herramienta, como veremos en el capítulo 18 (en la página Web del libro).

Prueba de la clase `EmpleadoBaseMasComision`

La clase `PruebaEmpleadoBaseMasComision` realiza la misma manipulación sobre un objeto `EmpleadoBaseMasComision` como en la figura 9.7, y produce los mismos resultados, por lo que no la mostraremos aquí. Aunque cada una de las clases `EmpleadoBaseMasComision` que hemos visto se comporta en forma idéntica, la versión de la figura 9.11 es la mejor diseñada. Al usar la herencia y llamar a los métodos que ocultan los datos y aseguran una consistencia, hemos construido de manera eficiente y efectiva una clase bien diseñada.

Resumen de los ejemplos de herencia en las secciones 9.4.1 a 9.4.5

Acabamos de ver un conjunto de ejemplos diseñados cuidadosamente para enseñar la buena ingeniería de software mediante el uso de la herencia. Aprendió a usar la palabra clave `extends` para crear una subclase mediante la herencia, a utilizar miembros `protected` de la superclase para permitir que una subclase acceda a las variables de instancia heredadas de la superclase, y cómo sobrescribir los métodos de la superclase para proporcionar versiones más apropiadas para los objetos de la subclase. Además, aprendió a aplicar las técnicas de ingeniería de software del capítulo 8 y de éste, para crear clases que sean fáciles de mantener, modificar y depurar.

9.5 Los constructores en las subclases

Como explicamos en la sección anterior, al crear una instancia de un objeto de una subclase se empieza una cadena de llamadas a los constructores, en los que el constructor de la subclase, antes de realizar sus propias tareas, invoca al constructor de su superclase directa, ya sea en forma explícita (por medio de la referencia `super`) o implícita (llamando al constructor predeterminado o sin argumentos de la superclase). De manera similar, si la superclase se deriva de otra clase (como sucede con cualquier clase, excepto `Object`), el constructor de la superclase invoca al constructor de la siguiente clase que se encuentre a un nivel más arriba en la jerarquía, y así en lo sucesivo. El último constructor que se llama en la cadena es *siempre* el de la clase `Object`. El cuerpo del constructor de la subclase original *termina* de ejecutarse al último. El constructor de cada superclase manipula las variables de instancia de la superclase que hereda el objeto de la subclase. Por ejemplo, considere de nuevo la jerarquía `EmpleadoPorComision`–`EmpleadoBaseMasComision` de las figuras 9.10 y 9.11. Cuando un programa crea un objeto `EmpleadoBaseMasComision`, se hace una llamada a su constructor. Ese constructor llama al constructor de `Empleado-`

PorComision, que a su vez llama al constructor de Object. El constructor de la clase Object tiene un cuerpo vacío, por lo que devuelve de inmediato el control al constructor de EmpleadoPorComision, el cual inicializa las variables de instancia private de EmpleadoPorComision que son parte del objeto EmpleadoBaseMasComision. Cuando el constructor EmpleadoPorComision termina de ejecutarse, devuelve el control al constructor de EmpleadoBaseMasComision, el cual inicializa el salarioBase del objeto EmpleadoBaseMasComision.

Observación de ingeniería de software 9.6

Java asegura que, aún si un constructor no asigna un valor a una variable de instancia, de todas formas se inicializa con su valor predeterminado (es decir, 0 para los tipos numéricos primitivos, false para los tipos boolean y null para las referencias).

9.6 Ingeniería de software mediante la herencia

Al extender una clase, la nueva clase hereda los miembros de la superclase, aunque sus miembros private están *ocultos* en la nueva clase. Podemos *personalizar* la nueva clase para cumplir nuestras necesidades, mediante la *inclusión de miembros adicionales* y la *sobrescritura* de miembros de la superclase. Para hacer esto, no es necesario que el programador de la subclase modifique el código fuente de la superclase; ni siquiera necesita tener acceso al mismo. Java sólo requiere el acceso al archivo .class de la superclase, para poder compilar y ejecutar cualquier programa que utilice o extienda la superclase. Esta poderosa capacidad es atractiva para los distribuidores independientes de software (ISV), quienes pueden desarrollar clases propietarias para vender o licenciar, y ponerlas a disposición de los usuarios en formato de código de bytes. Después, los usuarios pueden derivar con rapidez nuevas clases a partir de estas clases de biblioteca, sin necesidad de acceder al código fuente propietario del ISV.

Observación de ingeniería de software 9.7

Aunque al heredar de una clase no *se requiere acceso a su código fuente, los desarrolladores insisten con frecuencia en ver el código fuente para comprender cómo está implementada la clase. Los desarrolladores en la industria desean asegurarse que están extendiendo una clase sólida; por ejemplo, una clase que se desempeñe bien y que se implemente en forma tanto robusta como segura.*

Algunas veces es difícil apreciar el alcance de los problemas a los que se enfrentan los diseñadores que trabajan en proyectos de software a gran escala. Las personas experimentadas con esos proyectos dicen que la reutilización efectiva del software mejora el proceso de desarrollo del mismo. La programación orientada a objetos facilita la reutilización de software, con lo que comúnmente se obtiene una potencial reducción en el tiempo de desarrollo.

La disponibilidad de bibliotecas de clases extensas y útiles produce los máximos beneficios de la reutilización de software a través de la herencia. Las bibliotecas de clases estándar que se incluyen con Java tienden a ser de propósito general, y fomentan mucho la reutilización de software. Existen muchas otras bibliotecas de clases.

Puede ser confuso leer las declaraciones de subclases, ya que los miembros heredados no se declaran de manera explícita en las subclases, pero de todas formas están presentes en ellas. Existe un problema similar en la documentación de los miembros de las subclases.

Observación de ingeniería de software 9.8

En la etapa de diseño de un sistema orientado a objetos, es común encontrar que ciertas clases están muy relacionadas. Es conveniente "factorizar" las variables de instancia y los métodos comunes, y colocarlos en una superclase. Después hay que usar la herencia para desarrollar subclases, especializándolas con herramientas que estén más allá de las heredadas de parte de la superclase.

Observación de ingeniería de software 9.9

Declarar una subclase no afecta el código fuente de su superclase. La herencia preserva la integridad de la superclase.

Observación de ingeniería de software 9.10

Los diseñadores de sistemas orientados a objetos deben evitar la proliferación de clases. La cual crea problemas administrativos y puede obstaculizar la reutilización de software, ya que en una biblioteca de clases enorme es difícil para un cliente localizar las clases más apropiadas. La alternativa es crear menos clases que proporcionen una funcionalidad más substancial, pero dichas clases podrían volverse complejas.

9.7 La clase Object

Como vimos al principio en este capítulo, todas las clases en Java heredan, ya sea en forma directa o indirecta de la clase `Object` (paquete `java.lang`), por lo que todas las demás clases heredan sus 11 métodos (algunos de los cuales están sobrecargados). La figura 9.12 muestra un resumen de los métodos de `Object`. Hablaremos sobre varios métodos de `Object` a lo largo de este libro (como se indica en la figura 9.12).

Método	Descripción
`clone`	Este método `protected`, que no recibe argumentos y devuelve una referencia `Object`, realiza una copia del objeto en el que se llama. La implementación predeterminada de este método realiza algo que se conoce como **copia superficial**: los valores de las variables de instancia en un objeto se copian a otro objeto del mismo tipo. Para los tipos por referencia, sólo se copian las referencias. Una implementación típica del método `clone` sobrescrito sería realizar una **copia en profundidad**, que crea un nuevo objeto para cada variable de instancia de tipo por referencia. Es difícil implementar el método `clone` en forma correcta. Por esta razón, no se recomienda su uso. Muchos expertos de la industria sugieren que se utilice mejor la serialización de objetos. En el capítulo 17 (en la página Web del libro), Archivos, flujos y serialización de objetos hablaremos sobre este tema.
`equals`	Este método compara la igualdad entre dos objetos; devuelve `true` si son iguales y `false` en caso contrario. El método recibe cualquier objeto `Object` como argumento. Cuando debe compararse la igualdad entre objetos de una clase en particular, la clase debe sobrescribir el método `equals` para comparar el *contenido* de los dos objetos. Si desea ver los requerimientos para implementar este método, consulte la documentación del método en `download.oracle.com/javase/6/docs/api/java/lang/Object.html#equals (java.lang.Object)`. La implementación `equals` predeterminada utiliza el operador `==` para determinar si dos referencias *se refieren al mismo objeto* en la memoria. La sección 16.3.3 demuestra el método `equals` de la clase `String` y explica la diferencia entre comparar objetos `String` con `==` y con `equals`.
`finalize`	El recolector de basura llama a este método `protected` (que presentamos en la sección 8.10) para realizar las tareas de preparación para la terminación en un objeto, justo antes de que el recolector de basura reclame la memoria de éste. Recuerde: no se garantiza que se ejecute el método `finalize` del objeto, ni cuando se ejecutará. Por esta razón la mayoría de los programadores deberían evitar usar el método `finalize`.

Fig. 9.12 | Métodos de `Object` (parte 1 de 2).

Método	Descripción
`getClass`	Todo objeto en Java conoce su tipo en tiempo de ejecución. El método `getClass` (utilizado en las secciones 10.5, 14.5 y 24.3) devuelve un objeto de la clase `Class` (paquete `java.lang`), el cual contiene información acerca del tipo del objeto, como el nombre de su clase (devuelto por el método `getName` de `Class`).
`hashCode`	Los códigos de hash son valores `int` útiles para almacenar y obtener información en alta velocidad y en una estructura que se conoce como tabla de hash (la describiremos en la sección 20.11). Este método también se llama como parte de la implementación del método `toString` predeterminado de la clase `Object`.
`wait`, `notify`, `notifyAll`	Los métodos `notify`, `notifyAll` y las tres versiones sobrecargadas de `wait` están relacionados con la tecnología multihilo, que veremos en el capítulo 26 (en inglés, en la página Web del libro).
`toString`	Este método (presentado en la sección 9.4.1) devuelve una representación `String` de un objeto. La implementación predeterminada de este método devuelve el nombre del paquete y el nombre de la clase del objeto, seguidos por una representación hexadecimal del valor devuelto por el método `hashCode` del objeto.

Fig. 9.12 | Métodos de `Object` (parte 2 de 2).

En el capítulo 7 vimos que los arreglos son objetos. Como resultado, al igual que otros objetos, un arreglo hereda los miembros de la clase `Object`. Todo arreglo tiene un método `clone` sobrescrito, que copia el arreglo. No obstante, si el arreglo almacena referencias a objetos, éstos no se copian; se realiza una *copia superficial*.

9.8 (Opcional) Caso de estudio de GUI y gráficos: mostar texto e imágenes usando etiquetas

A menudo, los programas usan etiquetas cuando necesitan mostrar información o instrucciones al usuario, en una interfaz gráfica de usuario. Las **etiquetas** son una forma conveniente de identificar componentes de la GUI en la pantalla, y de mantener al usuario informado acerca del estado actual del programa. En Java, un objeto de la clase `JLabel` (del paquete `javax.swing`) puede mostrar texto, una imagen o ambos. El ejemplo de la figura 9.13 demuestra varias características de `JLabel`, incluyendo una etiqueta de texto simple, una etiqueta de imagen y una con texto e imagen.

Las líneas 3 a 6 importan las clases que necesitamos para mostrar los objetos `JLabel`. `BorderLayout` del paquete `java.awt` contiene constantes que especifican en dónde podemos colocar componentes de GUI en el objeto `JFrame`. La clase `ImageIcon` representa una imagen que puede mostrarse en un `JLabel`, y la clase `JFrame` representa la ventana que contiene todas las etiquetas.

```
1  // Fig 9.13: DemoLabel.java
2  // Demuestra el uso de etiquetas.
3  import java.awt.BorderLayout;
4  import javax.swing.ImageIcon;
5  import javax.swing.JLabel;
6  import javax.swing.JFrame;
7
```

Fig. 9.13 | `JLabel` con texto y con imágenes (parte 1 de 2).

```
8  public class DemoLabel
9  {
10    public static void main( String[] args )
11    {
12       // Crea una etiqueta con texto solamente
13       JLabel etiquetaNorte = new JLabel( "Norte" );
14
15       // crea un icono a partir de una imagen, para poder colocarla en un objeto JLabel
16       ImageIcon etiquetaIcono = new ImageIcon( "GUItip.gif" );
17
18       // crea una etiqueta con un icono en vez de texto
19       JLabel etiquetaCentro = new JLabel( etiquetaIcono );
20
21       // crea otra etiqueta con un icono
22       JLabel etiquetaSur = new JLabel( etiquetaIcono );
23
24       // establece la etiqueta para mostrar texto (así como un icono)
25       etiquetaSur.setText( "Sur" );
26
27       // crea un marco para contener las etiquetas
28       JFrame aplicacion = new JFrame();
29
30       aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
31
32       // agrega las etiquetas al marco; el segundo argumento especifica
33       // en qué parte del marco se va a agregar la etiqueta
34       aplicacion.add( etiquetaNorte, BorderLayout.NORTH );
35       aplicacion.add( etiquetaCentro, BorderLayout.CENTER );
36       aplicacion.add( etiquetaSur, BorderLayout.SOUTH );
37
38       aplicacion.setSize( 300, 300 ); // establece el tamaño del marco
39       aplicacion.setVisible( true ); // muestra el marco
40    } // fin de main
41 } // fin de la clase DemoLabel
```

Fig. 9.13 | `JLabel` con texto y con imágenes (parte 2 de 2).

La línea 13 crea un objeto `JLabel` que muestra el argumento de su constructor: la cadena `"Norte"`. La línea 16 declara la variable local `etiquetaIcono` y le asigna un nuevo objeto `ImageIcon`. El construc-

tor para `ImageIcon` recibe un objeto `String` que especifica la ruta del archivo de la imagen. Como sólo especificamos un nombre de archivo, Java supone que se encuentra en el mismo directorio que la clase `DemoLabel`. `ImageIcon` puede cargar imágenes en los formatos GIF, JPEG y PNG. La línea 19 declara e inicializa la variable local `etiquetaCentro` con un objeto `JLabel` que muestra el objeto `etiquetaIcono`. La línea 22 declara e inicializa la variable local `etiquetaSur` con un objeto `JLabel` similar al de la línea 19. Sin embargo, la línea 25 llama al método **`setText`** para modificar el texto que muestra la etiqueta. El método `setText` puede llamarse en cualquier objeto `JLabel` para modificar su texto. Este objeto `JLabel` muestra tanto el icono como el texto.

La línea 28 crea el objeto `JFrame` que muestra a los objetos `JLabel`, y la línea 30 indica que el programa debe terminar cuando se cierre el objeto `JFrame`. Para adjuntar las etiquetas al objeto `JFrame` en las líneas 34 a 36, llamamos a una versión sobrecargada del método `add` que recibe dos parámetros. El primero es el componente que deseamos adjuntar, y el segundo es la región en la que debe colocarse. Cada objeto `JFrame` tiene un **esquema** asociado, que ayuda al `JFrame` a posicionar los componentes de la GUI que tiene adjuntos. El esquema predeterminado para un objeto `JFrame` se conoce como **`BorderLayout`**, y tiene cinco regiones: `NORTH` (superior), `SOUTH` (inferior), `EAST` (lado derecho), `WEST` (lado izquierdo) y `CENTER` (centro). Cada una de estas regiones se declara como una constante en la clase `BorderLayout`. Al llamar al método `add` con un argumento, el objeto `JFrame` coloca el componente en la región `CENTER` de manera automática. Si una posición ya contiene un componente, entonces el nuevo toma su lugar. Las líneas 38 y 39 establecen el tamaño del objeto `JFrame` y lo hacen visible en pantalla.

Ejercicio del ejemplo práctico de GUI y gráficos

9.1 Modifique el ejercicio 8.1 del Caso de estudio de GUI y gráficos para incluir un objeto `JLabel` como barra de estado, que muestre las cuentas que representan el número de cada figura mostrada. La clase `PanelDibujo` debe declarar un método para devolver un objeto `String` que contenga el texto de estado. En `main`, primero cree el objeto `PanelDibujo`, y después el objeto `JLabel` con el texto de estado como argumento para el constructor de `JLabel`. Adjunte el objeto `JLabel` a la región `SOUTH` del objeto `JFrame`, como se muestra en la figura 9.14.

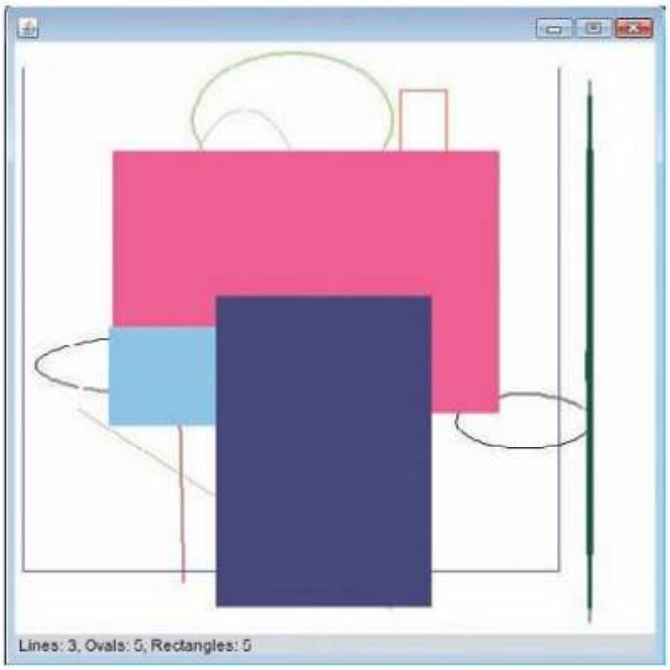


Fig. 9.14 | Objeto `JLabel` que muestra las estadísticas de las figuras.

9.9 Conclusión

En este capítulo se introdujo el concepto de la herencia: la habilidad de crear clases mediante la absorción de los miembros de una clase existente, mejorándolos con nuevas capacidades. Usted aprendió las nociones de las superclases y las subclases, y utilizó la palabra clave `extends` para crear una subclase que hereda miembros de una superclase. Le mostramos cómo usar la anotación `@Override` para evitar la sobrecarga accidental, al indicar que un método sobrescribe al método de una superclase. En este capítulo se introdujo también el modificador de acceso `protected`; los métodos de la subclase pueden acceder a los miembros `protected` de la superclase. Aprendió también cómo acceder a los miembros de la superclase mediante `super`. Vio además cómo se utilizan los constructores en las jerarquías de herencia. Por último, aprendió acerca de los métodos de la clase `Object`, la superclase directa o indirecta de todas las clases en Java.

En el capítulo 10, Programación orientada a objetos: polimorfismo, continuaremos con nuestra discusión sobre la herencia al introducir el polimorfismo: un concepto orientado a objetos que nos permite escribir programas que puedan manipular convenientemente, de una forma más general, objetos de una amplia variedad de clases relacionadas por la herencia. Después de estudiar el capítulo 10, estará familiarizado con las clases, los objetos, el encapsulamiento, la herencia y el polimorfismo: las tecnologías clave de la programación orientada a objetos.

Resumen

Sección 9.1 Introducción

- La herencia (pág. 360) reduce el tiempo de desarrollo de los programas.
- La superclase directa (pág. 360) de una subclase (que se especifica mediante la palabra `extends` en la primera línea de una declaración de clase) es la superclase a partir de la cual hereda la subclase. Una superclase indirecta (pág. 360) de una subclase se encuentra dos o más niveles arriba de esa subclase en la jerarquía de clases.
- En la herencia simple (pág. 360), una clase se deriva de una superclase directa. En la herencia múltiple, una clase se deriva de más de una superclase directa. Java no soporta la herencia múltiple.
- Una subclase es más específica que su superclase, y representa un grupo más pequeño de objetos (pág. 360).
- Cada objeto de una subclase es también un objeto de la superclase de esa clase. Sin embargo, el objeto de una superclase no es el de las subclases de su clase.
- Una relación *es un* (pág. 361) representa a la herencia. En una relación *es un*, un objeto de una subclase también puede tratarse como un objeto de su superclase.
- Una relación *tiene un* (pág. 361) representa a la composición. En una relación *tiene un*, el objeto de una clase contiene referencias a objetos de otras clases.

Sección 9.2 Superclases y subclases

- Las relaciones de herencia simple forman estructuras jerárquicas tipo árbol; una superclase existe en una relación jerárquica con sus subclases.

Sección 9.3 Miembros `protected`

- Los miembros `public` de una superclase son accesibles en cualquier parte en donde el programa tenga una referencia a un objeto de esa superclase, o de una de sus subclases.
- Los miembros `private` de una superclase son accesibles sólo dentro de la declaración de esa superclase.
- Los miembros `protected` de una superclase (pág. 363) tienen un nivel intermedio de protección entre acceso `public` y `private`. Pueden ser utilizados por los miembros de la superclase, los de sus subclases y los de otras clases en el mismo paquete.

- Los miembros `private` de una superclase están ocultos en sus subclases y sólo se puede acceder a ellos a través de los métodos `public` o `private` heredados de la superclase.
- Un método sobrescrito de una superclase se puede utilizar desde la subclase, si se antepone al nombre del método de la subclase la palabra clave `super` (pág. 363) y un separador punto (`.`).

Sección 9.4 Relación entre las superclases y las subclases

- Una subclase no puede acceder o heredar los miembros `private` de su superclase, pero puede acceder a los miembros no `private`.
- Una subclase puede invocar a un constructor de su superclase mediante el uso de la palabra clave `super`, seguida de un conjunto de paréntesis que contienen los argumentos del constructor de la superclase. Esto debe aparecer como la primera instrucción en el cuerpo del constructor de la subclase.
- El método de una superclase puede sobrescribirse en una subclase para declarar una implementación apropiada para la subclase.
- La anotación `@Override` (pág. 368) indica que un método debe sobrescribir al método de una superclase. Cuando el compilador encuentra un método declarado con `@Override`, compara la firma del método con las firmas del método de la superclase. Si no hay una coincidencia exacta, el compilador emite un mensaje de error, como "el método no sobrescribe o implementa un método a partir de un supertipo".
- El método `toString` no recibe argumentos y devuelve un objeto `String`. Por lo general, una subclase sobrescribe el método `toString` de la clase `Object`.
- Cuando se imprime un objeto usando el especificador de formato `%s`, se hace una llamada implícita al método `toString` del objeto para obtener su representación `String`.

Sección 9.5 Constructores en las subclases

- La primera tarea de cualquier constructor de subclase es llamar al constructor de su superclase directa (pág. 377), para asegurar que se inicialicen las variables de instancia heredadas de la superclase.

Sección 9.6 Ingeniería de software mediante la herencia

- Declarar variables de instancia `private`, al mismo tiempo que se proporcionan métodos no `private` para manipular y realizar la validación, ayuda a cumplir con la buena ingeniería de software.

Sección 9.7 La clase `Object`

- Consulte la tabla de los métodos de la clase `Object` en la figura 9.12.

Ejercicios de autoevaluación

9.1 Complete las siguientes oraciones:

a) _____ es una forma de reutilización de software, en la que nuevas clases adquieren los miembros de las clases existentes, y las mejoran con nuevas capacidades.

b) Los miembros _____ de una superclase pueden utilizarse en la declaración de la superclase *y en* las declaraciones de las subclases.

c) En una relación _____ , un objeto de una subclase puede ser tratado también como un objeto de su superclase.

d) En una relación _____ , el objeto de una clase tiene referencias a objetos de otras clases como miembros.

e) En la herencia simple, una clase existe en una relación _____ con sus subclases.

f) Los miembros _____ de una superclase son accesibles en cualquier parte en donde el programa tenga una referencia a un objeto de esa superclase, o a un objeto de una de sus subclases.

g) Cuando se crea la instancia de un objeto de una subclase, el _____ de una superclase se llama en forma implícita o explícita.

h) Los constructores de una subclase pueden llamar a los constructores de la superclase mediante la palabra clave _____ .

9.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Los constructores de la superclase no son heredados por las subclases.
b) Una relación *tiene un* se implementa mediante la herencia.
c) Una clase `Auto` tiene una relación *es un* con las clases `VolanteDireccion` y `Frenos`.
d) Cuando una subclase redefine al método de una superclase utilizando la misma firma, se dice que la subclase sobrecarga a ese método de la superclase.

Respuestas a los ejercicios de autoevaluación

9.1 a) Herencia. b) `public` y `protected`. c) *es un* o de herencia. d) *tiene un*, o composición. e) jerárquica. f) `public`. g) constructor. h) `super`.

9.2 a) Verdadero. b) Falso. Una relación *tiene un* se implementa mediante la composición. Una relación *es un* se implementa mediante la herencia. c) Falso. Éste es un ejemplo de una relación *tiene un*. La clase `Auto` tiene una relación *es un* con la clase `Vehiculo`. d) Falso. Esto se conoce como sobrescritura, no sobrecarga; un método sobrecargado tiene el mismo nombre, pero una firma distinta.

Ejercicios

9.3 Muchos programas escritos con herencia podrían escribirse mediante la composición, y viceversa. Vuelva a escribir la clase `EmpleadoBaseMasComision` (figura 9.11) de la jerarquía `EmpleadoPorComision-EmpleadoBaseMasComision` para usar la composición en vez de la herencia.

9.4 Describa las formas en las que la herencia fomenta la reutilización de software, ahorra tiempo durante el desarrollo de los programas y ayuda a prevenir errores.

9.5 Dibuje una jerarquía de herencia para los estudiantes en una universidad, de manera similar a la jerarquía que se muestra en la figura 9.2. Use a `Estudiante` como la superclase de la jerarquía, y después extienda `Estudiante` con las clases `EstudianteNoGraduado` y `EstudianteGraduado`. Siga extendiendo la jerarquía con el mayor número de niveles que sea posible. Por ejemplo, `EstudiantePrimerAnio`, `EstudianteSegundoAnio`, `EstudianteTercerAnio` y `EstudianteCuartoAnio` podrían extender a `EstudianteNoGraduado`, y `EstudianteDoctorado` y `EstudianteMaestria` podrían ser subclases de `EstudianteGraduado`. Después de dibujar la jerarquía, hable sobre las relaciones que existen entre las clases. [*Nota:* no necesita escribir código para este ejercicio].

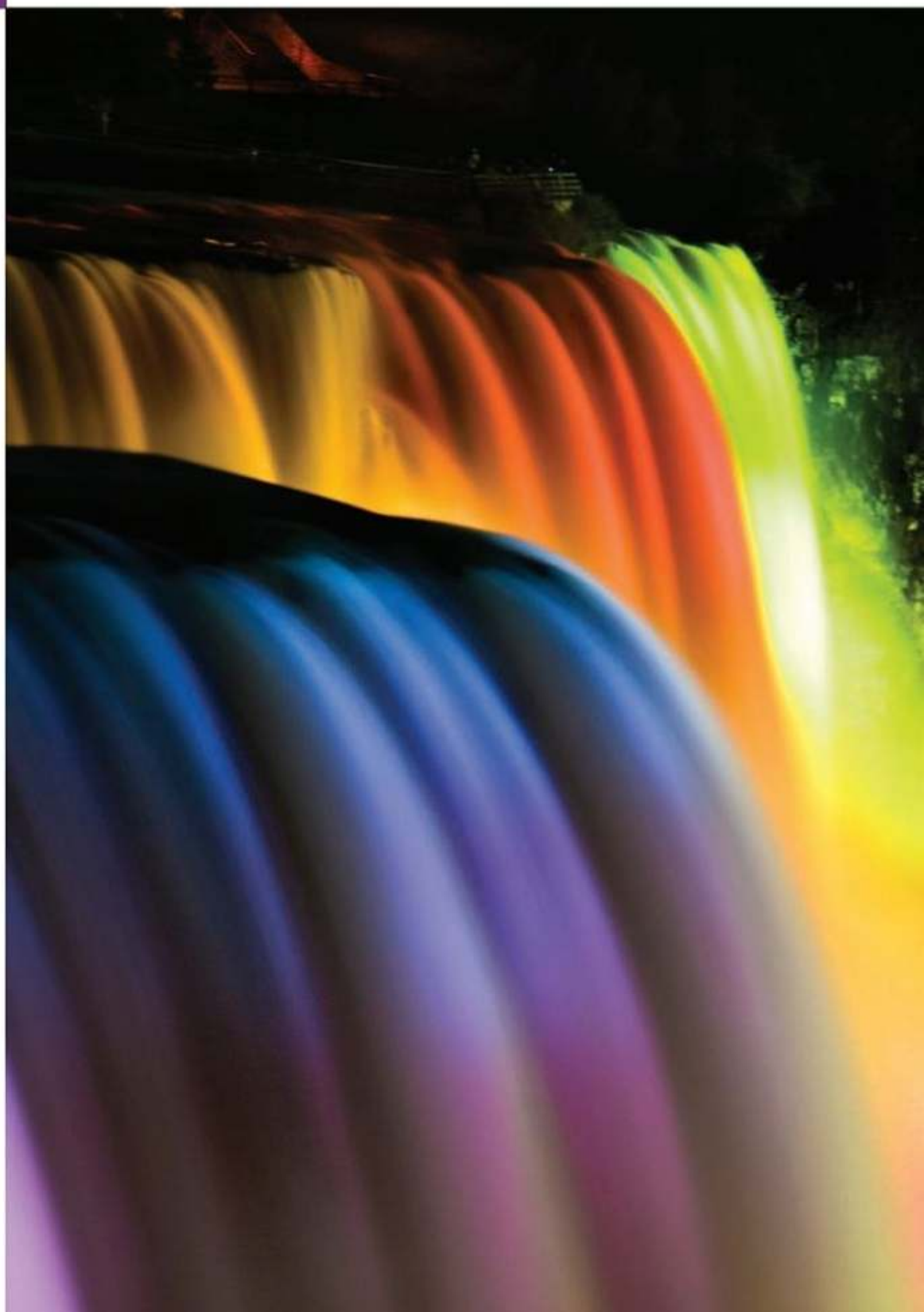
9.6 El mundo de las figuras es más extenso que las figuras incluidas en la jerarquía de herencia de la figura 9.3. Anote todas las figuras en las que pueda pensar (tanto bidimensionales como tridimensionales) e intégredas en una jerarquía `Figura` más completa, con todos los niveles que sea posible. Su jerarquía debe tener la clase `Figura` en la parte superior. Las clases `FiguraBidimensional` y `FiguraTridimensional` deben extender a `Figura`. Agregue subclases adicionales, como `Cuadrilatero` y `Esfera`, en sus ubicaciones correctas en la jerarquía, según sea necesario.

9.7 Algunos programadores prefieren no utilizar el acceso `protected`, pues piensan que quebranta el encapsulamiento de la superclase. Hable sobre los méritos relativos de utilizar el acceso `protected`, en comparación con el acceso `private` en las superclases.

9.8 Escriba una jerarquía de herencia para las clases `Cuadrilatero`, `Trapezoide`, `Paralelogramo`, `Rectangulo` y `Cuadrado`. Use `Cuadrilatero` como la superclase de la jerarquía. Cree y use una clase `Punto` para representar los puntos en cada figura. Agregue todos los niveles que sea posible a la jerarquía. Especifique las variables de instancia y los métodos para cada clase. Las variables de instancia `private` de `Cuadrilatero` deben ser los pares de coordenadas *x*-*y* para los cuatro puntos finales del `Cuadrilatero`. Escriba un programa que cree instancias de objetos de sus clases, y que imprima el área de cada objeto (excepto `Cuadrilatero`).

10

Programación orientada a objetos: polimorfismo



Un anillo para gobernarlos a todos, un anillo para encontrarlos, un anillo para traerlos a todos y en la oscuridad enlazarlos.
—John Ronald Reuel Tolkien

Las proposiciones generales no deciden casos concretos.
—Oliver Wendell Holmes

Un filósofo de imponente estatura no piensa en un vacío. Incluso sus ideas más abstractas son, en cierta medida, condicionadas por lo que se conoce o no en el tiempo en que vive.
—Alfred North Whitehead

¿Por qué, alma mía, desfalleces y te agitas por mí?
—Salmos 42:5

Objetivos

En este capítulo aprenderá a:

- Comprender el concepto de polimorfismo.
- Utilizar métodos sobrescritos para llevar a cabo el polimorfismo.
- Distinguir entre clases abstractas y concretas.
- Declarar métodos abstractos para crear clases abstractas.
- Apreciar la manera en que el polimorfismo hace que los sistemas puedan extenderse y mantenerse.
- Determinar el tipo de un objeto en tiempo de ejecución.
- Declarar e implementar interfaces.

Plan general

10.1 Introducción

Ahora continuaremos nuestro estudio de la programación orientada a objetos, en donde explicaremos y demostraremos el **polimorfismo** con las jerarquías de herencia. El polimorfismo nos permite "programar en forma general", en vez de "programar en forma específica." En particular, nos permite escribir programas que procesen objetos que compartan la misma superclase (ya sea de manera directa o indirecta) como si todos fueran objetos de la superclase; esto puede simplificar la programación.

Considere el siguiente ejemplo de polimorfismo. Suponga que crearemos un programa que simula el movimiento de varios tipos de animales para un estudio biológico. Las clases `Pez`, `Rana` y `Ave` representan los tipos de animales que se están investigando. Imagine que cada una de estas clases extiende a la superclase `Animal`, la cual contiene un método llamado `mover` y mantiene la posición actual de un animal, en forma de coordenadas *x-y*. Cada subclase implementa el método `mover`. Nuestro programa mantiene un arreglo tipo `Animal`, de referencias a objetos de las diversas subclases de `Animal`. Para simular los movimientos de los animales, el programa envía a cada objeto el *mismo* mensaje una vez por segundo; `mover`. Cada tipo específico de `Animal` responde a un mensaje `mover` de manera única; un `Pez` podría nadar tres pies, una `Rana`, saltar cinco pies y un `Ave`, volar diez pies. Cada objeto sabe cómo modificar sus coordenadas *x-y* en forma apropiada para su tipo *específico* de movimiento. Confiar en que cada objeto sepa cómo "hacer lo correcto" (es decir, lo que sea apropiado para ese tipo de objeto) en respuesta a la llamada al mismo método es el concepto clave del polimorfismo. El mismo mensaje (en este caso, `mover`) que se envía a una variedad de objetos tiene "muchas formas" de resultados; de aquí que se utilice el término polimorfismo.

Implementación para la extensibilidad

Con el polimorfismo podemos diseñar e implementar sistemas que puedan extenderse con facilidad; pueden agregarse nuevas clases con sólo modificar un poco (o nada) las porciones generales del programa,

siempre y cuando las nuevas clases sean parte de la jerarquía de herencia que el programa procesa en forma genérica. Las únicas partes de un programa que deben alterarse son las que requieren un conocimiento directo de las nuevas clases que agregamos a la jerarquía. Por ejemplo, si extendemos la clase `Animal` para crear la clase `Tortuga` (que podría responder a un mensaje mover caminando una pulgada), necesitamos escribir sólo la clase `Tortuga` y la parte de la simulación que crea una instancia de un objeto `Tortuga`. Las porciones de la simulación que indican a cada `Animal` que se mueva en forma genérica pueden permanecer iguales.

Generalidades del capítulo

Primero hablaremos sobre los ejemplos comunes del polimorfismo. Después proporcionaremos un ejemplo que demuestra el comportamiento polimórfico. Utilizaremos referencias a la superclase para manipular tanto a los objetos de la superclase como a los de las subclases mediante el polimorfismo.

Después presentaremos un ejemplo práctico en el que utilizaremos de nuevo la jerarquía de empleados de la sección 9.4.5. Desarrollaremos una aplicación simple de nómina que calcula mediante el polimorfismo el salario semanal de varios tipos distintos de empleados, con el método ingresos de cada trabajador. Aunque los ingresos de cada tipo de empleado se calculan de una manera específica, el polimorfismo nos permite procesar a los empleados "en general". En el caso de estudio ampliaremos la jerarquía para incluir dos nuevas clases: `EmpleadoAsalariado` (para las personas que reciben un salario semanal fijo) y `EmpleadoPorHoras` (para las personas que reciben un salario por horas y "tiempo y medio" por el tiempo extra). Declararemos un conjunto común de funcionalidad para todas las clases en la jerarquía actualizada en una clase "abstracta" llamada `Empleado`, a partir de la cual las clases "concretas" `EmpleadoAsalariado`, `EmpleadoPorHoras` y `EmpleadoPorComision` heredan en forma directa, y la clase "concreta" `EmpleadoBaseMasComision` hereda en forma indirecta. Como pronto verá, *al invocar el método ingresos de cada empleado desde una referencia a la superclase* `Empleado`, *se realiza el cálculo correcto de los ingresos* gracias a las capacidades polimórficas de Java.

Programación en forma específica

Algunas veces, cuando se lleva a cabo el procesamiento polimórfico, es necesario programar "en forma específica". Nuestro caso de estudio con `Empleado` demuestra que un programa puede determinar el tipo de un objeto en *tiempo de ejecución*, y actuar sobre él de manera acorde. En el caso de estudio decidimos que los empleados del tipo `EmpleadoBaseMasComision` deberían recibir aumentos del 10% en su salario base. Por lo tanto, usamos estas herramientas para determinar si un objeto empleado específico *es un* `EmpleadoBaseMasComision`. Si es así, incrementamos el salario base de ese empleado en un 10%.

Interfaces

El capítulo continúa con una introducción a las interfaces en Java. Una interfaz describe a un conjunto de métodos que pueden llamarse en un objeto, pero no proporciona implementaciones concretas para todos ellos. Podemos declarar clases que **implementen** a (es decir, que proporcionen implementaciones concretas para los métodos de) una o más interfaces. Cada método de una interfaz debe declararse en todas las clases que implementen a la interfaz. Una vez que una clase implementa a una interfaz, todos los objetos de esa clase tienen una relación *es un* con el tipo de la interfaz, y se garantiza que todos los objetos de la clase proporcionarán la funcionalidad descrita por la interfaz. Esto se aplica también para todas las subclases de esa clase.

En especial, las interfaces son útiles para asignar la funcionalidad común a clases que posiblemente no estén relacionadas. Esto permite que los objetos de clases no relacionadas se procesen en forma polimórfica; los objetos de las clases que implementan la misma interfaz pueden responder a todas las llamadas a los métodos de la interfaz. Para demostrar la creación y el uso de interfaces, modificaremos

nuestra aplicación de nómina para crear una aplicación general de cuentas por pagar, que puede calcular los pagos vencidos para los empleados de la compañía y los montos de las facturas a pagar por los bienes comprados. Como verá, las interfaces permiten capacidades polimórficas similares a las que permite la herencia.

10.2 Ejemplos del polimorfismo

Ahora consideraremos diversos ejemplos adicionales del polimorfismo.

Cuadriláteros

Si la clase `Rectangulo` se deriva de la clase `Cuadrilatero`, entonces un objeto `Rectangulo` es una versión más específica de un objeto `Cuadrilatero`. Cualquier operación (por ejemplo, calcular el perímetro o el área) que pueda realizarse en un objeto `Cuadrilatero` también puede realizarse en un objeto `Rectangulo`. Estas operaciones también pueden realizarse en otros objetos `Cuadrilatero`, como `Cuadrado`, `Paralelogramo` y `Trapezoide`. El polimorfismo ocurre cuando un programa invoca a un método a través de una variable de la superclase `Cuadrilatero`; en tiempo de ejecución, se hace una llamada a la versión correcta del método de la subclase, con base en el tipo de la referencia almacenada en la variable de la superclase. En la sección 10.3 veremos un ejemplo de código simple, en el cual se ilustra este proceso.

Objetos espaciales en un videojuego

Suponga que diseñaremos un videojuego que manipule objetos de las clases `Marciano`, `Venusino`, `Plutoniano`, `NaveEspacial` y `RayoLaser`. Imagine que cada clase hereda de la superclase común llamada `ObjetoEspacial`, la cual contiene el método `dibujar`. Cada subclase implementa a este método. Un programa administrador de la pantalla mantiene una colección (por ejemplo, un arreglo `ObjetoEspacial`) de referencias a objetos de las diversas clases. Para refrescar la pantalla, el administrador de pantalla envía en forma periódica el mismo mensaje a cada objeto; `dibujar`. No obstante, cada uno responde de una manera única, con base en su clase. Por ejemplo, un objeto `Marciano` podría dibujarse a sí mismo en color rojo, con ojos verdes y el número apropiado de antenas. Un objeto `NaveEspacial` podría dibujarse como un platillo volador de color plata brillante. Un objeto `RayoLaser` podría dibujarse como un rayo color rojo brillante a lo largo de la pantalla. De nuevo, el *mismo* mensaje (en este caso, dibujar) que se envía a una variedad de objetos tiene "muchas formas" de resultados.

Un administrador de pantalla podría utilizar el polimorfismo para facilitar el proceso de agregar nuevas clases a un sistema, con el menor número de modificaciones al código del mismo. Suponga que deseamos agregar objetos `Mercuriano` a nuestro videojuego. Para ello, debemos crear una clase `Mercuriano` que extienda a `ObjetoEspacial` y proporcione su propia implementación del método `dibujar`. Cuando aparezcan objetos de la clase `Mercuriano` en la colección `ObjetoEspacial`, el código del administrador de pantalla *invocará al método `dibujar`, de la misma forma que para cualquier otro objeto en la colección, sin importar su tipo.* Por lo tanto, los nuevos objetos `Mercuriano` simplemente se "integran" al videojuego sin necesidad de que el programador modifique el código del administrador de pantalla. Así, sin modificar el sistema (más que para crear nuevas clases y modificar el código que genera nuevos objetos), es posible utilizar el polimorfismo para incluir de manera conveniente tipos adicionales que no se hayan considerado a la hora de crear el sistema.

Observación de ingeniería de software 10.1

El polimorfismo nos permite tratar con las generalidades y dejar que el entorno en tiempo de ejecución se encargue de los detalles específicos. Podemos ordenar a los objetos que se comporten en formas apropiadas para ellos, sin necesidad de conocer los tipos de los objetos (siempre y cuando éstos pertenezcan a la misma jerarquía de herencia).

Observación de ingeniería de software 10.2

El polimorfismo promueve la extensibilidad: el software que invoca el comportamiento polimórfico es independiente de los tipos de los objetos a los cuales se envían los mensajes. Es posible incorporar en un sistema nuevos tipos de objetos que puedan responder a las llamadas de los métodos existentes, sin necesidad de modificar el sistema base. Sólo el código cliente que crea instancias de los nuevos objetos debe modificarse para dar cabida a los nuevos tipos.

10.3 Demostración del comportamiento polimórfico

En la sección 9.4 creamos una jerarquía de clases, en la cual la clase `EmpleadoBaseMasComision` heredó de la clase `EmpleadoPorComision`. Los ejemplos en esa sección manipularon objetos `EmpleadoPorComision` y `EmpleadoBaseMasComision` mediante el uso de referencias a ellos para invocar a sus métodos; dirigimos las referencias a la superclase a los objetos de la superclase, y las referencias a la subclase a los objetos de la subclase. Estas asignaciones son naturales y directas; las variables de la superclase están *diseñadas* para referirse a objetos de la superclase, y las variables de la subclase están *diseñadas* para referirse a objetos de la subclase. No obstante, como veremos pronto, es posible realizar otras asignaciones.

En el siguiente ejemplo, dirigiremos una referencia a la *superclase* a un objeto de la *subclase*. Después mostraremos cómo al invocar un método en un objeto de la subclase a través de una referencia a la superclase se invoca a la funcionalidad de la *subclase*; el tipo del *objeto referenciado*, y no el tipo de la *variable*, es el que determina cuál método se llamará. Este ejemplo demuestra que *un objeto de una subclase puede tratarse como un objeto de su superclase*, lo cual permite varias manipulaciones interesantes. Un programa puede crear un arreglo de variables de la superclase, que se refieran a objetos de muchos tipos de subclases. Esto se permite, ya que cada objeto de una subclase *es un* objeto de su superclase. Por ejemplo, podemos asignar la referencia de un objeto `EmpleadoBaseMasComision` a una variable de la superclase `EmpleadoPorComision`, ya que un `EmpleadoBaseMasComision` *es un* `EmpleadoPorComision`; por lo tanto, podemos tratar a un `EmpleadoBaseMasComision` como un `EmpleadoPorComision`.

Como veremos más adelante en este capítulo, *no podemos tratar a un objeto de la superclase como un objeto* de cualquiera de sus subclases, porque un objeto superclase *no* es un objeto de ninguna de sus subclases. Por ejemplo, no podemos asignar la referencia de un objeto `EmpleadoPorComision` a una variable de la subclase `EmpleadoBaseMasComision`, ya que un `EmpleadoPorComision` *no* es un `EmpleadoBaseMasComision`, *no* tiene una variable de instancia `salarioBase` y *no* tiene los métodos `establecerSalarioBase` y `obtenerSalarioBase`. La relación *es un* se aplica sólo *hacia arriba por la jerarquía*, de una subclase a sus superclases directas (e indirectas), pero *no* viceversa (es decir, no hacia debajo de la jerarquía, desde una superclase hacia sus subclases).

El compilador de Java *permite* asignar una referencia a la superclase a una variable de la subclase, si convertimos explícitamente la referencia a la superclase al tipo de la subclase; una técnica que veremos con más detalle en la sección 10.5. ¿Para qué nos serviría, en un momento dado, realizar una asignación así? Una referencia a la superclase puede usarse para invocar sólo a los métodos declarados en la superclase; si tratamos de invocar métodos que sólo pertenezcan a la subclase, a través de una referencia a la superclase, se producen errores de compilación. Si un programa necesita realizar una operación específica para la subclase en un objeto de la subclase al que se haga una referencia mediante una variable de la superclase, el programa primero debe convertir la referencia a la superclase en una referencia a la subclase, mediante una técnica conocida como **conversión descendente**. Esto permite al programa invocar métodos de la subclase que no se encuentren en la superclase. En la sección 10.5 presentaremos un ejemplo concreto de conversión descendente.

El ejemplo de la figura 10.1 demuestra tres formas de usar variables de la superclase y la subclase para almacenar referencias a objetos de la superclase y de la subclase. Las primeras dos formas son

simples: al igual que en la sección 9.4, asignamos una referencia a la superclase a una variable de la superclase, y asignamos una referencia a la subclase a una variable de la subclase. Después demostramos la relación entre las subclases y las superclases (es decir, la relación *es un*) mediante la asignación de una referencia a la subclase a una variable de la superclase. Este programa utiliza las clases `EmpleadoPorComision` y `EmpleadoBaseMasComision` de las figuras 9.10 y 9.11, respectivamente.

```
1  // Fig. 10.1: PruebaPolimorfismo.java
2  // Asignación de referencias a la superclase y la subclase, a
3  // variables de la superclase y la subclase.
4
5  public class PruebaPolimorfismo
6  {
7     public static void main( String[] args )
8     {
9        // asigna la referencia a la superclase a una variable de la superclase
10       EmpleadoPorComision empleadoPorComision = new EmpleadoPorComision(
11          "Sue", "Jones", "222-22-2222", 10000, .06 );
12
13       // asigna la referencia a la subclase a una variable de la subclase
14       EmpleadoBaseMasComision empleadoBaseMasComision =
15          new EmpleadoBaseMasComision(
16          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
17
18       // invoca a toString en un objeto de la superclase, usando una variable de la
            superclase
19       System.out.printf( "%s %s:\n\n%s\n\n",
20          "Llamada a toString de EmpleadoPorComision con referencia de superclase ",
21          "a un objeto de la superclase", empleadoPorComision.toString() );
22
23       // invoca a toString en un objeto de la subclase, usando una variable de la
            subclase
24       System.out.printf( "%s %s:\n\n%s\n\n",
25          "Llamada a toString de EmpleadoBaseMasComision con referencia",
26          "de subclase a un objeto de la subclase",
27          empleadoBaseMasComision.toString() );
28
29       // invoca a toString en un objeto de la subclase, usando una variable de la
            superclase
30       EmpleadoPorComision empleadoPorComision2 =
31          empleadoBaseMasComision;
32       System.out.printf( "%s %s:\n\n%s\n",
33          "Llamada a toString de EmpleadoBaseMasComision con referencia de superclase",
34          "a un objeto de la subclase", empleadoPorComision2.toString() );
35    } // fin de main
36 } // fin de la clase PruebaPolimorfismo
```

```
Llamada a toString de EmpleadoPorComision con referencia de superclase a un objeto de la
superclase:

empleado por comision: Sue Jones
numero de seguro social: 222-22-2222
ventas brutas: 10000.00
tarifa de comision: 0.06
```

Fig. 10.1 | Asignación de referencias de superclase y subclase a variables de superclase y subclase (parte 1 de 2).

```
Llamada a toString de EmpleadoBaseMasComision con referencia de subclase a un objeto de
la subclase:

con sueldo base empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
sueldo base: 300.00

Llamada a toString de EmpleadoBaseMasComision con referencia de superclase a un objeto
de la subclase:

con sueldo base empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
sueldo base: 300.00
```

Fig. 10.1 | Asignación de referencias de superclase y subclase a variables de superclase y subclase (parte 2 de 2).

En la figura 10.1, las líneas 10 y 11 crean un objeto `EmpleadoPorComision` y asignan su referencia a una variable `EmpleadoPorComision`. Las líneas 14 a 16 crean un objeto `EmpleadoBaseMasComision` y asignan su referencia a una variable `EmpleadoBaseMasComision`. Estas asignaciones son naturales; por ejemplo, el principal propósito de una variable `EmpleadoPorComision` es guardar una referencia a un objeto `EmpleadoPorComision`. Las líneas 19 a 21 utilizan `empleadoPorComision` para invocar a `toString` en forma explícita. Como `empleadoPorComision` hace referencia a un objeto `EmpleadoPorComision`, se hace una llamada a la versión de `toString` de la superclase `EmpleadoPorComision`. De manera similar, las líneas 24 a 27 utilizan a `empleadoBaseMasComision` para invocar a `toString` de forma explícita en el objeto `EmpleadoBaseMasComision`. Esto invoca a la versión de `toString` de la subclase `EmpleadoBaseMasComision`.

Después, las líneas 30 y 31 asignan la referencia al objeto `empleadoBaseMasComision` de la subclase a una variable de la superclase `EmpleadoPorComision`, que las líneas 32 a 34 utilizan para invocar al método `toString`. *Cuando la variable de una superclase contiene una referencia a un objeto de la subclase, y esa referencia se utiliza para llamar a un método, se hace una llamada a la versión del método de la subclase.* Por ende, `empleadoPorComision2.ToString()` en la línea 34 en realidad llama al método `toString` de la clase `EmpleadoBaseMasComision`. El compilador de Java permite este "cruzamiento", ya que un objeto de una subclase *es un* objeto de su superclase (pero no viceversa). Cuando el compilador encuentra una llamada a un método que se realiza a través de una variable, determina si el método puede llamarse verificando el tipo de clase de la variable. Si esa clase contiene la declaración del método apropiada (o hereda una), se compila la llamada. En tiempo de ejecución, el tipo del objeto al cual se refiere la variable es el que determina el método que se utilizará. En la sección 10.5 analizaremos con detalle este proceso, conocido como *vinculación dinámica*.

10.4 Clases y métodos abstractos

Cuando pensamos en un tipo de clase, asumimos que los programas crearán objetos de ese tipo. En algunos casos es conveniente declarar clases (conocidas como **clase abstractas**) para las cuales el programador *nunca* creará instancias de objetos. Puesto que sólo se utilizan como superclases en jerarquías de herencia, nos referimos a ellas como **superclases abstractas**. Estas clases no pueden utilizarse para

instanciar objetos, ya que como veremos pronto, las clases abstractas están *incompletas*. Las subclases deben declarar las "piezas faltantes" para convertirse en clases "concretas", a partir de las cuales podemos instanciar objetos. De lo contrario, estas subclases también serán abstractas. En la sección 10.5 demostraremos las clases abstractas.

Propósito de las clases abstractas

El propósito de una clase abstracta es proporcionar una superclase apropiada, a partir de la cual puedan heredar otras clases y, por ende, compartir un diseño común. Por ejemplo, en la jerarquía de `Figura` de la figura 9.3, las subclases heredan la noción de lo que significa ser una `Figura`: tal vez los atributos comunes como `posicion`, `color` y `grosorBorde`, y los comportamientos como `dibujar`, `mover`, `cambiarTamanio` y `cambiarColor`. Las clases que pueden utilizarse para instanciar objetos se llaman **clases concretas**, las cuales proporcionan implementaciones *de cada* método que declaran (algunas de las implementaciones pueden heredarse). Por ejemplo, podríamos derivar las clases concretas `Circulo`, `Cuadrado` y `Triangulo` de la superclase abstracta `FiguraBidimensional`. De manera similar, podríamos derivar las clases concretas `Esfera`, `Cubo` y `Tetraedro` de la superclase abstracta `FiguraTridimensional`. Las superclases abstractas son *demasiado generales* como para crear objetos reales; sólo especifican lo que tienen en común las subclases. Necesitamos ser más *específicos* para poder crear objetos. Por ejemplo, si envía el mensaje dibujar a la clase abstracta `FiguraBidimensional`, la clase sabe que las figuras bidimensionales deben poder dibujarse, pero no sabe qué figura específica dibujar, por lo que no puede implementar un verdadero método dibujar. Las clases concretas proporcionan los detalles específicos que hacen razonable la creación de instancias de objetos.

No todas las jerarquías contienen clases abstractas. Sin embargo, a menudo los programadores escriben código cliente que utiliza sólo tipos de superclases abstractas para reducir las dependencias del código cliente en un rango de tipos de subclases. Por ejemplo, un programador puede escribir un método con un parámetro de un tipo de superclase abstracta. Cuando se llama, ese método puede recibir un objeto de cualquier clase concreta que extienda en forma directa o indirecta a la superclase especificada como el tipo del parámetro.

Algunas veces las clases abstractas constituyen varios niveles de una jerarquía. Por ejemplo, la jerarquía de `Figura` de la figura 9.3 empieza con la clase abstracta `Figura`. En el siguiente nivel de la jerarquía están las clases *abstractas* `FiguraBidimensional` y `FiguraTridimensional`. El siguiente nivel de la jerarquía declara clases *concretas* para objetos `FiguraBidimensional` (`Circulo`, `Cuadrado` y `Triangulo`) y para objetos `FiguraTridimensional` (`Esfera`, `Cubo` y `Tetraedro`).

Declaración de una clase abstracta y de métodos abstractos

Para hacer una clase abstracta, ésta se declara con la palabra clave **`abstract`**. Por lo general, esta clase contiene uno o más **métodos abstractos**, cada uno con la palabra clave `abstract` en su declaración, como en

```
public abstract void dibujar(); // método abstracto
```

Los métodos abstractos *no* proporcionan implementaciones. Una clase que contiene uno o más métodos abstractos debe declararse de manera explícita como `abstract`, aun si esa clase contiene métodos concretos (no abstractos). Cada subclase concreta de una superclase abstracta también debe proporcionar implementaciones concretas de cada uno de los métodos abstractos de la superclase. Los constructores y los métodos `static` no pueden declararse como `abstract`. Los constructores no se heredan, por lo que nunca podría implementarse un constructor `abstract`. Aunque los métodos `static` que no son `private` se heredan, no pueden sobrescribirse. Como el propósito de los métodos `abstract` es sobrescribirlos para procesar objetos con base en sus tipos, no tendría sentido declarar un método `static` como `abstract`.

Observación de ingeniería de software 10.3

Una clase abstracta declara los atributos y comportamientos comunes (tanto abstractos como concretos) de las diversas clases en una jerarquía de clases. Por lo general, una clase abstracta contiene uno o más métodos abstractos, que las subclases deben sobrescribir, si van a ser concretas. Las variables de instancia y los métodos concretos de una clase abstracta están sujetos a las reglas normales de la herencia.

Error común de programación 10.1

Tratar de instanciar un objeto de una clase abstracta es un error de compilación.

Error común de programación 10.2

Si no se implementan los métodos abstractos de una superclase en una subclase, se produce un error de compilación, a menos que la subclase también se declare como `abstract`.

Uso de clases abstractas para declarar variables

Aunque no podemos instanciar objetos de superclases abstractas, pronto veremos que podemos usar superclases abstractas para declarar variables que puedan guardar referencias a objetos de cualquier clase concreta que se derive de esas superclases abstractas. Por lo general, los programas utilizan dichas variables para manipular los objetos de las subclases mediante el polimorfismo. Además, podemos usar los nombres de las superclases abstractas para invocar métodos `static` que estén declarados en esas superclases abstractas.

Considere otra aplicación del polimorfismo. Un programa de dibujo necesita mostrar en pantalla muchas figuras, incluyendo nuevos tipos de figuras que el programador agregará al sistema después de escribir el programa de dibujo. El cual podría necesitar mostrar figuras, como `Circulos`, `Triangulos`, `Rectangulos` u otras, que se deriven de la clase abstracta `Figura`. El programa de dibujo utiliza variables de `Figura` para administrar los objetos que se muestran en pantalla. Para dibujar cualquier objeto en esta jerarquía de herencia, utiliza una variable de la superclase `Figura` que contiene una referencia al objeto de la subclase para invocar al método dibujar del objeto. Este método se declara como `abstract` en la superclase `Figura`, por lo que cada subclase concreta *debe* implementar el método `dibujar` en una forma que sea específica para esa figura; cada objeto en la jerarquía de herencia de `Figura` *sabe cómo dibujarse a sí mismo*. El programa de dibujo no tiene que preocuparse acerca del tipo de cada objeto, o si alguna vez ha encontrado objetos de ese tipo.

Sistemas de software en capas

En especial, el polimorfismo es efectivo para implementar los denominados sistemas de software en capas. Por ejemplo, en los sistemas operativos cada tipo de dispositivo físico puede operar en forma muy distinta a los demás. Aun así, los comandos para leer o escribir datos desde y hacia los dispositivos pueden tener cierta uniformidad. Para cada dispositivo, el sistema operativo utiliza una pieza de software llamada *controlador de dispositivos* para controlar toda la comunicación entre el sistema y el dispositivo. El mensaje de escritura que se envía a un objeto controlador de dispositivo necesita interpretarse de manera específica en el contexto de ese controlador, y la forma en que manipula a un dispositivo de un tipo específico. No obstante, la llamada de escritura en sí no es distinta a la escritura en cualquier otro dispositivo en el sistema: colocar cierto número de bytes de memoria en ese dispositivo. Un sistema operativo orientado a objetos podría usar una superclase abstracta para proporcionar una "interfaz" apropiada para todos los controladores de dispositivos. Después, a través de la herencia de esa superclase abstracta, se forman subclases que se comporten todas de manera similar. Los métodos

del controlador de dispositivos se declaran como métodos abstractos en la superclase abstracta. Las implementaciones de estos métodos abstractos se proporcionan en las subclases concretas que corresponden a los tipos específicos de controladores de dispositivos. Siempre se están desarrollando nuevos dispositivos, a menudo mucho después de que se ha liberado el sistema operativo. Cuando usted compra un nuevo dispositivo, éste incluye un controlador de dispositivo proporcionado por el distribuidor. El dispositivo opera de inmediato, una vez que usted lo conecta a la computadora e instala el controlador de dispositivo. Éste es otro elegante ejemplo acerca de cómo el polimorfismo hace que los sistemas sean *extensibles*.

10.5 Caso de estudio: sistema de nómina utilizando polimorfismo

En esta sección analizamos de nuevo la jerarquía `EmpleadoPorComision-EmpleadoBaseMasComision` que exploramos a lo largo de la sección 9.4. Ahora podemos usar un método abstracto y polimorfismo para realizar cálculos de nómina, con base en una jerarquía de herencia de empleados mejorada que cumpla con los siguientes requerimientos:

> *Una compañía paga semanalmente a sus empleados, quienes se dividen en cuatro tipos: empleados asalariados que reciben un salario semanal fijo, sin importar el número de horas trabajadas; empleados por horas, que perciben un sueldo por hora y pago por tiempo extra (es decir, 1.5 veces la tarifa de su salario por horas), por todas las horas trabajadas que excedan a 40 horas; empleados por comisión, que perciben un porcentaje de sus ventas y empleados asalariados por comisión, que obtienen un salario base más un porcentaje de sus ventas. Para este periodo de pago, la compañía ha decidido recompensar a los empleados asalariados por comisión, agregando un 10% a sus salarios base. La compañía desea implementar una aplicación que realice sus cálculos de nómina en forma polimórfica.*

Utilizaremos la clase `abstract Empleado` para representar el concepto general de un empleado. Las clases que extienden a Empleado son `EmpleadoAsalariado`, `EmpleadoPorComision` y `EmpleadoPorHoras`. La clase `EmpleadoBaseMasComision` (que extiende a `EmpleadoPorComision`) representa el último tipo de empleado. El diagrama de clases de UML en la figura 10.2 muestra la jerarquía de herencia para nuestra aplicación polimórfica de nómina de empleados. El nombre de la clase abstracta `Empleado` está en cursivas, según la convención de UML.

Fig. 10.2 | Diagrama de clases de UML para la jerarquía de `Empleado`.

La superclase abstracta `Empleado` declara la "interfaz" para la jerarquía; esto es, el conjunto de métodos que puede invocar un programa en todos los objetos `Empleado`. Aquí utilizamos el término

"interfaz" en un sentido general, para referirnos a las diversas formas en que los programas pueden comunicarse con los objetos de cualquier subclase de `Empleado`. Tenga cuidado de no confundir la noción general de una "interfaz" con la noción formal de una interfaz en Java, el tema de la sección 10.7. Cada empleado, sin importar la manera en que se calculen sus ingresos, tiene un primer nombre, un apellido paterno y un número de seguro social, por lo que las variables de instancia `private primerNombre`, `apellidoPaterno` y `numeroSeguroSocial` aparecen en la superclase abstracta `Empleado`.

Las siguientes secciones implementan la jerarquía de clases de `Empleado` de la figura 10.2. La primera sección implementa la superclase abstracta `Empleado`. Cada una de las siguientes cuatro secciones implementan una de las clases concretas. La última sección implementa un programa de prueba que crea objetos de todas estas clases y procesa esos objetos mediante el polimorfismo.

10.5.1 La superclase abstracta `Empleado`

La clase `Empleado` (figura 10.4) proporciona los métodos `ingresos` y `toString`, además de los métodos *obtener* y *establecer* que manipulan las variables de instancia de `Empleado`. Es evidente que un método `ingresos` se aplica en forma genérica a todos los empleados. Pero cada cálculo de los ingresos depende de la clase de empleado. Por lo tanto, declaramos a `ingresos` como `abstract` en la superclase `Empleado`, ya que una implementación predeterminada no tiene sentido para ese método; no hay suficiente información para determinar qué monto debe devolver `ingresos`. Cada una de las subclases redefine a `ingresos` con una implementación apropiada. Para calcular los ingresos de un empleado, el programa asigna una referencia al objeto del empleado a una variable de la superclase `Empleado`, y después invoca al método `ingresos` en esa variable. Mantenemos un arreglo de variables `Empleado`, cada una de las cuales guarda una referencia a un objeto `Empleado` (desde luego que no puede haber objetos `Empleado`, ya que ésta es una clase abstracta. Sin embargo, debido a la herencia todos los objetos de todas las subclases de `Empleado` pueden considerarse como objetos `Empleado`). El programa itera a través del arreglo y llama al método `ingresos` para cada objeto `Empleado`. Java procesa estas llamadas a los métodos en forma polimórfica. Al declarar a `ingresos` como un método `abstract` en `Empleado`, es posible compilar las llamadas a `ingresos` que se realizan a través de las variables `Empleado`, y se obliga a cada subclase concreta directa de `Empleado` a sobrescribir el método ingresos.

El método `toString` en la clase `Empleado` devuelve un objeto `String` que contiene el primer nombre, el apellido paterno y el número de seguro social del empleado. Como veremos, cada subclase de `Empleado` sobrescribe el método `toString` para crear una representación `String` de un objeto de esa clase que contiene el tipo del empleado (por ejemplo, "`empleado asalariado:`"), seguido del resto de la información del empleado.

El diagrama en la figura 10.3 muestra cada una de las cinco clases en la jerarquía, hacia abajo en la columna de la izquierda, y los métodos `ingresos` y `toString` en la fila superior. Para cada clase, el diagrama muestra los resultados deseados de cada método. No enumeramos los métodos *establecer* y *obtener* de la superclase `Empleado` porque no se sobrescriben en ninguna de las subclases; las cuales heredan y utilizan cada uno de estos métodos "como están".

Consideremos ahora la declaración de la clase `Empleado` (figura 10.4). Esta clase tiene un constructor que recibe el primer nombre, el apellido paterno y el número de seguro social como argumentos (líneas 11 a 16); los métodos *obtener* que devuelven el primer nombre, apellido y número de seguro social (líneas 25 a 28, 37 a 40 y 49 a 52); los métodos *establecer* que establecen el primer nombre, el apellido paterno y el número de seguro social (líneas 19 a 22, 31 a 34 y 43 a 46); el método `toString` (líneas 55 a 60), el cual devuelve la representación `String` de `Empleado`; y el método `abstract ingresos` (línea 63), que cada una de las subclases concretas deben implementar. El constructor de `Empleado` no valida sus parámetros en este ejemplo; por lo general, se debe proporcionar esa validación.

	ingresos	toString
Empleado	abstract	*primerNombre apellidoPaterno* numero de seguro social: *NSS*
Empleado-Asalariado	salarioSemanal	empleado asalariado: *primerNombre apellidoPaterno* numero de seguro social: *NSS* salario semanal: *salarioSemanal*

Empleado-PorHoras	if (horas <= 40)
	  sueldo * horas
	else if (horas > 40)
	{
	  40 * sueldo +
	  (horas - 40) *
	  sueldo * 1.5
	}
		empleado por horas: *primerNombre apellidoPaterno*
		numero de seguro social: *NSS*
		sueldo por horas: *sueldo*; horas trabajadas: *horas*

Empleado PorComision	tarifaComision * ventasBrutas	empleado por comision: *primerNombre apellidoPaterno* numero de seguro social: *NSS* ventas brutas: *ventasBrutas*; tarifa de comision: *tarifaComision*
Empleado BaseMas Comision	(tarifaComision * ventasBrutas) + salarioBase	empleado por comision con salario base: *primerNombre apellidoPaterno* numero de seguro social: *NSS* ventas brutas: *ventasBrutas*; tarifa de comision: *tarifaComision*; salario base: *salarioBase*

Fig. 10.3 | Interfaz polimórfica para las clases de la jerarquía de `Empleado`.

```
1  // Fig. 10.4: Empleado.java
2  // La superclase abstracta Empleado.
3
4  public abstract class Empleado
5  {
6     private String primerNombre;
7     private String apellidoPaterno;
8     private String numeroSeguroSocial;
9
10    // constructor con tres argumentos
11    public Empleado( String nombre, String apellido, String nss )
12    {
13       primerNombre = nombre;
14       apellidoPaterno = apellido;
15       numeroSeguroSocial = nss;
16    } // fin del constructor de Empleado con tres argumentos
17
```

Fig. 10.4 | La superclase abstracta `Empleado` (parte 1 de 2).

```
18    // establece el primer nombre
19    public void establecerPrimerNombre( String nombre )
20    {
21       primerNombre = nombre; // debería validar
22    } // fin del método establecerPrimerNombre
23
24    // devuelve el primer nombre
25    public String obtenerPrimerNombre()
26    {
27       return primerNombre;
28    } // fin del método obtenerPrimerNombre
29
30    // establece el apellido paterno
31    public void establecerApellidoPaterno( String apellido )
32    {
33       apellidoPaterno = apellido; // debería validar
34    } // fin del método establecerApellidoPaterno
35
36    // devuelve el apellido paterno
37    public String obtenerApellidoPaterno()
38    {
39       return apellidoPaterno;
40    } // fin del método obtenerApellidoPaterno
41
42    // establece el número de seguro social
43    public void establecerNumeroSeguroSocial( String nss )
44    {
45       numeroSeguroSocial = nss; // debería validar
46    } // fin del método establecerNumeroSeguroSocial
47
48    // devuelve el número de seguro social
49    public String obtenerNumeroSeguroSocial()
50    {
51       return numeroSeguroSocial;
52    } // fin del método obtenerNumeroSeguroSocial
53
54    // devuelve representación String de un objeto Empleado
55    @Override
56    public String toString()
57    {
58       return String.format( "%s %s\nnumero de seguro social: %s",
59          obtenerPrimerNombre(), obtenerApellidoPaterno(), obtenerNumeroSeguroSocial() );
60    } // fin del método toString
61
62    // método abstracto sobrescrito por las subclases concretas
63    public abstract double ingresos(); // aquí no hay implementación
64 } // fin de la clase abstracta Empleado
```

Fig. 10.4 | La superclase abstracta `Empleado` (parte 2 de 2).

¿Por qué decidimos declarar a `ingresos` como un método `abstract`? Simplemente, no tiene sentido proporcionar una implementación de este método en la clase `Empleado`. No podemos calcular los ingresos para un `Empleado` *general*; primero debemos conocer el tipo de `Empleado` *específico* para determinar el cálculo apropiado de los ingresos. Al declarar este método `abstract`, indicamos que cada sub-

clase concreta *debe* proporcionar una implementación apropiada para `ingresos`, y que un programa podrá utilizar las variables de la superclase `Empleado` para invocar al método `ingresos` en forma polimórfica, para cualquier tipo de `Empleado`.

10.5.2 La subclase concreta `EmpleadoAsalariado`

La clase `EmpleadoAsalariado` (figura 10.5) extiende a la clase `Empleado` (línea 4) y sobrescribe el método abstracto `ingresos` (líneas 33 a 37), lo cual convierte a `EmpleadoAsalariado` en una clase concreta. La clase incluye un constructor (líneas 9 a 14) que recibe un primer nombre, un apellido paterno, un número de seguro social y un salario semanal como argumentos; un método *establecer* para asignar un nuevo valor no negativo a la variable de instancia `salarioSemanal` (líneas 17 a 24); un método *obtener* para devolver el valor de `salarioSemanal` (líneas 27 a 30); un método `ingresos` (líneas 33 a 37) para calcular los ingresos de un `EmpleadoAsalariado`; y un método `toString` (líneas 40 a 45), el cual devuelve un objeto `String` que tiene el tipo del empleado; "`empleado asalariado: `", seguido de la información específica para el empleado producida por el método `toString` de la superclase `Empleado` y el método `obtenerSalarioSemanal` de `EmpleadoAsalariado`. El constructor de la clase `EmpleadoAsalariado` pasa el primer nombre, el apellido paterno y el número de seguro social al constructor de `Empleado` (línea 12) para inicializar las variables de instancia `private` que no se heredan de la superclase. El método `ingresos` sobrescribe el método abstracto `ingresos` de `Empleado` para proporcionar una implementación concreta que devuelva el salario semanal del `EmpleadoAsalariado`. Si no implementamos `ingresos`, la clase `EmpleadoAsalariado` debe declararse como `abstract`; en caso contrario, se produce un error de compilación. Además, no hay duda de que `EmpleadoAsalariado` debe ser una clase concreta en este ejemplo.

```
1  // Fig. 10.5: EmpleadoAsalariado.java
2  // La clase concreta EmpleadoAsalariado extiende a la clase abstracta Empleado.
3
4  public class EmpleadoAsalariado extends Empleado
5  {
6     private double salarioSemanal;
7
8     // constructor de cuatro argumentos
9     public EmpleadoAsalariado( String nombre, String apellido, String nss,
10       double salario )
11    {
12       super( nombre, apellido, nss ); // los pasa al constructor de Empleado
13       establecerSalarioSemanal( salario ); // valida y almacena el salario
14    } // fin del constructor de EmpleadoAsalariado con cuatro argumentos
15
16    // establece el salario
17    public void establecerSalarioSemanal( double salario )
18    {
19       if (salario >= 0.0 )
20          salarioSemanal = salario;
21       else
22          throw new IllegalArgumentException(
23             "El salario semanal debe ser >= 0.0" );
24    } // fin del método establecerSalarioSemanal
25
```

Fig. 10.5 | La clase concreta `EmpleadoAsalariado` extiende a la clase `abstract` `Empleado` (parte 1 de 2).

```
26     // devuelve el salario
27     public double obtenerSalarioSemanal()
28     {
29        return salarioSemanal;
30     } // fin del método obtenerSalarioSemanal
31
32     // calcula los ingresos; sobrescribe el método abstracto ingresos en Empleado
33     @Override
34     public double ingresos()
35     {
36        return obtenerSalarioSemanal();
37     } // fin del método ingresos
38
39     // devuelve representación String de un objeto EmpleadoAsalariado
40     @Override
41     public String toString()
42     {
43        return String.format( "empleado asalariado: %s\n%s: $%,.2f",
44           super.toString(), "salario semanal", obtenerSalarioSemanal() );
45     } // fin del método toString
46  } // fin de la clase EmpleadoAsalariado
```

Fig. 10.5 | La clase concreta `EmpleadoAsalariado` extiende a la clase `abstract Empleado` (parte 2 de 2).

El método `toString` (líneas 40 a 45) sobrescribe al método `toString` de `Empleado`. Si la clase `EmpleadoAsalariado` no sobrescribiera a `toString`, `EmpleadoAsalariado` habría heredado la versión de `toString` de `Empleado`. En ese caso, el método `toString` de `EmpleadoAsalariado` simplemente devolvería el nombre completo del empleado y su número de seguro social, lo cual no representa en forma adecuada a un `EmpleadoAsalariado`. Para producir una representación `String` completa de un `EmpleadoAsalariado`, el método `toString` de la subclase devuelve `"empleado asalariado: "`, seguido de la información específica de la superclase `Empleado` (es decir, el primer nombre, el apellido paterno y el número de seguro social) que se obtiene al invocar el método `toString` de la superclase (línea 44); éste es un excelente ejemplo de reutilización de código. La representación `String` de un `EmpleadoAsalariado` también contiene el salario semanal del empleado, el cual se obtiene mediante la invocación del método `obtenerSalarioSemanal` de la clase.

10.5.3 La subclase concreta EmpleadoPorHoras

La clase `EmpleadoPorHoras` (figura 10.6) también extiende a `Empleado` (línea 4). La clase incluye un constructor (líneas 10 a 16) que recibe como argumentos un primer nombre, un apellido paterno, un número de seguro social, un sueldo por horas y el número de horas trabajadas. Las líneas 19 a 26 y 35 a 42 declaran los métodos *establecer* que asignan nuevos valores a las variables de instancia `sueldo` y `horas`. El método `establecerSueldo` (líneas 19 a 26) asegura que `sueldo` sea no negativo, y el método `establecerHoras` (líneas 35 a 42) asegura que `horas` esté entre 0 y 168 (el número total de horas en una semana), ambos valores inclusive. La clase `EmpleadoPorHoras` también incluye métodos *obtener* (líneas 29 a 32 y 45 a 48) para devolver los valores de `sueldo` y `horas`, respectivamente; un método `ingresos` (líneas 51 a 58) para calcular los ingresos de un `EmpleadoPorHoras`; y un método `toString` (líneas 61 a 67), que devuelve un objeto `String` con el tipo del empleado (`"empleado por horas: "`), e información específica para ese `Empleado`. El constructor de `EmpleadoPorHoras`, al igual que el constructor de `EmpleadoAsalariado`, pasa el primer nombre, el apellido paterno y el número de seguro social al constructor de la superclase `Empleado` (línea 13) para inicializar las variables de instancia `private`.

Además, el método `toString` llama al método `toString` de la superclase (línea 65) para obtener la información específica del `Empleado` (es decir, primer nombre, apellido paterno y número de seguro social); éste es otro excelente ejemplo de reutilización de código.

```
1  // Fig. 10.6: EmpleadoPorHoras.java
2  // La clase EmpleadoPorHoras extiende a Empleado.
3
4  public class EmpleadoPorHoras extends Empleado
5  {
6     private double sueldo; // sueldo por hora
7     private double horas; // horas trabajadas por semana
8
9     // constructor con cinco argumentos
10    public EmpleadoPorHoras( String nombre, String apellido, String nss,
11       double sueldoPorHoras, double horasTrabajadas )
12    {
13       super( nombre, apellido, nss );
14       establecerSueldo( sueldoPorHoras ); // valida y almacena el sueldo por horas
15       establecerHoras( horasTrabajadas ); // valida y almacena las horas trabajadas
16    } // fin del constructor de EmpleadoPorHoras con cinco argumentos
17
18    // establece el sueldo
19    public void establecerSueldo( double sueldoPorHoras )
20    {
21       if (sueldoPorHoras >= 0.0 )
22          sueldo = sueldoPorHoras;
23       else
24          throw new IllegalArgumentException(
25             "El sueldo por horas debe ser >= 0.0" );
26    } // fin del método establecerSueldo
27
28    // devuelve el sueldo
29    public double obtenerSueldo()
30    {
31       return sueldo;
32    } // fin del método obtenerSueldo
33
34    // establece las horas trabajadas
35    public void establecerHoras( double horasTrabajadas )
36    {
37       if ( ( horasTrabajadas >= 0.0 ) && ( horasTrabajadas <= 168.0 ) )
38          horas = horasTrabajadas;
39       else
40          throw new IllegalArgumentException(
41             "Las horas trabajadas deben ser >= 0.0 y <= 168.0" );
42    } // fin del método establecerHoras
43
44    // devuelve las horas trabajadas
45    public double obtenerHoras()
46    {
47       return horas;
48    } // fin del método obtenerHoras
```

Fig. 10.6 | La clase `EmpleadoPorHoras` extiende a `Empleado` (parte 1 de 2).

```
49
50    // calcula los ingresos; sobrescribe el método abstracto ingresos en Empleado
51    @Override
52    public double ingresos()
53    {
54       if ( obtenerHoras() <= 40 ) // no hay tiempo extra
55          return obtenerSueldo() * obtenerHoras();
56       else
57          return 40 * obtenerSueldo() + ( obtenerHoras() - 40 ) * obtenerSueldo() * 1.5;
58    } // fin del método ingresos
59
60    // devuelve representación String de un objeto EmpleadoPorHoras
61    @Override
62    public String toString()
63    {
64       return String.format( "empleado por horas: %s\n%s: $%,.2f; %s: %,.2f",
65          super.toString(), "sueldo por hora", obtenerSueldo(),
66          "horas trabajadas", obtenerHoras() );
67    } // fin del método toString
68 } // fin de la clase EmpleadoPorHoras
```

Fig. 10.6 | La clase `EmpleadoPorHoras` extiende a `Empleado` (parte 2 de 2).

10.5.4 La subclase concreta `EmpleadoPorComision`

La clase `EmpleadoPorComision` (figura 10.7) extiende a la clase `Empleado` (línea 4). Esta clase incluye a un constructor (líneas 10 a 16) que recibe como argumentos un primer nombre, un apellido, un número de seguro social, un monto de ventas y una tarifa de comisión; métodos *establecer* (líneas 19 a 26 y 35 a 42) para asignar nuevos valores a las variables de instancia `tarifaComision` y `ventasBrutas`, respectivamente; métodos *obtener* (líneas 29 a 32 y 45 a 48) que obtienen los valores de estas variables de instancia; el método `ingresos` (líneas 51 a 55) para calcular los ingresos de un `EmpleadoPorComision`; y el método `toString` (líneas 58 a 65) que devuelve el tipo del empleado;, "`empleado por comisión: `", e información específica del empleado. El constructor también pasa el primer nombre, el apellido y el número de seguro social al constructor de `Empleado` (línea 13) para inicializar las variables de instancia `private` de `Empleado`. El método `toString` llama al método `toString` de la superclase (línea 62) para obtener la información específica del `Empleado` (es decir, primer nombre, apellido paterno y número de seguro social).

```
1  // Fig. 10.7: EmpleadoPorComision.java
2  // La clase EmpleadoPorComision extiende a Empleado.
3
4  public class EmpleadoPorComision extends Empleado
5  {
6     private double ventasBrutas; // ventas totales por semana
7     private double tarifaComision; // porcentaje de comisión
8
9     // constructor con cinco argumentos
10    public EmpleadoPorComision( String nombre, String apellido, String nss,
11       double ventas, double tarifa )
12    {
13       super( nombre, apellido, nss );
```

Fig. 10.7 | La clase `EmpleadoPorComision` extiende a `Empleado` (parte 1 de 2).

```
14          establecerVentasBrutas( ventas );
15          establecerTarifaComision( tarifa );
16       } // fin del constructor de EmpleadoPorComision con cinco argumentos
17
18       // establece la tarifa de comisión
19       public void establecerTarifaComision( double tarifa )
20       {
21          if ( tarifa > 0.0 && tarifa < 1.0 )
22             tarifaComision = tarifa;
23          else
24             throw new IllegalArgumentException(
25                "La tarifa de comision debe ser > 0.0 y < 1.0" );
26       } // fin del método establecerTarifaComision
27
28       // devuelve la tarifa de comisión
29       public double obtenerTarifaComision()
30       {
31          return tarifaComision;
32       } // fin del método obtenerTarifaComision
33
34       // establece el monto de ventas brutas
35       public void establecerVentasBrutas( double ventas )
36       {
37          if (ventas >= 0.0 )
38             ventasBrutas = ventas;
39          else
40             throw new IllegalArgumentException(
41                "Las ventas brutas deben ser >= 0.0" );
42       } // fin del método establecerVentasBrutas
43
44       // devuelve el monto de ventas brutas
45       public double obtenerVentasBrutas()
46       {
47          return ventasBrutas;
48       } // fin del método obtenerVentasBrutas
49
50       // calcula los ingresos; sobrescribe el método abstracto ingresos en Empleado
51       @Override
52       public double ingresos()
53       {
54          return obtenerTarifaComision() * obtenerVentasBrutas();
55       } // fin del método ingresos
56
57       // devuelve representación String de un objeto EmpleadoPorComision
58       @Override
59       public String toString()
60       {
61          return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
62             "empleado por comision", super.toString(),
63             "ventas brutas", obtenerVentasBrutas(),
64             "tarifa de comision", obtenerTarifaComision() );
65       } // fin del método toString
66    } // fin de la clase EmpleadoPorComision
```

Fig. 10.7 | La clase `EmpleadoPorComision` extiende a `Empleado` (parte 2 de 2).

10.5.5 La subclase concreta indirecta EmpleadoBaseMasComision

La clase EmpleadoBaseMasComision (figura 10.8) extiende a la clase EmpleadoPorComision (línea 4) y, por lo tanto, es una subclase *indirecta* de la clase Empleado. La clase EmpleadoBaseMasComision tiene un constructor (líneas 9 a 14) que recibe como argumentos un primer nombre, un apellido paterno, un número de seguro social, un monto de ventas, una tarifa de comisión y un salario base. Después pasa todos estos parámetros, excepto el salario base, al constructor de EmpleadoPorComision (línea 12) para inicializar los miembros heredados. EmpleadoBaseMasComision también contiene un método *establecer* (líneas 17 a 24) para asignar un nuevo valor a la variable de instancia salarioBase y un método *obtener* (líneas 27 a 30) para devolver el valor de salarioBase. El método ingresos (líneas 33 a 37) calcula los ingresos de un EmpleadoBaseMasComision. Observe que la línea 36 en el método ingresos llama al método ingresos de la superclase EmpleadoPorComision para calcular la porción con base en la comisión de los ingresos del empleado; éste es otro buen ejemplo de reutilización de código. El método toString de EmpleadoBaseMasComision (líneas 40 a 46) crea una representación String de un EmpleadoBaseMasComision, la cual contiene "con salario base", seguida del objeto String que se obtiene al invocar el método toString de la superclase EmpleadoPorComision (otro buen ejemplo de reutilización de código), y después el salario base. El resultado es un objeto String que empieza con "con salario base empleado por comisión", seguido del resto de la información de EmpleadoBaseMasComision. Recuerde que el método toString de EmpleadoPorComision obtiene el primer nombre, el apellido paterno y el número de seguro social del empleado mediante la invocación al método toString de su superclase (es decir, Empleado); otro ejemplo más de reutilización de código. El método toString de EmpleadoBaseMasComision inicia una cadena de llamadas a métodos que abarcan los tres niveles de la jerarquía de Empleado.

```
1  // Fig. 10.8: EmpleadoBaseMasComision.java
2  // La clase EmpleadoBaseMasComision extiende a EmpleadoPorComision.
3
4  public class EmpleadoBaseMasComision extends EmpleadoPorComision
5  {
6     private double salarioBase; // salario base por semana
7
8     // constructor con seis argumentos
9     public EmpleadoBaseMasComision( String nombre, String apellido,
10       String nss, double ventas, double tarifa, double salario )
11    {
12       super( nombre, apellido, nss, ventas, tarifa );
13       establecerSalarioBase( salario ); // valida y almacena el salario base
14    } // fin del constructor de EmpleadoBaseMasComision con seis argumentos
15
16    // establece el salario base
17    public void establecerSalarioBase( double salario )
18    {
19       if (salario >= 0.0 )
20          salarioBase = salario;
21       else
22          throw new IllegalArgumentException(
23             "El salario base debe ser >= 0.0" );
24 } // fin del método establecerSalarioBase
25
```

Fig. 10.8 | La clase EmpleadoBaseMasComision extiende a EmpleadoPorComision (parte 1 de 2).

```
26      // devuelve el salario base
27      public double obtenerSalarioBase()
28      {
29         return salarioBase;
30      } // fin del método obtenerSalarioBase
31
32      // calcula los ingresos; sobrescribe el método ingresos en EmpleadoPorComision
33      @Override
34      public double ingresos()
35      {
36         return obtenerSalarioBase() + super.ingresos();
37      } // fin del método ingresos
38
39      // devuelve representación String de un objeto EmpleadoBaseMasComision
40      @Override
41      public String toString()
42      {
43         return String.format( "%s %s; %s: $%,.2f",
44            "con salario base", super.toString(),
45            "salario base", obtenerSalarioBase() );
46      } // fin del método toString
47   } // fin de la clase EmpleadoBaseMasComision
```

Fig. 10.8 | La clase `EmpleadoBaseMasComision` extiende a `EmpleadoPorComision` (parte 2 de 2).

10.5.6 El procesamiento polimórfico, el operador instanceof y la conversión descendente

Para probar nuestra jerarquía de `Empleado`, la aplicación en la figura 10.9 crea un objeto de cada una de las cuatro clases concretas `EmpleadoAsalariado`, `EmpleadoPorHoras`, `EmpleadoPorComision` y `EmpleadoBaseMasComision`. El programa manipula estos objetos, primero mediante variables del mismo tipo de cada objeto y después mediante el polimorfismo, utilizando un arreglo de variables `Empleado`. Al procesar los objetos mediante el polimorfismo, el programa incrementa el salario base de cada `EmpleadoBaseMasComision` en un 10%; para esto se requiere *determinar el tipo del objeto en tiempo de ejecución*. Por último, el programa determina e imprime en forma polimórfica el tipo de cada objeto en el arreglo `Empleado`. Las líneas 9 a 18 crean objetos de cada una de las cuatro subclases concretas de `Empleado`. Las líneas 22 a 30 imprimen en pantalla la representación `String` y los ingresos de cada uno de estos objetos *sin usar el polimorfismo*. El método `printf` llama en forma *implícita* al método `toString` de cada objeto, cuando éste se imprime en pantalla como un objeto `String` con el especificador de formato `%s`.

```
1   // Fig. 10.9: PruebaSistemaNomina.java
2   // Programa de prueba para la jerarquía de Empleado.
3
4   public class PruebaSistemaNomina
5   {
6      public static void main( String[] args )
7      {
8         // crea objetos de las subclases
9         EmpleadoAsalariado empleadoAsalariado =
10           new EmpleadoAsalariado( "John", "Smith", "111-11-1111", 800.00 );
```

Fig. 10.9 | Programa de prueba de la jerarquía de clases de `Empleado` (parte 1 de 4).

```
11         EmpleadoPorHoras empleadoPorHoras =
12            new EmpleadoPorHoras( "Karen", "Price", "222-22-2222", 16.75, 40 );
13         EmpleadoPorComision empleadoPorComision =
14            new EmpleadoPorComision(
15            "Sue", "Jones", "333-33-3333", 10000, .06 );
16         EmpleadoBaseMasComision empleadoBaseMasComision =
17            new EmpleadoBaseMasComision(
18            "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
19
20         System.out.println( "Empleados procesados por separado:\n" );
21
22         System.out.printf( "%s\n%s: $%,.2f\n\n",
23            empleadoAsalariado, "ingresos", empleadoAsalariado.ingresos() );
24         System.out.printf( "%s\n%s: $%,.2f\n\n",
25            empleadoPorHoras, "ingresos", empleadoPorHoras.ingresos() );
26         System.out.printf( "%s\n%s: $%,.2f\n\n",
27            empleadoPorComision, "ingresos", empleadoPorComision.ingresos() );
28         System.out.printf( "%s\n%s: $%,.2f\n\n",
29            empleadoBaseMasComision,
30            "ingresos", empleadoBaseMasComision.ingresos() );
31
32         // crea un arreglo Empleado de cuatro elementos
33         Empleado[] empleados = new Empleado[ 4 ];
34
35         // inicializa el arreglo con objetos Empleado
36         empleados[ 0 ] = empleadoAsalariado;
37         empleados[ 1 ] = empleadoPorHoras;
38         empleados[ 2 ] = empleadoPorComision;
39         empleados[ 3 ] = empleadoBaseMasComision;
40
41         System.out.println( "Empleados procesados en forma polimorfica:\n" );
42
43         // procesa en forma genérica a cada elemento en el arreglo de empleados
44         for ( Empleado empleadoActual : empleados )
45         {
46            System.out.println( empleadoActual ); // invoca a toString
47
48            // determina si el elemento es un EmpleadoBaseMasComision
49            if ( empleadoActual instanceof EmpleadoBaseMasComision )
50            {
51               // conversión descendente de la referencia de Empleado
52               // a una referencia de EmpleadoBaseMasComision
53               EmpleadoBaseMasComision empleado =
54                  ( EmpleadoBaseMasComision ) empleadoActual;
55
56               empleado.establecerSalarioBase( 1.10 * empleado.obtenerSalarioBase() );
57
58               System.out.printf(
59                  "el nuevo salario base con 10%% de aumento es : $%,.2f\n",
60                  empleado.obtenerSalarioBase() );
61            } // fin de if
62
```

Fig. 10.9 | Programa de prueba de la jerarquía de clases de `Empleado` (parte 2 de 4).

```
63            System.out.printf(
64               "ingresos $%,.2f\n\n", empleadoActual.ingresos() );
65         } // fin de for
66
67         // obtiene el nombre del tipo de cada objeto en el arreglo de empleados
68         for ( int j = 0; j < empleados.length; j++ )
69            System.out.printf( "El empleado %d es un %s\n", j,
70               empleados[ j ].getClass().getName() );
71      } // fin de main
72   } // fin de la clase PruebaSistemaNomina
```

```
Empleados procesados por separado:

empleado asalariado: John Smith
numero de seguro social: 111-11-1111
salario semanal: $800.00
ingresos: $800.00

empleado por horas: Karen Price
numero de seguro social: 222-22-2222
sueldo por hora: $16.75; horas trabajadas: 40.00
ingresos: $670.00

empleado por comision: Sue Jones
numero de seguro social: 333-33-3333
ventas brutas: $10,000.00; tarifa de comision: 0.06
ingresos: $600.00

con salario base empleado por comision: Bob Lewis
numero de seguro social: 444-44-4444
ventas brutas: $5,000.00; tarifa de comision: 0.04; salario base: $300.00
ingresos: $500.00

Empleados procesados en forma polimorfica:

empleado asalariado: John Smith
numero de seguro social: 111-11-1111
salario semanal: $800.00
ingresos $800.00

empleado por horas: Karen Price
numero de seguro social: 222-22-2222
sueldo por hora: $16.75; horas trabajadas: 40.00
ingresos $670.00

empleado por comision: Sue Jones
numero de seguro social: 333-33-3333
ventas brutas: $10,000.00; tarifa de comision: 0.06
ingresos $600.00

con salario base empleado por comision: Bob Lewis
numero de seguro social: 444-44-4444
ventas brutas: $5,000.00; tarifa de comision: 0.04; salario base: $300.00
el nuevo salario base con 10% de aumento es : $330.00
ingresos $530.00
```

Fig. 10.9 | Programa de prueba de la jerarquía de clases de `Empleado` (parte 3 de 4).

```
El empleado 0 es un EmpleadoAsalariado
El empleado 1 es un EmpleadoPorHoras
El empleado 2 es un EmpleadoPorComision
El empleado 3 es un EmpleadoBaseMasComision
```

Fig. 10.9 | Programa de prueba de la jerarquía de clases de `Empleado` (parte 4 de 4).

Creación del arreglo de objetos `Empleado`

La línea 33 declara a `empleados` y le asigna un arreglo de cuatro variables `Empleado`. La línea 36 asigna la referencia a un objeto `EmpleadoAsalariado` a `empleados[0]`. La línea 37 asigna la referencia a un objeto `EmpleadoPorHoras` a `empleados[1]`. La línea 38 asigna la referencia a un objeto `EmpleadoPorComision` a `empleados[2]`. La línea 39 asigna la referencia a un objeto `EmpleadoBaseMasComision` a `empleados[3]`. Estas asignaciones se permiten, ya que un `EmpleadoAsalariado` *es un* `Empleado`, un `EmpleadoPorHoras` *es un* `Empleado`, un `EmpleadoPorComision` *es un* `Empleado` y un `EmpleadoBaseMasComision` *es un* `Empleado`. Por lo tanto, podemos asignar las referencias de los objetos `EmpleadoAsalariado`, `EmpleadoPorHoras`, `EmpleadoPorComision` y `EmpleadoBaseMasComision` a variables de la superclase `Empleado`, *aun cuando ésta es una clase abstracta.*

Procesamiento de objetos `Empleado` *mediante el polimorfismo*

Las líneas 44 a 65 iteran a través del arreglo de empleados e invocan los métodos `toString` e `ingresos` con la variable `empleadoActual` de `Empleado`, a la cual se le asigna la referencia a un `Empleado` distinto en el arreglo, durante cada iteración. Los resultados ilustran que en definitivo se invocan los métodos apropiados para cada clase. Todas las llamadas a los métodos `toString` e `ingresos` se resuelven en tiempo de ejecución, con base en el tipo del objeto al que `empleadoActual` hace referencia. Este proceso se conoce como **vinculación dinámica** o **vinculación postergada**. Por ejemplo, la línea 46 invoca en forma *implícita* al método `toString` del objeto al que `empleadoActual` hace referencia. Como resultado de la vinculación dinámica, Java decide qué método `toString` de cuál clase llamará *en tiempo de ejecución, en vez de hacerlo en tiempo de compilación*. Sólo los métodos de la clase `Empleado` pueden llamarse a través de una variable `Empleado` (y desde luego que `Empleado` incluye los métodos de la clase `Object`). Una referencia a la superclase puede utilizarse para invocar sólo a métodos de la superclase; las implementaciones de los métodos de las subclases se invocan mediante el polimorfismo.

Realización de operaciones de tipos específicos en objetos `EmpleadoBasePorComision`

Realizamos un procesamiento especial en los objetos `EmpleadoBasePorComision`; a medida que los encontramos en tiempo de ejecución, incrementamos su salario base en un 10%. Cuando procesamos objetos en forma polimórfica, por lo general no necesitamos preocuparnos por los "detalles específicos", pero para ajustar el salario base, *tenemos* que determinar el tipo específico de cada objeto `Empleado` en tiempo de ejecución. La línea 49 utiliza el operador `instanceof` para determinar si el tipo de cierto objeto Empleado es `EmpleadoBaseMasComision`. La condición en la línea 49 es verdadera si el objeto al que hace referencia `empleadoActual` *es un* `EmpleadoBaseMasComision`. Esto también sería verdadero para cualquier objeto de una subclase de `EmpleadoBaseMasComision`, debido a la relación *es un* que tiene una subclase con su superclase. Las líneas 53 y 54 realizan una conversión descendente en `empleadoActual`, del tipo `Empleado` al tipo `EmpleadoBaseMasComision`; esta conversión se permite sólo si el objeto tiene una relación *es un* con `EmpleadoBaseMasComision`. La condición en la línea 49 asegura que éste sea el caso. Esta conversión se requiere si vamos a invocar los métodos `obtenerSalarioBase` y `establecerSalarioBase` de la subclase `EmpleadoBaseMasComision` en el objeto `Empleado` actual; como veremos en

un momento, *si tratamos de invocar a un método que pertenezca sólo a la subclase directamente en una referencia a la superclase, se produce un error de compilación.*

Error común de programación 10.3

Asignar una variable de la superclase a una variable de la subclase (sin una conversión descendente explícita) es un error de compilación.

Observación de ingeniería de software 10.4

Si en tiempo de ejecución se asigna la referencia a un objeto de la subclase a una variable de una de sus superclases directas o indirectas, es aceptable convertir la referencia almacenada en esa variable de la superclase, de vuelta a una referencia del tipo de la subclase. Antes de realizar dicha conversión, use el operador `instanceof` *para asegurar que el objeto sea indudablemente de una subclase apropiada.*

Error común de programación 10.4

Al realizar una conversión descendente sobre una referencia, se produce una excepción `ClassCastException` *si, en tiempo de ejecución, el objeto al que se hace referencia no tiene una relación* es un *con el tipo especificado en el operador de conversión.*

Si la expresión `instanceof` en la línea 49 es `true`, las líneas 53 a 60 realizan el procesamiento especial requerido para el objeto `EmpleadoBaseMasComision`. Mediante el uso de la variable empleado de `EmpleadoBaseMasComision`, la línea 56 invoca a los métodos `obtenerSalarioBase` y `establecerSalarioBase`, que sólo pertenecen a la subclase, para obtener y actualizar el salario base del empleado con el aumento del 10%.

Llamada a `ingresos` *mediante el polimorfismo*

Las líneas 63 y 64 invocan al método `ingresos` en `empleadoActual`, el cual llama al método `ingresos` del objeto de la subclase apropiada en forma polimórfica. Al obtener en forma polimórfica los ingresos del `EmpleadoAsalariado`, el `EmpleadoPorHoras` y el `EmpleadoPorComision` en las líneas 63 y 64, se produce el mismo resultado que obtener los ingresos de estos empleados en forma individual, en las líneas 22 a 27. El monto de los ingresos obtenidos para el `EmpleadoBaseMasComision` en las líneas 63 y 64 es más alto que el que se obtiene en las líneas 28 a 30, debido al aumento del 10% en su salario base.

Uso de la reflexión para obtener el nombre de la clase de cada `Empleado`

Las líneas 68 a 70 imprimen en pantalla el tipo de cada empleado como un objeto `String`, mediante el uso de las características básicas de lo que se conoce como capacidades de reflexión de Java. Todos los objetos en Java conocen su propia clase y pueden acceder a esta información a través del método `getClass`, que todas las clases heredan de la clase `Object`. El método `getClass` devuelve un objeto de tipo `Class` (del paquete `java.lang`), el cual contiene información acerca del tipo del objeto, incluyendo el nombre de su clase. La línea 70 invoca al método `getClass` en el objeto actual para obtener su clase en tiempo de ejecución. El resultado de la llamada a `getClass` se utiliza para invocar al método `getName` y obtener el nombre de la clase del objeto.

Evite los errores de compilación mediante la conversión descendente

En el ejemplo anterior, evitamos varios errores de compilación mediante la conversión descendente de una variable de `Empleado` a una variable de `EmpleadoBaseMasComision` en las líneas 53 y 54. Si eliminamos el operador de conversión (`EmpleadoBaseMasComision`) de la línea 54 y tratamos de asignar la variable `empleadoActual` de `Empleado` directamente a la variable `empleado` de `EmpleadoBaseMasComision`, recibiremos un error de compilación del tipo "`incompatible types`" (incompatibilidad de tipos). Este error indica que el intento de asignar la referencia del objeto `empleadoPorComision` de la superclase a la variable `empleadoBaseMasComision` de la subclase no se permite. El compilador evita esta asignación debido a que un `EmpleadoPorComision` no es un `EmpleadoBaseMasComision`; *la relación* es un *se aplica sólo entre la subclase y sus superclases, no viceversa.*

De manera similar, si las líneas 56 y 60 utilizaran la variable `empleadoActual` de la superclase para invocar a los métodos `obtenerSalarioBase` y `establecerSalarioBase` que sólo pertenecen a la subclase, recibiríamos errores de compilación del tipo "`cannot find symbol`" (no se puede encontrar el símbolo) en estas líneas. No se permite tratar de invocar métodos que pertenezcan sólo a la subclase, a través de una variable de la superclase, aun cuando las líneas 56 y 60 se ejecutan sólo si `instanceof` en la línea 49 devuelve `true` para indicar que a `empleadoActual` se le asignó una referencia a un objeto `EmpleadoBaseMasComision`. Si utilizamos una variable `Empleado` de la superclase, sólo podemos invocar a los métodos que se encuentran en la clase `Empleado`: `ingresos`, `toString`, y los métodos *obtener* y *establecer* de `Empleado`.

Observación de ingeniería de software 10.5

Aunque el método que se vaya a llamar depende del tipo en tiempo de ejecución del objeto al que una variable hace referencia, puede utilizarse una variable para invocar sólo a los métodos que sean miembros del tipo de ésta, lo cual verifica el compilador.

10.5.7 Resumen de las asignaciones permitidas entre variables de la superclase y de la subclase

Ahora que hemos visto una aplicación completa que procesa diversos objetos de las subclases en forma polimórfica, sintetizaremos lo que puede y no hacer con los objetos y variables de las superclases y las subclases. Aunque un objeto de una subclase también *es un* objeto de su superclase, los dos son, sin embargo, distintos. Como vimos antes, los objetos de una subclase pueden tratarse como si fueran objetos de la superclase. Sin embargo, como la subclase puede tener miembros adicionales que sólo pertenezcan a ella, no se permite asignar una referencia de la superclase a una variable de la subclase sin una conversión explícita; dicha asignación dejaría los miembros de la subclase indefinidos para el objeto de la superclase.

Hemos visto cuatro maneras de asignar referencias de una superclase y de una subclase a las variables de sus tipos:

1. Asignar una referencia de la superclase a una variable de ella es un proceso simple y directo.
2. Asignar una referencia de la subclase a una variable de ella es un proceso simple y directo.
3. Asignar una referencia de la subclase a una variable de la superclase es seguro, ya que el objeto de la subclase *es un* objeto de su superclase. No obstante, la variable de la superclase puede usarse para referirse *sólo* a los miembros de la superclase. Si este código hace referencia a los miembros que pertenezcan sólo a la subclase, a través de la variable de la superclase, el compilador reporta errores.
4. Tratar de asignar una referencia de la superclase a una variable de la subclase produce un error de compilación. Para evitar este error, la referencia de la superclase debe convertirse en forma explícita a un tipo de la subclase. En *tiempo de ejecución*, si el objeto que menciona la referencia *no* es un objeto de la subclase, se producirá una excepción. (Para más información sobre el manejo de excepciones vea el capítulo 11). Es conveniente usar el operador `instanceof` para asegurar que dicha conversión se realice sólo si el objeto es de la subclase.

10.6 Métodos y clases `final`

En las secciones 6.3 y 6.10 vimos que las variables pueden declararse como `final` para indicar que no pueden modificarse una vez que se inicializan; dichas variables representan valores constantes. También es posible declarar métodos, parámetros de los métodos y clases con el modificador `final`.

Los métodos final no se pueden sobrescribir

Un método final en una superclase no puede sobrescribirse en una subclase; esto garantiza que todas las subclases directas e indirectas en la jerarquía utilicen la implementación del método final. Los méto-

dos que se declaran como `private` son implícitamente `final`, ya que es imposible sobrescribirlos en una subclase. Los métodos que se declaran como `static` también son implícitamente `final`. La declaración de un método `final` nunca puede cambiar, por lo cual todas las subclases utilizan la misma implementación del método, y las llamadas a los métodos `final` se resuelven en tiempo de compilación; a esto se le conoce como **vinculación estática**.

Las clases final no pueden ser superclases

Una **clase `final`** que se declara como `final` no puede ser una superclase (es decir, una clase no puede extender a una clase `final`). Todos los métodos en una clase `final` son implícitamente `final`. La clase `String` es un ejemplo de una clase `final`. Si pudiéramos crear una subclase de `String`, los objetos de esa subclase podrían usarse en cualquier lugar en donde se esperaran objetos `String`. Como esta clase no puede extenderse, lo que los programas que utilizan objetos `String` pueden depender de la funcionalidad de los objetos `String`, según lo especificado en la API de java. Al hacer la clase `final` también se evita que los programadores creen subclases que podrían ignorar las restricciones de seguridad. Para obtener más información sobre el uso de la palabra clave `final`, visite

```
download.oracle.com/javase/tutorial/java/IandI/final.html
```

y

```
www.ibm.com/developerworks/java/library/j-jtp1029.html
```

Error común de programación 10.5

Tratar de declarar una subclase de una clase `final` es un error de compilación.

Observación de ingeniería de software 10.6

En la API de Java, la vasta mayoría de clases no *se declara como `final`. Esto permite la herencia y el polimorfismo. Sin embargo, en algunos casos es importante declarar las clases como `final`; generalmente por razones de seguridad.*

10.7 Caso de estudio: creación y uso de interfaces

En nuestro siguiente ejemplo (figuras 10.11 a 10.15) analizaremos de nuevo el sistema de nómina de la sección 10.5. Suponga que la compañía involucrada desea realizar varias operaciones de contabilidad en una sola aplicación de cuentas por pagar; además de valuar los ingresos de nómina que deben pagarse a cada empleado, la compañía debe también calcular el pago vencido en cada una de varias facturas (por los bienes comprados). Aunque se aplican a cosas no relacionadas (es decir, empleados y facturas), ambas operaciones tienen que ver con el cálculo de algún tipo de monto a pagar. Para un empleado, el pago se refiere a sus ingresos. Para una factura, el pago se refiere al costo total de los bienes listados en la misma. ¿Podemos calcular esas cosas *distintas*, como los pagos vencidos para los empleados y las facturas, en forma polimórfica en *una sola* aplicación? ¿Ofrece Java una herramienta que requiera que las clases *no relacionadas* implementen un conjunto de métodos *comunes* (por ejemplo, un método que calcule un monto a pagar)? Las **interfaces** de Java ofrecen exactamente esta herramienta.

Estandarización de las interacciones

Las interfaces definen y estandarizan las formas en que pueden interactuar las cosas entre sí, como las personas y los sistemas. Por ejemplo, los controles en una radio sirven como una interfaz entre los usuarios de la radio y sus componentes internos. Los controles permiten a los usuarios realizar un conjunto

limitado de operaciones (por ejemplo, cambiar la estación, ajustar el volumen, seleccionar AM o FM), y distintas radios pueden implementar los controles de distintas formas (por ejemplo, el uso de botones, perillas, comandos de voz). La interfaz especifica *qué* operaciones debe permitir la radio que realicen los usuarios, pero no *cómo* deben hacerse.

Los objetos de software se comunican a través de interfaces

Los objetos de software también se comunican a través de interfaces. Una interfaz de Java describe un conjunto de métodos que pueden llamarse sobre un objeto; por ejemplo, para indicar al objeto que realice cierta tarea, o que devuelva cierta pieza de información. El siguiente ejemplo introduce una interfaz llamada `PorPagar`, la cual describe la funcionalidad de cualquier objeto que deba ser capaz de recibir un pago y, por lo tanto, debe ofrecer un método para determinar el monto de pago vencido apropiado. La **declaración de una interfaz** empieza con la palabra clave `interface` y sólo puede contener constantes y métodos `abstract`. A diferencia de las clases, todos los miembros de la interfaz deben ser `public`, y *las interfaces no pueden especificar ningún detalle de implementación*, como las declaraciones de métodos concretos y variables de instancia. Todos los métodos que se declaran en una interfaz son `public abstract` de manera implícita, y todos los campos son implícitamente `public`, `static` y `final`. [*Nota:* a partir de Java SE 5, declarar conjuntos de constantes como enumeraciones mediante la palabra `enum` se convirtió en una mejor práctica de programación. Vea la sección 6.10 para una introducción a `enum` y la sección 8.9 para detalles adicionales sobre `enum`].

Buena práctica de programación 10.1

De acuerdo con el capítulo 9 de la Especificación del lenguaje Java, *es un estilo apropiado declarar los métodos de una interfaz sin las palabras clave `public` y `abstract`, ya que son redundantes en las declaraciones de los métodos de la interfaz. De manera similar, las constantes deben declararse sin las palabras clave `public`, `static` y `final`, ya que también son redundantes.*

Uso de una interfaz

Para utilizar una interfaz, una clase concreta debe especificar que implementa (`implements`) a esa interfaz y debe declarar cada uno de sus métodos con la firma especificada en la declaración de la interfaz. Para especificar que una clase implementa a una interfaz, agregamos la palabra clave `implements` y el nombre de la interfaz al final de la primera línea de la declaración de nuestra clase. Una clase que no implementa a *todos* los métodos de la interfaz es una clase *abstracta*, y debe declararse como `abstract`. Implementar una interfaz es como firmar un *contrato* con el compilador que diga, "Declararé todos los métodos especificados por la interfaz, o declararé mi clase como `abstract`".

Error común de programación 10.6

Si no declaramos ningún miembro de una interfaz en una clase concreta que implemente (`implements`) a esa interfaz, se produce un error de compilación indicando que la clase debe declararse como `abstract`.

Relacionar tipos dispares

Por lo general, una interfaz se utiliza cuando clases dispares (es decir, no relacionadas) necesitan compartir métodos y constantes comunes. Esto permite que los objetos de clases no relacionadas se procesen en forma polimórfica; los objetos de clases que implementan la misma interfaz pueden responder a las mismas llamadas a métodos. Usted puede crear una interfaz que describa la funcionalidad deseada y después implementar esta interfaz en cualquier clase que requiera esa funcionalidad. Por ejemplo, en la aplicación de cuentas por pagar que desarrollaremos en esta sección, implementamos la interfaz `PorPagar` en cualquier clase que deba tener la capacidad de calcular el monto de un pago (por ejemplo, `Empleado`, `Factura`).

Comparación entre interfaces y clases abstractas

A menudo, una interfaz se utiliza en vez de una clase `abstract` *cuando no hay una implementación predeterminada que heredar*; esto es, no hay campos ni implementaciones de métodos predeterminados. Al igual que las clases `public abstract`, las interfaces son comúnmente de tipo `public`. De igual forma que una clase `public`, una interfaz `public` se debe declarar en un archivo con el mismo nombre que la interfaz, y con la extensión de archivo `.java`.

Etiquetado de interfaces

En el capítulo 17, Archivos, flujos y serialización de objetos (en el sitio Web del libro), veremos la noción de "etiquetado de interfaces": interfaces vacías que *no* tienen métodos ni valores constantes. Se utilizan para agregar relaciones del tipo *es un* a las clases. Por ejemplo en el capítulo 17 también veremos un mecanismo conocido como serialización de objetos, el cual puede convertir objetos a representaciones de bytes, y puede convertir esas representaciones de bytes de vuelta en objetos. Para que este mecanismo pueda funcionar con sus objetos, sólo tiene que marcarlos como `Serializables`, para lo cual debe agregar el texto `implements Serializable` al final de la primera línea de la declaración de su clase. Así, todos los objetos de su clase tendrán la relación *es un* con `Serializable`.

10.7.1 Desarrollo de una jerarquía PorPagar

Para crear una aplicación que pueda determinar los pagos para los empleados y facturas por igual, primero crearemos una interfaz llamada `Porpagar`, la cual contiene el método `obtenerMontoPago`, que devuelve un monto `double` que debe pagarse para un objeto de cualquier clase que implemente a la interfaz. El método `obtenerMontoPago` es una versión de propósito general del método `ingresos` de la jerarquía de `Empleado`; el método `ingresos` calcula un monto de pago específicamente para un `Empleado`, mientras que `obtenerMontoPago` puede aplicarse a un amplio rango de objetos no relacionados. Después de declarar la interfaz `PorPagar` presentaremos la clase `Factura`, la cual implementa a la interfaz `PorPagar`. Luego modificaremos la clase `Empleado` de tal forma que también implemente a la interfaz `PorPagar`. Por último, actualizaremos la subclase `EmpleadoAsalariado` de `Empleado` para "ajustarla" en la jerarquía de `PorPagar`; para ello cambiaremos el nombre del método `ingresos` de `EmpleadoAsalariado` por el de `obtenerMontoPago`.

Buena práctica de programación 10.2

Al declarar un método en una interfaz, seleccione un nombre para el método que describa su propósito en forma general, *ya que podría implementarse por muchas clases no relacionadas.*

Las clases `Factura` y `Empleado` representan cosas para las cuales la compañía debe calcular un monto a pagar. Ambas clases implementan la interfaz `PorPagar`, por lo que un programa puede invocar al método `obtenerMontoPago` en objetos `Factura` y `Empleado` por igual. Como pronto veremos, esto permite el procesamiento polimórfico de objetos `Factura` y `Empleado` requerido para la aplicación de cuentas por pagar de nuestra compañía.

El diagrama de clases de UML en la figura 10.10 muestra la jerarquía utilizada en nuestra aplicación de cuentas por pagar. La jerarquía comienza con la interfaz `PorPagar`. UML diferencia a una interfaz de otras clases colocando la palabra "interface" entre los signos « y », por encima del nombre de la interfaz. UML expresa la relación entre una clase y una interfaz a través de una relación conocida como **realización**. Se dice que una clase "realiza", o implementa, los métodos de una interfaz. Un diagrama de clases modela una realización como una flecha punteada con punta hueca, que parte de la clase que realizará la implementación, hasta la interfaz. El diagrama en la figura 10.10 indica que cada una de las clases `Factura` y `Empleado` pueden realizar (es decir, implementar) la interfaz `PorPagar`.

Al igual que en el diagrama de clases de la figura 10.2, la clase `Empleado` aparece en cursivas, lo cual indica que es una clase abstracta. La clase concreta `EmpleadoAsalariado` extiende a `Empleado` y hereda la relación de realización de su superclase con la interfaz `PorPagar`.

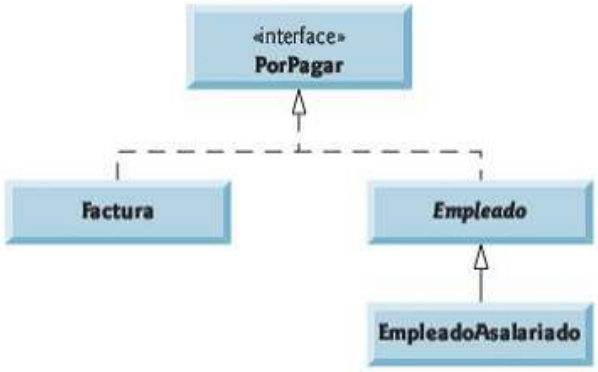


Fig. 10.10 | Diagrama de clases de UML de la jerarquía de la interfaz `PorPagar`.

10.7.2 La interfaz PorPagar

La declaración de la interfaz `PorPagar` empieza en la figura 10.11, línea 4. La interfaz `PorPagar` contiene el método `public abstract obtenerMontoPago` (línea 6). Este método no puede declararse en forma explícita como `public` o `abstract`. Los métodos de una interfaz siempre son `public` y `abstract`, por lo cual no necesitan declararse como tales. La interfaz `PorPagar` sólo tiene un método; las interfaces pueden tener cualquier número de métodos. Además, el método `obtenerMontoPago` no tiene parámetros, pero los métodos de las interfaces *pueden* tenerlos. Las interfaces también pueden contener campos que sean implícitamente `final` y `static`.

```
1  // Fig. 10.11: PorPagar.java
2  // Declaración de la interfaz PorPagar.
3
4  public interface PorPagar
5  {
6     double obtenerMontoPago(); // calcula el pago; no hay implementación
7  } // fin de la interfaz PorPagar
```

Fig. 10.11 | Declaración de la interfaz PorPagar.

10.7.3 La clase Factura

Ahora crearemos la clase `Factura` (figura 10.12) para representar una factura simple que contiene información de facturación para cierto tipo de pieza. La clase declara las variables de instancia `private numeroPieza`, `descripcionPieza`, `cantidad` y `precioPorArticulo` (líneas 6 a 9), las cuales indican el número de pieza, su descripción, la cantidad de piezas ordenadas y el precio por artículo. La clase `Factura` también contiene un constructor (líneas 12 a 19), métodos *obtener* y *establecer* (líneas 22 a 74) que manipulan las variables de instancia de la clase y un método `toString` (líneas 73 a 83) que devuelve una representación `String` de un objeto `Factura`. Los métodos `establecerCantidad` (líneas 46 a 52) y `establecerPrecioPorArticulo` (líneas 61 a 68) aseguran que `cantidad` y `precioPorArticulo` obtengan sólo valores no negativos.

```
1  // Fig. 10.12: Factura.java
2  // La clase Factura implementa a PorPagar.
3
4  public class Factura implements PorPagar
5  {
6     private String numeroPieza;
7     private String descripcionPieza;
8     private int cantidad;
9     private double precioPorArticulo;
10
11    // constructor con cuatro argumentos
12    public Factura( String pieza, String descripcion, int cuenta,
13       double precio )
14    {
15       numeroPieza = pieza;
16       descripcionPieza = descripcion;
17       establecerCantidad( cuenta ); // valida y almacena la cantidad
18       establecerPrecioPorArticulo( precio ); // valida y almacena el precio por
                                                   articulo
19    } // fin del constructor de Factura con cuatro argumentos
20
21    // establece el número de pieza
22    public void establecerNumeroPieza( String pieza )
23    {
24       numeroPieza = pieza;  // debería validar
25    } // fin del método establecerNumeroPieza
26
27    // obtiene el número de pieza
28    public String obtenerNumeroPieza()
29    {
30       return numeroPieza;
31    } // fin del método obtenerNumeroPieza
32
33    // establece la descripción
34    public void establecerDescripcionPieza( String descripcion )
35    {
36       descripcionPieza = descripcion;  // debería validar
37    } // fin del método establecerDescripcionPieza
38
39    // obtiene la descripción
40    public String obtenerDescripcionPieza()
41    {
42       return descripcionPieza;
43    } // fin del método obtenerDescripcionPieza
44
45    // establece la cantidad
46    public void establecerCantidad( int cuenta )
47    {
48       if ( cuenta >= 0 )
49          cantidad = cuenta;
50       else
51          throw new IllegalArgumentException ( "Cantidad debe ser >= 0" );
52    } // fin del método establecerCantidad
53
```

Fig. 10.12 | La clase `Factura`, que implementa a `Porpagar` (parte 1 de 2).

```
54      // obtener cantidad
55      public int obtenerCantidad()
56      {
57         return cantidad;
58      } // fin del método obtenerCantidad
59
60      // establece el precio por artículo
61      public void establecerPrecioPorArticulo( double precio )
62      {
63         if ( precio >= 0.0 )
64            precioPorArticulo = precio;
65         else
66            throw new IllegalArgumentException(
67               "El precio por articulo debe ser >= 0" );
68      } // fin del método establecerPrecioPorArticulo
69
70      // obtiene el precio por artículo
71      public double obtenerPrecioPorArticulo()
72      {
73         return precioPorArticulo;
74      } // fin del método obtenerPrecioPorArticulo
75
76      // devuelve representación String de un objeto Factura
77      @Override
78      public String toString()
79      {
80         return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
81            "factura", "numero de pieza", obtenerNumeroPieza(), obtenerDescripcionPieza(),
82            "cantidad", obtenerCantidad(), "precio por articulo",
               obtenerPrecioPorArticulo() );
83      } // fin del método toString
84
85      // método requerido para realizar el contrato con la interfaz PorPagar
86      @Override
87      public double obtenerMontoPago()
88      {
89         return obtenerCantidad() * obtenerPrecioPorArticulo(); // calcula el costo total
90      } // fin del método obtenerMontoPago
91   } // fin de la clase Factura
```

Fig. 10.12 | La clase `Factura`, que implementa a `Porpagar` (parte 2 de 2).

La línea 4 indica que la clase `Factura` implementa a la interfaz `PorPagar`. Al igual que todas las clases, la clase `Factura` también extiende a `Object` de manera implícita. Java no permite que las subclases hereden de más de una superclase, pero sí que una clase herede de una superclase e implemente tantas interfaces como necesite. Para implementar más de una interfaz, utilice una lista separada por comas de nombres de interfaz después de la palabra clave `implements` en la declaración de la clase, como se muestra a continuación:

```
public class NombreClase extends NombreSuperClase implements PrimeraInterfaz,
    SegundaInterfaz, ...
```

Observación de ingeniería de software 10.7

Todos los objetos de una clase que implementan varias interfaces tienen la relación es un *con cada tipo de interfaz implementada.*

La clase `Factura` implementa el único método de la interfaz `PorPagar`. El método `obtenerMontoPago` se declara en las líneas 86 a 90. Este método calcula el pago total requerido para pagar la factura. El método multiplica los valores de cantidad y `precioPorArticulo` (que se obtienen a través de los métodos *obtener* apropiados) y devuelve el resultado (línea 89). Este método cumple con el requerimiento de implementación del mismo en la interfaz `PorPagar`; hemos cumplido el contrato de interfaz con el compilador.

10.7.4 Modificación de la clase `Empleado` para implementar la interfaz `PorPagar`

Ahora modificaremos la clase `Empleado` para que implemente la interfaz `PorPagar`. La figura 10.13 contiene la clase `Empleado` modificada, la cual es idéntica a la de la figura 10.4, con sólo dos excepciones. En primer lugar, la línea 4 de la figura 10.13 indica que la clase `Empleado` ahora implementa a la interfaz `PorPagar`. Por ende, debemos cambiar el nombre de `ingresos` por el de `obtenerMontoPago` en toda la jerarquía de `Empleado`. Sin embargo, al igual que con el método `ingresos` en la versión de la clase `Empleado` de la figura 10.4, no tiene sentido implementar el método `obtenerMontoPago` en la clase `Empleado`, ya que no podemos calcular el pago de los ingresos para un `Empleado` general; primero debemos conocer el tipo específico de `Empleado`. En la figura 10.4 declaramos el método `ingresos` como `abstract` por esta razón y, como resultado, la clase `Empleado` tuvo que declararse como `abstract`. Esto obliga a cada clase derivada de `Empleado` a sobrescribir el método `ingresos` con una implementación.

```
1  // Fig. 10.13: Empleado.java
2  // La superclase abstracta Empleado que implementa a PorPagar.
3
4  public abstract class Empleado implements PorPagar
5  {
6     private String primerNombre;
7     private String apellidoPaterno;
8     private String numeroSeguroSocial;
9
10    // constructor con tres argumentos
11    public Empleado( String nombre, String apellido, String nss )
12    {
13       primerNombre = nombre;
14       apellidoPaterno = apellido;
15       numeroSeguroSocial = nss;
16    } // fin del constructor de Empleado con tres argumentos
17
18    // establece el primer nombre
19    public void establecerPrimerNombre( String nombre )
20    {
21       primerNombre = nombre;  // debería validar
22    } // fin del método establecerPrimerNombre
23
24    // devuelve el primer nombre
25    public String obtenerPrimerNombre()
26    {
27       return primerNombre;
28    } // fin del método obtenerPrimerNombre
29
```

Fig. 10.13 | La clase `Empleado`, que implementa a `PorPagar` (parte 1 de 2).

```
30      // establece el apellido paterno
31      public void establecerApellidoPaterno( String apellido )
32      {
33         apellidoPaterno = apellido;  // debería validar
34      } // fin del método establecerApellidoPaterno
35
36      // devuelve el apellido paterno
37      public String obtenerApellidoPaterno()
38      {
39         return apellidoPaterno;
40      } // fin del método obtenerApellidoPaterno
41
42      // establece el número de seguro social
43      public void establecerNumeroSeguroSocial( String nss )
44      {
45         numeroSeguroSocial = nss; // debe validar
46      } // fin del método establecerNumeroSeguroSocial
47
48      // devuelve el número de seguro social
49      public String obtenerNumeroSeguroSocial()
50      {
51         return numeroSeguroSocial;
52      } // fin del método obtenerNumeroSeguroSocial
53
54      // devuelve representación String de un objeto Empleado
55      @Override
56      public String toString()
57      {
58         return String.format( "%s %s\nnumero de seguro social: %s",
59            obtenerPrimerNombre(), obtenerApellidoPaterno(), obtenerNumeroSeguroSocial() );
60      } // fin del método toString
61
62      // Nota: Aquí no implementamos el método obtenerMontoPago de PorPagar, así que
63      // esta clase debe declararse como abstract para evitar un error de compilación.
64   } // fin de la clase abstracta Empleado
```

Fig. 10.13 | La clase `Empleado`, que implementa a `PorPagar` (parte 2 de 2).

En la figura 10.13, manejamos esta situación en forma distinta. Recuerde que cuando una clase implementa a una interfaz, hace un *contrato* con el compilador, en el que se establece que la clase implementará *cada uno* de los métodos en la interfaz, o de lo contrario la clase se declara como `abstract`. Si se elige la última opción, no necesitamos declarar los métodos de la interfaz como `abstract` en la clase abstracta; ya están declarados como tales de manera implícita en la interfaz. Cualquier subclase concreta de la clase `abstract` debe implementar a los métodos de la interfaz para cumplir con el contrato de la superclases con el compilador. Si la subclase no lo hace, también debe declararse como `abstract`. Como lo indican los comentarios en las líneas 62 y 63, la clase `Empleado` de la figura 10.13 *no* implementa al método `obtenerMontoPago`, por lo que la clase se declara como `abstract`. Cada subclase directa de `Empleado` *hereda el contrato de la superclase* para implementar el método `obtenerMontoPago` y, por ende, debe implementar este método para convertirse en una clase concreta, para la cual puedan crearse instancias de objetos. Una clase que extienda a una de las subclases concretas de `Empleado` heredará una implementación de `obtenerMontoPago` y, por ende, también será una clase concreta.

10.7.5 Modificación de la clase EmpleadoAsalariado para usarla en la jerarquía PorPagar

La figura 10.14 contiene una versión modificada de la clase EmpleadoAsalariado, que extiende a Empleado y cumple con el contrato de la superclase Empleado para implementar el método obtenerMontoPago de la interfaz PorPagar. Esta versión de EmpleadoAsalariado es idéntica a la de la figura 10.5, con la excepción de que reemplaza el método ingresos con el método obtenerMontoPago (líneas 34 a 38). Recuerde que la versión de PorPagar del método tiene un nombre más *general* para que pueda aplicarse a clases que sean posiblemente *dispares*. El resto de las subclases de Empleado (EmpleadoPorHoras, EmpleadoPorComision y EmpleadoBaseMasComision) también deben modificarse para que contengan el método obtenerMontoPago en vez de ingresos, y así reflejar el hecho de que ahora Empleado implementa a PorPagar. Dejaremos estas modificaciones como un ejercicio (ejercicio 10.11) y sólo utilizaremos a EmpleadoAsalariado en nuestro programa de prueba en esta sección. El ejercicio 10.12 le pide que implemente la interfaz PorPagar en toda la jerarquía de la clase Empleado de las figuras 10.4 a 10.9, sin modificar las subclases de Empleado.

Cuando una clase implementa a una interfaz, se aplica la misma relación *es un* que proporciona la herencia. Por ejemplo, la clase Empleado implementa a PorPagar, por lo que podemos decir que un objeto Empleado *es un* objeto PorPagar. De hecho, los objetos de cualquier clase que extienda a Empleado son también objetos PorPagar. Por ejemplo, los objetos EmpleadoAsalariado son objetos PorPagar. Los objetos de cualquier subclase de la clase que implementa (implements) a la interfaz también pueden considerarse como objetos del tipo de la interfaz. Por ende, así como podemos asignar la referencia de un objeto EmpleadoAsalariado a una variable de la superclase Empleado, también podemos asignar la referencia de un objeto EmpleadoAsalariado a una variable de la interfaz PorPagar. Factura implementa a PorPagar, por lo que un objeto Factura también *es un* objeto PorPagar, y podemos asignar la referencia de un objeto Factura a una variable PorPagar.

Observación de ingeniería de software 10.8

Cuando el parámetro de un método se declara con un tipo de superclase o de interfaz, el método procesa en forma polimórfica al objeto que recibe como argumento.

Observación de ingeniería de software 10.9

Al utilizar una referencia a la superclase, podemos invocar de manera polimórfica a cualquier método declarado en la superclase y sus superclases (por ejemplo, en la clase Object). Al utilizar una referencia a la interfaz, podemos invocar de manera polimórfica a cualquier método declarado en la interfaz, en sus superinterfaces (una interfaz puede extender a otra) y en la clase Object; una variable de un tipo de interfaz debe hacer referencia a un objeto para llamar a los métodos, y todos los objetos contienen los métodos de la clase Object.

```
1  // Fig. 10.14: EmpleadoAsalariado.java
2  // La clase EmpleadoAsalariado extiende a Empleado, que implementa a PorPagar.
3
4  public class EmpleadoAsalariado extends Empleado
5  {
6     private double salarioSemanal;
7
8     // constructor con cuatro argumentos
9     public EmpleadoAsalariado( String nombre, String apellido, String nss,
10       double salario )
11    {
```

Fig. 10.14 | La clase EmpleadoAsalariado, que implementa el método obtenerMontoPago de la interfaz PorPagar (parte 1 de 2).

```
12          super( nombre, apellido, nss ); // pasa argumentos al constructor de Empleado
13          establecerSalarioSemanal( salario ); // valida y almacena el salario
14       } // fin del constructor de EmpleadoAsalariado con cuatro argumentos
15
16       // establece el salario
17       public void establecerSalarioSemanal( double salario )
18       {
19          if (salario >= 0.0 )
20             salarioSemanal = salario;
21          else
22             throw new IllegalArgumentException(
23                "El salario semanal debe ser >= 0.0" );
24       } // fin del método establecerSalarioSemanal
25
26       // devuelve el salario
27       public double obtenerSalarioSemanal()
28       {
29          return salarioSemanal;
30       } // fin del método obtenerSalarioSemanal
31
32       // calcula los ingresos; implementa el método de la interfaz PorPagar
33       // que era abstracto en la superclase Empleado
34       @Override
35       public double obtenerMontoPago()
36       {
37          return obtenerSalarioSemanal();
38       } // fin del método obtenerMontoPago
39
40       // devuelve representación String de un objeto EmpleadoAsalariado
41       @Override
42       public String toString()
43       {
44          return String.format( "empleado asalariado: %s\n%s: $%,.2f",
45             super.toString(), "salario semanal", obtenerSalarioSemanal() );
46       } // fin del método toString
47    } // fin de la clase EmpleadoAsalariado
```

Fig. 10.14 | La clase `EmpleadoAsalariado`, que implementa el método `obtenerMontoPago` de la interfaz `PorPagar` (parte 2 de 2).

10.7.6 Uso de la interfaz PorPagar para procesar objetos Factura y Empleado mediante el polimorfismo

`PruebaInterfazPorPagar` (figura 10.15) ilustra que la interfaz `PorPagar` puede usarse para procesar un conjunto de objetos `Factura` y `Empleado` en forma polimórfica en una sola aplicación. La línea 9 declara a `objetosPorPagar` y le asigna un arreglo de cuatro variables `PorPagar`. Las líneas 12 y 13 asignan las referencias de objetos `Factura` a los primeros dos elementos de `objetosPorPagar`. Después, las líneas 14 a 17 asignan las referencias de objetos `EmpleadoAsalariado` a los dos elementos restantes de `objetosPorPagar`. Estas asignaciones se permiten debido a que un objeto `Factura` *es un* objeto `PorPagar`, un `EmpleadoAsalariado` *es un* `Empleado`, y un `Empleado` *es un* objeto `PorPagar`. Las líneas 23 a 29 utilizan una instrucción `for` mejorada para procesar cada objeto `PorPagar` en `objetosPorPagar` de manera polimórfica,e imprime en pantalla el objeto como un `String`, junto con el pago vencido.

La línea 27 invoca al método `toString` desde una referencia de la interfaz `PorPagar`, aun cuando `toString` no se declara en la interfaz `PorPagar`; *todas las referencias (entre ellas las de los tipos de interfaces) se refieren a objetos que extienden a `Object` y, por lo tanto, tienen un método `toString`.* (Aquí también podemos invocar a `toString` en forma implícita.) La línea 28 invoca al método `obtenerMontoPago` de `PorPagar` para obtener el monto a pagar para cada objeto en `objetosPorPagar`, sin importar el tipo actual del objeto. Los resultados revelan que las llamadas a los métodos en las líneas 27 y 28 invocan a la implementación de la clase apropiada de los métodos `toString` y `obtenerMontoPago`. Por ejemplo, cuando `empleadoActual` hace referencia a un objeto `Factura` durante la primera iteración del ciclo `for`, se ejecutan los métodos `toString` y `obtenerMontoPago` de la clase `Factura`.

```
1  // Fig. 10.15: PruebaInterfazPorPagar.java
2  // Prueba la interfaz PorPagar.
3
4  public class PruebaInterfazPorPagar
5  {
6     public static void main( String[] args )
7     {
8        // crea arreglo PorPagar con cuatro elementos
9        PorPagar[] objetosPorPagar = new PorPagar[ 4 ];
10
11       // llena el arreglo con objetos que implementan la interfaz PorPagar
12       objetosPorPagar[ 0 ] = new Factura( "01234", "asiento", 2, 375.00 );
13       objetosPorPagar[ 1 ] = new Factura( "56789", "llanta", 4, 79.95 );
14       objetosPorPagar[ 2 ] =
15          new EmpleadoAsalariado( "John", "Smith", "111-11-1111", 800.00 );
16       objetosPorPagar[ 3 ] =
17          new EmpleadoAsalariado( "Lisa", "Barnes", "888-88-8888", 1200.00 );
18
19       System.out.println(
20          "Facturas y Empleados procesados en forma polimorfica:\n" );
21
22       // procesa en forma genérica cada elemento en el arreglo objetosPorPagar
23       for ( PorPagar porPagarActual : objetosPorPagar )
24       {
25          // imprime porPagarActual y su monto de pago apropiado
26          System.out.printf( "%s \n%s: $%,.2f\n\n",
27             porPagarActual.toString(),
28             "pago vencido", porPagarActual.obtenerMontoPago() );
29       } // fin de for
30    } // fin de main
31 } // fin de la clase PruebaInterfazPorPagar
```

```
Facturas y Empleados procesados en forma polimorfica:

factura:
numero de pieza: 01234 (asiento)
cantidad: 2
precio por articulo: $375.00
pago vencido: $750.00
```

Fig. 10.15 | Programa de prueba de la interfaz `PorPagar`, que procesa objetos `Factura` y `Empleado` de manera polimórfica (parte 1 de 2).

```
factura:
numero de pieza: 56789 (llanta)
cantidad: 4
precio por articulo: $79.95
pago vencido: $319.80

empleado asalariado: John Smith
numero de seguro social: 111-11-1111
salario semanal: $800.00
pago vencido: $800.00

empleado asalariado: Lisa Barnes
numero de seguro social: 888-88-8888
salario semanal: $1,200.00
pago vencido: $1,200.00
```

Fig. 10.15 | Programa de prueba de la interfaz `PorPagar`, que procesa objetos `Factura` y `Empleado` de manera polimórfica (parte 2 de 2).

10.7.7 Interfaces comunes de la API de Java

En esta sección veremos las generalidades acerca de varias interfaces comunes que se encuentran en la API de Java. El poder y la flexibilidad de las interfaces se utilizan con frecuencia a lo largo de la API de Java. Estas interfaces se implementan y usan de la misma forma que las que usted crea (por ejemplo, la interfaz `PorPagar` en la sección 10.7.2). Las interfaces de la API de Java le permiten utilizar sus propias clases dentro de los marcos de trabajo que proporciona Java, como el comparar objetos de sus propios tipos y crear tareas que se ejecuten de manera concurrente con otras tareas en el mismo programa. La figura 10.16 presenta una breve sinopsis de las interfaces más populares de la API de Java que utilizamos en este libro.

Interfaz	Descripción
`Comparable`	Java contiene varios operadores de comparación (`<`, `<=`, `>`, `>=`, `==`, `!=`) que nos permiten comparar valores primitivos. Sin embargo, estos operadores *no* se pueden utilizar para comparar objetos. La interfaz `Comparable` se utiliza para permitir que los objetos de una clase que implementa a la interfaz se comparen entre sí. La interfaz `Comparable` se utiliza comúnmente para ordenar objetos en una colección, como un arreglo. En el capítulo 20 y en el capítulo 21 (ambos en inglés en el sitio Web del libro), utilizaremos a `Comparable`.
`Serializable`	Una interfaz que se utiliza para identificar clases cuyos objetos pueden escribirse en (serializarse), o leerse desde (deserializarse) algún tipo de almacenamiento (archivo en disco, campo de base de datos) o transmitirse a través de una red. En el capítulo 17, Archivos, flujos y serialización de objetos, y en el capítulo 27 (en inglés en el sitio Web del libro), utilizaremos a `Serializable`.
`Runnable`	La implementa cualquier clase para la cual sus objetos deban poder ejecutarse en paralelo, usando una técnica llamada multihilos, que veremos en el capítulo 26 (en inglés en el sitio Web del libro). La interfaz contiene un método, `run`, que describe el comportamiento de un objeto al ejecutarse.

Fig. 10.16 | Interfaces comunes de la API de Java (parte 1 de 2).

Interfaz	Descripción
Interfaces de escucha de eventos de la GUI	Usted trabaja con interfaces gráficas de usuario (GUI) a diario. Por ejemplo, en su navegador Web, podría escribir en un campo de texto la dirección de un sitio Web para visitarlo, o podría hacer clic en un botón para regresar al sitio anterior que visitó. El navegador Web responde a su interacción y realiza la tarea que usted desea. Su interacción se conoce como un evento, y el código que utiliza el navegador para responder a un evento se conoce como manejador de eventos. En el capítulo 14 Componentes de la GUI: Parte 1 (en el sitio Web del libro), y en el capítulo 25 (en inglés, en el sitio Web), aprenderá a crear interfaces tipo GUI en Java y manejadores de eventos para responder a las interacciones del usuario. Éstos se declaran en clases que implementan una interfaz de escucha de eventos apropiada. Cada interfaz de escucha de eventos especifica uno o más métodos que deben implementarse para responder a las interacciones de los usuarios.
`SwingConstants`	Contiene un conjunto de constantes que se utilizan en la programación de GUI para posicionar los elementos de la GUI en la pantalla. En los capítulos 14 y 25 (en el sitio Web del libro) exploraremos la programación de GUI.

Fig. 10.16 | Interfaces comunes de la API de Java (parte 2 de 2).

10.8 (Opcional) Caso de estudio de GUI y gráficos: realizar dibujos usando polimorfismo

Tal vez haya observado en el programa de dibujo que creamos en el ejercicio 8.1 del caso de estudio de GUI y gráficos (y que modificamos en el ejercicio 9.1 del caso de estudio de GUI y gráficos) que existen muchas similitudes entre las clases de figuras. Mediante la herencia, podemos "factorizar" las características comunes de las tres clases y colocarlas en una sola superclase de figura. Después, podemos manipular objetos de los tres tipos de figuras en forma polimórfica, usando variables del tipo de la superclase. Al eliminar la redundancia en el código se producirá un programa más pequeño y flexible, que será más fácil de mantener.

Ejercicios del caso de estudio de GUI y gráficos

10.1 Modifique las clases `MiLinea`, `MiOvalo` y `MiRectangulo` de los ejercicios 8.1 y 9.1 del caso de estudio de GUI y gráficos, para crear la jerarquía de clases de la figura 10.17. Las clases de la jerarquía `MiFigura` deben ser clases de figuras "inteligentes", que sepan cómo dibujarse a sí mismas (si se les proporciona un objeto `Graphics` que les indique en dónde deben dibujarse). Una vez que el programa cree un objeto a partir de esta jerarquía, podrá manipularlo de manera polimórfica por el resto de su duración como un objeto `MiFigura`.

En su solución, la clase `MiFigura` en la figura 10.17 *debe* ser `abstract`. Como `MiFigura` representa a cualquier figura en general, no es posible implementar un método `dibujar` sin saber exactamente qué figura es. Los datos que representan las coordenadas y el color de las figuras en la jerarquía deben declararse como miembros `private` de la clase `MiFigura`. Además de los datos comunes, la clase `MiFigura` debe declarar los siguientes métodos:

a) Un constructor sin argumentos que establezca todas las coordenadas de la figura en `0`, y el color en `Color.BLACK`.
b) Un constructor que inicialice las coordenadas y el color con los valores de los argumentos suministrados.
c) Métodos *establecer* para las coordenadas individuales y el color, que permitan al programador establecer cualquier pieza de datos de manera independiente, para una figura en la jerarquía.
d) Métodos *obtener* para las coordenadas individuales y el color, que permitan al programador obtener cualquier pieza de datos de manera independiente, para una figura en la jerarquía.

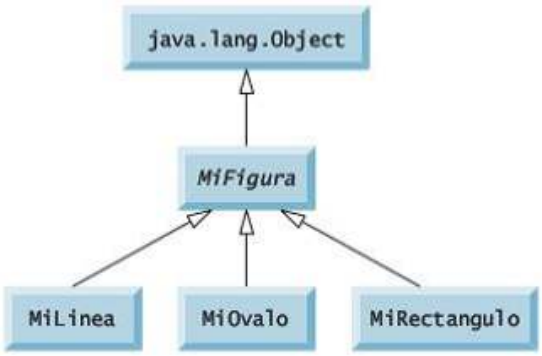


Fig. 10.17 | La jerarquía `MiFigura`.

e) El método `abstract`.

```
public abstract void draw( Graphics g );
```

que se llamará desde el método `paintComponent` del programa para dibujar una figura en la pantalla.

Para asegurar un correcto encapsulamiento, todos los datos en la clase `MiFigura` deben ser `private`. Para esto se requiere declarar métodos *establecer* y *obtener* apropiados para manipular los datos. La clase `MiLinea` debe proporcionar un constructor sin argumentos y uno con argumentos para las coordenadas y el color. Las clases `MiOvalo` y `MiRectangulo` deben proporcionar un constructor sin argumentos y uno con argumentos para las coordenadas, el color y para determinar si la figura es rellena. El constructor sin argumentos debe, además, establecer los valores predeterminados, y la figura como una figura sin relleno.

Puede dibujar líneas, rectángulos y óvalos si conoce dos puntos en el espacio. Las líneas requieren coordenadas *x1*, *y1*, *x2* y *y2*. El método `drawLine` de la clase `Graphics` conectará los dos puntos suministrados con una línea. Si tiene los mismos cuatro valores de coordenadas (*x1*, *y1*, *x2* y *y2*) para óvalos y rectángulos, puede calcular los cuatro argumentos necesarios para dibujarlos. Cada uno requiere un valor de coordenada *x* superior izquierda (el menor de los dos valores de coordenada *x*), un valor de coordenada *y* superior izquierda (el menor de los dos valores de coordenada *y*), una *anchura* (el valor absoluto de la diferencia entre los dos valores de coordenada *x*) y una *altura* (el valor absoluto de la diferencia entre los dos valores de coordenada *y*). Los rectángulos y óvalos también deben tener una bandera `relleno`, que determine si se dibujará la figura con un relleno.

No debe haber variables `MiLinea`, `MiOvalo` o `MiRectangulo` en el programa; sólo variables `MiFigura` que contengan referencias a objetos `MiLinea`, `MiOvalo` y `MiRectangulo`. El programa debe generar figuras aleatorias y almacenarlas en un arreglo de tipo `MiFigura`. El método `paintComponent` debe recorrer el arreglo `MiFigura` y dibujar cada una de las figuras (es decir, mediante una llamada polimórfica al método `dibujar` de cada figura).

Permita al usuario que especifique (mediante un diálogo de entrada) el número de figuras a generar. Después, el programa generará y mostrará las figuras en pantalla, junto con una barra de estado para informar al usuario cuántas figuras de cada tipo se crearon.

10.2 ***(Modificación de la aplicación de dibujo)*** En el ejercicio 10.1, usted creó una jerarquía `MiFigura` en la cual las clases `MiLinea`, `MiOvalo` y `MiRectangulo` extienden a `MiFigura` directamente. Si su jerarquía estuviera diseñada de manera apropiada, debería poder ver las similitudes entre las clases `MiOvalo` y `MiRectangulo`. Rediseñe y vuelva a implementar el código de las clases `MiOvalo` y `MiRectangulo`, para "factorizar" las características comunes en la clase abstracta `MiFiguraDelimitada`, para producir la jerarquía de la figura 10.18.

La clase `MiFiguraDelimitada` debe declarar dos constructores que imiten a los de `MiFigura`, sólo con un parámetro adicional para ver si la figura es rellena. La clase `MiFiguraDelimitada` también debe declarar métodos *obtener* y *establecer* para manipular la bandera de relleno y los métodos que calculan la coordenada *x* superior izquierda, la coordenada *y* superior izquierda, la anchura y la altura. Recuerde que los valores necesarios para dibujar un óvalo o un rectángulo se

pueden calcular a partir de dos coordenadas (*x*, *y*). Si se diseñan de manera apropiada, las nuevas clases `MiOvalo` y `MiRectangulo` deberán tener dos constructores y un método `dibujar` cada una.

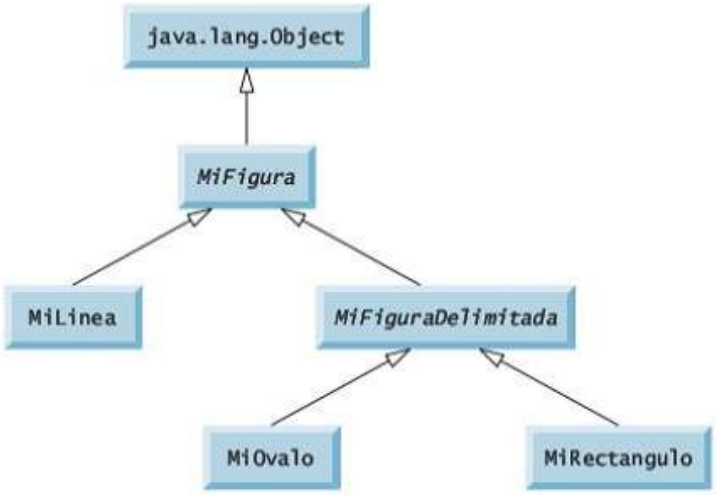


Fig. 10.18 | Jerarquía `MiFigura` con `MiFiguraDelimitada`.

10.9 Conclusión

En este capítulo se introdujo el polimorfismo: la habilidad de procesar objetos que comparten la misma superclase en una jerarquía de clases, como si todos fueran objetos de la superclase. También hablamos sobre cómo el polimorfismo facilita la extensibilidad y el mantenimiento de los sistemas, y después demostramos cómo utilizar métodos sobrescritos para llevar a cabo el comportamiento polimórfico. Presentamos la noción de las clases abstractas, las cuales permiten a los programadores proporcionar una superclase apropiada, a partir de la cual otras clases pueden heredar. Aprendió que una clase abstracta puede declarar métodos abstractos que cada una de sus subclases debe implementar para convertirse en clase concreta, y que un programa puede utilizar variables de una clase abstracta para invocar implementaciones en las subclases de los métodos abstractos en forma polimórfica. También aprendió a determinar el tipo de un objeto en tiempo de ejecución. Vimos los conceptos de los métodos y clases `final`. Por último, hablamos también sobre la declaración e implementación de una interfaz, como otra manera de obtener el comportamiento polimórfico.

Ahora deberá estar familiarizado con las clases, los objetos, el encapsulamiento, la herencia, las interfaces y el polimorfismo: los aspectos más esenciales de la programación orientada a objetos.

En el siguiente capítulo aprenderá sobre las excepciones, que son útiles para manejar errores durante la ejecución de un programa. El manejo de excepciones nos permite generar programas más robustos.

Resumen

Sección 10.1 Introducción

- El polimorfismo (pág. 395) nos permite escribir programas para procesar objetos que compartan la misma superclase, como si todos fueran objetos de la superclase; esto puede simplificar la programación.
- Con el polimorfismo, podemos diseñar e implementar sistemas que puedan extenderse con facilidad. Las únicas partes de un programa que deben alterarse para dar cabida a las nuevas clases son las que requieren un conocimiento directo de las nuevas clases que el programador agregará a la jerarquía.

Sección 10.3 Demostración del comportamiento polimórfico

- Cuando el compilador encuentra una llamada a un método que se realiza a través de una variable, determina si el método puede llamarse verificando el tipo de clase de la variable. Si esa clase contiene la declaración del método apropiada (o hereda una), se compila la llamada. En tiempo de ejecución, el tipo del objeto al cual se refiere la variable es el que determina el método que se utilizará.

Sección 10.4 Clases y métodos abstractos

- Las clases abstractas (pág. 400) no se pueden utilizar para instanciar objetos, ya que están incompletas.
- El propósito principal de una clase abstracta es proporcionar una superclase apropiada, a partir de la cual puedan heredar otras clases y, por ende, compartir un diseño común.
- Las clases que pueden utilizarse para instanciar objetos se llaman clases concretas (pág. 401). Dichas clases proporcionan implementaciones de cada método que declaran (algunas de las implementaciones pueden heredarse).
- Los programadores escriben código cliente que utiliza sólo tipos de superclases abstractas (pág. 401) para reducir las dependencias del código cliente en tipos de subclases específicas.
- Algunas veces las clases abstractas constituyen varios niveles de la jerarquía.
- Por lo general, una clase abstracta contiene uno o más métodos abstractos (pág. 401).
- Los métodos abstractos no proporcionan implementaciones.
- Una clase que contiene métodos abstractos debe declararse como clase `abstracta` (pág. 401). Cada subclase concreta debe proporcionar implementaciones de cada uno de los métodos abstractos de la superclase.
- Los constructores y los métodos `static` no pueden declararse como `abstract`.
- Las variables de las superclases abstractas pueden guardar referencias a objetos de cualquier clase concreta que se derive de esas superclases. Por lo general, los programas utilizan dichas variables para manipular los objetos de las subclases mediante el polimorfismo.
- En especial, el polimorfismo es efectivo para implementar los sistemas de software en capas.

Sección 10.5 Caso de estudio: sistema de nómina utilizando polimorfismo

- El diseñador de una jerarquía de clases puede exigir que cada subclase concreta proporcione una implementación apropiada del método, para lo cual incluye un método `abstract` en una superclase.
- La mayoría de las llamadas a los métodos se resuelven en tiempo de ejecución, con base en el tipo del objeto que se está manipulando. Este proceso se conoce como vinculación dinámica (pág. 416) o vinculación postergada.
- La variable de una superclase puede utilizarse para invocar sólo a los métodos declarados en la superclase.
- El operador `instanceof` (pág. 416) determina si un objeto tiene la relación *es un* con un tipo específico.
- Todos los objetos en Java conocen su propia clase y pueden acceder a esta información a través del método `getClass` de la clase `Object` (pág. 417), el cual devuelve un objeto de tipo `Class` (paquete `java.lang`).
- La relación *es un* se aplica sólo entre la subclase y sus superclases, no viceversa.

Sección 10.6 Métodos y clases `final`

- Un método que se declara como `final` (pág. 418) en una superclase no se puede sobrescribir en una subclase.
- Los métodos que se declaran como `private` son `final` de manera implícita, ya que es imposible sobrescribirlos en una subclase.
- Los métodos que se declaran como `static` son `final` de manera implícita.
- La declaración de un método `final` no puede cambiar, por lo que todas las subclases utilizan la misma implementación del método, y las llamadas a los métodos `final` se resuelven en tiempo de compilación; a esto se le conoce como vinculación estática (pág. 419).
- Como el compilador sabe que los métodos `final` no se pueden sobrescribir, puede optimizar los programas al eliminar las llamadas a los métodos `final` y sustituirlas con el código expandido de sus declaraciones en cada una de las ubicaciones de las llamadas al método; a esta técnica se le conoce como poner el código en línea.
- Una clase que se declara como `final` no puede ser una superclase (pág. 419).
- Todos los métodos en una clase `final` son implícitamente `final`.

Sección 10.7 Caso de estudio: creación y uso de interfaces

- Una interfaz (pág. 419) especifica *qué* operaciones están permitidas, pero no determina *cómo* se realizan.
- Una interfaz de Java describe a un conjunto de métodos que pueden llamarse en un objeto.
- La declaración de una interfaz empieza con la palabra clave `interface` (pág. 420).
- Todos los miembros de una interfaz deben ser `public`, y las interfaces no pueden especificar ningún detalle de implementación, como las declaraciones de métodos concretos y las variables de instancia.
- Todos los métodos que se declaran en una interfaz son `public abstract` de manera implícita, y todos los campos son `public`, `static` y `final` de manera implícita.
- Para utilizar una interfaz, una clase concreta debe especificar que implementa (`implements`; pág. 420) a esa interfaz, y debe declarar cada uno de los métodos de la interfaz con la firma especificada en su declaración. Una clase que no implementa a todos los métodos de una interfaz debe declararse como `abstract`.
- Implementar una interfaz es como firmar un contrato con el compilador que diga, "Declararé todos los métodos especificados por la interfaz, o de lo contrario declararé mi clase como `abstract`".
- Por lo general, una interfaz se utiliza cuando clases dispares (es decir, no relacionadas) necesitan compartir métodos y constantes comunes. Esto permite que los objetos de clases no relacionadas se procesen en forma polimórfica; los objetos de clases que implementan la misma interfaz pueden responder a las mismas llamadas a métodos.
- Usted puede crear una interfaz que describa la funcionalidad deseada, y después implementar esa interfaz en cualquier clase que requiera esa funcionalidad.
- A menudo, una interfaz se utiliza en vez de una clase `abstract` cuando no hay una implementación predeterminada que heredar; esto es, no hay variables de instancia ni implementaciones de métodos predeterminadas.
- Al igual que las clases `public abstract`, las interfaces son comúnmente de tipo `public`, por lo que se declaran en archivos por sí solas con el mismo nombre que la interfaz, y la extensión de archivo `.java`.
- Java no permite que las subclases hereden de más de una superclase, pero sí permite que una clase herede de una superclase e implemente más de una interfaz.
- Todos los objetos de una clase que implementan varias interfaces tienen la relación *es un* con cada tipo de interfaz implementada.
- Una interfaz puede declarar constantes. Las constantes son implícitamente `public`, `static` y `final`.

Ejercicios de autoevaluación

10.1 Complete las siguientes oraciones:

a) Si una clase contiene al menos un método abstracto, es una clase _____ .
b) Las clases a partir de las cuales pueden instanciarse objetos se llaman clases _____ .
c) El _____ implica el uso de una variable de superclase para invocar métodos en objetos de superclase y subclase, lo cual nos permite "programar en general".
d) Los métodos que no son métodos de interfaz y que no proporcionan implementaciones deben declararse utilizando la palabra clave _____ .
e) Al proceso de convertir una referencia almacenada en una variable de una superclase a un tipo de una subclase se le conoce como _____ .

10.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Todos los métodos en una clase `abstract` deben declararse como métodos `abstract`.
b) No está permitido invocar a un método que sólo pertenece a una subclase, a través de una variable de subclase.
c) Si una superclase declara a un método como `abstract`, una subclase debe implementar a ese método.
d) Un objeto de una clase que implementa a una interfaz puede considerarse como un objeto de ese tipo de interfaz.

Respuestas a los ejercicios de autoevaluación

10.1 a) abstracta. b) concretas c) polimorfismo d) `abstract`. e) conversión descendente.

10.2 a) Falso. Una clase abstracta puede incluir métodos con implementaciones y métodos `abstract`. b) Falso. No está permitido tratar de invocar un método que sólo pertenece a una subclase, con una variable de la superclase. c) Falso. Sólo una subclase concreta debe implementar el método. d) Verdadero.

Ejercicios

10.3 ¿Cómo es que el polimorfismo le permite programar "en forma general", en lugar de hacerlo "en forma específica"? Hable sobre las ventajas clave de la programación "en forma general".

10.4 ¿Qué son los métodos abstractos? Describa las circunstancias en las que un método abstracto sería apropiado.

10.5 ¿Cómo es que el polimorfismo fomenta la extensibilidad?

10.6 Describa cuatro formas en las que podemos asignar referencias de superclases y subclases a variables de los tipos de las superclases y las subclases.

10.7 Compare y contraste las clases abstractas y las interfaces. ¿Para qué podría usar una clase abstracta? ¿Para qué podría usar una interfaz?

10.8 ***(Modificación al sistema de nómina)*** Modifique el sistema de nómina de las figuras 10.4 a 10.9 para incluir la variable de instancia `private` llamada `fechaNacimiento` en la clase `Empleado`. Use la clase `Fecha` de la figura 8.7 para representar el cumpleaños de un empleado. Agregue métodos *obtener* a la clase `Fecha`. Suponga que la nómina se procesa una vez al mes. Cree un arreglo de variables `Empleado` para guardar referencias a los diversos objetos empleado. En un ciclo, calcule la nómina para cada `Empleado` (mediante el polimorfismo) y agregue una bonificación de $100.00 a la cantidad de pago de nómina de la persona, si el mes actual es el mes en el que ocurre el cumpleaños de ese `Empleado`.

10.9 ***(Proyecto: Jerarquía de figuras)*** Implemente la jerarquía `Figura` que se muestra en la figura 9.3. Cada `FiguraBidimensional` debe contener el método `obtenerArea` para calcular el área de la figura bidimensional. Cada `FiguraTridimensional` debe tener los métodos `obtenerArea` y `obtenerVolumen` para calcular el área superficial y el volumen, respectivamente, de la figura tridimensional. Cree un programa que utilice un arreglo de referencias `Figura` a objetos de cada clase concreta en la jerarquía. El programa deberá imprimir una descripción de texto del objeto al cual se refiere cada elemento del arreglo. Además, en el ciclo que procesa a todas las figuras en el arreglo, determine si cada figura es `FiguraBidimensional` o `FiguraTridimensional`. Si es `FiguraBidimensional`, muestre su área. Si es `FiguraTridimensional`, muestre su área y su volumen.

10.10 ***(Modificación al sistema de nómina)*** Modifique el sistema de nómina de las figuras 10.4 a 10.9, para incluir una subclase adicional de `Empleado` llamada `TrabajadorPorPiezas`, que represente a un empleado cuyo sueldo se base en el número de piezas de mercancía producidas. La clase `TrabajadorPorPiezas` debe contener las variables de instancia `private` llamadas `sueldo` (para almacenar el sueldo del empleado por pieza) y `piezas` (para almacenar el número de piezas producidas). Proporcione una implementación concreta del método `ingresos` en la clase `TrabajadorPorPiezas` que calcule los ingresos del empleado, multiplicando el número de piezas producidas por el sueldo por pieza. Cree un arreglo de variables `Empleado` para almacenar referencias a objetos de cada clase concreta en la nueva jerarquía `Empleado`. Para cada `Empleado`, muestre su representación de cadena y los ingresos.

10.11 ***(Modificación al sistema de cuentas por pagar)*** En este ejercicio modificaremos la aplicación de cuentas por pagar de las figuras 10.11 a 10.15, para incluir la funcionalidad completa de la aplicación de nómina de las figuras 10.4 a 10.9. La aplicación debe aún procesar dos objetos `Factura`, pero ahora debe procesar un objeto de cada una de las cuatro subclases de `Empleado`. Si el objeto que se está procesando en un momento dado es `EmpleadoBasePorComision`, la aplicación debe incrementar el salario base del `EmpleadoBasePorComision` por un 10%. Por último, la aplicación debe imprimir el monto del pago para cada objeto. Complete los siguientes pasos para crear la nueva aplicación:

a) Modifique las clases `EmpleadoPorHoras` (figura 10.6) y `EmpleadoPorComision` (figura 10.7) para colocarlas en la jerarquía `PorPagar` como subclases de la versión de `Empleado` (figura 10.13) que implementa a `PorPagar`. [*Sugerencia:* cambie el nombre del método ingresos a `obtenerMontoPago` en cada subclase, de manera que la clase cumpla con su contrato heredado con la interfaz `PorPagar`].

b) Modifique la clase `EmpleadoBaseMasComision` (figura 10.8), de tal forma que extienda la versión de la clase `EmpleadoPorComision` que se creó en la parte (a).
c) Modifique `PruebaInterfazPorPagar` (figura 10.15) para procesar mediante el polimorfismo dos objetos `Factura`, un `EmpleadoAsalariado`, un `EmpleadoPorHoras`, un `EmpleadoPorComision` y un `EmpleadoBaseMasComision`. Primero imprima una representación de cadena de cada objeto `PorPagar`. Después, si un objeto es un `EmpleadoBaseMasComision`, aumente su salario base por un 10%. Por último, imprima el monto del pago para cada objeto `PorPagar`.

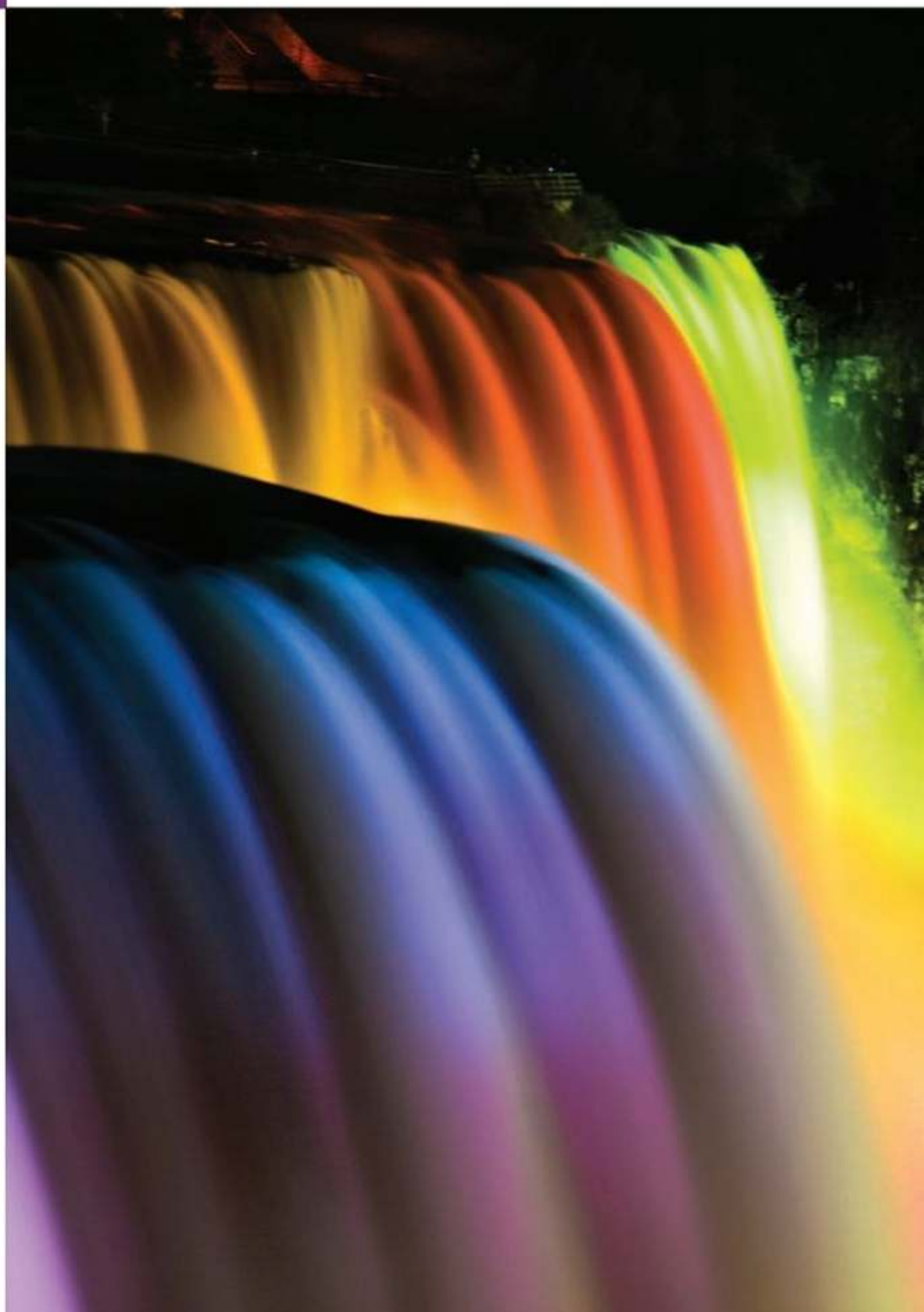
10.12 ***(Modificación al sistema de cuentas por pagar)*** Es posible incluir la funcionalidad de la aplicación de nómina (figuras 10.4 a 10.9) en la aplicación de cuentas por pagar sin necesidad de modificar las subclases de `Empleado` llamadas `EmpleadoAsalariado`, `EmpleadoPorHoras`, `EmpleadoPorComision` o `EmpleadoBaseMasComision`. Para ello, puede modificar la clase `Empleado` (figura 10.4) de modo que implemente la interfaz `PorPagar` y declare el método `obtenerMontoPago` para invocar al método `ingresos`. Así, el método `obtenerMontoPago` sería heredado por las subclases en la jerarquía de `Empleado`. Cuando se haga una llamada a `obtenerMontoPago` para un objeto de una subclase específica, se invocará mediante el polimorfismo al método `ingresos` apropiado para esa subclase. Vuelva a implementar el ejercicio 10.11, usando la jerarquía de `Empleado` original de la aplicación de nómina de las figuras 10.4 a 10.9. Modifique la clase `Empleado` como se describe en este ejercicio, y no modifique ninguna de las subclases de `Empleado`.

Marcar la diferencia

10.13 ***(Interfaz `ImpactoEcologico`: polimorfismo)*** Mediante el uso de interfaces, como aprendió en este capítulo, es posible especificar comportamientos similares para clases que pueden ser dispares. Los gobiernos y las compañías en todo el mundo se están preocupando cada vez más por el impacto ecológico del carbono (las liberaciones anuales de dióxido de carbono en la atmósfera), debido a los edificios que consumen diversos tipos de combustibles para obtener calor, los vehículos que queman combustibles para producir energía, y demás. Muchos científicos culpan a estos gases de invernadero por el fenómeno conocido como calentamiento global. Cree tres pequeñas clases no relacionadas por herencia: las clases `Edificio`, `Auto` y `Bicicleta`. Proporcione a cada clase ciertos atributos y comportamientos apropiados que sean únicos, que no tengan en común con otras clases. Escriba la interfaz `ImpactoEcologico` con un método `obtenerImpactoEcologico`. Haga que cada una de sus clases implementen a esa interfaz, de modo que su método `obtenerImpactoEcologico` calcule el impacto ecológico del carbono apropiado para esa clase (consulte sitios Web que expliquen cómo calcular el impacto ecológico del carbono). Escriba una aplicación que cree objetos de cada una de las tres clases, coloque referencias a esos objetos en `ArrayList<ImpactoEcologico>` y después itere a través del objeto `ArrayList`, invocando en forma polimórfica el método `obtenerImpactoEcologico` de cada objeto. Para cada objeto imprima cierta información de identificación, además de su impacto ecológico.

11

Manejo de excepciones: un análisis más detallado



Es cuestión de sentido común tomar un método y probarlo. Si falla, admítalo francamente y pruebe otro. Pero sobre todo, inténtelo.

—Franklin Delano Roosevelt

¡Oh! Arroja la peor parte de ello, y vive en forma más pura con la otra mitad.

—William Shakespeare

Si están corriendo y no saben hacia dónde se dirigen tengo que salir de alguna parte y atraparlos.

—Jerome David Salinger

Objetivos

En este capítulo aprenderá a:

- Comprender qué son las excepciones y cómo se manejan.
- Saber cuándo usar el manejo de excepciones.
- Utilizar bloques `try` para delimitar el código en el que podrían ocurrir excepciones.
- Lanzar excepciones mediante `throw` para indicar un problema.
- Usar bloques `catch` para especificar manejadores de excepciones.
- Utilizar el bloque `finally` para liberar recursos.
- Comprender la jerarquía de clases de excepciones.
- Crear excepciones definidas por el usuario.

Plan general

11.1 Introducción

Como vimos en el capítulo 7, una excepción (`exception`) es la indicación de un problema que ocurre durante la ejecución de un programa. El manejo de excepciones le permite crear aplicaciones que puedan resolver (o manejar) las excepciones. En muchos casos, el manejo de una excepción permite que el programa continúe su ejecución como si no se hubiera encontrado el problema. Las características que presentamos en este capítulo permiten a los programadores escribir programas tolerantes a fallas y robustos, que traten con los problemas que puedan surgir sin dejar de ejecutarse o que terminen sin causar estragos. El manejo de excepciones en Java se basa, en parte, en el trabajo de Andrew Koenig y Bjarne Stroustrup.[1]

Primero demostraremos las técnicas básicas de manejo de excepciones mediante un ejemplo que señala cómo manejar una excepción que ocurre cuando un método intenta realizar una división entre cero. Después del ejemplo, presentaremos varias clases de la parte superior de la jerarquía de clases de Java para el manejo de excepciones. Como verá posteriormente, sólo las clases que extienden a `Throwable` (paquete `java.lang`) en forma directa o indirecta pueden usarse para manejar excepciones. Después le mostraremos cómo usar excepciones encadenadas. Cuando invocamos a un método que indica una excepción, podemos lanzar otra excepción y encadenar la original a la nueva; esto nos permite agregar información específica de la aplicación a la excepción original. Luego le presentaremos las precondiciones y poscondiciones, que deben ser verdaderas cuando se hacen llamadas a sus métodos y cuando éstos regresan. A continuación presentaremos las aserciones, que los programadores utilizan en tiempo de desarrollo para facilitar el proceso de depurar su código. Por último, introduciremos dos nuevas características de manejo de excepciones en Java SE 7: atrapar varias excepciones con un solo manejador `catch` y la nueva instrucción `try` con recursos (`try-with-resources`), que libera de manera automática un recurso después de usarlo en el bloque `try`.

11.2 Ejemplo: división entre cero sin manejo de excepciones

Demostraremos primero qué ocurre cuando surgen errores en una aplicación que no utiliza el manejo de errores. En la figura 11.1 se pide al usuario que introduzca dos enteros y éstos se pasan al método

1 A. Koenig y B. Stroustrup, "Exception Handling for C++ (versión revisada)", *Proceedings of the Usenix C++ Conference*, págs. 149-176, San Francisco, abril de 1990.

cociente, que calcula el cociente y devuelve un resultado int. En este ejemplo veremos que las excepciones se **lanzan** (es decir, la excepción ocurre) cuando un método detecta un problema y no puede manejarlo.

```
1  // Fig. 11.1: DivisionEntreCeroSinManejoDeExcepciones.java
2  // División entre cero sin manejo de excepciones.
3  import java.util.Scanner;
4
5  public class DivisionEntreCeroSinManejoDeExcepciones
6  {
7     // demuestra el lanzamiento de una excepción cuando ocurre una división entre cero
8     public static int cociente( int numerador, int denominador )
9     {
10       return numerador / denominador; // posible división entre cero
11    } // fin del método cociente
12
13    public static void main( String[] args )
14    {
15       Scanner explorador = new Scanner( System.in ); // objeto Scanner para entrada
16
17       System.out.print( "Introduzca un numerador entero: " );
18       int numerador = explorador.nextInt();
19       System.out.print( "Introduzca un denominador entero: " );
20       int denominador = explorador.nextInt();
21
22       int resultado = cociente( numerador, denominador );
23       System.out.printf(
24          "\nResultado: %d / %d = %d\n", numerador, denominador, resultado );
25    } // fin de main
26 } // fin de la clase DivisionEntreCeroSinManejoDeExcepciones
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: 7

Resultado: 100 / 7 = 14
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at  DivisionEntreCeroSinManejoDeExcepciones.cociente(
            DivisionEntreCeroSinManejoDeExcepciones.java:10)
        at  DivisionEntreCeroSinManejoDeExcepciones.main(
            DivisionEntreCeroSinManejoDeExcepciones.java:22)
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: hola
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at DivisionEntreCeroSinManejoDeExcepciones.main(
            DivisionEntreCeroSinManejoDeExcepciones.java:20)
```

Fig. 11.1 | División entera sin manejo de excepciones.

La primera de las tres ejecuciones de ejemplo en la figura 11.1 muestra una división exitosa. En la segunda ejecución de ejemplo, el usuario introduce el valor `0` como denominador. Se muestran varias líneas de información en respuesta a esta entrada inválida. Esta información se conoce como el **rastreo de la pila**, la cual lleva el nombre de la excepción (`java.lang.ArithmeticException`) en un mensaje descriptivo, que indica el problema que ocurrió y la pila de llamadas a métodos (es decir, la cadena de llamadas) al momento en que ocurrió la excepción. El rastreo de la pila incluye la ruta de ejecución que condujo a la excepción, método por método. Esta información nos ayuda a depurar un programa. La primera línea especifica que ha ocurrido una excepción `ArithmeticException`. El texto después del nombre de la excepción (`"/ by zero"`) indica que esta excepción ocurrió como resultado de un intento de dividir entre cero. Java no permite la división entre cero en la aritmética de enteros. Cuando ocurre esto, Java lanza una excepción `ArithmeticException`. Este tipo de excepciones pueden surgir debido a varios problemas distintos en aritmética, por lo que los datos adicionales (`"/ by zero"`) nos proporcionan información más específica. Java *sí* permite la división entre cero con valores de punto flotante. Dicho cálculo produce como resultado el valor de infinito positivo o negativo, que se representa en Java como un valor de punto flotante (pero en realidad aparece como la cadena `Infinity` o `-Infinity`). Si se divide `0.0` entre 0.0, el resultado es NaN (no es un número), que también se representa en Java como un valor de punto flotante (pero se visualiza como `NaN`).

Empezando a partir de la última línea del rastreo de la pila, podemos ver que la excepción se detectó en la línea 22 del método `main`. Cada línea del rastreo de la pila contiene el nombre de la clase y el método (`DivideByZeroNoExceptionHandling.main`) seguido por el nombre del archivo y el número de línea (`DivideByZeroNoExceptionHandling.java:22`). Siguiendo el rastreo de la pila, podemos ver que la excepción ocurre en la línea 10, en el método `cociente`. La fila superior de la cadena de llamadas indica el **punto de lanzamiento**: el punto inicial en el que ocurre la excepción. El punto de lanzamiento de esta excepción está en la línea 10 del método `cociente`.

En la tercera ejecución, el usuario introduce la cadena `"hola"` como denominador. Observe de nuevo que se muestra un rastreo de la pila. Esto nos informa que ha ocurrido una excepción `InputMismatchException` (paquete `java.util`). En nuestros ejemplos anteriores, en donde se leían valores numéricos del usuario, se suponía que éste debía introducir un valor entero apropiado. Sin embargo, algunas veces los usuarios cometen errores e introducen valores no enteros. Una excepción `InputMismatchException` ocurre cuando el método `nextInt` de `Scanner` recibe una cadena que no representa un entero válido. Empezando desde el final del rastreo de la pila, podemos ver que la excepción se detectó en la línea 20 del método `main`. Siguiendo el rastreo de la pila, podemos ver que la excepción ocurre en el método `nextInt`. Observe que en vez del nombre de archivo y del número de línea, se proporciona el texto `Unknown Source`. Esto significa que la JVM no tiene acceso a los supuestos símbolos de depuración que proveen la información sobre el nombre del archivo y el número de línea para la clase de ese método; por lo general, éste es el caso para las clases de la API de Java. Muchos IDE tienen acceso al código fuente de la API de Java, por lo que muestran los nombres de archivos y números de línea en los rastreos de la pila.

En las ejecuciones de ejemplo de la figura 11.1, cuando ocurren excepciones y se muestran los rastreos de la pila, el programa también termina. Esto no siempre ocurre en Java; algunas veces un programa puede continuar, aun cuando haya ocurrido una excepción y se imprima un rastreo de pila. En tales casos, la aplicación puede producir resultados inesperados. Por ejemplo, una aplicación de interfaz gráfica de usuario (GUI) por lo general se seguirá ejecutando. En la siguiente sección le mostraremos cómo manejar estas excepciones.

En la figura 11.1, ambos tipos de excepciones se detectaron en el método `main`. En el siguiente ejemplo, veremos cómo manejar estas excepciones para permitir que el programa se ejecute hasta terminar de manera normal.

11.3 Ejemplo: manejo de excepciones tipo ArithmeticException e InputMismatchException

La aplicación de la figura 11.2, que se basa en la figura 11.1, utiliza el manejo de excepciones para procesar cualquier excepción tipo `ArithmeticException` e `InputMismatchException` que pueda ocurrir. La aplicación todavía pide dos enteros al usuario y los pasa al método `cociente`, que calcula el cociente y devuelve un resultado `int`. Esta versión de la aplicación utiliza el manejo de excepciones de manera que, si el usuario comete un error, el programa atrapa y maneja (es decir, se encarga de) la excepción; en este caso, le permite al usuario tratar de introducir los datos de entrada otra vez.

```
1  // Fig. 11.2: DivisionEntreCeroConManejoDeExcepciones.java
2  // Manejo de excepciones ArithmeticException e InputMismatchException.
3  import java.util.InputMismatchException;
4  import java.util.Scanner;
5
6  public class DivisionEntreCeroConManejoDeExcepciones
7  {
8     // demuestra cómo se lanza una excepción cuando ocurre una división entre cero
9     public static int cociente( int numerador, int denominador )
10       throws ArithmeticException
11    {
12       return numerador / denominador; // posible división entre cero
13    } // fin del método cociente
14
15    public static void main( String[] args )
16    {
17       Scanner explorador = new Scanner( System.in ); // objeto Scanner para entrada
18       boolean continuarCiclo = true; // determina si se necesitan más datos de entrada
19
20       do
21       {
22          try // lee dos números y calcula el cociente
23          {
24             System.out.print( "Introduzca un numerador entero: " );
25             int numerador = explorador.nextInt();
26             System.out.print( "Introduzca un denominador entero: " );
27             int denominador = explorador.nextInt();
28
29             int resultado = cociente( numerador, denominador );
30             System.out.printf( "\nResultado: %d / %d = %d\n", numerador,
31                denominador, resultado );
32             continuarCiclo = false; // entrada exitosa; termina el ciclo
33          } // fin de bloque try
34          catch ( InputMismatchException inputMismatchException )
35          {
36             System.err.printf( "\nExcepcion: %s\n",
37                inputMismatchException );
38             explorador.nextLine(); // descarta entrada para que el usuario intente
                                      otra vez
39             System.out.println(
40                "Debe introducir enteros. Intente de nuevo.\n" );
41          } // fin de bloque catch
```

Fig. 11.2 | Manejo de excepciones `ArithmeticException` e `InputMismatchException` (parte 1 de 2).

```
42          catch ( ArithmeticException arithmeticException )
43          {
44             System.err.printf( "\nExcepcion: %s\n", arithmeticException );
45             System.out.println(
46                "Cero es un denominador invalido. Intente de nuevo.\n" );
47          } // fin de catch
48       } while ( continuarCiclo ); // fin de do...while
49    } // fin de main
50 } // fin de la clase DivisionEntreCeroConManejoDeExcepciones
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: 7

Resultado: 100 / 7 = 14
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: 0

Excepcion: java.lang.ArithmeticException: / by zero
Cero es un denominador invalido. Intente de nuevo.

Introduzca un numerador entero: 100
Introduzca un denominador entero: 7

Resultado: 100 / 7 = 14
```

```
Introduzca un numerador entero: 100
Introduzca un denominador entero: hola

Excepcion: java.util.InputMismatchException
Debe introducir enteros. Intente de nuevo.

Introduzca un numerador entero: 100
Introduzca un denominador entero: 7

Resultado: 100 / 7 = 14
```

Fig. 11.2 | Manejo de excepciones `ArithmeticException` e `InputMismatchException` (parte 2 de 2)

La primera ejecución de ejemplo de la figura 11.2 es una ejecución exitosa que no se encuentra con ningún problema. En la segunda ejecución, el usuario introduce un denominador cero y ocurre una excepción `ArithmeticException`. En la tercera ejecución, el usuario introduce la cadena "`hola`" como el denominador, y ocurre una excepción `InputMismatchException`. Para cada excepción, se informa al usuario sobre el error y se le pide que intente de nuevo; después el programa le pide dos nuevos enteros. En cada ejecución de ejemplo, el programa se ejecuta hasta terminar sin problemas.

La clase `InputMismatchException` se importa en la línea 3. La clase `ArithmeticException` no necesita importarse, ya que se encuentra en el paquete `java.lang`. En la línea 18 se crea la variable `boolean` llamada `continuarCiclo`, la cual es verdadera si el usuario no ha introducido aún datos de entrada válidos. En las líneas 20 a 48 se pide repetidas veces a los usuarios que introduzcan datos, hasta recibir una entrada válida.

Encerrar código en un bloque `try`

Las líneas 22 a 33 contienen un **bloque `try`**, que encierra el código que podría lanzar (`throw`) una excepción y el código que no debería ejecutarse en caso de que ocurra una excepción (es decir, si ocurre una excepción, se omitirá el resto del código en el bloque `try`). Un bloque `try` consiste en la palabra clave `try` seguida de un bloque de código, encerrado entre llaves. [*Nota:* el término "bloque `try`" se refiere algunas veces sólo al bloque de código que va después de la palabra clave `try` (sin incluir a la palabra `try` en sí). Para simplificar, usaremos el término "bloque `try`" para referirnos al bloque de código que va después de la palabra clave `try`, incluyendo esta palabra]. Las instrucciones que leen los enteros del teclado (líneas 25 y 27) utilizan el método `nextInt` para leer un valor `int`. El método `nextInt` lanza una excepción `InputMismatchException` si el valor leído no es un entero válido.

La división que puede provocar una excepción `ArithmeticException` no se ejecuta en el bloque `try`. En vez de ello, la llamada al método `cociente` (línea 29) invoca al código que intenta realizar la división (línea 12); la JVM lanza un objeto `ArithmeticException` cuando el denominador es cero.

Observación de ingeniería de software 11.1

Las excepciones pueden surgir a través de código mencionado en forma explícita en un bloque `try`, a través de llamadas a otros métodos, de llamadas a métodos con muchos niveles de anidamiento, iniciadas por código en un bloque `try` o desde la máquina virtual de Java, al momento en que ejecute códigos de byte de Java.

Atrapar excepciones

El bloque `try` en este ejemplo va seguido de dos bloques `catch`: uno que maneja una excepción `InputMismatchException` (líneas 34 a 41) y uno que maneja una excepción `ArithmeticException` (líneas 42 a 47). Un **bloque `catch`** (también conocido como **cláusula `catch`** o **manejador de excepciones**) atrapa (es decir, recibe) y maneja una excepción. Un bloque `catch` empieza con la palabra clave `catch` y va seguido por un parámetro entre paréntesis (conocido como el parámetro de excepción, que veremos en breve) y un bloque de código encerrado entre llaves. [*Nota:* el término "cláusula `catch`" se utiliza algunas veces para hacer referencia a la palabra clave `catch`, seguida de un bloque de código, mientras que el término "bloque `catch`" se refiere sólo al bloque de código que va después de la palabra clave `catch`, sin incluirla. Para simplificar, usaremos el término "bloque `catch`" para referirnos al bloque de código que va después de la palabra clave `catch`, incluyendo esta palabra].

Por lo menos debe ir un bloque `catch` o un **bloque `finally`** (que veremos en la sección 11.6) justo después del bloque `try`. Cada bloque `catch` especifica entre paréntesis un parámetro de excepción, que identifica el tipo de excepción que puede procesar el manejador. Cuando ocurre una excepción en un bloque `try`, el bloque `catch` que se ejecuta es el *primero* cuyo tipo coincide con el tipo de la excepción que ocurrió (es decir, el tipo en el bloque `catch` coincide exactamente con el tipo de la excepción que se lanzó, o es una superclase de ésta). El nombre del parámetro de excepción permite al bloque `catch` interactuar con un objeto de excepción atrapada; por ejemplo, para invocar en forma implícita el método `toString` de la excepción que se atrapó (como en las líneas 37 y 44), que muestra información básica acerca de la excepción. Observe que usamos el objeto **`System.err`** (**flujo de error estándar**) para mostrar los mensajes de error en pantalla. Los métodos print de `System.err`, al igual que los de `System.out`, muestran datos en el símbolo del sistema de manera predeterminada.

La línea 38 del primer bloque `catch` llama al método `nextLine` de `Scanner`. Como ocurrió una excepción `InputMismatchException`, la llamada al método `nextInt` nunca leyó con éxito los datos del usuario; por lo tanto, leemos esa entrada con una llamada al método `nextLine`. No hacemos nada con la entrada en este punto, ya que sabemos que es inválida. Cada bloque `catch` muestra un mensaje de error y pide al usuario que intente de nuevo. Al terminar alguno de los bloques `catch`, se pide al usuario

que introduzca datos. Pronto veremos con más detalle la manera en que trabaja este flujo de control en el manejo de excepciones.

Error común de programación 11.1

Es un error de sintaxis colocar código entre un bloque `try` *y sus correspondientes bloques* `catch`.

Error común de programación 11.2

Cada bloque catch sólo puede tener un parámetro; especificar una lista de parámetros de excepción separados por comas es un error de sintaxis.

Una **excepción no atrapada** es una para la que no hay bloques `catch` que coincidan. En el segundo y tercer resultado de ejemplo de la figura 11.1, vio las excepciones no atrapadas. Recuerde que cuando ocurrieron excepciones en ese ejemplo, la aplicación terminó antes de tiempo (después de mostrar el rastreo de pila de la excepción). Esto no siempre ocurre como resultado de las excepciones no atrapadas. Java utiliza un modelo "multihilos" de ejecución de programas: cada **hilo** es una actividad paralela. Un programa puede tener muchos hilos. Si un programa sólo tiene un hilo, una excepción no atrapada hará que el programa termine. Si uno tiene múltiples hilos, una excepción no atrapada terminará *sólo* el hilo en el cual ocurrió la excepción. Sin embargo, en dichos programas ciertos hilos pueden depender de otros, y si un hilo termina debido a una excepción no atrapada, puede haber efectos adversos para el resto del programa. En el capítulo 26 (en inglés en el sitio Web del libro), analizaremos estas cuestiones con detalle.

Modelo de terminación del manejo de excepciones

Si ocurre una excepción en un bloque `try` (por ejemplo, si se lanza una excepción `InputMismatchException` como resultado del código de la línea 25 en la figura 11.2), el bloque `try` termina de inmediato y el control del programa se transfiere al *primero* de los siguientes bloques `catch` en los que el tipo del parámetro de excepción coincide con el tipo de la excepción que se lanzó. En la figura 11.2, el primer bloque `catch` atrapa excepciones `InputMismatchException` (que ocurren si se introducen datos de entrada inválidos) y el segundo bloque `catch` atrapa excepciones `ArithmeticException` (que ocurren si hay un intento por dividir entre cero). Una vez que se maneja la excepción, el control del programa *no* regresa al punto de lanzamiento, ya que el bloque `try` ha *expirado* (y se han perdido sus variables locales). En vez de ello, el control se reanuda después del último bloque `catch`. Esto se conoce como el **modelo de terminación del manejo de excepciones**. Algunos lenguajes utilizan el **modelo de reanudación del manejo de excepciones** en el que, después de manejar una excepción, el control se reanuda justo después del punto de lanzamiento.

Observe que nombramos a nuestros parámetros de excepción (`inputMismatchException` y `arithmeticException`) con base en su tipo. A menudo, los programadores de Java utilizan simplemente la letra `e` como el nombre de sus parámetros de excepción.

Buena práctica de programación 11.1

El uso del nombre de un parámetro de excepción que refleje el tipo del parámetro fomenta la claridad, al recordar al programador el tipo de excepción que se está manejando.

Después de ejecutar un bloque `catch`, el flujo de control de este programa procede a la primera instrucción después del último bloque `catch` (línea 48 en este caso). La condición en la instrucción `do...while` es `true` (la variable `continuarCiclo` contiene su valor inicial de `true`), por lo que el control regresa al principio del ciclo y se le pide al usuario una vez más que introduzca datos. Esta instruc-

ción de control iterará hasta que se introduzcan datos de entrada válidos. En ese punto, el control del programa llega a la línea 32, en donde se asigna `false` a la variable `continuarCiclo`. Después, el bloque `try` termina. Si no se lanzan excepciones en el bloque `try`, se omiten los bloques `catch` y el control continúa con la primera instrucción después de los bloques `catch` (en la sección 11.6 aprenderemos acerca de otra posibilidad, cuando hablemos sobre el bloque `finally`). Ahora la condición del ciclo `do...while` es `false`, y el método `main` termina.

El bloque `try` y sus correspondientes bloques `catch` y/o `finally` forman en conjunto una **instrucción try**. Es importante no confundir los términos "bloque `try`" e "instrucción `try`; éste último incluye el bloque `try`, así como los siguientes bloques `catch` y/o un bloque `finally`.

Al igual que con cualquier otro bloque de código, cuando termina un bloque `try`, las variables locales declaradas en ese bloque quedan fuera de alcance y ya no son accesibles; por ende, las variables locales de un bloque `try` no son accesibles en los correspondientes bloques `catch`. Cuando termina un bloque `catch`, las variables locales declaradas dentro de este bloque (incluyendo el parámetro de excepción de ese bloque `catch`) también quedan fuera de alcance y se destruyen. Cualquier bloque `catch` restante en la instrucción `try` se ignora, y la ejecución se reanuda en la primera línea de código después de la secuencia `try...catch`; ésta será un bloque `finally`, en caso de que haya uno presente.

Uso de la cláusula `throws`

Ahora examinaremos el método `cociente` (figura 11.2; líneas 9 a 13). La porción de la declaración del método ubicada en la línea 10 se conoce como **cláusula throws**. Esta cláusula especifica las excepciones que lanza el método. La cláusula aparece *después* de la lista de parámetros del método y *antes* de su cuerpo. Contiene una lista separada por comas de las excepciones que lanzará el método, en caso de que ocurran varios problemas. Dichas excepciones pueden lanzarse mediante instrucciones en el cuerpo del método, o a través de métodos que se llamen desde el cuerpo. Un método puede lanzar excepciones de las clases que se listen en su cláusula `throws`, o en la de sus subclases. Hemos agregado la cláusula `throws` a esta aplicación, para indicar al resto del programa que este método puede lanzar una excepción `ArithmeticException`. Por ende, a los clientes del método `cociente` se les informa que el método puede lanzar una excepción `ArithmeticException`. En la sección 11.5 aprenderá más acerca de la cláusula `throws`.

Tip para prevenir errores 11.1

Lea la documentación de la API en línea para saber acerca de un método, antes de usarlo en un programa. La documentación especifica las excepciones que lanza el método (si la hay), y también indica las razones por las que pueden ocurrir dichas excepciones. Después, lea la documentación de la API en línea para ver las clases de excepciones especificadas. Por lo general, la documentación para una clase de excepción contiene las razones potenciales por las que pueden ocurrir dichas excepciones. Por último, incluya el código adecuado para manejar esas excepciones en su programa.

Cuando se ejecuta la línea 12, si el `denominador` es cero, la JVM lanza un objeto `ArithmeticException`. Este objeto será atrapado por el bloque `catch` en las líneas 42 a 47, que muestra información básica acerca de la excepción, invocando de manera implícita al método `toString` de la excepción, y después pide al usuario que intente de nuevo.

Si el denominador no es cero, el método `cociente` realiza la división y devuelve el resultado al punto de la invocación al método `cociente` en el bloque `try` (línea 29). Las líneas 30 y 31 muestran el resultado del cálculo y la línea 32 establece `continuarCiclo` en `false`. En este caso, el bloque `try` se completa con éxito, por lo que el programa omite los bloques `catch` y la condición falla en la línea 48, y el método `main` termina de ejecutarse en forma normal.

Cuando cociente lanza una excepción `ArithmeticException`, `cociente` termina y no devuelve un valor, y sus variables locales quedan fuera de alcance (y se destruyen). Si `cociente` contiene varia-

bles locales que sean referencias a objetos y no hay otras referencias a esos objetos, éstos se marcan para la recolección de basura. Además, cuando ocurre una excepción, el bloque `try` desde el cual se llamó `cociente` termina antes de que puedan ejecutarse las líneas 30 a 32. Aquí también, si las variables locales se crearon en el bloque `try` antes de que se lanzara la excepción, estas variables quedarían fuera de alcance.

Si se genera una excepción `InputMismatchException` mediante las líneas 25 o 27, el bloque `try` termina y la ejecución continúa con el bloque `catch` en las líneas 34 a 41. En este caso, no se hace una llamada al método `cociente`. Entonces, el método `main` continúa después del último bloque `catch` (línea 48).

11.4 Cuándo utilizar el manejo de excepciones

El manejo de excepciones está diseñado para procesar **errores sincrónicos**, que ocurren cuando se ejecuta una instrucción. Ejemplos comunes de estos errores que veremos en este libro son los índices fuera de rango, el desbordamiento aritmético (es decir, un valor fuera del rango representable de valores), la división entre cero, los parámetros inválidos de un método, la interrupción de hilos (como veremos en el capítulo 26, en inglés en el sitio Web del libro) y la asignación fallida de memoria (debido a la falta de ésta). El manejo de excepciones no está diseñado para procesar los problemas asociados con los **eventos asíncronos** (por ejemplo, completar las operaciones de E/S de disco, la llegada de mensajes de red, clics del ratón y pulsaciones de teclas), los cuales ocurren en paralelo con, y en forma independiente de, el flujo de control del programa.

Observación de ingeniería de software 11.2

Incorpore su estrategia de manejo de excepciones en sus sistemas, partiendo desde el principio del proceso de diseño. Puede ser difícil incluir un manejo efectivo de las excepciones, después de haber implementado un sistema.

Observación de ingeniería de software 11.3

El manejo de excepciones proporciona una sola técnica uniforme para procesar los problemas. Esto ayuda a los programadores que trabajan en proyectos extensos a comprender el código de procesamiento de errores de los demás programadores.

11.5 Jerarquía de excepciones en Java

Todas las clases de excepciones heredan, ya sea en forma directa o indirecta, de la clase `Exception`, formando una jerarquía de herencias. Los programadores pueden extender esta jerarquía para crear sus propias clases de excepciones.

La figura 11.3 muestra una pequeña porción de la jerarquía de herencia para la clase `Throwable` (una subclase de `Object`), que es la superclase de la clase `Exception`. Sólo pueden usarse objetos `Throwable` con el mecanismo para manejar excepciones. La clase `Throwable` tiene dos subclases: `Exception` y `Error`. La clase `Exception` y sus subclases (por ejemplo, `RuntimeException`, del paquete `java.lang`, e `IOException`, del paquete `java.io`) representan situaciones excepcionales que pueden ocurrir en un programa en Java, y que pueden ser atrapadas por la aplicación. La clase `Error` y sus subclases representan las situaciones anormales que ocurren en la JVM. La mayoría de los *errores tipo `Error` ocurren con poca frecuencia y no deben ser atrapados por las aplicaciones; por lo general no es posible que las aplicaciones se recuperen de los errores tipo `Error`.*

La jerarquía de excepciones de Java contiene cientos de clases. En la API de Java puede encontrar información acerca de las clases de excepciones de Java. La documentación para la clase `Throwable` se encuentra en `download.oracle.com/javase/6/docs/api/java/lang/Throwable.html`. En este sitio puede buscar las subclases de esta clase para obtener más información acerca de los objetos `Exception` y `Error` de Java.

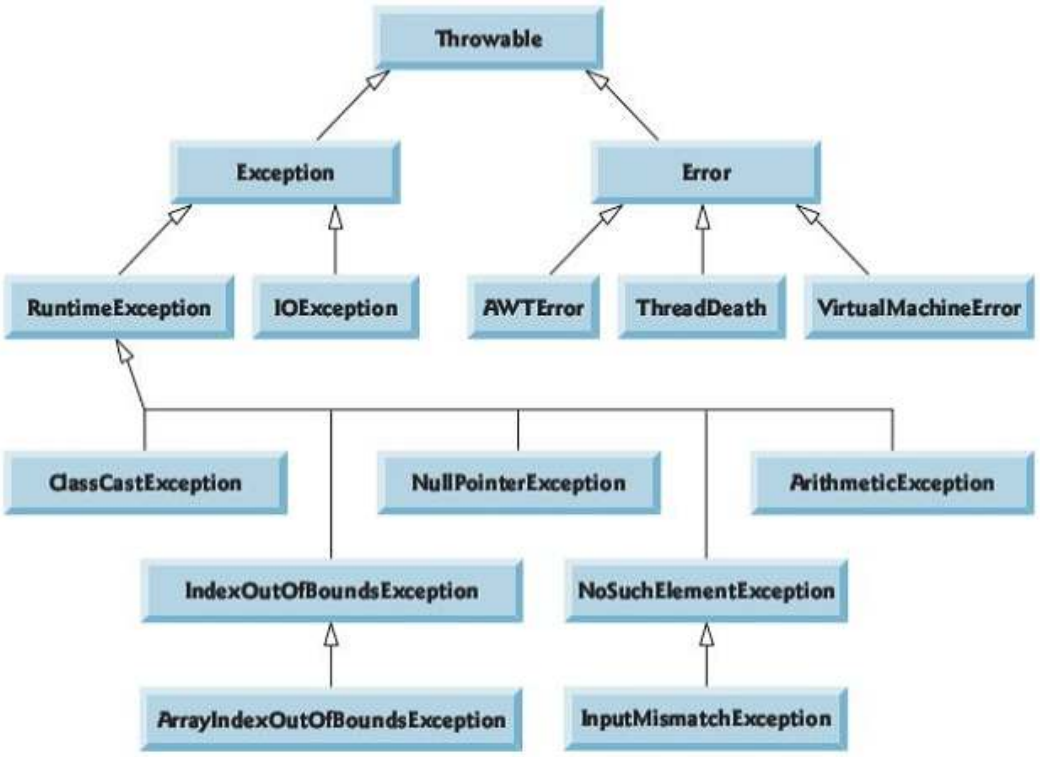


Fig. 11.3 | Porción de la jerarquía de herencia de la clase `Throwable`.

Comparación entre excepciones verificadas y no verificadas

Java clasifica a las excepciones en dos categorías: **excepciones verificadas** y **excepciones no verificadas**. Esta distinción es importante, ya que el compilador de Java implementa un **requerimiento de atrapar o declarar** para las excepciones verificadas. El tipo de una excepción determina si es verificada o no verificada. Todos los tipos de excepciones que son subclases directas o indirectas de la clase `RuntimeException` (paquete `java.lang`) son excepciones no verificadas. Por lo general, se deben a los defectos en el código de nuestros programas. Algunos ejemplos de excepciones no verificadas son las excepciones `ArrayIndexOutOfBoundsException` (que vimos en el capítulo 7) y `ArithmeticException` (que se muestran en la figura 11.3). Todas las clases que heredan de la clase `Exception` pero no de la clase `RuntimeException` se consideran como excepciones verificadas. Por lo general, dichas excepciones son provocadas por condiciones que no están bajo el control del programa; por ejemplo, en el procesamiento de archivos, el programa no puede abrir un archivo debido a que no existe. Las clases que heredan de la clase `Error` se consideran como no verificadas.

El compilador *verifica* cada una de las llamadas a un método, junto con su declaración, para determinar si el método lanza excepciones verificadas. De ser así, el compilador asegura que la excepción verificada sea atrapada o declarada en una cláusula `throws`. En los siguientes ejemplos le mostraremos cómo atrapar y declarar excepciones verificadas. En la sección 11.3 vimos que la cláusula `throws` especifica las excepciones que lanza un método. Dichas excepciones no se atrapan en el cuerpo del método. Para satisfacer la parte relacionada con *atrapar* del requerimiento de atrapar o declarar, el código que genera la excepción debe envolverse en un bloque `try`, y debe proporcionar un manejador `catch` para el tipo de excepción verificada (o uno de los tipos de su superclase). Para satisfacer la parte relacionada con *declarar* del requerimiento de atrapar o declarar, el método que contiene el código que genera la excepción debe proporcionar una cláusula `throws` que contenga el tipo de excepción verificada, después de su lista de parámetros y antes de su cuerpo. Si el requerimiento de atrapar o declarar no se satisface, el compilador emitirá un mensaje de error, indicando que la excepción debe ser atrapada o

declarada. Esto obliga a los programadores a pensar acerca de los problemas que pueden ocurrir cuando se hace una llamada a un método que lanza excepciones verificadas.

Observación de ingeniería de software 11.4

Los programadores se ven obligados a tratar con las excepciones verificadas. Esto produce un código más robusto que el que se crearía si los programadores pudieran simplemente ignorar las excepciones.

Error común de programación 11.3

Si un método intenta lanzar de manera explícita una excepción verificada (o si llama a otro método que lance una excepción verificada), y ésta no se enumera en la cláusula `throws` *de ese método, se produce un error de compilación.*

Error común de programación 11.4

Si el método de una subclase sobrescribe al método de una superclase, es un error para el método de la subclase mencionar más expresiones en su cláusula `throws` *de las que tiene el método sobrescrito de la superclase. Sin embargo, la cláusula* `throws` *de una subclase puede contener un subconjunto de la lista* `throws` *de una superclase.*

Observación de ingeniería de software 11.5

Si su método llama a otros métodos que lanzan explícitamente excepciones verificadas, éstas deben atraparse o declararse en su método. Si una expresión puede manejarse de manera significativa en un método, éste debe atrapar la excepción en vez de declararla.

A diferencia de las excepciones verificadas, el compilador de Java no verifica el código para determinar si una excepción no verificada es atrapada o declarada. Por lo general, las excepciones no verificadas se pueden evitar mediante una codificación apropiada. Por ejemplo, la excepción `ArithmeticException` no verificada que lanza el método `cociente` (líneas 9 a 13) en la figura 11.2 puede evitarse si el método se asegura de que el denominador no sea cero antes de tratar de realizar la división. No es obligatorio que se enumeren las excepciones no verificadas en la cláusula `throws` de un método; aun si se hace, no es obligatorio que una aplicación atrape dichas excepciones.

Observación de ingeniería de software 11.6

Aunque el compilador no implementa el requerimiento de atrapar o declarar para las excepciones no verificadas, usted deberá proporcionar un código apropiado para el manejo de excepciones cuando sepa que podrían ocurrir. Por ejemplo, un programa debería procesar la excepción `NumberFormatException` *del método* `parseInt` *de la clase* `Integer`*, aun cuando las excepciones* `NumberFormatException` *(una subclase indirecta de* `RuntimeException`*) sean del tipo de excepciones no verificadas. Esto hará que sus programas sean más robustos.*

Atrapar excepciones de subclases

Si se escribe un manejador `catch` para atrapar objetos de excepción de un tipo de superclase, también se pueden atrapar todos los objetos de las subclases de esa clase. Esto permite que un bloque `catch` maneje los errores relacionados con una notación concisa, y permite el procesamiento polimórfico de las excepciones relacionadas. Sin duda, se podría atrapar a cada uno de los tipos de las subclases en forma individual, si estas excepciones requirieran un procesamiento distinto.

Sólo se ejecuta la primera cláusula `catch` *que coincida*

Si hay *varios* bloques `catch` que coinciden con un tipo específico de excepción, sólo se ejecuta el *primer* bloque `catch` que coincida cuando ocurra una excepción de ese tipo. Es un error de compilación tratar de atrapar el *mismo tipo exacto* en dos bloques `catch` distintos asociados con un bloque `try` específico. Sin embargo, puede haber varios bloques `catch` que coincidan con una excepción; es decir, varios blo-

ques `catch` cuyos tipos sean los mismos que el tipo de excepción, o de una superclase de ese tipo. Por ejemplo, podríamos colocar un bloque `catch` para el tipo `ArithmeticException` después de un bloque `catch` para el tipo `Exception`; ambos coincidirían con las excepciones `ArithmeticException`, pero sólo se ejecutaría el primer bloque `catch` que coincidiera.

Tip para prevenir errores 11.2

Atrapar los tipos de las subclases en forma individual puede ocasionar errores si usted olvida evaluar uno o más de los tipos de subclase en forma explícita; al atrapar a la superclase se garantiza que se atraparán los objetos de todas las subclases. Al colocar un bloque `catch` *para el tipo de la superclase después de los demás bloques* `catch` *para todas las subclases de esa superclase aseguramos que todas las excepciones de las subclases se atrapen en un momento dado.*

Error común de programación 11.5

Al colocar un bloque `catch` *para un tipo de excepción de la superclase antes de los demás bloques* `catch` *que atrapan los tipos de excepciones de las subclases, evitamos que esos bloques* `catch` *se ejecuten, por lo cual se produce un error de compilación.*

11.6 Bloque `finally`

Los programas que obtienen ciertos tipos de recursos deben devolver esos recursos al sistema en forma explícita, para evitar las denominadas **fugas de recursos**. En lenguajes de programación como C y C++, el tipo más común de fuga de recursos es la fuga de memoria. Java realiza la recolección automática de basura en la memoria que ya no es utilizada por los programas, evitando así la mayoría de las fugas de memoria. Sin embargo, pueden ocurrir otros tipos de fugas de recursos en Java. Por ejemplo, los archivos, las conexiones de bases de datos y conexiones de red que no se cierran apropiadamente cuando ya no se necesitan, podrían no estar disponibles para su uso en otros programas.

Tip para prevenir errores 11.3

Hay una pequeña cuestión en Java: no elimina completamente las fugas de memoria. Java no hace recolección de basura en un objeto, sino hasta que no existen más referencias a ese objeto. Por lo tanto, si los programadores mantienen por error referencias a objetos no deseados, pueden ocurrir fugas de memoria. Para ayudar a evitar este problema, asigne `null` *a las variables de tipo por referencia cuando ya no las necesite.*

El bloque `finally` (que consiste en la palabra clave `finally`, seguida de código encerrado entre llaves) es opcional, y algunas veces se le llama **cláusula `finally`**. Si está presente, se coloca después del último bloque `catch`. Si no hay bloques `catch`, el bloque `finally` sigue justo después del bloque `try`.

El bloque `finally` se ejecutará, se lance o no una excepción en el bloque `try` correspondiente. El bloque `finally` también se ejecutará si un bloque `try` se sale mediante el uso de una instrucción `return`, `break` o `continue`, o simplemente al llegar a la llave derecha de cierre del bloque `try`. El bloque `finally` *no* se ejecutará si la aplicación sale antes de tiempo de un bloque `try`, llamando al método `System.exit`. Este método, que demostraremos en el capítulo 17 (en el sitio Web del libro), termina de inmediato una aplicación.

Como un bloque `finally` casi siempre se ejecuta, por lo general contiene código para liberar recursos. Suponga que se asigna un recurso en un bloque `try`. Si no ocurre una excepción, se ignoran los bloques `catch` y el control pasa al bloque `finally`, que libera el recurso. Después, el control pasa a la primera instrucción después del bloque `finally`. Si ocurre una excepción en el bloque `try`, éste termina. Si el programa atrapa la excepción en uno de los bloques `catch` correspondientes, procesa la excepción,

después el bloque `finally` *libera* el recurso y el control pasa a la primera instrucción después del bloque `finally`. Si el programa no atrapa la excepción, el bloque `finally` de todas formas *libera* el recurso y se hace un intento por atrapar la excepción en uno de los métodos que hacen la llamada.

Tip para prevenir errores 11.4

El bloque `finally` es un lugar ideal para liberar los recursos adquiridos en un bloque `try` (como los archivos abiertos), lo cual ayuda a eliminar fugas de recursos.

Tip de rendimiento 11.1

Siempre debe liberar cada recurso de manera explícita y lo más pronto posible, una vez que ya no sea necesario. Esto hace que los recursos estén disponibles para que su programa los reutilice lo más pronto posible, con lo cual se mejora la utilización de recursos.

Si una excepción que ocurre en un bloque `try` no puede ser atrapada por uno de los manejadores `catch` de ese bloque `try`, el programa ignora el resto del bloque `try` y el control procede al bloque `finally`. Después el programa pasa la excepción al siguiente bloque `try` exterior (por lo general, en el método que hizo la llamada), en donde un bloque `catch` asociado podría atraparla. Este proceso puede ocurrir a través de muchos niveles de bloques `try`. También es posible que la excepción no se atrape.

Si un bloque `catch` lanza una excepción, el bloque `finally` de todas formas se ejecuta. Después, la excepción se pasa al siguiente bloque `try` exterior; de nuevo, lo común es que sea en el método que hizo la llamada.

La figura 11.4 demuestra que el bloque `finally` se ejecuta, aún y cuando no se lance una excepción en el bloque `try` correspondiente. El programa contiene los métodos `static main` (líneas 6 a 18), `lanzaExcepcion` (líneas 21 a 44) y `noLanzaExcepcion` (líneas 47 a 64). Los métodos `lanzaExcepcion` y `noLanzaExcepcion` se declaran como `static`, por lo que `main` puede llamarlos directamente sin instanciar un objeto `UsoDeExcepciones`.

```
1  // Fig. 11.4: UsoDeExcepciones.java
2  // El mecanismo de manejo de excepciones try...catch...finally.
3
4  public class UsoDeExcepciones
5  {
6     public static void main( String[] args )
7     {
8        try
9        {
10          lanzaExcepcion(); //llama al método lanzaExcepcion
11       } // fin de try
12       catch ( Exception excepcion ) // excepción lanzada por lanzaExcepcion
13       {
14          System.err.println( "La excepcion se manejo en main" );
15       } // fin de catch
16
17       noLanzaExcepcion();
18    } // fin de main
19
20    // demuestra los bloques try...catch...finally
21    public static void lanzaExcepcion() throws Exception
22    {
```

Fig. 11.4 | Mecanismo de manejo de excepciones `try...catch...finally` (parte 1 de 2).

```
23          try // lanza una excepción y la atrapa de inmediato
24          {
25             System.out.println( "Metodo lanzaExcepcion" );
26             throw new Exception(); // genera la excepción
27          } // fin de try
28          catch ( Exception excepcion ) // atrapa la excepción lanzada en el bloque try
29          {
30             System.err.println(
31                "La excepcion se manejo en el metodo lanzaExcepcion" );
32             throw excepcion; // vuelve a lanzar para procesarla más adelante
33
34             // no se llegaría al código que se coloque aquí; se producirían errores de
                  compilación
35
36          } // fin de catch
37          finally // se ejecuta sin importar lo que ocurra en los bloques try...catch
38          {
39             System.err.println( "Se ejecuto finally en lanzaExcepcion" );
40          } // fin de finally
41
42          // no se llegaría al código que se coloque aquí; se producirían errores de
               compilación
43
44       } // fin del método lanzaExcepcion
45
46       // demuestra el uso de finally cuando no ocurre una excepción
47       public static void noLanzaExcepcion()
48       {
49          try // el bloque try no lanza una excepción
50          {
51             System.out.println( "Metodo noLanzaExcepcion" );
52          } // fin de try
53          catch ( Exception excepcion ) // no se ejecuta
54          {
55             System.err.println( excepcion );
56          } // fin de catch
57          finally // se ejecuta sin importar lo que ocurra en los bloques try...catch
58          {
59             System.err.println(
60                "Se ejecuto Finally en noLanzaExcepcion" );
61          } // fin de bloque finally
62
63          System.out.println( "Fin del metodo noLanzaExcepcion" );
64       } // fin del método noLanzaExcepcion
65    } // fin de la clase UsoDeExcepciones
```

```
Metodo lanzaExcepcion
La excepcion se manejo en el metodo lanzaExcepcion
Se ejecuto finally en lanzaExcepcion
La excepcion se manejo en main
Metodo noLanzaExcepcion
Se ejecuto Finally en noLanzaExcepcion
Fin del metodo noLanzaExcepcion
```

Fig. 11.4 | Mecanismo de manejo de excepciones `try...catch...finally` (parte 2 de 2).

Tanto System.out como System.err son **flujos**: una secuencia de bytes. Mientras que System.out (conocido como el **flujo de salida estándar**) se utiliza para mostrar la salida de un programa, System.err (**flujo de error estándar**) se utiliza para mostrar los errores de un programa. La salida de estos flujos se puede redirigir (es decir, enviar a otra parte que no sea el símbolo del sistema, como a un archivo). El uso de dos flujos distintos permite al programador separar fácilmente los mensajes de error de cualquier otra información de salida. Por ejemplo, los datos que se imprimen de System.err se podrían enviar a un archivo de registro, mientras que los que se imprimen de System.out se podrían mostrar en la pantalla. Para simplificar, en este capítulo no redigiremos la salida de System.err, sino que mostraremos dichos mensajes en el símbolo del sistema. En el capítulo 17 (en el sitio Web del libro) aprenderá más acerca de los flujos.

Lanzar excepciones mediante la instrucción throw

El método main (figura 11.4) empieza a ejecutarse, entra a su bloque try y de inmediato llama al método lanzaExcepcion (línea 10). El método lanzaExcepcion lanza una excepción tipo Exception. La instrucción en la línea 26 se conoce como **instrucción throw**; la cual se ejecuta para indicar que ha ocurrido una excepción. Hasta ahora sólo hemos atrapado las excepciones que lanzan los métodos que son llamados. Los programadores pueden lanzar excepciones mediante el uso de la instrucción throw. Al igual que con las excepciones lanzadas por los métodos de la API de Java, esto indica a las aplicaciones cliente que ha ocurrido un error. Una instrucción throw especifica un objeto que se lanzará. El operando de throw puede ser de cualquier clase derivada de Throwable.

Observación de ingeniería de software 11.7

Cuando se invoca el método toString en cualquier objeto Throwable, su cadena resultante incluye la cadena descriptiva que se suministró al constructor, o simplemente el nombre, si no se suministró una cadena.

Observación de ingeniería de software 11.8

Un objeto puede lanzarse sin contener información acerca del problema que ocurrió. En este caso, el simple conocimiento de que ocurrió una excepción de cierto tipo puede proporcionar suficiente información para que el manejador procese el problema en forma correcta.

Observación de ingeniería de software 11.9

Las excepciones pueden lanzarse desde constructores. Cuando se detecta un error en un constructor, debe lanzarse una excepción para evitar crear un objeto mal formado.

Volver a lanzar excepciones

La línea 32 de la figura 11.4 **vuelve a lanzar la excepción**. Las excepciones se vuelven a lanzar cuando un bloque catch, al momento de recibir una excepción, decide que no puede procesar la excepción o que sólo puede procesarla en forma parcial. Al volver a lanzar una excepción, se difiere el manejo de la misma (o tal vez una porción de ella) hacia otro bloque catch asociado con una instrucción try exterior. Para volver a lanzar una excepción se utiliza la **palabra clave throw**, seguida de una referencia al objeto excepción que se acaba de atrapar. Las excepciones no se pueden volver a lanzar desde un bloque finally, ya que el parámetro de la excepción (una variable local) del bloque catch ha dejado de existir.

Cuando se vuelve a lanzar una excepción, el *siguiente bloque try circundante* la detecta, y la instrucción catch de ese bloque try trata de manejarla. En este caso, el siguiente bloque try circundante se encuentra en las líneas 8 a 11 en el método main. Sin embargo, antes de manejar la excepción que se volvió a lanzar, se ejecuta el bloque finally (líneas 37 a 40). Después, el método main detecta la excepción que se volvió a lanzar en el bloque try, y la maneja en el bloque catch (líneas 12 a 15).

A continuación, `main` llama al método `noLanzaExcepcion` (línea 17). Como no se lanza una excepción en el bloque `try` de `noLanzaExcepcion` (líneas 49 a 52), el programa ignora el bloque `catch` (líneas 53 a 56), pero el bloque `finally` (líneas 57 a 61) se ejecuta de todas formas. El control pasa a la instrucción que está después del bloque `finally` (línea 63). Después, el control regresa a `main` y el programa termina.

Error común de programación 11.6

Si no se ha atrapado una excepción cuando el control entra a un bloque `finally`, y éste lanza una excepción que no se atrapa en el bloque `finally`, se perderá la primera excepción y se devolverá la del bloque `finally` al método que hizo la llamada.

Tip para prevenir errores 11.5

Evite colocar código que pueda lanzar (`throw`) una excepción en un bloque `finally`. Si se requiere dicho código, enciérrelo en bloques `try...catch` dentro del bloque `finally`.

Error común de programación 11.7

Suponer que una excepción lanzada desde un bloque `catch` se procesará por ese bloque `catch`, o por cualquier otro bloque `catch` asociado con la misma instrucción `try`, puede provocar errores lógicos.

Buena práctica de programación 11.2

El mecanismo de manejo de excepciones de Java está diseñado para eliminar el código de procesamiento de errores de la línea principal del código de un programa, para así mejorar su legibilidad. No coloque bloques `try...catch...finally` alrededor de cada instrucción que pueda lanzar una excepción. Esto dificulta la legibilidad de los programas. En vez de ello, coloque un bloque `try` alrededor de una porción considerable de su código, y después de ese bloque `try` coloque bloques `catch` para manejar cada posible excepción, y después de esos bloques `catch` coloque un solo bloque `finally` (si se requiere).

11.7 Limpieza de la pila y obtención de información de un objeto excepción

Cuando se lanza una excepción, pero no se atrapa en un alcance específico, la pila de llamadas a métodos se "limpia" y se hace un intento de atrapar (`catch`) la excepción en el siguiente bloque `try` exterior. A este proceso se le conoce como **limpieza de la pila**. Limpiar la pila de llamadas a métodos significa que el método en el que no se atrapó la excepción *termina*, todas las variables locales en ese método quedan fuera de alcance y el control regresa a la instrucción que invocó originalmente a ese método. Si un bloque `try` encierra a esa instrucción, se hace un intento de atrapar (`catch`) esa excepción. Si un bloque `try` no encierra a esa instrucción o si no se atrapa la excepción, se lleva a cabo la limpieza de la pila otra vez. La figura 11.5 demuestra la limpieza de la pila, y el manejador de excepciones en `main` muestra cómo acceder a los datos en un objeto excepción.

Limpieza de la pila

En `main`, el bloque `try` (líneas 8 a 11) llama a `metodo1` (declarado en las líneas 35 a 38), el cual a su vez llama a `metodo2` (declarado en las líneas 41 a 44), que a su vez llama a `metodo3` (declarado en las líneas 47 a 50). La línea 49 de `metodo3` lanza un objeto `Exception`: éste es el *punto de lanzamiento*. Puesto que la instrucción `throw` en la línea 49 *no* está encerrada en un bloque `try`, se produce la *limpieza de la pila*; `metodo3` termina en la línea 49 y después devuelve el control a la instrucción en `metodo2` que invocó a `metodo3` (es decir, la línea 43). Debido a que *ningún* bloque `try` encierra la línea 43, se produce

```
1  // Fig. 11.5: UsoDeExcepciones.java
2  // Limpieza de la pila y obtención de datos de un objeto excepción.
3
4  public class UsoDeExcepciones
5  {
6     public static void main( String[] args )
7     {
8        try
9        {
10          metodo1(); // llama a metodo1
11       } // fin de try
12       catch ( Exception excepcion ) // atrapa la excepción lanzada en metodo1
13       {
14          System.err.printf( "%s\n\n", excepcion.getMessage() );
15          excepcion.printStackTrace(); // imprime el rastreo de la pila de la excepción
16
17          // obtiene la información de rastreo de la pila
18          StackTraceElement[] elementosRastreo = excepcion.getStackTrace();
19
20          System.out.println( "\nRastreo de la pila de getStackTrace:" );
21          System.out.println( "Clase\t\t\tArchivo\t\t\tLinea\tMetodo" );
22
23          // itera a través de elementosRastreo para obtener la descripción de la
               excepción
24          for ( StackTraceElement elemento : elementosRastreo )
25          {
26             System.out.printf( "%s\t", elemento.getClassName() );
27             System.out.printf( "%s\t", elemento.getFileName() );
28             System.out.printf( "%s\t", elemento.getLineNumber() );
29             System.out.printf( "%s\n", elemento.getMethodName() );
30          } // fin de for
31       } // fin de catch
32    } // fin de main
33
34    // llama a metodo2; lanza las excepciones de vuelta a main
35    public static void metodo1() throws Exception
36    {
37       metodo2();
38    } // fin del método metodo1
39
40    // llama a metodo3; lanza las excepciones de vuelta a metodo1
41    public static void metodo2() throws Exception
42    {
43       metodo3();
44    } // fin del método metodo2
45
46    // lanza la excepción Exception de vuelta a metodo2
47    public static void metodo3() throws Exception
48    {
49       throw new Exception( "La excepcion se lanzo en metodo3" );
50    } // fin del método metodo3
51 } // fin de la clase UsoDeExcepciones
```

Fig. 11.5 | Limpieza de la pila y obtención de datos de un objeto excepción (parte 1 de 2).

```
La excepcion se lanzo en metodo3

java.lang.Exception: La excepcion se lanzo en metodo3
        at UsoDeExcepciones.metodo3(UsoDeExcepciones.java:49)
        at UsoDeExcepciones.metodo2(UsoDeExcepciones.java:43)
        at UsoDeExcepciones.metodo1(UsoDeExcepciones.java:37)
        at UsoDeExcepciones.main(UsoDeExcepciones.java:10)

Rastreo de la pila de getStackTrace:
Clase              Archivo                    Linea   Metodo
UsoDeExcepciones   UsoDeExcepciones.java      49      metodo3
UsoDeExcepciones   UsoDeExcepciones.java      43      metodo2
UsoDeExcepciones   UsoDeExcepciones.java      37      metodo1
UsoDeExcepciones   UsoDeExcepciones.java      10      main
```

Fig. 11.5 | Limpieza de la pila y obtención de datos de un objeto excepción (parte 2 de 2).

la *limpieza de la pila* otra vez; `metodo2` termina en la línea 43 y devuelve el control a la instrucción en `metodo1` que invocó a `metodo2` (es decir, la línea 37). Como *ningún* bloque `try` encierra la línea 37, se produce una vez más la *limpieza de la pila*; `metodo1` termina en la línea 37 y devuelve el control a la instrucción en `main` que invocó a `metodo1` (la línea 10). El bloque `try` de las líneas 8 a 11 encierra a esta instrucción. Ya que no se manejó la excepción, el bloque `try` termina y el primer bloque `catch` concordante (líneas 12 a 31) atrapa y procesa la excepción. Si no hubiera bloques `catch` que coincidieran, y la excepción no se declara en cada método que la lanza, se produciría un error de compilación. Recuerde que éste no es siempre el caso; para las excepciones *no verificadas* la aplicación se compilará, pero se ejecutará con resultados inesperados.

Obtención de datos de un objeto excepción

Recuerde que las excepciones se derivan de la clase `Throwable`. Esta clase ofrece un método llamado `printStackTrace`, que envía al flujo de error estándar el rastreo de la pila (lo cual se describe en la sección 11.2). A menudo, esto ayuda en la prueba y en la depuración. La clase `Throwable` también proporciona un método llamado `getStackTrace`, que obtiene la información de rastreo de la pila que podría imprimir `printStackTrace`. El método `getMessage` de la clase `Throwable` devuelve la cadena descriptiva almacenada en una excepción.

Tip para prevenir errores 11.6

Una excepción que no sea atrapada en una aplicación hará que se ejecute el manejador de excepciones predeterminado *de Java. Éste muestra el nombre de la excepción, un mensaje descriptivo que indica el problema que ocurrió y un rastreo completo de la pila de ejecución. En una aplicación con un solo hilo de ejecución, la aplicación termina. En una aplicación con varios hilos, termina el hilo que produjo la excepción.*

Tip para prevenir errores 11.7

El método `toString` de `Throwable` (heredado en todas las subclases de `Throwable`) devuelve un objeto `String` que contiene el nombre de la clase de la excepción y un mensaje descriptivo.

El manejador de `catch` en la figura 11.5 (líneas 12 a 31) demuestra el uso de `getMessage`, `printStackTrace` y `getStackTrace`. Si queremos mostrar la información de rastreo de la pila a flujos que no

sean el flujo de error estándar, podemos utilizar la información devuelta por `getStackTrace` y enviar estos datos a otro flujo, o usar las versiones sobrecargadas del método `printStackTrace`. En el capítulo 17 (en el sitio Web del libro) veremos cómo enviar datos a otros flujos.

En la línea 14 se invoca al método `getMessage` de la excepción, para obtener la descripción de la misma. En la línea 15 se invoca al método `printStackTrace` de la excepción, para mostrar el rastreo de la pila, el cual indica en dónde ocurrió la excepción. En la línea 18 se invoca al método `getStackTrace` de la excepción, para obtener la información del rastreo de la pila, como un arreglo de objetos `StackTraceElement`. En las líneas 24 a 30 se obtiene cada uno de los objetos `StackTraceElement` en el arreglo, y se invocan sus métodos `getClassName`, `getFileName`, `getLineNumber` y `getMethodName` para obtener el nombre de la clase, el nombre del archivo, el número de línea y el nombre del método, respectivamente, para ese objeto `StackTraceElement`. Cada objeto `StackTraceElement` representa la llamada a un método en la pila de llamadas a métodos.

Los resultados del programa muestran que la información de rastreo de la pila que imprime `printStackTrace` sigue el patrón: *nombreClase.nombreMétodo(nombreArchivo:númeroLínea)*, en donde *nombreClase*, *nombreMétodo* y *nombreArchivo* indican los nombres de la clase, el método y el archivo en los que ocurrió la excepción, respectivamente, y *númeroLínea* indica en qué parte del archivo ocurrió la excepción. Usted vio esto en los resultados para la figura 11.1. El método `getStackTrace` permite un procesamiento personalizado de la información sobre la excepción. Compare la salida de `printStackTrace` con la salida creada a partir de los objetos `StackTraceElement`, y podrá ver que ambos contienen la misma información de rastreo de la pila.

Observación de ingeniería de software 11.10

Nunca proporcione un manejador `catch` con un cuerpo vacío; eso en definitiva ignora la excepción. Use por lo menos el método `printStackTrace` para imprimir un mensaje de error e indicar que existe un problema.

11.8 Excepciones encadenadas

Algunas veces un método responde a una excepción lanzando un tipo distinto de excepción, específico para la aplicación actual. Si un bloque `catch` lanza una nueva excepción, se *pierden* tanto la información como el rastreo de la pila de la excepción original. En las primeras versiones de Java, no había mecanismo para envolver la información de la excepción original con la de la nueva excepción, para proporcionar un rastreo completo de la pila, indicando en dónde ocurrió el problema original en el programa. Esto hacía que depurar dichos problemas fuera un proceso bastante difícil. Las **excepciones encadenadas** permiten que un objeto excepción mantenga la información completa sobre el rastreo de la pila de la excepción original. En la figura 11.6 se demuestran las excepciones encadenadas.

```
1  // Fig. 11.6: UsoDeExcepcionesEncadenadas.java
2  // Las excepciones encadenadas.
3
4  public class UsoDeExcepcionesEncadenadas
5  {
6     public static void main( String[] args )
7     {
8        try
9        {
```

Fig. 11.6 | Excepciones encadenadas (parte 1 de 2).

```
10            metodo1(); // llama a metodo1
11         } // fin de try
12         catch ( Exception excepcion ) // excepciones lanzadas desde metodo1
13         {
14            excepcion.printStackTrace();
15         } // fin de catch
16      } // fin de main
17
18      // llama a metodo2; lanza las excepciones de vuelta a main
19      public static void metodo1() throws Exception
20      {
21         try
22         {
23            metodo2(); // llama a metodo2
24         } // fin de try
25         catch ( Exception excepcion ) // excepción lanzada desde metodo2
26         {
27            throw new Exception( "La excepcion se lanzo en metodo1", excepcion );
28         } // fin de catch
29      } // fin del método metodo1
30
31      // llama a metodo3; lanza las excepciones de vuelta a metodo1
32      public static void metodo2() throws Exception
33      {
34         try
35         {
36            metodo3(); // llama a metodo3
37         } // fin de try
38         catch ( Exception excepcion ) // excepción lanzada desde metodo3
39         {
40            throw new Exception( "La excepcion se lanzo en metodo2", excepcion );
41         } // fin de catch
42      } // fin del método metodo2
43
44      // lanza excepción Exception de vuelta a metodo2
45      public static void metodo3() throws Exception
46      {
47         throw new Exception( "La excepcion se lanzo en metodo3" );
48      } // fin del método metodo3
49   } // fin de la clase UsoDeExcepcionesEncadenadas
```

```
java.lang.Exception: La excepcion se lanzo en metodo1
        at UsoDeExcepcionesEncadenadas.metodo1(UsoDeExcepcionesEncadenadas.java:27)
        at UsoDeExcepcionesEncadenadas.main(UsoDeExcepcionesEncadenadas.java:10)
Caused by: java.lang.Exception: La excepcion se lanzo en metodo2
        at UsoDeExcepcionesEncadenadas.metodo2(UsoDeExcepcionesEncadenadas.java:40)
        at UsoDeExcepcionesEncadenadas.metodo1(UsoDeExcepcionesEncadenadas.java:23)
        ... 1 more
Caused by: java.lang.Exception: La excepcion se lanzo en metodo3
        at UsoDeExcepcionesEncadenadas.metodo3(UsoDeExcepcionesEncadenadas.java:47)
        at UsoDeExcepcionesEncadenadas.metodo2(UsoDeExcepcionesEncadenadas.java:36)
        ... 2 more
```

Fig. 11.6 | Excepciones encadenadas (parte 2 de 2).

El programa consiste de cuatro métodos: `main` (líneas 6 a 16), `metodo1` (líneas 19 a 29), `metodo2` (líneas 32 a 42) y `metodo3` (líneas 45 a 48). La línea 10 en el bloque `try` de `main` llama a `metodo1`. La línea 23 en el bloque `try` de `metodo1` llama a `metodo2`. La línea 36 en el bloque `try` de `metodo2` llama a `metodo3`. En `metodo3`, la línea 47 lanza una nueva excepción `Exception`. Como esta instrucción no se encuentra dentro de un bloque `try`, el `metodo3` termina y la excepción se devuelve al método que hace la llamada (`metodo2`), en la línea 36. Esta instrucción *se encuentra* dentro de un bloque `try`; por lo tanto, el bloque `try` termina y la excepción es atrapada en las líneas 38 a 41. En la línea 40, en el bloque `catch`, se lanza una nueva excepción. En este caso, se hace una llamada al constructor `Exception` con *dos* argumentos). El segundo argumento representa a la excepción que era la causa original del problema. En este programa, la excepción ocurrió en la línea 47. Como se lanza una excepción desde el bloque `catch`, el `metodo2` termina y devuelve la nueva excepción al método que hace la llamada (`metodo1`), en la línea 23. Una vez más, esta instrucción se encuentra dentro de un bloque `try`, por lo tanto este bloque termina y la excepción es atrapada en las líneas 25 a 28. En la línea 27, en el bloque `catch` se lanza una nueva excepción y se utiliza la excepción que se atrapó como el segundo argumento para el constructor de `Exception`. Puesto que se lanza una excepción desde el bloque `catch`, el `metodo1` termina y devuelve la nueva excepción al método que hace la llamada (`main`), en la línea 10. El bloque `try` en `main` termina y la excepción es atrapada en las líneas 12 a 15. En la línea 14 se imprime un rastreo de la pila.

Observe en la salida del programa que las primeras tres líneas muestran la excepción más reciente que fue lanzada (es decir, la del `metodo1` en la línea 27). Las siguientes cuatro líneas indican la excepción que se lanzó desde el `metodo2`, en la línea 40. Por último, las siguientes cuatro líneas representan la excepción que se lanzó desde el `metodo3`, en la línea 47. Además observe que, si lee la salida en forma inversa, muestra cuántas excepciones encadenadas más quedan pendientes.

11.9 Declaración de nuevos tipos de excepciones

La mayoría de los programadores de Java utilizan las clases existentes de la API de Java, de distribuidores independientes y de bibliotecas de clases gratuitas (que por lo general se pueden descargar de Internet) para crear aplicaciones de Java. Los métodos de esas clases suelen declararse para lanzar las excepciones apropiadas cuando ocurren problemas. Los programadores escriben código para procesar esas excepciones existentes, de modo que sus programas sean más robustos.

Si usted crea clases que otros programadores utilizarán en sus programas, tal vez le sea conveniente declarar sus propias clases de excepciones que sean específicas para los problemas que pueden ocurrir cuando otro programador utilice sus clases reutilizables.

Observación de ingeniería de software 11.11

De ser posible, indique las excepciones de sus métodos mediante el uso de las clases de excepciones existentes, en vez de crear nuevas. La API de Java contiene muchas clases de excepciones que podrían ser adecuadas para el tipo de problema que su método necesite indicar.

Una nueva clase de excepción debe extender a una clase de excepción existente, para poder asegurar que la clase pueda utilizarse con el mecanismo de manejo de excepciones. De la misma manera que cualquier otra clase, una clase de excepción puede contener campos y métodos. Una nueva clase de excepción por lo general contiene sólo cuatro constructores; uno que no toma argumentos y pasa un objeto `String` como mensaje de error predeterminado al constructor de la superclase; uno que recibe un mensaje de error personalizado como un objeto `String` y lo pasa al constructor de la superclase; uno que recibe un mensaje de error personalizado como un objeto `String` y un objeto `Throwable` (para encadenar excepciones), y pasa ambos objetos al constructor de la superclase; y uno que recibe un objeto `Throwable` (para encadenar excepciones) y pasa sólo este objeto al constructor de la superclase.

Buena práctica de programación 11.3

Asociar cada uno de los tipos de fallas graves en tiempo de ejecución con una clase `Exception` con nombre apropiado ayuda a mejorar la claridad del programa.

Observación de ingeniería de software 11.12

Al definir su propio tipo de excepción, estudie las clases de excepción existentes en la API de Java y trate de extender una clase de excepción relacionada. Por ejemplo, si va a crear una nueva clase para representar cuando un método intenta realizar una división entre cero, podría extender la clase `ArithmeticException`, ya que la división entre cero ocurre durante la aritmética. Si las clases existentes no son superclases apropiadas para su nueva clase de excepción, debe decidir si su nueva clase debe ser una clase de excepción verificada o no verificada. La nueva clase de excepción debe ser una excepción verificada (es decir, debe extender a `Exception` pero no a `RuntimeException`) si los clientes deben tener que manejar la excepción. La aplicación cliente debe ser capaz de recuperarse en forma razonable de una excepción de este tipo. La nueva clase de excepción debe extender a `RuntimeException` si el código cliente debe ser capaz de ignorar la excepción (es decir, si la excepción es una excepción no verificada).

En el capítulo 22 (en inglés en el sitio Web del libro), proporcionaremos un ejemplo de una clase de excepción personalizada. Declararemos una clase reutilizable llamada `Lista`, la cual es capaz de almacenar una lista de referencias a objetos. Algunas operaciones que se realizan comúnmente en una `Lista` no se permitirán si la `Lista` está vacía, como eliminar un elemento de la parte frontal o posterior de la lista. Por esta razón, algunos métodos de `Lista` lanzan excepciones de la clase de excepción `ListaVaciaException`.

Buena práctica de programación 11.4

Por convención, todos los nombres de las clases de excepciones deben terminar con la palabra `Exception`.

11.10 Precondiciones y poscondiciones

Los programadores invierten una gran parte de su tiempo en mantener y depurar código. Para facilitar estas tareas y mejorar el diseño en general, comúnmente especifican los estados esperados antes y después de la ejecución de un método. A estos estados se les llama precondiciones y poscondiciones, respectivamente.

Una **precondición** debe ser verdadera cuando se *invoca* a un método. Las precondiciones describen las restricciones en los parámetros de un método, y en cualquier otra expectativa que tenga el método en relación con el estado actual de un programa, justo antes de empezar a ejecutarse. Si no se cumplen las precondiciones, entonces el comportamiento del método es *indefinido*; puede lanzar una excepción, continuar con un valor ilegal o tratar de recuperarse del error. Nunca hay que esperar un comportamiento consistente si no se cumplen las precondiciones.

Una **poscondición** es verdadera *una vez que el método regresa con éxito*. Las poscondiciones describen las restricciones en el valor de retorno, y en cualquier otro efecto secundario que pueda tener el método. Al definir un método, debe documentar todas las poscondiciones, de manera que otros sepan qué pueden esperar al llamar a su método, y usted debe asegurarse que su método cumpla con todas sus poscondiciones, si en definitiva se cumplen sus precondiciones.

Cuando no se cumplen sus precondiciones o poscondiciones, los métodos por lo general lanzan excepciones. Como ejemplo, examine el método `charAt` de `String`, que tiene un parámetro `int`: un índice en el objeto `String`. Para una precondición, el método `charAt` asume que `indice` es mayor o igual

que cero, y menor que la longitud del objeto `String`. Si se cumple la precondición, ésta establece que el método devolverá el carácter en la posición en el objeto `String` especificada por el parámetro `indice`. En caso contrario, el método lanza una excepción `IndexOutOfBoundsException`. Confiamos en que el método `charAt` satisfaga su poscondición, siempre y cuando cumplamos con la precondición. No necesitamos preocuparnos por los detalles acerca de cómo el método obtiene en realidad el carácter en el índice.

Por lo general, las precondiciones y poscondiciones de un método se describen como parte de su especificación. Al diseñar sus propios métodos, debe indicar las precondiciones y poscondiciones en un comentario antes de la declaración del método.

11.11 Aserciones

Al implementar y depurar una clase, algunas veces es conveniente establecer condiciones que deban ser verdaderas en un punto específico de un método. Estas condiciones, conocidas como aserciones, ayudan a asegurar la validez de un programa al atrapar los errores potenciales e identificar los posibles errores lógicos durante el desarrollo. Las precondiciones y las poscondiciones son dos tipos de aserciones. Las precondiciones son aserciones acerca del estado de un programa a la hora de invocar un método, y las poscondiciones son aserciones acerca del estado de un programa cuando el método termina.

Aunque las aserciones pueden establecerse como comentarios para guiar al programador durante el desarrollo del programa, Java incluye dos versiones de la instrucción `assert` para validar aserciones mediante la programación. La instrucción `assert` evalúa una expresión `boolean` y, si es `false`, lanza una excepción `AssertionError` (una subclase de `Error`). La primera forma de la instrucción `assert` es

```
assert expresión;
```

la cual lanza una excepción `AssertionError` si *expresión* es `false`. La segunda forma es

```
assert expresión1 : expresión2;
```

que evalúa *expresión1* y lanza una excepción `AssertionError` con *expresión2* como el mensaje de error, en caso de que *expresión1* sea `false`.

Puede utilizar aserciones para implementar las precondiciones y poscondiciones mediante la programación, o para verificar cualquier otro estado intermedio que le ayude a asegurar que su código esté funcionando en forma correcta. La figura 11.7 demuestra la instrucción `assert`. En la línea 11 se pide al usuario que introduzca un número entre 0 y 10, y después en la línea 12 se lee el número. La línea 15 determina si el usuario introdujo un número dentro del rango válido. Si el número está fuera de rango, la instrucción `assert` reporta un error; en caso contrario, el programa continúa en forma normal.

```
1  // Fig. 11.7: PruebaAssert.java
2  // Comprobar mediante assert que un valor esté dentro del rango
3  import java.util.Scanner;
4
5  public class PruebaAssert
6  {
7     public static void main( String[] args )
8     {
9        Scanner entrada = new Scanner( System.in );
10
```

Fig. 11.7 | Verificar con `assert` que un valor se encuentre dentro del rango (parte 1 de 2).

```
11         System.out.print( "Escriba un numero entre 0 y 10: " );
12         int numero = entrada.nextInt();
13
14         // asegura que el valor sea >= 0 y <= 10
15         assert ( numero >= 0 && numero <= 10 ) : "numero incorrecto: " + numero;
16
17         System.out.printf( "Usted escribio %d\n", numero );
18      } // fin de main
19   } // fin de la clase PruebaAssert
```

```
Escriba un numero entre 0 y 10: 5
Usted escribio 5
```

```
Escriba un numero entre 0 y 10: 50
Exception in thread "main" java.lang.AssertionError: numero incorrecto: 50
        at PruebaAssert.main(PruebaAssert.java:15)
```

Fig. 11.7 | Verificar con `assert` que un valor se encuentre dentro del rango (parte 2 de 2).

El programador utiliza las aserciones principalmente para depurar e identificar errores lógicos en una aplicación. Hay que habilitar las aserciones de manera explícita al ejecutar un programa, ya que reducen el rendimiento y son innecesarias para el usuario del mismo. Para ello, use la opción de línea de comandos `-ea` del comando `java`, como en

```
java -ea PruebaAssert
```

Los usuarios no deben encontrar ningún error tipo `AssertionError` durante la ejecución normal de un programa escrito en forma apropiada. Dichos errores sólo deben indicar errores (bugs) en la implementación. Como resultado, nunca se debe atrapar una excepción tipo `AssertionError`. En vez de ello, debemos permitir que el programa termine al ocurrir el error, para poder ver el mensaje de error; después hay que localizar y corregir el origen del problema. Como los usuarios de las aplicaciones pueden elegir no habilitar las aserciones en tiempo de ejecución, no debemos usar la instrucción `assert` para indicar problemas en tiempo de ejecución en el código de producción. En vez de ello, debemos usar el mecanismo de las excepciones.

11.12 (Nuevo en Java SE 7) Cláusula `catch` múltiple: atrapar varias excepciones en un `catch`

Es muy común que después de un bloque `try` haya varios bloques `catch` para manejar diversos tipos de excepciones. Si los cuerpos de varios bloques `catch` son idénticos, el programador puede usar la nueva característica **multi-catch** de Java SE 7 para atrapar esos tipos de excepciones en un solo manejador `catch` y realizar la misma tarea. La sintaxis para el multi-`catch` es:

```
catch ( Tipo1 | Tipo2 | Tipo3 e )
```

Cada tipo de excepción se separa del siguiente con una barra vertical (|). La línea anterior de código indica que es posible atrapar uno de los tipos especificados (o cualquier subclase de esos tipos) en el manejador de excepciones. En una cláusula multi-`catch`, se puede especificar cualquier cantidad de tipos `Throwable`.

11.13 (Nuevo en Java SE 7): Cláusula try con recursos (try-with-resources): desasignación automática de recursos

Por lo general, el código para liberar recursos debe colocarse en un bloque `finally`, para asegurar que se libere un recurso sin importar que se hayan lanzado excepciones cuando se utilizó ese recurso en el bloque `try` correspondiente. Hay una notación alternativa, la instrucción **try con recursos** (nueva en Java SE 7), que simplifica la escritura de código en donde se obtienen uno o más recursos, se utilizan en un bloque `try` y se liberan en el correspondiente bloque `finally`. Por ejemplo, una aplicación de procesamiento de archivos (capítulo 17, en el sitio Web del libro) podría procesar un archivo con una instrucción `try` con recursos, para asegurar que el archivo se cierre de manera apropiada cuando ya no se necesite. Cada recurso debe ser un objeto de una clase que implemente a la interfaz `AutoCloseable`; dicha clase tiene un método llamado `close`. La forma general de una instrucción `try` con recursos es:

```
try ( NombreClase elObjeto = new NombreClase() )
{
   // aquí se usa elObjeto
}
catch ( Exception e )
{
   // atrapa las excepciones que ocurren al usar el recurso
}
```

en donde *NombreClase* es una clase que implementa a la interfaz `AutoCloseable`. Este código crea un objeto de tipo *NombreClase* y lo utiliza en el bloque `try`, después llama a su método `close` para liberar los recursos utilizados por el objeto. La instrucción `try` con recursos llama de manera *implícita* al método `close` de `elObjeto` *al final del bloque* `try`. Usted puede asignar varios recursos en los paréntesis que van después de `try`, separándolos con un signo de punto y coma (;).

11.14 Conclusión

En este capítulo aprendió a utilizar el manejo de excepciones para lidiar con los errores. Aprendió que el manejo de excepciones permite a los programadores eliminar el código para manejar errores de la "línea principal" de ejecución del programa. Le mostramos cómo utilizar los bloques `try` para encerrar código que puede lanzar una excepción, y cómo utilizar los bloques `catch` para lidiar con las excepciones que puedan surgir. Aprendió acerca del modelo de terminación del manejo de excepciones, el cual indica que una vez que se maneja una excepción, el control del programa no regresa al punto de lanzamiento. Vimos la diferencia entre las excepciones verificadas y no verificadas, y cómo especificar mediante la cláusula `throws` las excepciones que podría lanzar un método. Aprendió a utilizar el bloque `finally` para liberar recursos, ya sea que ocurra o no una excepción. También aprendió a lanzar y volver a lanzar excepciones. Después, aprendió a obtener información acerca de una excepción, mediante el uso de los métodos `printStackTrace`, `getStackTrace` y `getMessage`. Luego le presentamos las excepciones encadenadas, que permiten a los programadores envolver la información de la excepción original con la información de la nueva excepción. Después, le enseñamos a crear sus propias clases de excepciones. Después presentamos las precondiciones y poscondiciones para ayudar a los programadores que utilizan sus métodos a comprender las condiciones que deben ser verdaderas cuando se hace la llamada al método y cuando éste regresa, respectivamente. Cuando no se cumplen las precondiciones y poscondiciones, los métodos por lo general lanzan excepciones. Hablamos sobre la instrucción `assert` y cómo puede utilizarse para ayudarnos a depurar los programas. En especial, esta instrucción se puede utilizar para asegurar que se cumplan las precondiciones y poscondiciones. Por último, le presentamos las nuevas herramientas para ma-

nejar excepciones de Java SE 7, incluyendo la cláusula `catch` múltiple para procesar varios tipos de excepciones en el mismo manejador `catch`, y la instrucción `try` con recursos para desasignar de manera automática un recurso después de usarlo en el bloque `try`. En el siguiente capítulo empezaremos nuestro caso de estudio opcional, que abarca dos capítulos, sobre el diseño orientado a objetos con el UML.

Resumen

Sección 11.1 Introducción

- Una excepción es una indicación de un problema que ocurre durante la ejecución de un programa.
- El manejo de excepciones permite a los programadores crear aplicaciones que puedan resolver las excepciones.

Sección 11.2 Ejemplo: división entre cero sin manejo de excepciones

- Las excepciones se lanzan cuando un método detecta un problema y no puede manejarlo.
- El rastreo de la pila de una excepción (pág. 441) incluye el nombre de la excepción en un mensaje que el problema y la pila de llamadas a métodos completa en el momento en el que ocurrió la excepción.
- El punto en el programa en el cual ocurre una excepción se conoce como punto de lanzamiento (pág. 441).

Sección 11.3 Ejemplo: manejo de excepciones `ArithmeticException` e `InputMismatchException`

- Un bloque `try` (pág. 444) encierra el código que podría lanzar (`throw`) una excepción, y el código que no debe ejecutarse si se produce esa excepción.
- Las excepciones pueden surgir a través de código mencionado explícitamente en un bloque `try`, a través de llamadas a otros métodos, o incluso a través de llamadas a métodos anidados, iniciadas por el código en el bloque `try`.
- Un bloque `catch` (pág. 444) empieza con la palabra clave `catch` y un parámetro de excepción, seguido de un bloque de código que maneja la excepción. Este código se ejecuta cuando el bloque `try` detecta la excepción.
- Una excepción no atrapada es una excepción que ocurre y no tiene bloques `catch` que coincidan.
- Una excepción no atrapada (pág. 445) hará que un programa termine antes de tiempo, si éste sólo contiene un hilo. De lo contrario, sólo terminará el hilo en el que ocurrió la excepción. El resto del programa se ejecutará, pero puede producir efectos adversos.
- Justo después del bloque `try` debe ir por lo menos un bloque `catch` o un bloque `finally` (pág. 444).
- Un bloque `catch` especifica entre paréntesis un parámetro de excepción, el cual identifica el tipo de excepción a manejar. El nombre del parámetro permite al bloque `catch` interactuar con un objeto de excepción atrapada.
- Si ocurre una excepción en un bloque `try`, éste termina de inmediato y el control del programa se transfiere al primero de los siguientes bloques `catch` cuyo parámetro de excepción coincida con el tipo de la excepción que se lanzó.
- Una vez que se maneja una excepción, el control del programa no regresa al punto de lanzamiento, ya que el bloque `try` ha expirado. A esto se le conoce como el modelo de terminación del manejo de excepciones (pág. 445).
- Si hay varios bloques `catch` que coinciden cuando ocurre una excepción, sólo se ejecuta el primero.
- Una cláusula `throws` (pág. 446) especifica una lista de excepciones separadas por comas que el método podría lanzar, y aparece después de la lista de parámetros del método, pero antes de su cuerpo.

Sección 11.4 Cuándo utilizar el manejo de excepciones

- El manejo de excepciones procesa errores sincrónicos (pág. 447), que ocurren cuando se ejecuta una instrucción.
- El manejo de excepciones no está diseñado para procesar los problemas asociados con eventos asíncronos (pág. 447), que ocurren en paralelo con (y en forma independiente de) el flujo de control del programa.

Sección 11.5 Jerarquía de excepciones de Java

- Todas las clases de excepciones de Java heredan, ya sea en forma directa o indirecta, de la clase `Exception`.
- Los programadores pueden extender la jerarquía de excepciones de Java con sus propias clases de excepciones.
- La clase `Throwable` es la superclase de la clase `Exception`, y por lo tanto es también la superclase de todas las excepciones. Sólo pueden usarse objetos `Throwable` con el mecanismo para manejar excepciones.
- La clase `Throwable` (pág. 447) tiene dos subclases: `Exception` y `Error`.
- La clase `Exception` y sus subclases representan situaciones excepcionales que podrían ocurrir en un programa de Java y ser atrapados por la aplicación.
- La clase `Error` y sus subclases representan problemas que podrían ocurrir en el sistema en tiempo de ejecución de Java. Los errores tipo `Error` ocurren con poca frecuencia, y por lo general no deben ser atrapados por una aplicación.
- Java clasifica a las excepciones en dos categorías (pág. 448): verificadas y no verificadas.
- El compilador de Java no verifica el código para determinar si una excepción no verificada se atrapa o se declara. Por lo general, las excepciones no verificadas se pueden evitar mediante una codificación apropiada.
- Las subclases de `RuntimeException` representan excepciones no verificadas. Todos los tipos de excepciones que heredan de la clase `Exception`, pero no de `RuntimeException` (pág. 448), son verificadas.
- Si se escribe un bloque `catch` para atrapar los objetos de excepción de un tipo de la superclase, también puede atrapar a todos los objetos de las subclases de esa clase. Esto permite el procesamiento polimórfico de las excepciones relacionadas.

Sección 11.6 Bloque `finally`

- Los programas que obtienen ciertos tipos de recursos deben devolverlos al sistema para evitar las denominadas fugas de recursos (pág. 450). Por lo general, el código para liberar recursos se coloca en un bloque `finally` (pág. 450).
- El bloque `finally` es opcional. Si está presente, se coloca después del último bloque `catch`.
- El bloque `finally` se ejecutará sin importar que se lance o no una excepción en el bloque `try` correspondiente, o en uno de sus correspondientes bloques `catch`.
- Si una excepción no se puede atrapar mediante uno de los manejadores `catch` asociados a ese bloque `try`, el control pasa al bloque `finally`. Después, la excepción se pasa al siguiente bloque `try` exterior.
- Si un bloque `catch` lanza una excepción, de todas formas se ejecuta el bloque `finally`. Después, la excepción se pasa al siguiente bloque `try` exterior.
- Una instrucción `throw` (pág. 453) puede lanzar cualquier objeto `Throwable`.
- Las excepciones se vuelven a lanzar (pág. 453) cuando un bloque `catch`, al momento de recibir una excepción, decide que no puede procesarla, o que sólo puede procesarla en forma parcial. Al volver a lanzar una excepción se difiere el manejo de excepciones (o tal vez una parte de éste) a otro bloque `catch`.
- Cuando se vuelve a lanzar una excepción, el siguiente bloque `try` circundante detecta la excepción que se volvió a lanzar, y los bloques `catch` de ese bloque `try` tratan de manejarla.

Sección 11.7 Limpieza de la pila y obtención de información de un objeto excepción

- Cuando se lanza una excepción, pero no se atrapa en un alcance específico, se limpia la pila de llamadas a métodos y se hace un intento por atrapar la excepción en la siguiente instrucción `try` exterior.
- La clase `Throwable` ofrece un método `printStackTrace`, el cual imprime la pila de llamadas a métodos. A menudo, esto es útil en la prueba y depuración.
- La clase `Throwable` también proporciona un método `getStackTrace`, que obtiene la misma información de rastreo de la pila que `printStackTrace` imprime (pág. 456).
- El método `getMessage` de la clase `Throwable` (pág. 456) devuelve la cadena descriptiva almacenada en una excepción.
- El método `getStackTrace` (pág. 456) obtiene la información de rastreo de la pila como un arreglo de objetos `StackTraceElement`. Cada objeto `StackTraceElement` representa una llamada a un método en la pila de llamadas a métodos.

- Los métodos `getClassName`, `getFileName`, `getLineNumber` y `getMethodName` de la clase `StackTraceElement` (pág. 457) obtienen el nombre de la clase, el nombre de archivo, el número de línea y el nombre del método.

Sección 11.8 Excepciones encadenadas

- Las excepciones encadenadas (pág. 457) permiten que un objeto de excepción mantenga la información de rastreo de la pila completa, incluyendo la información acerca de las excepciones anteriores que provocaron la excepción actual.

Sección 11.9 Declaración de los nuevos tipos de excepciones

- Una nueva clase de excepción debe extender a una existente, para asegurar que la clase pueda usarse con el mecanismo de manejo de excepciones.

Sección 11.10 Precondiciones y poscondiciones

- La precondición de un método (pág. 460) debe ser verdadera al momento de invocar el método.
- La poscondición de un método (pág. 460) es verdadera una vez que regresa el método con éxito.
- Al diseñar sus propios métodos, debe establecer las precondiciones y poscondiciones en un comentario antes de la declaración del método.

Sección 11.11 Aserciones

- Las aserciones (pág. 461) ayudan a atrapar errores potenciales e identificar posibles errores lógicos.
- La instrucción `assert` (pág. 461) permite validar las aserciones mediante la programación.
- Para habilitar las aserciones en tiempo de ejecución, use el modificador `-ea` al ejecutar el comando java.

Sección 11.12 (Nuevo en Java SE7) Cláusula `catch` múltiple: atrapar varias excepciones en un `catch`

- La cláusula multi-catch de Java SE 7 (pág. 462) le permite atrapar varios tipos de excepciones en un solo manejador `catch` y realizar la misma tarea para cada tipo de excepción. La sintaxis para una instrucción `catch` múltiple es:

```
catch ( Tipo1 | Tipo2 | Tipo3 e )
```

- Cada tipo de excepción se separa del siguiente con una barra vertical (`|`).

Sección 11.13 (Nuevo en Java SE7) Cláusula `try` con recursos (`try-with-resources`): desasignación automática de recursos

- La instrucción `try` con recursos (pág. 463) simplifica la escritura de código en donde se obtiene un recurso, se utiliza en un bloque `try` y se libera el recurso en el correspondiente bloque `finally`. En cambio, se asigna el recurso en los paréntesis que van después de la palabra clave `try` y se utiliza el recurso en el bloque `try`; después, la instrucción llama de manera implícita al método `close` del recurso al final del bloque `try`.
- Cada recurso debe ser un objeto de una clase que implemente a la interfaz `AutoCloseable` (pág. 463); dicha clase tiene un método `close`.
- Puede asignar varios recursos en los paréntesis que van después de `try`, separándolos con un signo de punto y coma (`;`).

Ejercicios de autoevaluación

11.1 Mencione cinco ejemplos comunes de excepciones.

11.2 Dé varias razones por las cuales no deban utilizarse las técnicas de manejo de excepciones para el control convencional de los programas.

11.3 ¿Por qué son las excepciones apropiadas para tratar con los errores producidos por los métodos de las clases en la API de Java?

11.4 ¿Qué es una "fuga de recursos"?

11.5 Si no se lanzan excepciones en un bloque `try`, ¿hacia dónde procede el control cuando el bloque `try` completa su ejecución?

11.6 Mencione una ventaja clave del uso de `catch( Exception` *nombreExcepción* `)`.

11.7 ¿Debe una aplicación convencional atrapar los objetos `Error`? Explique.

11.8 ¿Qué ocurre si ningún manejador `catch` coincide con el tipo de un objeto lanzado?

11.9 ¿Qué ocurre si varios bloques `catch` coinciden con el tipo del objeto lanzado?

11.10 ¿Por qué debería un programador especificar un tipo de superclase como el tipo en un bloque `catch`?

11.11 ¿Cuál es la razón clave de utilizar bloques `finally`?

11.12 ¿Qué ocurre cuando un bloque `catch` lanza una excepción `Exception`?

11.13 ¿Qué hace la instrucción `throw` *referenciaExcepción* en un bloque `catch`?

11.14 ¿Qué ocurre a una referencia local en un bloque `try`, cuando ese bloque lanza una excepción `Exception`?

Respuestas a los ejercicios de autoevaluación

11.1 Agotamiento de memoria, índice de arreglo fuera de límites, desbordamiento aritmético, división entre cero, parámetros inválidos de método.

11.2 (a) El manejo de excepciones está diseñado para manejar las situaciones que ocurren con poca frecuencia y que a menudo provocan la terminación del programa, no situaciones que surjan todo el tiempo. (b) Por lo general, el flujo de control con estructuras de control convencionales es más claro y eficiente que con las excepciones. (c) Las excepciones adicionales pueden interponerse en el camino de las excepciones de tipos de errores genuinos. Es más difícil para el programador llevar el registro de un número más extenso de casos de excepciones.

11.3 Es muy poco probable que los métodos de clases en la API de Java puedan realizar un procesamiento de errores que cumpla con las necesidades únicas de todos los usuarios.

11.4 Una "fuga de recursos" ocurre cuando un programa en ejecución no libera apropiadamente un recurso cuando éste ya no es necesario.

11.5 Los bloques `catch` para esa instrucción `try` se ignoran y el programa reanuda su ejecución después del último bloque `catch`. Si hay bloque `finally`, se ejecuta primero y luego el programa reanuda su ejecución después del bloque `finally`.

11.6 La forma `catch( Exception` *nombreExcepción* `)` atrapa cualquier tipo de excepción lanzada en un bloque `try`. Una ventaja es que ninguna excepción `Exception` lanzada puede escabullirse sin ser atrapada. El programador puede entonces decidir si manejará la excepción o si posiblemente vuelva a lanzarla.

11.7 Las excepciones `Error` son generalmente problemas graves con el sistema de Java subyacente; en la mayoría de los programas no es conveniente atrapar excepciones `Error`, ya que el programa no podrá recuperarse de dichos problemas.

11.8 Esto hace que la búsqueda de una coincidencia continúe en la siguiente instrucción `try` circundante. Si hay un bloque `finally`, éste se ejecutará antes de que la excepción pase a la siguiente instrucción `try` circundante. Si no hay instrucciones `try` circundantes para las cuales haya bloques `catch` que coincidan, y las excepciones son declaradas (o no verificadas), se imprime un rastreo de la pila y el subproceso actual termina antes de tiempo. Si las excepciones son verificadas, pero no se atrapan o se declaran, ocurren errores de compilación.

11.9 Se ejecuta el primer bloque `catch` que coincida después del bloque `try`.

11.10 Esto permite a un programa atrapar tipos relacionados de excepciones, y procesarlos en una manera uniforme. Sin embargo, a menudo es conveniente procesar los tipos de subclases en forma individual, para un manejo de excepciones más preciso.

11.11 El bloque `finally` es el medio preferido para liberar recursos y evitar las fugas de éstos.

11.12 Primero, el control pasa al bloque `finally`, si existe uno. Después, la excepción se procesará mediante un bloque `catch` (si existe uno) asociado con un bloque `try` circundante (si existe uno).

11.13 Vuelve a lanzar la excepción para que la procese un manejador de excepciones de un bloque `try` circundante, una vez que se ejecuta el bloque `finally` de la instrucción `try` actual.

11.14 La referencia queda fuera de alcance. Si el objeto al que se hace referencia es inalcanzable, se marca para la recolección de basura.

Ejercicios

11.15 *(Condiciones excepcionales)* Enumere las diversas condiciones excepcionales que han ocurrido en programas, a lo largo de este texto hasta ahora. Mencione todas las condiciones excepcionales adicionales que pueda. Para cada una de ellas, describa brevemente la manera en que un programa manejaría la excepción, utilice las técnicas de manejo de excepciones que se describen en este capítulo. Algunas excepciones típicas son la división entre cero, y el índice de arreglo fuera de límites.

11.16 *(Excepciones y falla de los constructores)* Hasta este capítulo, hemos visto que tratar con los errores detectados por los constructores es algo difícil. Explique por qué el manejo de excepciones es un medio efectivo para tratar con las fallas en los constructores.

11.17 *(Atrapar excepciones con las superclases)* Use la herencia para crear una superclase de excepción (llamada `ExcepcionA`) y las subclases de excepción `ExcepcionB` y `ExcepcionC`, en donde `ExcepcionB` hereda de `ExcepcionA` y `ExcepcionC` hereda de `ExcepcionB`. Escriba un programa para demostrar que el bloque `catch` para el tipo `ExcepcionA` atrapa excepciones de los tipos `ExcepcionB` y `ExcepcionC`.

11.18 *(Atrapar excepciones mediante el uso de la clase `Exception`)* Escriba un programa que demuestre cómo se atrapan las diversas excepciones con

```
catch ( Exception excepcion )
```

Esta vez, defina las clases `ExcepcionA` (que hereda de la clase `Exception`) y `ExcepcionB` (que hereda de la clase `ExcepcionA`). En su programa, cree bloques `try` que lancen excepciones de los tipos `ExcepcionA`, `ExcepcionB`, `NullPointerException` e `IOException`. Todas las excepciones deberán atraparse con bloques `catch` que especifiquen el tipo `Exception`.

11.19 *(Orden de los bloques catch)* Escriba un programa que demuestre que el orden de los bloques `catch` es importante. Si trata de atrapar un tipo de excepción de superclase antes de un tipo de subclase, el compilador debe generar errores.

11.20 *(Falla del constructor)* Escribe un programa que muestre cómo un constructor pasa información sobre la falla del constructor a un manejador de excepciones. Defina la clase `UnaClase`, que lance una excepción `Exception` en el constructor. Su programa deberá tratar de crear un objeto de tipo `UnaClase` y atrapar la excepción que se lance desde el constructor.

11.21 *(Volver a lanzar expresiones)* Escriba un programa que ilustre cómo volver a lanzar una excepción. Defina los métodos `unMetodo` y `unMetodo2`. El método `unMetodo2` debe lanzar al principio una excepción. El método `unMetodo` debe llamar a `unMetodo2`, atrapar la excepción y volver a lanzarla. Llame a `unMetodo` desde el método `main`, y atrape la excepción que se volvió a lanzar. Imprima el rastreo de la pila de esta excepción.

11.22 *(Atrapar excepciones mediante el uso de alcances exteriores)* Escriba un programa que muestre que un método con su propio bloque `try` no tiene que atrapar todos los posibles errores que se generen dentro del `try`. Algunas excepciones pueden pasarse hacia otros alcances, en donde se manejan.

Los capítulos 12 a 19 se encuentran en español en el sitio Web del libro

Los capítulos 20 a 31 y los apéndices M a Q se encuentran en inglés en el sitio Web del libro

A

Tabla de precedencia de operadores

Los operadores se muestran en orden decreciente de precedencia, de arriba hacia abajo (figura A.1).

Operador	Descripción	Asociatividad
++	unario de postincremento	de derecha a izquierda
--	unario de postdecremento	
++	unario de preincremento	de derecha a izquierda
--	unario de predecremento	
+	unario de suma	
-	unario de resta	
!	unario de negación lógica	
~	unario de complemento a nivel de bits	
(*tipo*)	unario de conversión	
*	multiplicación	de izquierda a derecha
/	división	
%	residuo	
+	suma o concatenación de cadenas	de izquierda a derecha
-	resta	
<<	desplazamiento a la izquierda	de izquierda a derecha
>>	desplazamiento a la derecha con signo	
>>>	desplazamiento a la derecha sin signo	
<	menor que	de izquierda a derecha
<=	menor o igual que	
>	mayor que	
>=	mayor o igual que	
instanceof	comparación de tipos	
==	es igual que	de izquierda a derecha
!=	no es igual que	
&	AND a nivel de bits AND lógico booleano	de izquierda a derecha
^	OR excluyente a nivel de bits OR excluyente lógico booleano	de izquierda a derecha

Fig. A.1 | Tabla de precedencia de operadores (parte 1 de 2).

Operador	Descripción	Asociatividad
|	OR incluyente a nivel de bits OR incluyente lógico booleano	de izquierda a derecha
&&	AND condicional	de izquierda a derecha
||	OR condicional	de izquierda a derecha
?:	condicional	de derecha a izquierda
=	asignación	de derecha a izquierda
+=	asignación, suma	
-=	asignación, resta	
*=	asignación, multiplicación	
/=	asignación, división	
%=	asignación, residuo	
&=	asignación, AND a nivel de bits	
^=	asignación, OR excluyente a nivel de bits	
|=	asignación, OR incluyente a nivel de bits	
<<=	asignación, desplazamiento a la izquierda a nivel de bits	
>>=	asignación, desplazamiento a la derecha a nivel de bits con signo	
>>>=	asignación, desplazamiento a la derecha a nivel de bits sin signo	

Fig. A.1 | Tabla de precedencia de operadores (parte 2 de 2).

B

Conjunto de caracteres ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\'	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Fig. B.1 | El conjunto de caracteres ASCII.

Los dígitos a la izquierda de la tabla son los dígitos izquierdos de los equivalentes decimales (0-127) de los códigos de caracteres, y los dígitos en la parte superior de la tabla son los dígitos derechos de los códigos de caracteres. Por ejemplo, el código de carácter para la "F" es 70, mientras que para el "&" es 38.

La mayoría de los usuarios de este libro estarán interesados en el conjunto de caracteres ASCII utilizado para representar los caracteres del idioma español en muchas computadoras. El conjunto de caracteres ASCII es un subconjunto del conjunto de caracteres Unicode utilizado por Java para representar caracteres de la mayoría de los lenguajes existentes en el mundo. Para obtener más información acerca del conjunto de caracteres Unicode, vea el apéndice N que se incluye como bono Web.

C

Palabras clave y palabras reservadas

Palabras clave en Java				
`abstract`	`assert`	`boolean`	`break`	`byte`
`case`	`catch`	`char`	`class`	`continue`
`default`	`do`	`double`	`else`	`enum`
`extends`	`final`	`finally`	`float`	`for`
`if`	`implements`	`import`	`instanceof`	`int`
`interface`	`long`	`native`	`new`	`package`
`private`	`protected`	`public`	`return`	`short`
`static`	`strictfp`	`super`	`switch`	`synchronized`
`this`	`throw`	`throws`	`transient`	`try`
`void`	`volatile`	`while`		
Palabras clave que no se utilizan actualmente				
`const`	`goto`			

Fig. C.1 | Palabras clave de Java.

Java también contiene las palabras reservadas `true` y `false`, las cuales son literales `boolean`, y `null`, que es la literal que representa una referencia a nada. Al igual que las palabras clave, esas palabras reservadas no se pueden utilizar como identificadores.

D

Tipos primitivos

Tipo	Tamaño en bits	Valores	Estándar
`boolean`		`true` o `false`	
[*Nota:* una representación `boolean` es específica para la Máquina virtual de Java en cada plataforma].			
`char`	16	`'\u0000'` a `'\uFFFF'` (0 a 65535)	(ISO, conjunto de caracteres Unicode)
`byte`	8	–128 a +127 (-2^7 a $2^7 - 1$)	
`short`	16	–32,768 a +32,767 (-2^{15} a $2^{15} - 1$)	
`int`	32	–2,147,483,648 a +2,147,483,647 (-2^{31} a $2^{31} - 1$)	
`long`	64	–9,223,372,036,854,775,808 a +9,223,372,036,854,775,807 (-2^{63} a $2^{63} - 1$)	
`float`	32	*Rango negativo:* –3.4028234663852886E+38 a –1.40129846432481707e–45	(IEEE 754, punto flotante)
		Rango positivo: 1.40129846432481707e–45 a 3.4028234663852886E+38	
`double`	64	*Rango negativo:* –1.7976931348623157E+308 a –4.94065645841246544e–324	(IEEE 754, punto flotante)
		Rango positivo: 4.94065645841246544e–324 a 1.7976931348623157E+308	

Fig. D.1 | Tipos primitivos de Java.

Para obtener más información acerca de IEEE 754, visite `grouper.ieee.org/groups/754/`. Para obtener más información sobre Unicode, vea el apéndice N.

E

Uso de la documentación de la API de Java

E.1 Introducción

La biblioteca de clases de Java contiene miles de clases e interfaces predefinidas, que los programadores pueden usar para escribir sus propias aplicaciones. Estas clases se agrupan en paquetes con base en su funcionalidad. Por ejemplo, las clases e interfaces que se utilizan para el procesamiento de archivos se agrupan en el paquete `java.io`, mientras que las clases e interfaces para las aplicaciones en red se agrupan en el paquete `java.net`. La **documentación de la API de Java** enumera los miembros `public` y `protected` de cada clase, además de los miembros `public` de cada interfaz en la biblioteca de clases de Java. La documentación muestra las generalidades de todas las clases e interfaces, sintetiza sus miembros (es decir, los campos, constructores y métodos de las clases, además de los campos y métodos de las interfaces) y proporciona descripciones detalladas de cada miembro. La mayoría de los programadores de Java consultan esta documentación al momento de escribir programas. Por lo general, los programadores buscan en la API lo siguiente:

1. El paquete que contiene una clase o interfaz específica.
2. Las relaciones entre una clase o interfaz específica, y otras clases e interfaces.
3. Constantes de una clase o interfaz; por lo general, se declaran como campos `public static final`.
4. Constructores para determinar cómo se puede inicializar un objeto de la clase.
5. Los métodos de una clase, para determinar si son `static` o no, el número y tipos de argumentos que se deben pasar, los tipos de valores de retorno y cualquier excepción que podría lanzarse desde el método.

Además, los programadores dependen a menudo de la documentación para descubrir clases e interfaces que no han utilizado antes. Por esta razón, demostraremos la documentación con clases que usted ya conoce y con clases que tal vez no haya estudiado aún. Le mostraremos cómo usar la documentación para localizar la información que necesita para usar una clase o interfaz con efectividad.

E.2 Navegación por la API de Java

Es posible descargar la documentación de la API de Java en el disco duro, o verla en línea. Para descargarla, vaya a `www.oracle.com/technetwork/java/javase/downloads/index.html`, desplácese hasta la sección **Additional Resources** y haga clic en el botón **Download** a la derecha de **Java SE 6 Documentation**. A continuación, aparecerá un mensaje pidiéndole que acepte un acuerdo de licencia. Si está de acuerdo, haga clic en **Aceptar** y después en el vínculo hacia el archivo ZIP para empezar a descargarlo. Después de descargar el archivo, puede usar un programa como WinZip (`www.winzip.`

com) para extraer los archivos. Si utiliza Windows, extraiga el contenido en su directorio de instalación del JDK. Para ver la documentación de la API en su disco duro local en Microsoft Windows, abra la página `C:\Archivos de programa\Java\SuVersionDeJDK\docs\api\index.html` en su navegador. Si desea ver la documentación de la API en línea, vaya a `download.oracle.com/javase/6/docs/api/` (figura E.1).

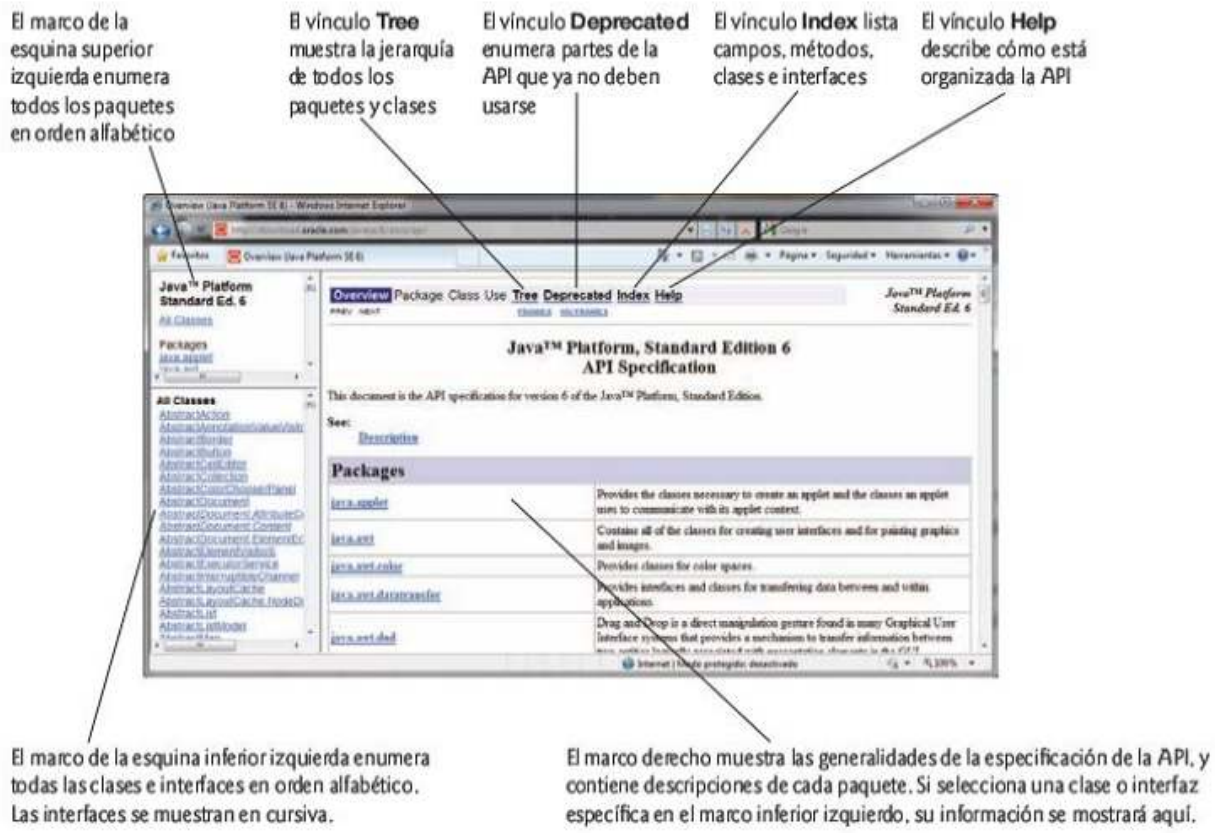


Fig. E.1 | Generalidades de la API de Java (cortesía de Oracle Corporation).

Marcos en la página `index.html` de la documentación de la API

La documentación de la API se divide en tres marcos (vea la figura E.1). El marco superior izquierdo enumera todos los paquetes de la API de Java en orden alfabético. En un principio, el marco inferior izquierdo enumera las clases e interfaces de la API de Java en orden alfabético. Los nombres de las interfaces se muestran en cursiva. Si hace clic en un paquete específico en el marco superior izquierdo, el marco inferior izquierdo lista las clases e interfaces del paquete seleccionado. En un principio, el marco derecho provee una descripción breve de cada paquete de la especificación de la API de Java; lea esta descripción general para familiarizarse con las capacidades generales de las API de Java. Si selecciona una clase o interfaz en el marco inferior izquierdo, el marco derecho muestra información sobre esa clase o interfaz.

Vínculos importantes en la página `index.html`

En la parte superior del marco derecho (figura E.1), hay cuatro vínculos: **`Tree`**, **`Deprecated`**, **`Index`** y **`Help`**. El vínculo **Tree** muestra la jerarquía de todos los paquetes, clases e interfaces en una estructura tipo árbol. El vínculo **Deprecated** muestra las interfaces, clases, excepciones, campos, constructores y métodos que ya no deben usarse. El vínculo **Index** muestra clases, interfaces, campos, constructores y

métodos en orden alfabético. El vínculo **Help** describe la forma en que está organizada la documentación de la API. Tal vez usted deba empezar por leer la página **Help**.

Vista de la página **Index**

Si no conoce el nombre de la clase que busca, pero sabe el nombre de un método o campo, puede usar el índice de la documentación para localizar la clase. El vínculo **Index** se encuentra cerca de la esquina superior derecha del marco derecho. La página del índice (figura E.2) muestra los campos, constructores, métodos, interfaces y clases en orden alfabético. Por ejemplo, si busca el método `hasNextInt` de `Scanner` pero no conoce el nombre de la clase, puede hacer clic en el vínculo **H** para ir al listado alfabético de todos los elementos en la API de Java que empiecen con "`h`". Desplácese hasta el método `hasNextInt` (figura E.3). Una vez ahí, cada método llamado `hasNextInt` se enumera con el nombre del paquete y la clase a la que pertenece ese método. De aquí usted puede hacer clic en el nombre de la clase para ver sus detalles completos, o puede hacer clic en el nombre del método para ver sus detalles.

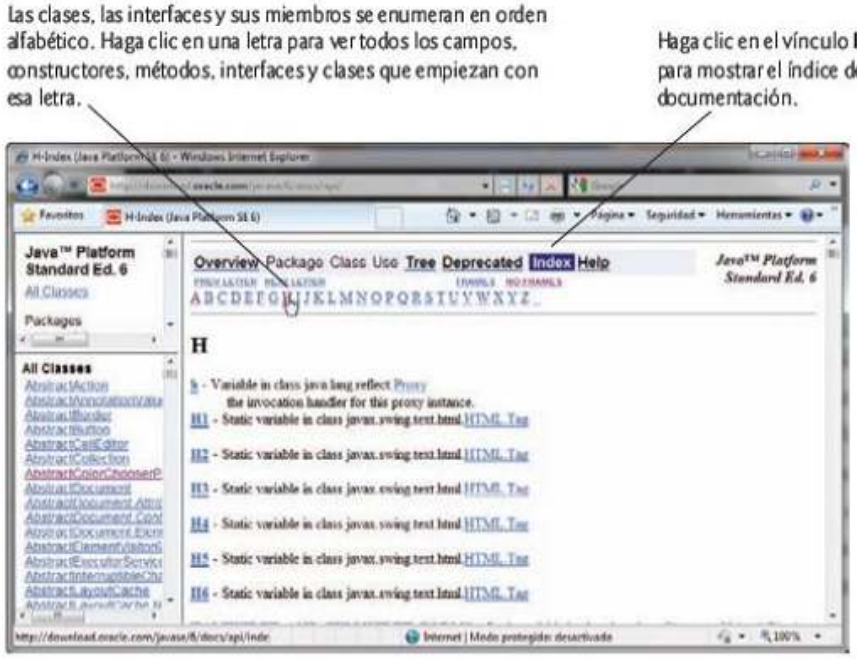


Fig. E.2 | Vista de la página **Index** (cortesía de Oracle Corporation).

Vista de una página específica

Al hacer clic en el nombre del paquete en el marco superior izquierdo, se muestran todas las clases e interfaces de ese paquete en el marco inferior izquierdo y se dividen en cinco subsecciones: **Interfaces**, **Classes**, **Enums**, **Exceptions** y **Errors**; cada una de ellas se lista por orden alfabético. Por ejemplo, el contenido del paquete `javax.swing` se muestra en el marco inferior izquierdo (figura E.4) al hacer clic en `javax.swing` en el marco superior izquierdo. Puede hacer clic en el nombre del paquete en el marco inferior izquierdo para tener una vista general del paquete. Si cree que un paquete contiene varias clases que podrían ser útiles en su aplicación, la vista general del paquete puede ser bastante útil.

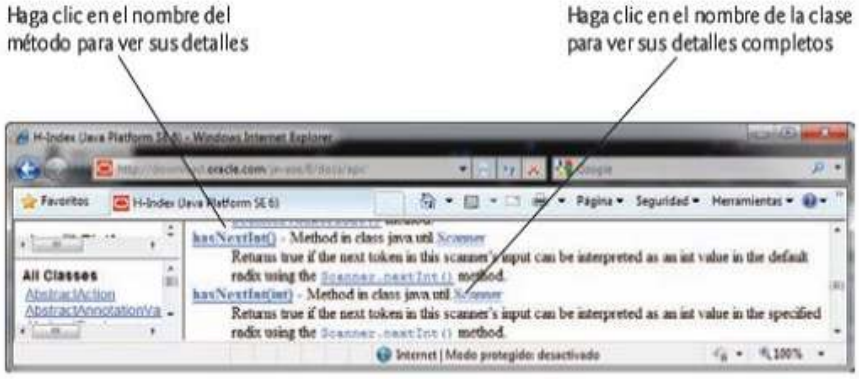


Fig. E.3 | Desplácese hasta el método `hasNextInt` (cortesía de Oracle Corporation).

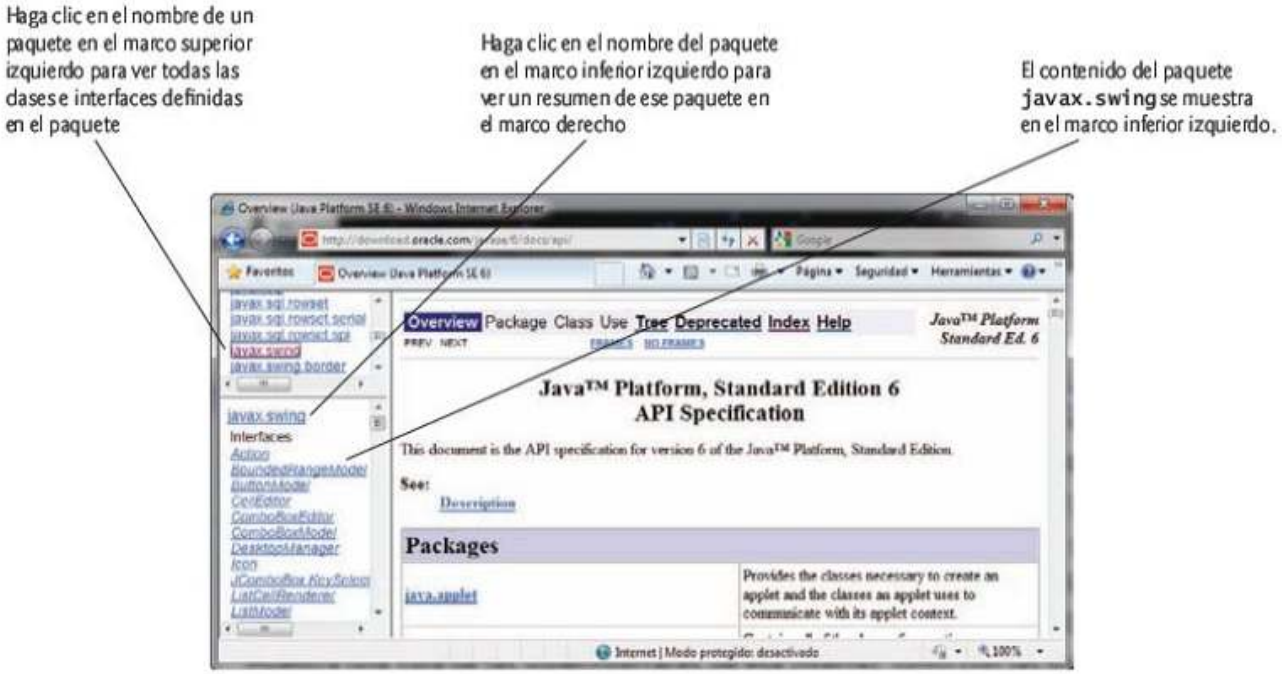


Fig. E.4 | Haga clic en el nombre de un paquete en el marco superior izquierdo para ver todas las clases e interfaces declaradas en este paquete (cortesía de Oracle Corporation).

Vista de los detalles de una clase

Al hacer clic en el nombre de una clase o de una interfaz en el marco inferior izquierdo, el marco derecho muestra los detalles de esa clase o interfaz. Primero verá el nombre del paquete de la clase, seguido de una jerarquía que muestra la relación que tiene esa clase con respecto a otras clases. También verá una lista de las interfaces implementadas por la clase y las subclases conocidas de esa clase. La figura E.5 muestra el principio de la página de documentación de la clase `JButton`, del paquete `javax.swing`. Primero, la página muestra el nombre del paquete en donde aparece la clase. Esto va seguido de la jerarquía de la clase que conduce a la clase `JButton`, las interfaces que implementa la clase `JButton` y las subclases de la clase `JButton`. La parte inferior del marco derecho muestra el principio de la descripción de la clase `JButton`. Al analizar la documentación de una interfaz, el marco derecho no muestra una jerarquía para esa interfaz. En cambio, el marco derecho muestra una lista de las superinterfaces de esa interfaz, junto con las subinterfaces conocidas y las clases conocidas que la implementan.

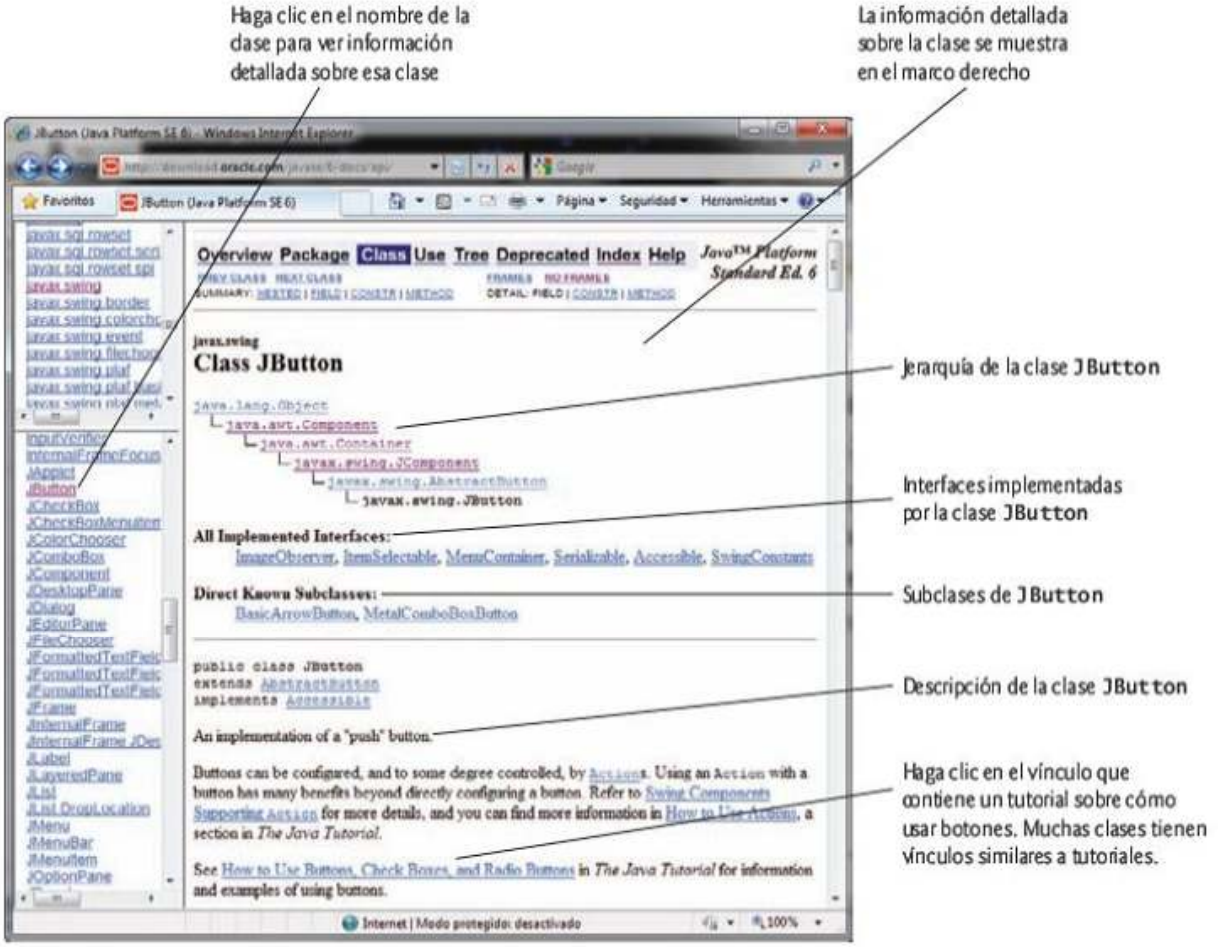


Fig. E.5 | Haga clic en el nombre de una clase para ver información detallada sobre esa clase (cortesía de Oracle Corporation).

Secciones de resumen en la página de documentación de una clase

A continuación se muestra una lista con otras partes de cada página de API. Cada parte se presenta sólo si la clase contiene o hereda los elementos especificados. Los miembros de clases que se muestran en las secciones de resumen son `public`, a menos que se marquen de manera explícita como `protected`. Los miembros `private` de una clase no se muestran en la documentación, ya que no se pueden usar de manera directa en los programas.

1. La sección **Nested Class Summary** muestra un resumen de las clases `public` y `protected` anidadas de la clase; es decir, las clases que se definen dentro de la clase. A menos que se especifique de manera explícita, estas clases son `public` y no son `static`.

2. La sección **Field Summary** muestra un resumen de los campos `public` y `protected` de la clase. A menos que se especifique de manera explícita, estos campos son `public` y no son `static`. La figura E.6 muestra la sección **Field Summary** de la clase `Color`.

3. La sección **Constructor Summary** muestra un resumen de los constructores de la clase. Los constructores no son heredados, por lo que esta sección aparece en la documentación para una clase, sólo si la clase declara uno o más constructores. La figura E.7 muestra la sección **Constructor Summary** de la clase `JButton`.

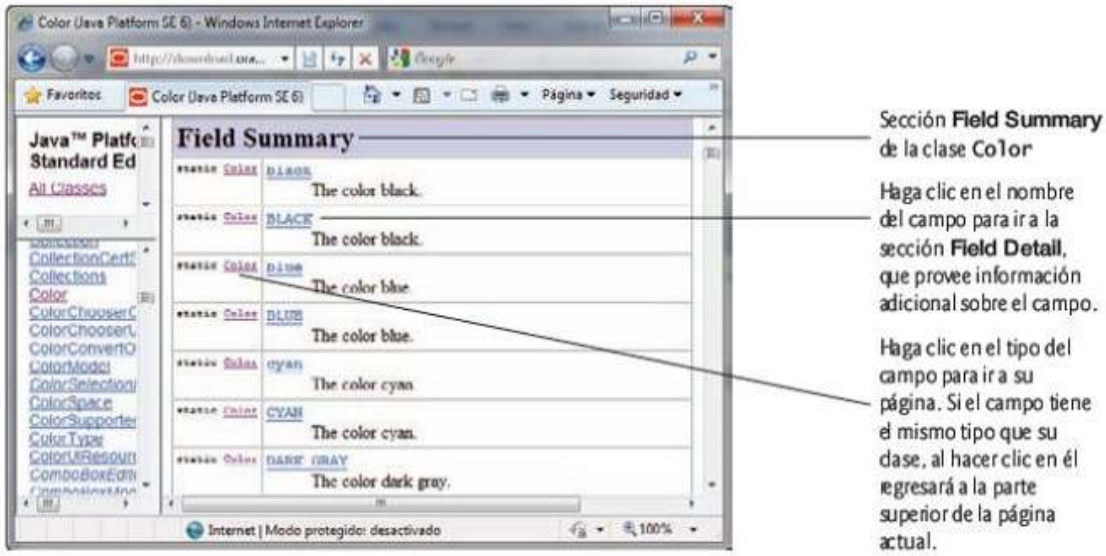


Fig. E.6 | La sección **Field Summary** de la clase `Color` (cortesía de Oracle Corporation).

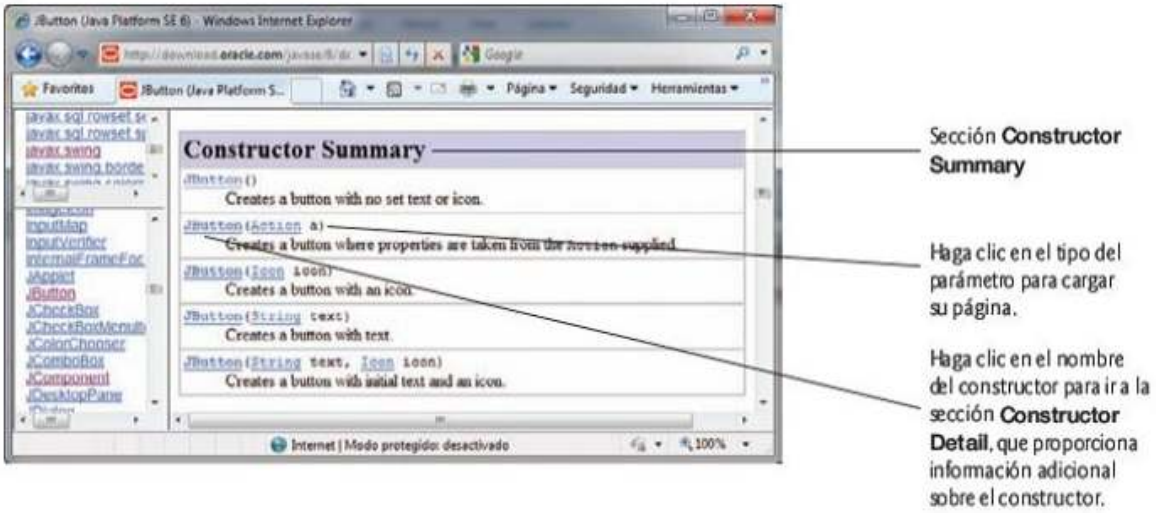


Fig. E.7 | La sección **Constructor Summary** de la clase `JButton` (cortesía de Oracle Corporation).

4. La sección **Method Summary** muestra un resumen de los métodos `public` y `protected` de la clase. A menos que se especifique de manera explícita, estos métodos son `public` y no `static`. La figura E.8 muestra la sección **Method Summary** de la clase `BufferedInputStream`.

Por lo general, las secciones de resumen sólo proveen una descripción de un solo enunciado del miembro de una clase. Los detalles adicionales se presentan en las secciones de detalles que veremos a continuación.

Secciones de detalles en la página de documentación de una clase

Después de las secciones de resumen se encuentran las secciones de detalles, que por lo general proveen información más específica de los miembros de las clases. No hay una sección de detalles para las clases

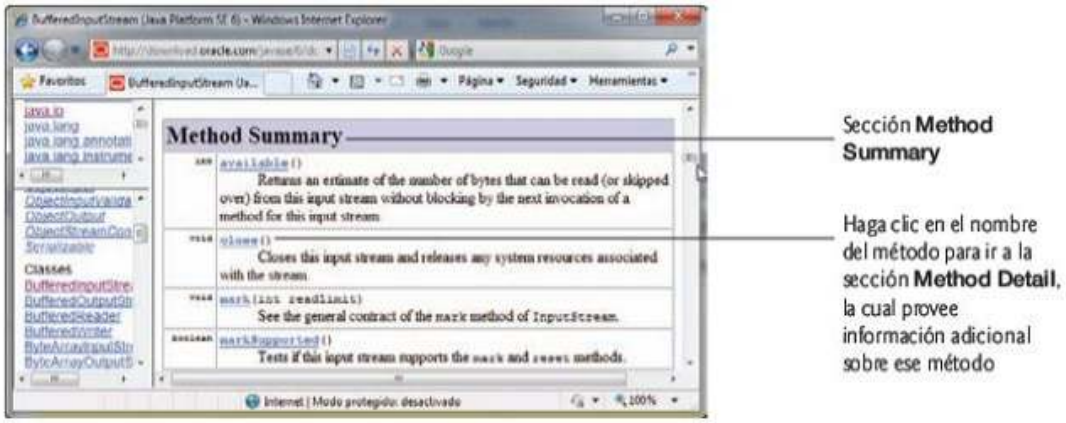


Fig. E.8 | La sección **Method Summary** de la clase `BufferedInputStream` (cortesía de Oracle Corporation).

anidadas. Al hacer clic en uno de los vínculos de la sección **Nested Class Summary** para una clase anidada específica, se muestra una página de documentación que describe a esa clase anidada. A continuación se describen las secciones de detalles.

1. La sección **Field Detail** provee la declaración de cada campo. También analiza cada campo, incluyendo sus modificadores y su significado. La figura E.9 muestra la sección **Field Detail** de la clase `Color`.

2. La sección **Constructor Detail** provee la primera línea de la declaración de cada constructor y un análisis de los constructores. Este análisis incluye los modificadores de cada constructor, una descripción de cada constructor, los parámetros de cada constructor y cualquier excepción lanzada por cada uno de ellos. La figura E.10 muestra la sección **Constructor Detail** de la clase `JButton`.

3. La sección **Method Detail** provee la primera línea de cada método. El análisis de cada método incluye sus modificadores, una descripción más completa del método, sus parámetros, el tipo de valor de retorno y cualquier excepción lanzada por el método. La figura E.11 muestra la sección **Method Detail** de `BufferedInputStream`.

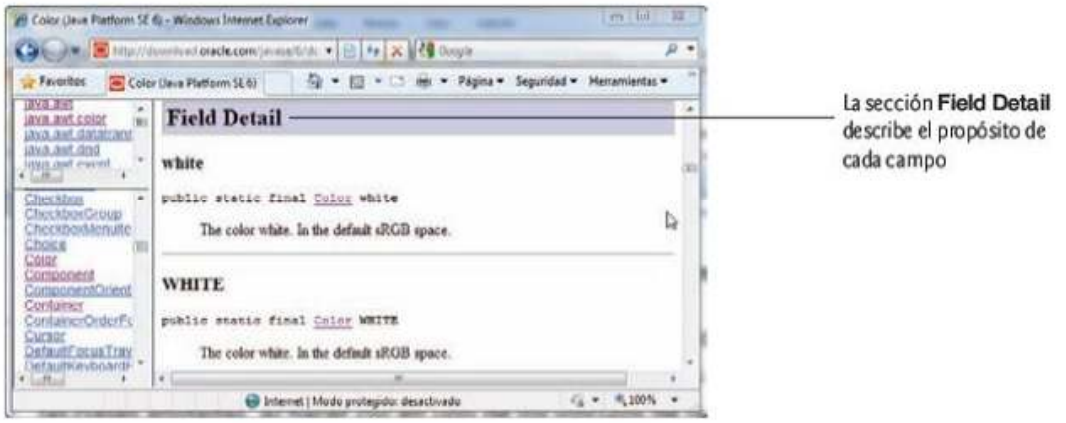


Fig. E.9 | La sección **Field Detail** de la clase `Color` (cortesía de Oracle Corporation).

Fig. E.10 | La sección **Constructor Detail** de la clase `JButton` (cortesía de Oracle Corporation).

Los detalles de los métodos nos muestran otros métodos que podrían ser de interés (etiquetados como **See Also**). Si el método sobrescribe a un método de la superclase, se proporcionan tanto el nombre del método de la superclase como el de la superclase, para que podamos hacer clic en el vínculo a ese método o esa superclase y obtener más información.

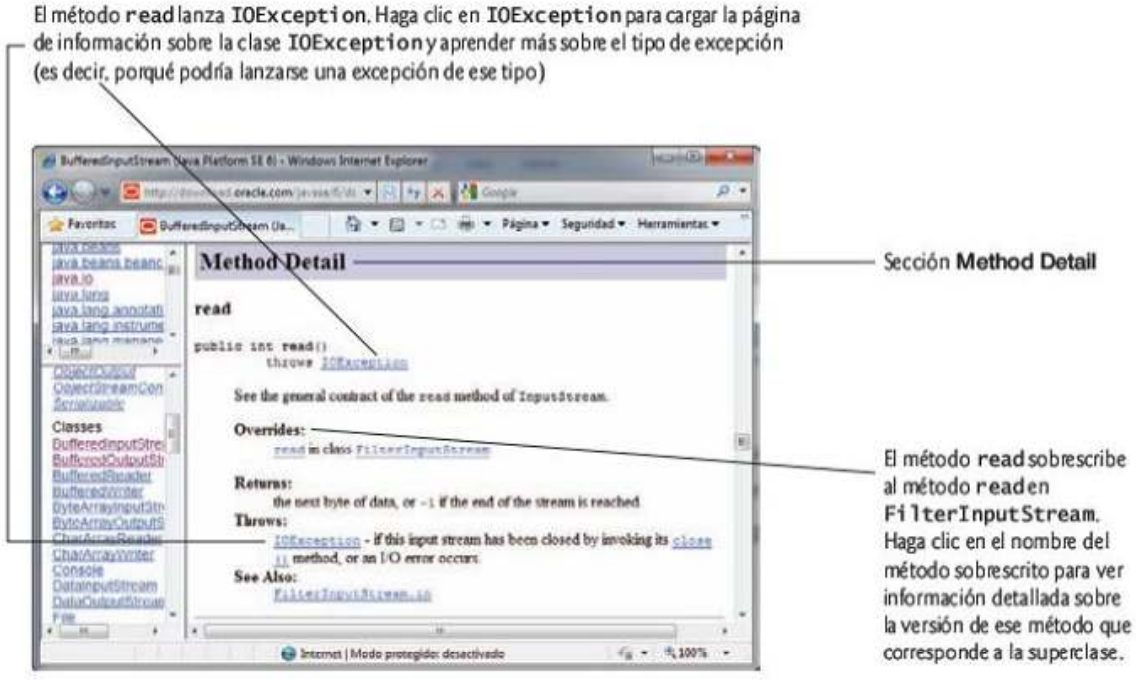
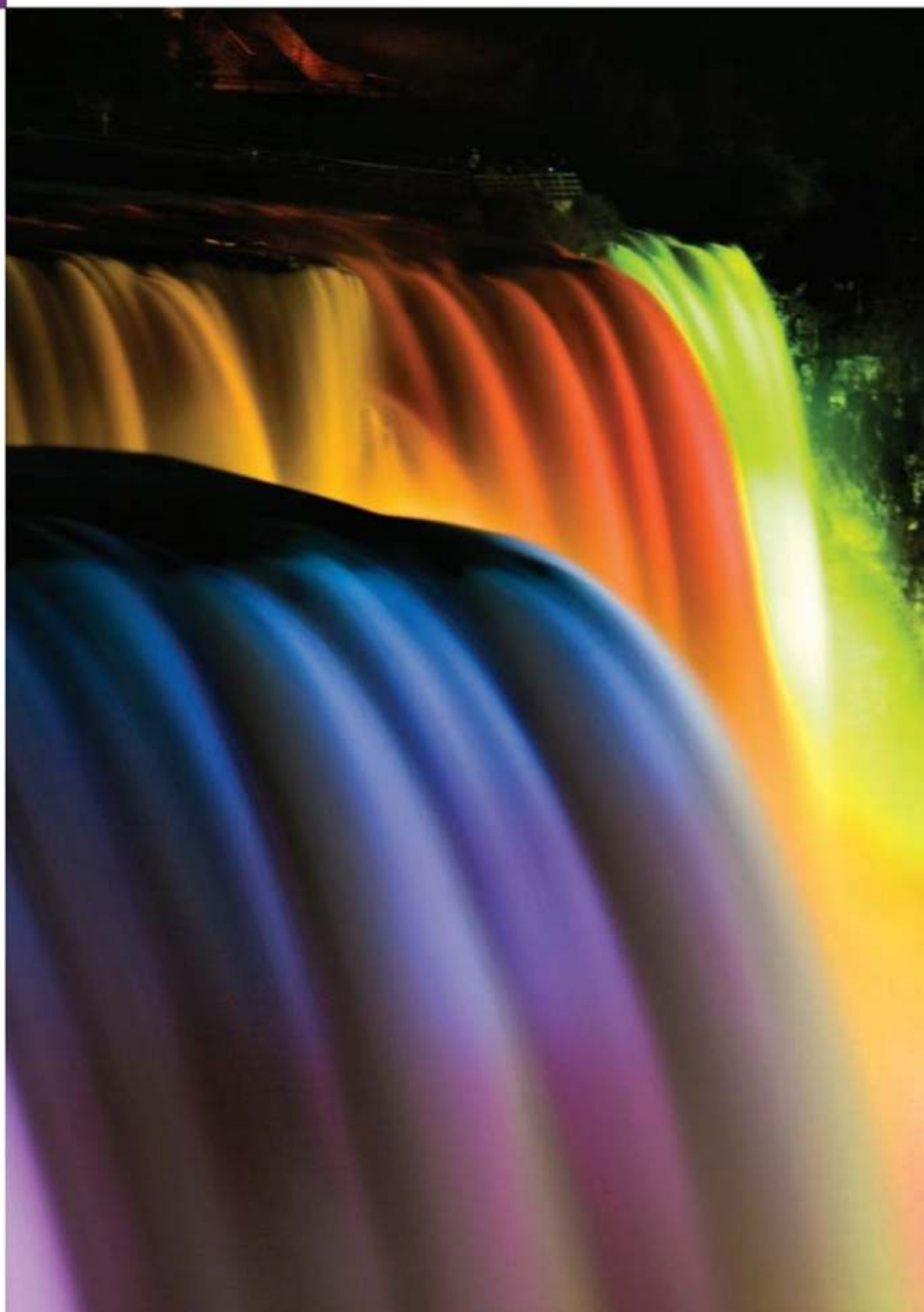


Fig. E.11 | La sección **Method Detail** de la clase `BufferedInputStream` (cortesía de Oracle Corporation).

A medida que analice la documentación, podrá observar que a menudo existen vínculos a otros campos, métodos, clases anidadas y clases de nivel superior. Estos vínculos le permiten saltar de la clase que está viendo a otra porción relevante de la documentación.

F

Uso del depurador



Por lo tanto, debo atrapar a la mosca.

—William Shakespeare

Estamos creados para cometer equivocaciones, codificados para el error.

—Lewis Thomas

Lo que anticipamos raras veces ocurre; lo que menos esperamos es lo que generalmente pasa.

—Benjamin Disraeli

Objetivos

En este apéndice aprenderá a:

- Establecer puntos de interrupción para depurar aplicaciones.
- Usar el comando `run` para ejecutar una aplicación a través del depurador
- Usar el comando `stop` para establecer un punto de interrupción.
- Usar el comando `cont` para continuar la ejecución.
- Usar el comando `print` para evaluar expresiones.
- Usar el comando `set` para cambiar los valores de las variables durante la ejecución del programa.
- Usar los comandos `step`, `step up` y `next` para controlar la ejecución.
- Usar el comando `watch` para ver cómo se modifica un campo durante la ejecución del programa.
- Usar el comando `clear` para listar los puntos de interrupción o eliminar uno de ellos.

Plan general

F.1 Introducción

En el capítulo 2 vimos que hay dos tipos de errores (errores de sintaxis y errores lógicos), y usted aprendió a eliminar los errores de sintaxis de su código. Los errores lógicos no evitan que la aplicación se compile con éxito, pero pueden hacer que produzca resultados erróneos al ejecutarse. El JDK cuenta con software conocido como **depurador**, el cual nos permite supervisar la ejecución de las aplicaciones para localizar y eliminar errores lógicos. El depurador será una de sus herramientas más importantes para el desarrollo de aplicaciones. En la actualidad, muchos IDE cuentan con sus propios depuradores, similares al que se viene en el JDK, o proveen una interfaz gráfica de usuario para el depurador del JDK.

En este apéndice demostramos las características clave del depurador del JDK, mediante el uso de aplicaciones de línea de comandos que no reciben entrada por parte del usuario. Las mismas características del depurador que veremos aquí pueden usar ese para depurar aplicaciones que reciben entrada del usuario, pero para depurar ese tipo de aplicaciones se requiere una configuración un poco más compleja. Para enfocarnos en las características del depurador, optamos por demostrar su funcionamiento con aplicaciones simples de línea de comandos que no requieren entrada del usuario. Para obtener más información sobre el depurador de Java, visite `download.oracle.com/javase/6/docs/technotes/tools/windows/jdb.html`.

F.2 Los puntos de interrupción y los comandos `run`, `stop`, `cont` y `print`

Para empezar con nuestro estudio del depurador, vamos a investigar los **puntos de interrupción**, que son marcadores que pueden establecerse en cualquier línea de código ejecutable. Cuando la ejecución de la aplicación llega a un punto de interrupción, la ejecución se detiene, lo cual nos permite examinar los valores de las variables para ayudarnos a determinar si existen errores lógicos. Por ejemplo, podemos examinar el valor de una variable que almacena el resultado de un cálculo para determinar si el cálculo se realizó en forma correcta. Si establece un punto de interrupción en una línea de código que no es ejecutable (como un comentario), el depurador mostrará un mensaje de error.

Para ilustrar las características del depurador, vamos a usar la aplicación `PruebaCuenta` (figura F.1), la cual crea y manipula un objeto de la clase `Cuenta` (figura 3.13). La ejecución de `PruebaCuenta` empieza en `main` (líneas 7 a 24). En la línea 9 se crea un objeto `Cuenta` con un saldo inicial de $50.00. Recuerde que el constructor de `Cuenta` acepta un argumento, el cual especifica el saldo inicial de la `Cuenta`. En las líneas 12 y 13 se imprime el saldo inicial de la cuenta mediante el uso del método `obtenerSaldo` de `Cuenta`. En la línea 15 se declara e inicializa una variable local llamada `montoDeposito`. Después, en las líneas 17 a 19 se imprime `montoDeposito` y se agrega al `saldo` de `Cuenta` mediante el uso de su método `abonar`. Por último, en las líneas 22 y 23 se muestra el nuevo `saldo`. [*Nota:* el direc-

torio de ejemplos del apéndice F contiene una copia del archivo Cuenta.java, el cual es idéntico al de la figura 3.13].

```
1  // Fig. F.1: PruebaCuenta.java
2  // Crea y manipula un objeto Cuenta.
3
4  public class PruebaCuenta
5  {
6     // el método main empieza la ejecución
7     public static void main (String [] args )
8     {
9        Cuenta cuenta = new Cuenta( 50.00 );  // crea objeto Cuenta
10
11       // muestra el saldo inicial del objeto Cuenta
12       System.out.printf( "saldo inicial de cuenta: $%.2f\n",
13          cuenta.obtenerSaldo() );
14
15       double montoDeposito = 25.0;  // monto del depósito
16
17       System.out.printf( "\nagregando %.2f al saldo de la cuenta\n\n",
18          montoDeposito );
19       cuenta.abonar( montoDeposito );  // se suma al saldo de la cuenta
20
21       // muestra el nuevo saldo
22       System.out.printf( "nuevo saldo de la cuenta: $%.2f\n",
23          cuenta.obtenerSaldo() );
24    } // fin de main
25
26 } // fin de la clase PruebaCuenta
```

```
saldo inicial de cuenta: $50.00

agregando 25.00 al saldo de la cuenta

nuevo saldo de la cuenta: $75.00
```

Fig. F.1 | La clase PruebaCuenta crea y manipula un objeto Cuenta.

En los siguientes pasos, utilizaremos puntos de interrupción y varios comandos del depurador para examinar el valor de la variable montoDeposito declarada en PruebaCuenta (figura F.1).

Abrir la ventana* Símbolo del sistema *y cambiar directorios. Para abrir la ventana **Símbolo del sistema**, seleccione **Inicio | Programas | Accesorios | Símbolo del sistema**. Ahora hay que cambiar al directorio que contiene los ejemplos del apéndice F. Escriba `cd C:\ejemplos\depurador`. [*Nota:* si sus ejemplos están en un directorio distinto, use ese directorio aquí].

2. ***Compilar la aplicación para depurarla.*** El depurador de Java trabaja sólo con archivos `.class` que se hayan compilado con la opción `-g` del compilador, la cual genera información adicional que el depurador necesita para ayudar al programador a depurar sus aplicaciones. Para compilar la aplicación con la opción de línea de comandos `-g`, escriba `javac -g PruebaCuenta.java Cuenta.java`. En el capítulo 3 vimos que este comando compila tanto a PruebaCuenta.java como a Cuenta.java. El comando `javac -g *.java` compila todos los archivos `.java` del directorio de trabajo para la depuración.

3. ***Iniciar el depurador.*** Escriba `jdb` en la ventana **Símbolo del sistema** (figura F.2). Este comando iniciará el depurador y le permitirá usar sus características. [*Nota:* modificamos los colores de nuestra ventana **Símbolo del sistema** para mejorar la legibilidad].

Fig. F.2 | Iniciar el depurador de Java.

4. ***Ejecutar una aplicación en el depurador.*** Ejecute la aplicación `PruebaCuenta` a través del depurador, escribiendo `run PruebaCuenta` (figura F.3). Si no establece puntos de interrupción antes de ejecutar su aplicación en el depurador, ésta se ejecutará hasta justo igual que si se utilizara el comando `java`.

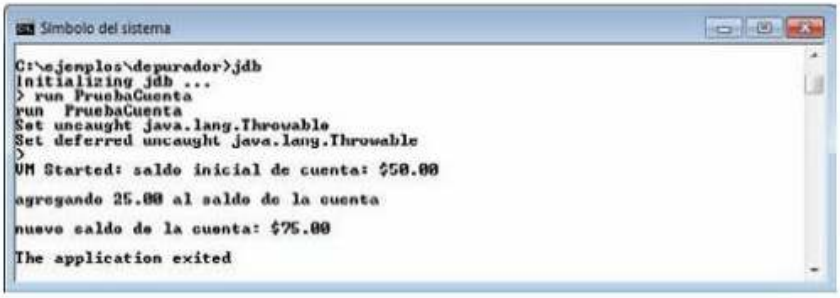


Fig. F.3 | Ejecución de la aplicación `PruebaCuenta` a través del depurador.

5. ***Reiniciar el depurador.*** Para hacer un uso apropiado del depurador, debe establecer por lo menos un punto de interrupción antes de ejecutar la aplicación. Para reiniciar el depurador, escriba `jdb`.

6. ***Insertar puntos de interrupción en Java.*** Puede establecer un punto de interrupción en una línea específica de código en su aplicación. Los números de línea que usamos en estos pasos corresponden al código fuente de la figura F.1. Establezca un punto de interrupción en la línea 12 del código fuente, escribiendo `stop` en `PruebaCuenta:12` (figura F.4). El **comando `stop`** inserta un punto de interrupción en el número de línea especificado después del comando. Puede establecer todos los puntos de interrupción que sean necesarios. Establezca otro punto de interrupción en la línea 19, escribiendo `stop at PruebaCuenta:19` (figura F.4). Cuando se ejecuta la aplicación, suspende la ejecución en cualquier línea que contiene un punto de interrupción. Se dice que la aplicación está en **modo de interrupción** cuando el depurador detiene la ejecución de la aplicación. Pueden establecerse puntos de interrupción, incluso hasta después de que haya empezado el proceso de depuración. El comando `stop in` del depurador, seguido del nombre de una clase, un punto y el nombre de un método (por ejemplo, `stop in Cuenta.abonar`), instruye al depurador para que establezca un punto de interrupción en la primera instrucción ejecutable en el método especificado. El depurador suspende la ejecución cuando el control del programa entra al método.

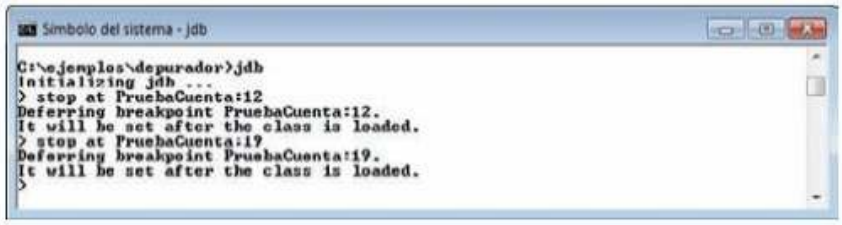


Fig. F.4 | Cómo establecer puntos de interrupción en las líneas 12 y 19.

7. ***Ejecutar la aplicación e iniciar el proceso de depuración.*** Escriba `run PruebaCuenta` para ejecutar la aplicación y empezar el proceso de depuración (figura F.5). El depurador imprime texto para indicar que se establecieron puntos de interrupción en las líneas 12 y 19. Llama a cada punto de interrupción un "punto de interrupción diferido", debido a que cada uno se estableció antes de que se empezara a ejecutar la aplicación en el depurador. La aplicación se detiene cuando la ejecución llega al punto de interrupción en la línea 12. En este punto, el depurador notifica que ha llegado a un punto de interrupción y muestra el código fuente en esa línea (12). Esa línea de código será la siguiente instrucción en ejecutarse.

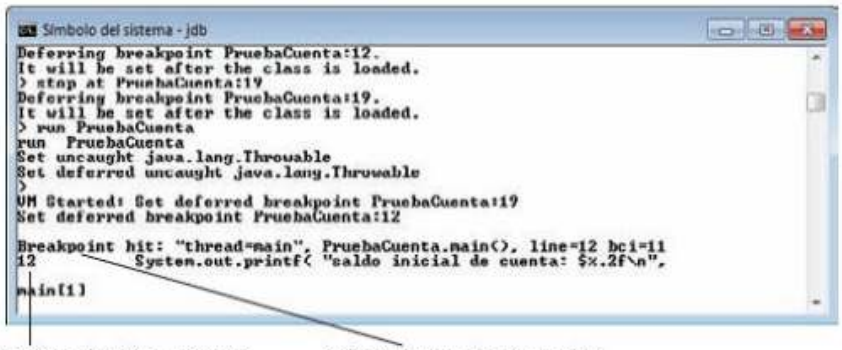


Fig. F.5 | Reinicio de la aplicación `PruebaCuenta`.

8. ***Uso del comando cont para reanudar la ejecución.*** Escriba `cont`. El **comando cont** hace que la aplicación siga ejecutándose hasta llegar al siguiente punto de interrupción (línea 19), y aquí es donde el depurador le notificará a usted (figura F.6). La salida normal de `PruebaCuenta` aparece entre los mensajes del depurador.

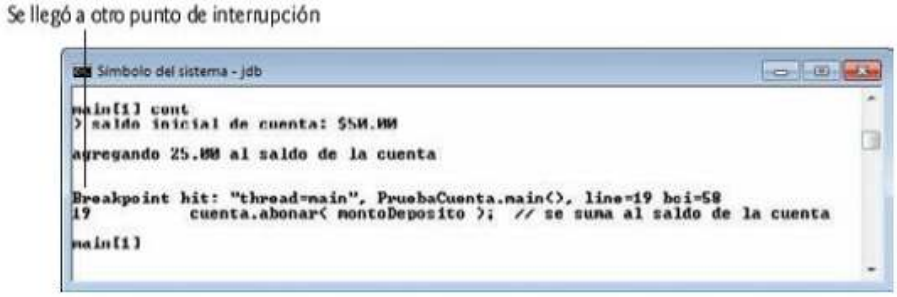


Fig. F.6 | La ejecución llega al segundo punto de interrupción.

9. ***Examinar el valor de una variable.*** Escriba `print montoDeposito` para mostrar el valor actual almacenado en la variable `montoDeposito` (figura F.7). El **comando print** nos permite espiar dentro de la computadora el valor de una de las variables. Este comando le ayudará a encontrar y eliminar los errores lógicos en su código. El valor mostrado es `25.0`: el valor asignado a `montoDeposito` en la línea 15 de la figura F.1.

Fig. F.7 | Examinar el valor de la variable `montoDeposito`.

10. ***Continuar la ejecución de la aplicación.*** Escriba `cont` para continuar la ejecución de la aplicación. Puesto que no hay más puntos de interrupción, la aplicación ya no se encuentra en modo de interrupción. La aplicación continúa su ejecución hasta terminar en forma normal (figura F.8). El depurador se detendrá cuando la aplicación termine.

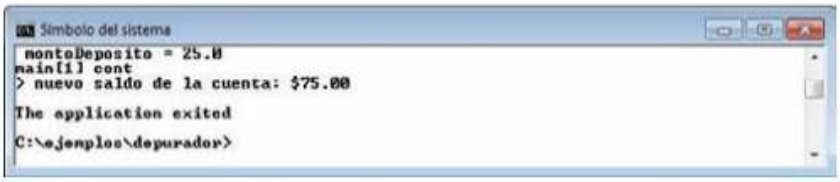


Fig. F.8 | Continuar la ejecución de la aplicación y salir del depurador.

F.3 Los comandos print y set

En la sección anterior, aprendió a usar el comando `print` del depurador para examinar el valor de una variable durante la ejecución de un programa. En esta sección, aprenderá a usar el comando `print` para examinar el valor de expresiones más complejas. También aprenderá sobre el **comando set**, que permite al programador asignar nuevos valores a las variables.

Para esta sección, vamos a suponer que usted siguió los *Pasos 1* y *2* en la sección F.2 para abrir la ventana **Símbolo del sistema**, cambiar al directorio que contiene los ejemplos del apéndice F (por ejemplo, `C:\ejemplos\depurador`) y compilar la aplicación `PruebaCuenta` (junto con la clase `Cuenta`) para su depuración.

1. ***Iniciar la depuración.*** En la ventana **Símbolo del sistema**, escriba `jdb` para iniciar el depurador de Java.
2. ***Insertar un punto de interrupción.*** Establezca un punto de interrupción en la línea 19 del código fuente, escribiendo `stop at CuentaPrueba:19`.
3. ***Ejecutar la aplicación y llegar a un punto de interrupción.*** Escriba `run PruebaCuenta` para empezar el proceso de depuración (figura F.9). Esto hará que se ejecute el método `main` de `PruebaCuenta` hasta llegar al punto de interrupción en la línea 19. Aquí se suspenderá la ejecución de la aplicación y ésta cambiará al modo de interrupción. Hasta este punto, las instrucciones en las líneas 9 a 13 crearon un objeto `Cuenta` e imprimieron el saldo inicial del

objeto `Cuenta`, el cual se obtiene llamando a su método `obtenerSaldo`. La instrucción en la línea 15 (figura F.1) declaró e inicializó la variable local `montoDeposito` con `25.0`. La instrucción en la línea 19 es la siguiente línea que se va a ejecutar.

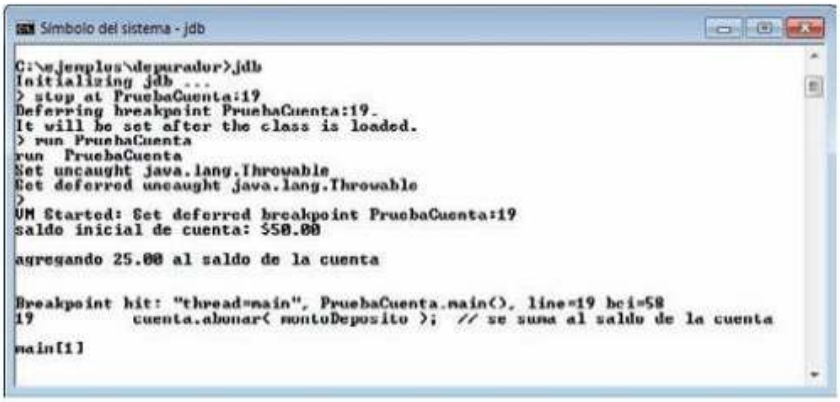


Fig. F.9 | La ejecución de la aplicación se suspende cuando el depurador llega al punto de interrupción en la línea 19.

4. ***Evaluar expresiones aritméticas y booleanas.*** En la sección F.2 vimos que, una vez que la aplicación entra al modo de interrupción, es posible explorar los valores de las variables de la aplicación mediante el comando `print` del depurador. También es posible usar el comando `print` para evaluar expresiones aritméticas y `boolean`. En la ventana **Símbolo del sistema**, escriba `print montoDeposito - 2.0`. El comando `print` devuelve el valor `23.0` (figura F.10). Sin embargo, este comando en realidad no cambia el valor de `montoDeposito`. En la ventana **Símbolo del sistema**, escriba `print montoDeposito == 23.0`. Las expresiones que contienen el símbolo `==` se tratan como expresiones `boolean`. El valor devuelto es `false` (figura F.10), ya que `montoDeposito` no contiene en ese momento el valor `23.0`; `montoDeposito` sigue siendo `25.0`.

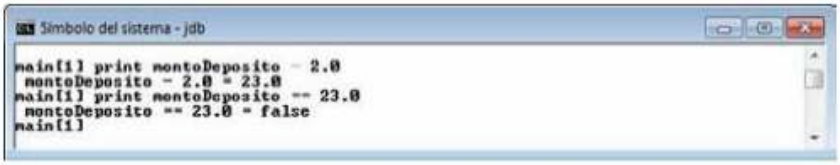
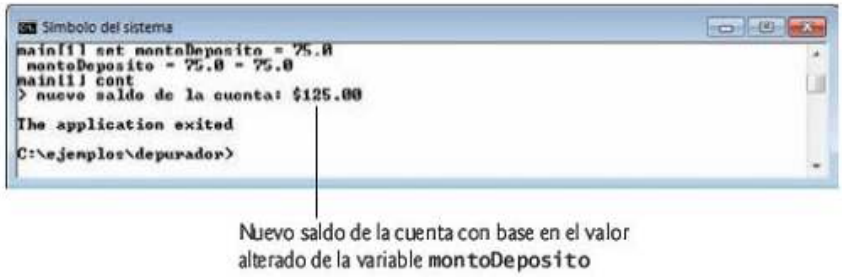


Fig. F.10 | Examinar los valores de una expresión aritmética y `boolean`.

5. ***Modificar valores.*** El depurador le permite modificar los valores de las variables durante la ejecución de la aplicación. Esto puede ser valioso para experimentar con distintos valores y para localizar los errores lógicos en las aplicaciones. Puede usar el comando `set` del depurador para modificar el valor de una variable. Escriba `set montoDeposito = 75.0`. El depurador modifica el valor de `montoDeposito` y muestra su nuevo valor (figura F.11).

Fig. F.11 | Modificando los valores.

6. ***Ver el resultado de la aplicación.*** Escriba `cont` para continuar con la ejecución de la aplicación. A continuación se ejecuta la línea 19 de `PruebaCuenta` (figura F.1) y pasa `montoDeposito` al método `abonar` de `Cuenta`. Después el método `main` muestra el nuevo saldo. El resultado es `$125.00` (figura F.12). Esto muestra que el paso anterior modificó el valor de `montoDeposito`, de su valor inicial (`25.0`) a `75.0`.



Nuevo saldo de la cuenta con base en el valor alterado de la variable `montoDeposito`

Fig. F.12 | La salida que aparece después del proceso de depuración.

F.4 Cómo controlar la ejecución mediante los comandos `step`, `step up` y `next`

Algunas veces es necesario ejecutar una aplicación línea por línea, para encontrar y corregir los errores. Puede ser útil avanzar a través de una porción de la aplicación de esta forma, para verificar que el código de un método se ejecute correctamente. En esta sección aprenderá a usar el depurador para esta tarea. Los comandos en esta sección nos permiten ejecutar un método línea por línea, ejecutar todas las instrucciones de un método a la vez o ejecutar sólo el resto de las instrucciones de un método (si ya hemos ejecutado algunas instrucciones dentro del método).

Una vez más, vamos a suponer que está trabajando en el directorio que contiene los ejemplos del apéndice F, y que compiló los archivos para depuración, con la opción `-g` del compilador.

1. ***Iniciar el depurador.*** Para iniciar el depurador, escriba `jdb`.
2. ***Establecer un punto de interrupción.*** Escriba `stop at PruebaCuenta:19` para establecer un punto de interrupción en la línea 19.
3. ***Ejecutar la aplicación.*** Para ejecutar la aplicación, escriba `run PruebaCuenta`. Después de que la aplicación muestre sus dos mensajes de salida, el depurador indicará que se llegó al punto de interrupción y mostrará el código en la línea 19 (figura F.13). A continuación, el depurador y la aplicación se detendrán y esperarán a que se introduzca el siguiente comando.
4. ***Usar el comando `step`.*** El **comando `step`** ejecuta la siguiente instrucción en la aplicación. Si la siguiente instrucción a ejecutar es la llamada a un método, el control se transfiere al método

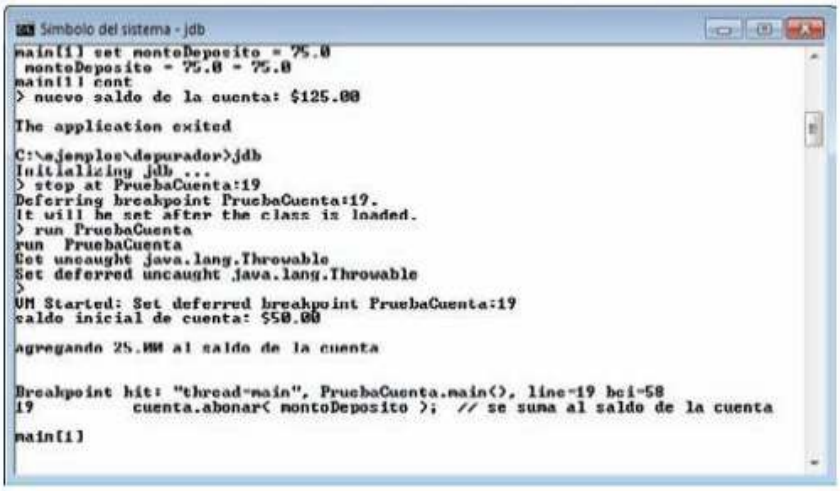


Fig. F.13 | Cómo llegar al punto de interrupción en la aplicación `PruebaCuenta`.

que se llamó. El comando `step` nos permite entrar a un método y estudiar cada una de las instrucciones de ese método. Por ejemplo, puede usar los comandos `print` y `set` para ver y modificar las variables dentro del método. Ahora usará el comando `step` para entrar al método `abonar` de la clase `Cuenta` (figura 3.13), escribiendo `step` (figura F.14). El depurador indicará que el paso se completó y mostrará la siguiente instrucción ejecutable; en este caso, la línea 21 de la clase `Cuenta` (figura 3.13).

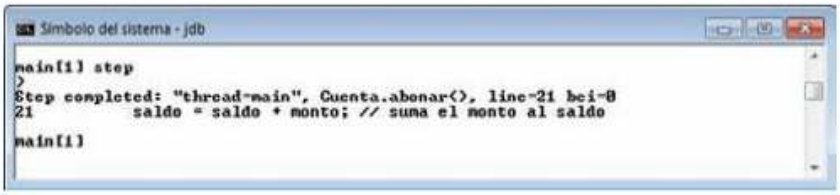


Fig. F.14 | Cómo entrar en el método `abonar`.

5. ***Usar el comando step up.*** Una vez que haya entrado al método `abonar`, escriba `step up`. Este comando ejecuta el resto de las instrucciones en el método y devuelve el control al lugar en donde se llamó al método. El método `abonar` sólo contiene una instrucción para sumar el parámetro `monto` del método a la variable de instancia `saldo`. El comando `step up` ejecuta esta instrucción y después se detiene antes de la línea 22 en `PruebaCuenta`. Por ende, la siguiente acción que ocurrirá será imprimir el nuevo saldo de la cuenta (figura F.15). En métodos extensos, tal vez sea conveniente analizar unas cuantas líneas clave de código y después continuar depurando el código del método que hizo la llamada. El comando `step up` es útil para situaciones en las que no deseamos seguir avanzando por todo el método completo, línea por línea.
6. ***Usar el comando cont para continuar la ejecución.*** Escriba el comando `cont` (figura F.16) para continuar la ejecución. A continuación se ejecutará la instrucción en las líneas 22 y 23, mostrando el nuevo saldo, y luego terminarán tanto la aplicación como el depurador.
7. ***Reiniciar el depurador.*** Para reiniciar el depurador, escriba `jdb`.

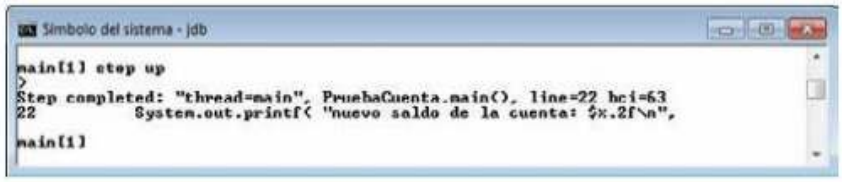


Fig. F.15 | Cómo salir de un método.

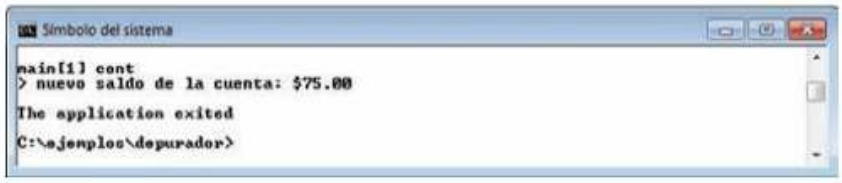


Fig. F.16 | Continuar la ejecución de la aplicación `PruebaCuenta`.

8. ***Establecer un punto de interrupción.*** Los puntos de interrupción sólo persisten hasta el fin de la sesión de depuración en la que se establecieron; una vez que el depurador deja de ejecutarse, se eliminan todos los puntos de interrupción (en la sección F.6 aprenderá cómo borrar en forma manual un punto de interrupción antes de terminar la sesión de depuración). Por ende, el punto de interrupción establecido para la línea 19 en el *paso 2* ya no existe al momento de volver a iniciar el depurador en el *paso 7*. Para restablecer el punto de interrupción en la línea 19, escriba una vez más `stop at CuentaPrueba:19`.
9. ***Ejecutar la aplicación.*** Escriba `run PruebaCuenta` para ejecutar la aplicación. Como en el *paso 3*, `PruebaCuenta` se ejecuta hasta llegar al punto de interrupción en la línea 19, y después el depurador se detiene y espera el siguiente comando (figura F.17).

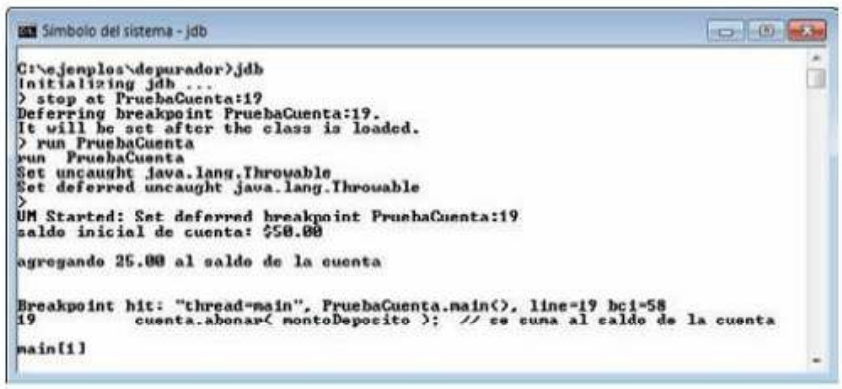


Fig. F.17 | Cómo llegar al punto de interrupción en la aplicación `PruebaCuenta`.

10. ***Usar el comando next.*** Escriba `next`. Este comando se comporta como el comando `step`, excepto cuando la siguiente instrucción a ejecutar contiene la llamada a un método. En ese

caso, el método llamado se ejecuta en su totalidad y la aplicación avanza a la siguiente línea ejecutable después de la llamada al método (figura F.18). En el *paso 4* vimos que el comando `step` entraría al método llamado. En este ejemplo, el comando `next` provoca la ejecución del método `abonar` de `Cuenta`, y después el depurador se detiene en la línea 22 de `PruebaCuenta`.

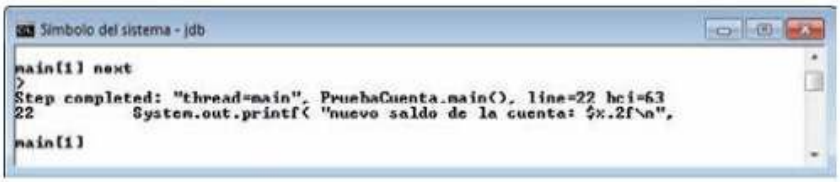


Fig. F.18 | Avanzar por toda una llamada a un método.

11. ***Usar el comando `exit`.*** Use el **comando `exit`** para salir de la sesión de depuración (figura F.19). Este comando hace que la aplicación `PruebaCuenta` termine de inmediato, en vez de ejecutar el resto de las instrucciones en `main`. Al depurar ciertos tipos de aplicaciones (como las aplicaciones de GUI), la aplicación continúa su ejecución incluso después de que termina la sesión de depuración.

Fig. F.19 | Salir del depurador.

F.5 El comando watch

En esta sección presentaremos el **comando watch**, el cual indica al depurador que debe vigilar un campo. Cuando ese campo esté a punto de cambiar, el depurador se lo notificará al programador. En esta sección aprenderá a usar el comando `watch` para ver cómo se modifica el campo `saldo` del objeto `Cuenta` durante la ejecución de la aplicación `PruebaCuenta`.

Como en las dos secciones anteriores, vamos a suponer que siguió los *pasos 1* y *2* de la sección F.2 para abrir la ventana **Símbolo del sistema**, cambiar al directorio de ejemplos correcto y compilar las clases `PruebaCuenta` y `Cuenta` para su depuración (es decir, con la opción `-g` del compilador).

1. ***Iniciar el depurador.*** Para iniciar el depurador, escriba `jdb`.
2. ***Vigilar el campo de una clase.*** Establezca un punto de inspección en el campo `saldo` de `Cuenta`, escribiendo `watch Cuenta.saldo` (figura F.20). Puede establecer un punto de inspección en cualquier campo durante la ejecución del depurador. Cada vez que está a punto de cambiar el valor en un campo, el depurador entra al modo de interrupción y notifica al programador que el valor va a cambiar. Los puntos de inspección se pueden colocar sólo en campos, no en las variables locales.
3. ***Ejecutar la aplicación.*** Ejecute la aplicación con el comando `run PruebaCuenta`. Ahora el depurador le notificará que el valor del campo `saldo` va a cambiar (figura F.21). Cuando empie-

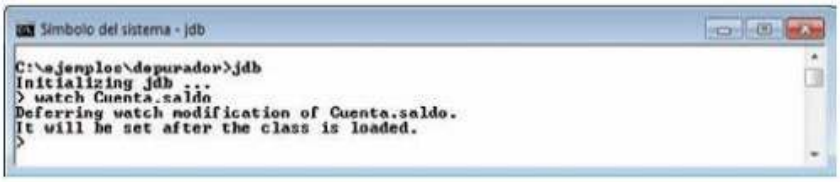


Fig. F.20 | Establecer un punto de inspección en el campo `saldo` de `Cuenta`.

za la aplicación, se crea una instancia de `Cuenta` con un saldo inicial de $50.00 y a la variable local `cuenta` se le asigna una referencia al objeto `Cuenta` (línea 9, figura F.1). En la figura 3.13 vimos que, cuando se ejecuta el constructor para este objeto, si el parámetro `saldoInicial` es mayor que `0.0`, a la variable de instancia `saldo` se le asigna el valor del parámetro `saldoInicial`. El depurador le notificará que el valor de `saldo` se establecerá en `50.0`.

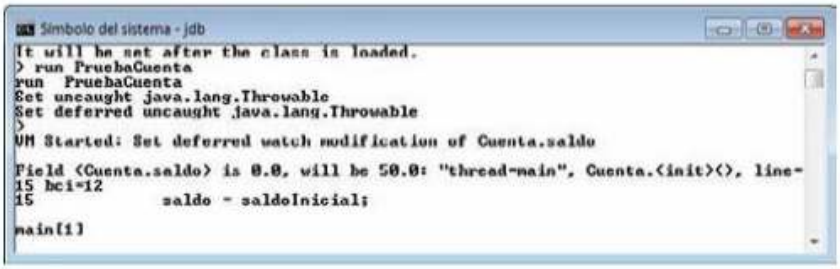


Fig. F.21 | La aplicación `PruebaCuenta` se detiene al crear la `cuenta`, y se modificará su campo `saldo`.

4. ***Agregar dinero a la cuenta.*** Escriba `cont` para continuar con la ejecución de la aplicación. Ésta se ejecutará en forma normal antes de llegar al código en la línea 19 de la figura F.1, que llama al método `abonar` de `Cuenta` para aumentar el saldo del objeto `Cuenta` por un `monto` especificado. El depurador le notificará que la variable de instancia `saldo` va a cambiar (figura F.22). Aunque la línea 19 de la clase `PruebaCuenta` llama al método `abonar`, la línea 21 en el método `abonar` de `Cuenta` es la que modifica el valor de `saldo`.

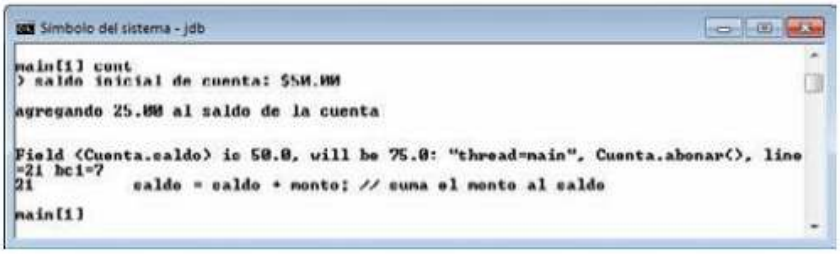


Fig. F.22 | Modificar el valor de `saldo` llamando al método `abonar` de `Cuenta`.

5. ***Continuar la ejecución.*** Escriba `cont`; la aplicación terminará de ejecutarse debido a que no intentará realizar ningún cambio adicional al `saldo` (figura F.23).

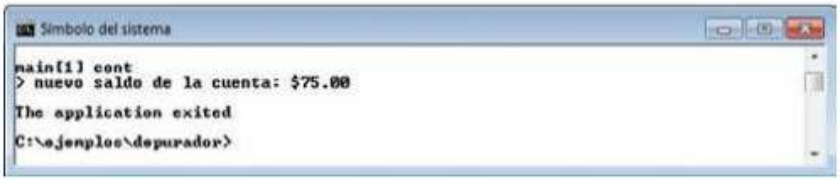


Fig. F.23 | Continuar la ejecución de `PruebaCuenta`.

6. ***Reiniciar el depurador y restablecer el punto de inspección en la variable.*** Escriba `jdb` para reiniciar el depurador. Una vez más, establezca un punto de inspección en la variable de instancia `saldo` de `Cuenta`, escribiendo `watch Cuenta.saldo`, y después escriba `run PruebaCuenta` para ejecutar la aplicación (figura F.24).

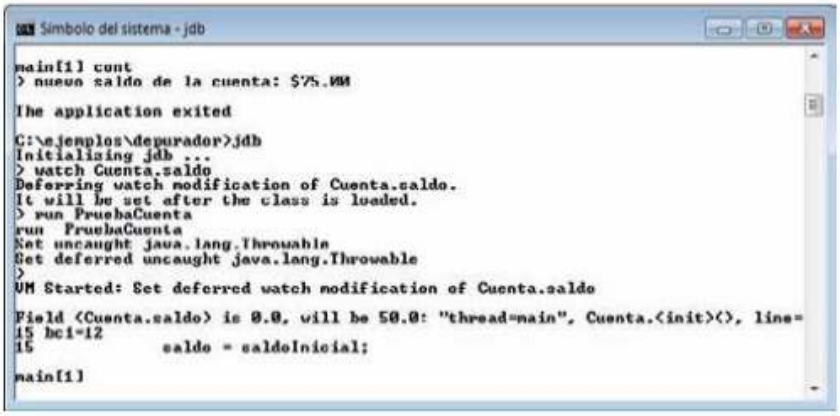


Fig. F.24 | Reiniciar el depurador y restablecer el punto de inspección en la variable `saldo`.

7. ***Eliminar el punto de inspección en el campo.*** Suponga que desea inspeccionar un campo durante sólo una parte de la ejecución de un programa. Puede eliminar el punto de inspección del depurador en la variable `saldo` si escribe `unwatch Cuenta.saldo` (figura F.25). Escriba `cont`; la aplicación terminará su ejecución sin volver a entrar al modo de interrupción.

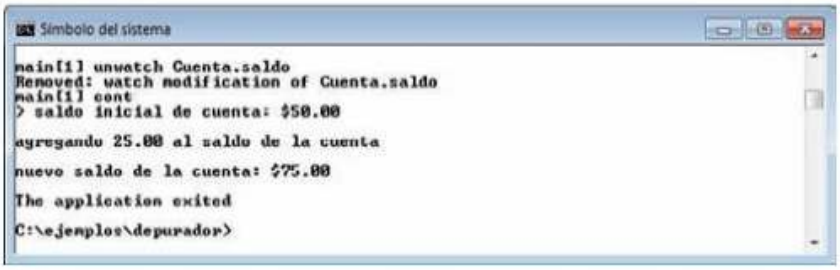


Fig. F.25 | Eliminar el punto de inspección en la variable `saldo`.

8. ***Cerrar la ventana Símbolo del sistema.*** Si desea cerrar la ventana **Símbolo del sistema**, haga clic en el botón para cerrarla.

F.6 El comando clear

En la sección anterior, aprendió a usar el comando unwatch para eliminar un punto de inspección en un campo. El depurador también cuenta con el comando `clear` para eliminar un punto de interrupción de una aplicación. A menudo es necesario depurar aplicaciones que contienen acciones repetitivas, como un ciclo. Tal vez quiera examinar los valores de las variables durante varias, pero posiblemente no todas las iteraciones del ciclo. Si establece un punto de interrupción en el cuerpo de un ciclo, el depurador se detendrá antes de cada ejecución de la línea que contenga un punto de interrupción. Después de determinar que el ciclo funciona en forma apropiada, tal vez desee eliminar el punto de interrupción y dejar que el resto de las iteraciones procedan en forma usual. En esta sección usaremos la aplicación de interés compuesto de la figura 5.6 para demostrar cómo se comporta el depurador al establecer un punto de interrupción en el cuerpo de una instrucción `for`, y cómo eliminar un punto de interrupción en medio de una sesión de depuración.

1. ***Abrir la ventana Símbolo del sistema, cambiar de directorio y compilar la aplicación para su depuración.*** Abra la ventana **Símbolo del sistema**, y después cambie al directorio que contiene los ejemplos del apéndice F. Para su conveniencia, hemos proporcionado una copia del archivo `Interes.java` en este directorio. Compile la aplicación para su depuración, escribiendo `javac -g Interes.java`.
2. ***Iniciar el depurador y establecer puntos de interrupción.*** Para iniciar el depurador escriba `jdb`. Establezca puntos de interrupción en las líneas 13 y 22 de la clase `Interes`, escribiendo `stop at Interes:13` y luego `stop at Interes:22` (figura F.26).

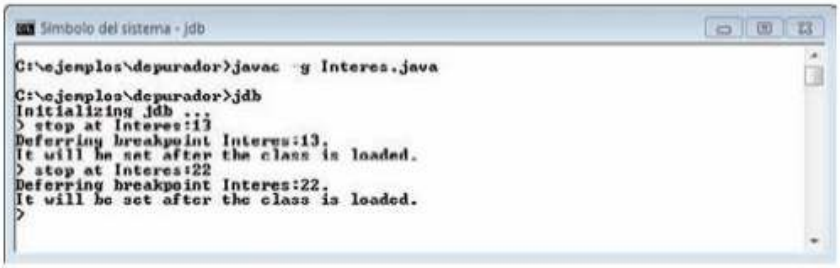


Fig. F.26 | Establecer puntos de interrupción en la aplicación `Interes`.

3. ***Ejecutar la aplicación.*** Para ejecutar la aplicación, escriba `run Interes`. La aplicación se ejecutará hasta llegar al punto de interrupción en la línea 13 (figura F.27).
4. ***Continuar la ejecución.*** Escriba `cont` para continuar; la aplicación ejecutará la línea 13, imprimiendo los encabezados de columna "`Anio`" y "`Monto en deposito`". La línea 13 aparece antes de la instrucción `for` en las líneas 16 a 23 en `Interes` (figura 5.6), y por lo tanto se ejecuta sólo una vez. La ejecución continúa después de la línea 13, hasta llegar al punto de interrupción en la línea 22 durante la primera iteración de la instrucción `for` (figura F.28).
5. ***Examinar los valores de las variables.*** Escriba `print anio` para examinar el valor actual de la variable `anio` (es decir, la variable de control del `for`). Imprima también el valor de la variable `monto` (figura F.29).

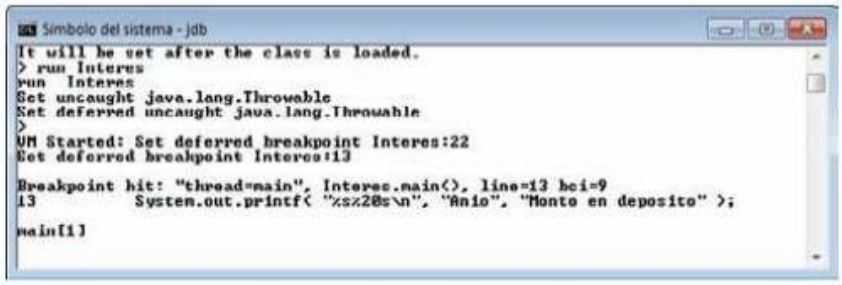


Fig. F.27 | Llegar al punto de interrupción en la línea 13 de la aplicación `Interes`.

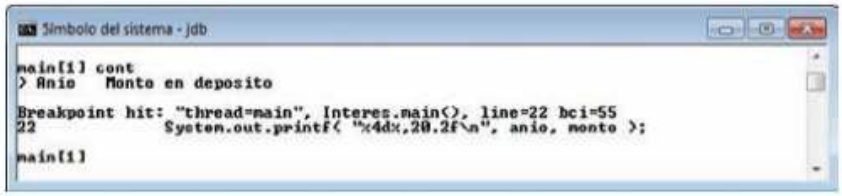


Fig. F.28 | Llegar al punto de interrupción en la línea 22 de la aplicación `Interes`.

Fig. F.29 | Imprimir `anio` y `monto` durante la primera iteración del `for` en `Interes`.

6. ***Continuar la ejecución.*** Escriba `cont` para continuar la ejecución. Se ejecutará la línea 22 e imprimirá los valores actuales de `anio` y `monto`. Después de que el `for` entre a su segunda iteración, el depurador le notificará que ha llegado al punto de interrupción en la línea 22 por segunda vez. El depurador se detiene cada vez que esté a punto de ejecutarse una línea en donde se haya establecido un punto de interrupción; cuando el punto de interrupción aparece en un ciclo, el depurador se detiene durante cada iteración. Imprima los valores de `anio` y `monto` de nuevo, para ver cómo han cambiado los valores desde la primera iteración del `for` (figura F.30).

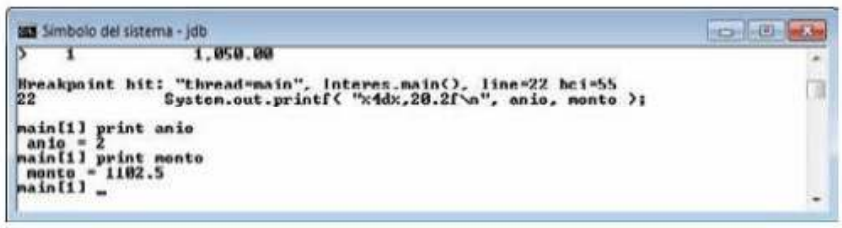


Fig. F.30 | Imprimir `monto` y `anio` durante la segunda iteración del `for` en `Interes`.

7. ***Eliminar un punto de interrupción.*** Para mostrar una lista de todos los puntos de interrupción en la aplicación, escriba `clear` (figura F.31). Suponga que está satisfecho de que la ins-

trucción `for` de la aplicación `Interes` esté trabajando en forma apropiada, por lo que desea eliminar el punto de interrupción en la línea 22 y dejar que el resto de las iteraciones del ciclo procedan en forma normal. Para eliminar el punto de interrupción en la línea 22, escriba `clear Interes:22`. Ahora escriba `clear` para ver una lista de los puntos de interrupción restantes en la aplicación. El depurador debe indicar que sólo queda el punto de interrupción en la línea 13 (figura F.31). Ya se llegó a este punto de interrupción, por lo que la ejecución no se verá afectada.

Fig. F.31 | Eliminar el punto de interrupción en la línea 22.

8. ***Continuar la ejecución después de eliminar un punto de interrupción.*** Escriba `cont` para continuar la ejecución. Recuerde que la ejecución se detuvo por última vez antes de la instrucción `printf` en la línea 22. Si el punto de interrupción de la línea 22 se eliminó con éxito, al continuar con la aplicación se producirá la salida correcta para la iteración actual y el resto de las iteraciones de la instrucción `for`, sin que se detenga la aplicación (figura F.32).

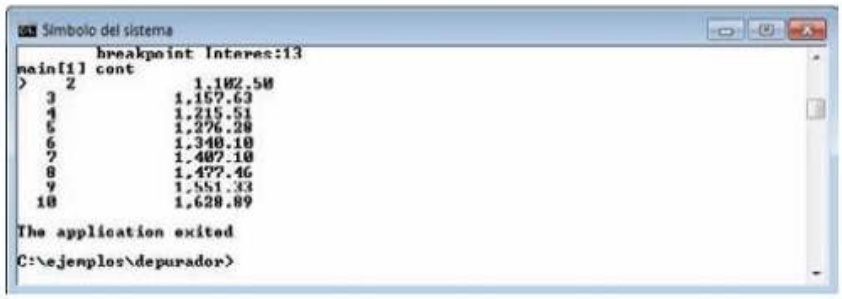


Fig. F.32 | La aplicación se ejecuta sin el punto de interrupción establecido en la línea 22.

F.7 Conclusión

En este apéndice aprendió a insertar y eliminar puntos de interrupción en el depurador. Los puntos de interrupción nos permiten detener la ejecución de la aplicación, para poder examinar los valores de las variables mediante el comando `print` del depurador. Esta herramienta nos ayuda a localizar y corregir errores lógicos en las aplicaciones. Vio cómo usar el comando `print` para examinar el valor de una expresión, y cómo usar el comando `set` para modificar el valor de una variable. También aprendió acerca de los comandos del depurador (incluyendo los comandos `step`, `step up` y `next`) que se pueden utilizar para determinar si un método se está ejecutando en forma correcta. Aprendió también a utilizar el comando `watch` para llevar el registro de un campo a lo largo de la vida de una aplicación. Por último, aprendió a utilizar el comando `clear` para listar todos los puntos de interrupción establecidos para una aplicación, o eliminar puntos de interrupción individuales para continuar la ejecución sin puntos de interrupción.

Ejercicios de autoevaluación

F.1 Complete cada uno de los siguientes enunciados:

a) Un punto de interrupción no se puede establecer en un _____ .
b) Para examinar el valor de una expresión, puede usar el comando _____ del depurador.
c) Para modificar el valor de una variable, puede usar el comando _____ del depurador.
d) Durante la depuración, el comando _____ ejecuta el resto de las instrucciones en el método actual y devuelve el control del programa al lugar en donde se hizo la llamada al método.
e) El comando _____ del depurador se comporta como el comando step, cuando la siguiente instrucción a ejecutar no contiene la llamada a un método.
f) El comando `watch` del depurador nos permite ver todos los cambios en un(a) _____ .

F.2 Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.

a) Cuando la ejecución de la aplicación se suspende en un punto de interrupción, la siguiente instrucción a ejecutarse es la instrucción que va después del punto de interrupción.
b) Los puntos de inspección se pueden eliminar mediante el comando `clear` del depurador.
c) Hay que usar la opción `-g` del compilador cuando se compilen clases para su depuración.
d) Cuando aparece un punto de interrupción en un ciclo, el depurador se detiene sólo la primera vez que se encuentra con el punto de interrupción.

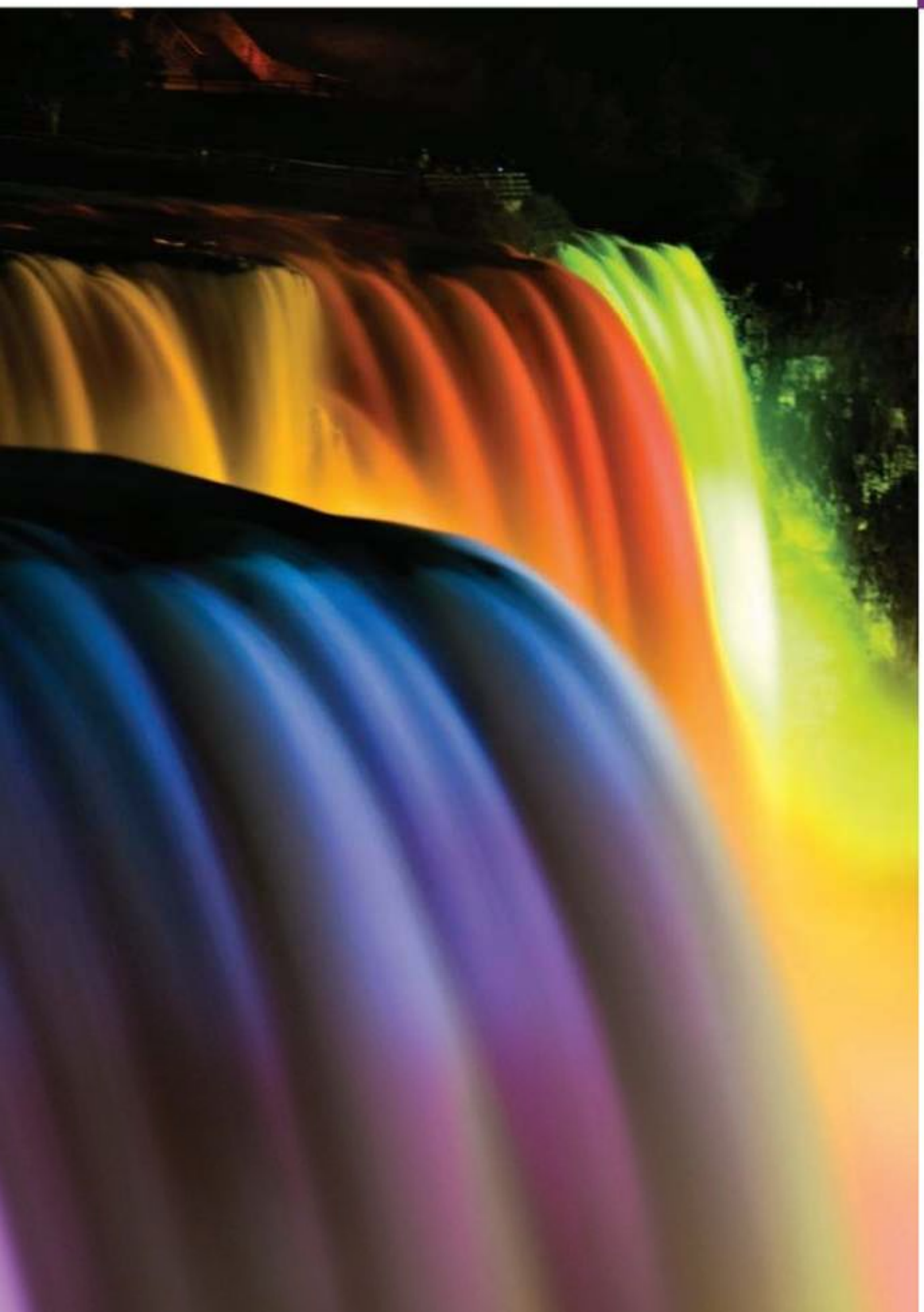
Respuestas a los ejercicios de autoevaluación

F.1 a) comentario. b) `print`. c) `set`. d) `step up`. e) `next`. f) campo.

F.2 a) Falso. Cuando se suspende la ejecución de la aplicación en un punto de interrupción, la siguiente instrucción a ejecutarse es la que está en el punto de interrupción. b) Falso. Los puntos de inspección se pueden eliminar mediante el comando `unwatch` del depurador. c) Verdadero. d) Falso. Cuando aparece un punto de interrupción en un ciclo, el depurador se detiene durante cada iteración.

Salida con formato

G



Todas las noticias que puede imprimir.

—Adolph S. Ochs

¿Qué loca persecución?
¿De qué aprieto escapar?

—John Keats

No elimines el punto de referencia en el límite de los campos.

—Amenehope

Objetivos

En este apéndice aprenderá a:

- Comprender los flujos de entrada y salida.
- Usar el formato con `printf`.
- Imprimir con anchuras de campo y precisiones.
- Usar banderas de formato en la cadena de formato de `printf`.
- Imprimir con un índice como argumento.
- Imprimir literales y secuencias de escape en pantalla.
- Dar formato a la salida con la clase `Formatter`.

Plan general

G.1 Introducción

En este apéndice, hablaremos sobre las características de formato del método `printf` y la clase `Formatter` (paquete `java.util`). La clase **`Formatter`** aplica formato a los datos y los imprime en un destino especificado, como una cadena o un flujo de salida de archivo. Anteriormente en este libro hablamos sobre muchas de las características de `printf`. En este apéndice se sintetizan esas características y se presentan otras, como mostrar datos de fecha y hora en varios formatos, reordenar la salida con base en el índice del argumento, y mostrar números y cadenas con varias banderas.

G.2 Flujos

Por lo general, las operaciones de entrada y salida se llevan a cabo con flujos, los cuales son secuencias de bytes. En las operaciones de entrada, los bytes fluyen de un dispositivo (como un teclado, una unidad de disco, una conexión de red) a la memoria principal. En las operaciones de salida, los bytes fluyen de la memoria principal a un dispositivo (como una pantalla, una impresora, una unidad de disco, una conexión de red).

Cuando empieza la ejecución de un programa, se crean tres flujos. Por lo común, el flujo de entrada estándar lee bytes mediante el teclado, y el flujo de salida estándar envía caracteres a una ventana de comandos. Un tercer flujo, el **flujo de error estándar** (`System.err`), por lo general envía caracteres a una ventana de comandos y se utiliza para imprimir mensajes de error en ésta, de manera que puedan verse de inmediato. Comúnmente, los sistemas operativos permiten redirigir estos flujos a otros dispositivos. En el capítulo 17, Archivos, flujos y serialización de objetos, y en el capítulo 24 (en inglés, en el sitio Web) se describen los flujos con detalle.

G.3 Aplicación de formato a la salida con `printf`

Con `printf` podemos lograr un formato preciso en la salida. Java tomó prestada (y mejoró) esta característica del lenguaje de programación C. El método `printf` cuenta con las siguientes herramientas de formato, cada una de las cuales se verá en este apéndice:

1. Redondeo de valores de punto flotante a un número indicado de posiciones decimales.
2. Alineación de una columna de números con puntos decimales, que aparezcan uno encima de otro.
3. Justificación a la derecha y justificación a la izquierda de los resultados.

4. Inserción de caracteres literales en posiciones precisas de una línea de salida.
5. Representación de números de punto flotante en formato exponencial.
6. Representación de enteros en formato octal y hexadecimal.
7. Visualización de todo tipo de datos con anchuras de campo de tamaño fijo y precisiones.
8. Visualización de fechas y horas en diversos formatos.

Cada llamada a `printf` proporciona como primer argumento una **cadena de formato**, la cual describe el formato de salida. La cadena de formato puede consistir en **texto fijo** y **especificadores de formato**. El texto fijo se imprime mediante `printf` de igual forma que como se imprimiría mediante los métodos print o `println` de `System.out`. Cada especificador de formato es un receptáculo para un valor y especifica el tipo de datos a imprimir. Los especificadores de formato también pueden incluir información de formato opcional.

En su forma más simple, cada especificador de formato empieza con un signo de porcentaje (%) y va seguido de un **carácter de conversión** que representa el tipo de datos del valor a imprimir. Por ejemplo, el especificador de formato `%s` es un receptáculo para una cadena, y el especificador de formato `%d` es un receptáculo para un valor `int`. La información de formato opcional, como un índice de argumento, banderas, anchura de campo y precisión, se especifica entre el signo de porcentaje y el carácter de conversión. Más adelante demostraremos cada una de estas capacidades.

G.4 Impresión de enteros

En la figura G.1 se describen los caracteres de conversión de enteros (en el apéndice H podrá ver las generalidades sobre los sistemas binario, octal, decimal y hexadecimal). En la figura G.2 se usa cada uno de estos caracteres para imprimir un entero. En las líneas 9 a 10, el signo positivo no se muestra de manera predeterminada, pero el signo negativo sí. Más adelante en este apéndice (figura G.14) veremos cómo forzar a que se impriman los signos positivos.

Carácter de conversión	Descripción
`d`	Muestra un entero decimal (base 10).
`o`	Muestra un entero octal (base 8).
`x` o `X`	Muestra un entero hexadecimal (base 16). `X` usa letras mayúsculas.

Fig. G.1 | Caracteres de conversión de enteros.

```
1  // Fig. G.2: PruebaConversionEnteros.java
2  // Uso de los caracteres de conversión integrales.
3
4  public class PruebaConversionEnteros
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf( "%d\n", 26 );
9        System.out.printf( "%d\n", +26 );
10       System.out.printf( "%d\n", -26 );
11       System.out.printf( "%o\n", 26 );
12       System.out.printf( "%x\n", 26 );
```

Fig. G.2 | Uso de los caracteres de conversión de enteros (parte 1 de 2).

```
13          System.out.printf( "%X\n", 26 );
14       } // fin de main
15    } // fin de la clase PruebaConversionEnteros
```

```
26
26
-26
32
1a
1A
```

Fig. G.2 | Uso de los caracteres de conversión de enteros (parte 2 de 2).

El método `printf` tiene la forma

```
printf( cadena-de-formato, lista-de-argumentos );
```

en donde *cadena-de-formato* describe el formato de salida, y la *lista-de-argumentos* opcional contiene los valores que corresponden a cada especificador de formato en *cadena-de-formato*. Puede haber muchos especificadores de formato en una cadena de formato.

Cada cadena de formato en las líneas 8 a 10 especifica que `printf` debe imprimir un entero decimal (`%d`) seguido de un carácter de nueva línea. En la posición del especificador de formato, `printf` sustituye el valor del primer argumento después de la cadena de formato. Si la cadena de formato contiene varios especificadores de formato, en cada posición del siguiente especificador de formato, `printf` sustituirá el valor del siguiente argumento en la lista de argumentos. El especificador de formato `%o` en la línea 11 imprime el entero en formato octal. El especificador de formato `%x` en la línea 12 imprime el entero en formato hexadecimal. El especificador de formato `%X` en la línea 13 imprime el entero en formato hexadecimal, con letras mayúsculas.

G.5 Impresión de números de punto flotante

En la figura G.3 se describen las conversiones de punto flotante. Los **caracteres de conversión e** y **E** muestran valores de punto flotante en **notación científica computarizada** (también conocida como **notación exponencial**). La notación exponencial es el equivalente computacional de la notación científica que se utiliza en las matemáticas. Por ejemplo, el valor `150.4582` se representa en notación científica matemática de la siguiente manera:

$$1.504582 \times 10^2$$

y se representa en notación exponencial como

```
1.504582e+02
```

en Java. Esta notación indica que `1.504582` se multiplica por 10 elevado a la segunda potencia (`e+02`). La `e` representa al "exponente".

Los valores que se imprimen con los caracteres de conversión `e`, `E` y `f` se muestran con seis dígitos de precisión en el lado derecho del punto decimal de manera predeterminada (por ejemplo, `1.045921`); otras precisiones se deben especificar de manera explícita. Para los valores impresos con el carácter de conversión `g`, la precisión representa el número total de dígitos mostrados, excluyendo el exponente. El valor predeterminado es de seis dígitos (por ejemplo, `12345678.9` se muestra como `1.23457e+07`). El **carácter de conversión f** siempre imprime por lo menos un dígito a la izquierda del punto decimal. Los caracteres

Carácter de conversión	Descripción
e o E	Muestra un valor de punto flotante en notación exponencial. El carácter de conversión E muestra la salida en letras mayúsculas.
f	Muestra un valor de punto flotante en formato decimal.
g o G	Muestra un valor de punto flotante en el formato de punto flotante f o en el formato exponencial e, con base en la magnitud del valor. Si la magnitud es menor que 10^{-3}, o si es mayor o igual que 10^{7}, el valor de punto flotante se imprime con e (o E). En cualquier otro caso, el valor se imprime en el formato f. Cuando se utiliza el carácter de conversión G, la salida se muestra en letras mayúsculas.
a o A	Muestra un número de punto flotante en formato hexadecimal. El carácter de conversión A muestra la salida en letras mayúsculas.

Fig. G.3 | Caracteres de conversión de enteros.

de conversión e y E imprimen una e minúscula y una E mayúscula antes del exponente, y siempre imprimen sólo un dígito a la izquierda del punto decimal. El redondeo ocurre si el valor al que se está dando formato tiene más dígitos significativos que la precisión.

El **carácter de conversión g** (o G) imprime en formato e (E) o f, dependiendo del valor de punto flotante. Por ejemplo, los valores 0.0000875, 87500000.0, 8.75, 87.50 y 875.0 se imprimen como 8.750000e-05, 8.750000e+07, 8.750000, 87.500000 y 875.000000 con el carácter de conversión g. El valor 0.0000875 utiliza la notación e ya que la magnitud es menor que 10-3. El valor 87500000.0 utiliza la notación e, debido a que la magnitud es mayor que 10^{7}. En la figura G.4 se muestra cada uno de los caracteres de conversión de punto flotante.

```
1  // Fig. G.4: PruebaPuntoFlotante.java
2  // Uso de los caracteres de conversión de punto flotante.
3
4  public class PruebaPuntoFlotante
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf( "%e\n", 12345678.9 );
9        System.out.printf( "%e\n", +12345678.9 );
10       System.out.printf( "%e\n", -12345678.9 );
11       System.out.printf( "%E\n", 12345678.9 );
12       System.out.printf( "%f\n", 12345678.9 );
13       System.out.printf( "%g\n", 12345678.9 );
14       System.out.printf( "%G\n", 12345678.9 );
15    } // fin de main
16 } // fin de la clase PruebaPuntoFlotante
```

```
1.234568e+07
1.234568e+07
-1.234568e+07
```

Fig. G.4 | Uso de los caracteres de conversión de punto flotante (parte 1 de 2).

```
1.234568E+07
12345678.900000
1.23457e+07
1.23457E+07
```

Fig. G.4 | Uso de los caracteres de conversión de punto flotante (parte 2 de 2).

G.6 Impresión de cadenas y caracteres

Los caracteres de conversión `c` y `s` se utilizan para imprimir caracteres individuales y cadenas. Los **caracteres de conversión `c` y `C`** requieren un argumento `char`. Los **caracteres de conversión `s` y `S`** pueden recibir un objeto `String` o cualquier objeto `Object` como argumento. Cuando se utilizan los caracteres de conversión `C` y `S`, la salida se muestra en letras mayúsculas. La figura G.5 imprime en pantalla caracteres, cadenas y objetos con los caracteres de conversión `c` y `s`. En la línea 9 se realiza una conversión autoboxing , cuando se asigna una constante `int` a un objeto `Integer`. En la línea 15 se imprime en pantalla un argumento `Integer` con el carácter de conversión `s`, el cual invoca de manera implícita al método `toString` para obtener el valor entero. También puede imprimir en pantalla un objeto `Integer` mediante el uso del especificador de formato `%d`. En este caso, se realizará una conversión `unboxing` con el valor `int` en el objeto `Integer` y se imprimirá en pantalla.

Error común de programación G.1

El uso de `%c` para imprimir objeto `String` produce una excepción `IllegalFormatConversionException`; un objeto `String` no se puede convertir en un carácter.

```
1  // Fig. G.5: ConversionCadenasChar.java
2  // Uso de los caracteres de conversión de cadenas y caracteres.
3  public class ConversionCadenasChar
4  {
5     public static void main( String[] args )
6     {
7        char caracter = 'A';  // inicializa el char
8        String cadena = "Esta tambien es una cadena";  // objeto String
9        Integer entero = 1234;  // inicializa el entero (autoboxing)
10
11       System.out.printf( "%c\n", caracter );
12       System.out.printf( "%s\n", "Esta es una cadena" );
13       System.out.printf( "%s\n", cadena );
14       System.out.printf( "%S\n", cadena );
15       System.out.printf( "%s\n", entero ); // llamada implicita a toString
16    } // fin de main
17 } // fin de la clase ConversionCadenasChar
```

```
A
Esta es una cadena
Esta tambien es una cadena
ESTA TAMBIEN ES UNA CADENA
1234
```

Fig. G.5 | Uso de los caracteres de conversión de cadenas y caracteres.

G.7 Impresión de fechas y horas

El **carácter de conversión t** (o `T`) se utiliza para imprimir fechas y horas en diversos formatos. Siempre va seguido de un **carácter de sufijo de conversión** que especifica el formato de fecha y/o de hora. Cuando se utiliza el carácter de conversión `T`, la salida se muestra en letras mayúsculas. En la figura G.6 se listan los caracteres de sufijo de conversión comunes para aplicar formato a las **composiciones de fecha y hora** que muestran tanto la fecha como la hora. En la figura G.7 se listan los caracteres de sufijo de conversión comunes para aplicar formato a las fechas. En la figura G.8 se listan los caracteres de sufijo de conversión comunes para aplicar formato a las horas. Para ver la lista completa de caracteres de sufijo de conversión, visite el sitio Web `java.sun.com/javase/6/docs/api/java/util/Formatter.html`.

Carácter de sufijo de conversión	Descripción
`c`	Muestra la fecha y hora con el formato `dia mes fecha hora:minuto:segundo zona-horaria año` con tres caracteres para `dia` y `mes`, dos dígitos para `fecha`, `hora`, `minuto` y `segundo`, y cuatro dígitos para `año`; por ejemplo, `Mié Mar 03 16:30:25 GMT -05:00 2004`. Se utiliza el reloj de 24 horas. `GMT -05:00` es la zona horaria.
`F`	Muestra la fecha con el formato `año-mes-dia` con cuatro dígitos para el `año` y dos dígitos para el `mes` y la `fecha` (por ejemplo, `2004-05-04`).
`D`	Muestra la fecha con el formato `mes/dia/año`, con dos dígitos para el `mes`, `dia` y `año` (por ejemplo, `03/03/04`).
`r`	Muestra la hora en formato de 12 horas como `hora:minuto:segundo AM|PM`, con dos dígitos para la `hora`, `minuto` y `segundo` (por ejemplo, `04:30:25 PM`).
`R`	Muestra la hora con el formato `hora:minuto`, con dos dígitos para la `hora` y el `minuto` (por ejemplo, `16:30`). Se utiliza el reloj de 24 horas.
`T`	Muestra la hora con el formato `hora:minuto:segundo`, con dos dígitos para la `hora`, `minuto` y `segundo` (por ejemplo, `16:30:25`). Se utiliza el reloj de 24 horas.

Fig. G.6 | Caracteres de sufijo de conversión de composiciones de fecha y hora.

Carácter de sufijo de conversión	Descripción
`A`	Muestra el nombre completo del día de la semana (por ejemplo, `Miércoles`).
`a`	Muestra el nombre corto de tres caracteres del día de la semana (por ejemplo, `Mié`).
`B`	Muestra el nombre completo del mes (por ejemplo, `Marzo`).
`b`	Muestra el nombre corto de tres caracteres del mes (por ejemplo, `Mar`).
`d`	Muestra el día del mes con dos dígitos, rellenando con ceros a la izquierda si es necesario (por ejemplo, `03`).
`m`	Muestra el mes con dos dígitos, rellenando con ceros a la izquierda si es necesario (por ejemplo, `07`).

Fig. G.7 | Caracteres de sufijo de conversión para aplicar formato a las fechas (parte 1 de 2).

Carácter de sufijo de conversión	Descripción
`e`	Muestra el día del mes sin ceros a la izquierda (por ejemplo, 3).
`Y`	Muestra el año con cuatro dígitos (por ejemplo, 2004).
`y`	Muestra los dos últimos dígitos del año con ceros a la izquierda (por ejemplo, 04).
`j`	Muestra el día del año con tres dígitos, rellenando con ceros a la izquierda según sea necesario (por ejemplo, 016).

Fig. G.7 | Caracteres de sufijo de conversión para aplicar formato a las fechas (parte 2 de 2).

Carácter de sufijo de conversión	Descripción
`H`	Muestra la hora en el reloj de 24 horas, con un cero a la izquierda si es necesario (por ejemplo, 16).
`I`	Muestra la hora en el reloj de 12 horas, con un cero a la izquierda si es necesario (por ejemplo, 04).
`k`	Muestra la hora en el reloj de 24 horas sin ceros a la izquierda (por ejemplo, 16).
`l`	Muestra la hora en el reloj de 12 horas sin ceros a la izquierda (por ejemplo, 4).
`M`	Muestra los minutos con un cero a la izquierda, si es necesario (por ejemplo, 06).
`S`	Muestra los segundos con un cero a la izquierda, si es necesario (por ejemplo, 05).
`Z`	Muestra la abreviación para la zona horaria (por ejemplo, `EST` representa a la Hora estándar occidental, la cual se encuentra a 5 horas de retraso de la Hora del Meridiano de Greenwich).
`p`	Muestra el marcador de mañana o tarde en minúsculas (por ejemplo, `pm`).
`P`	Muestra el marcador de mañana o tarde en mayúsculas (por ejemplo, `PM`).

Fig. G.8 | Caracteres de sufijo de conversión para aplicar formato a las horas.

En la figura G.9 se utiliza el carácter de conversión `t` y `T` con los caracteres de sufijo de conversión para mostrar fechas y horas en diversos formatos. El carácter de conversión `t` requiere que su correspondiente argumento sea una fecha u hora de tipo `long`, `Long`, `Calendar` (paquete `java.util`) o `Date` (paquete `java.util`); los objetos de cada una de estas clases pueden representar fechas y horas. La clase `Calendar` es la preferida para este propósito, ya que ciertos constructores y métodos en la clase `Date` se

```
1  // Fig. G.9: PruebaFechaHora.java
2  // Aplicación de formato a fechas y horas con los caracteres de conversión t y T.
3  import java.util.Calendar;
4
5  public class PruebaFechaHora
6  {
7     public static void main( String[] args )
8     {
9        // obtiene la fecha y hora actuales
10       Calendar fechaHora = Calendar.getInstance();
```

Fig. G.9 | Aplicación de formato a fechas y horas con los caracteres de conversión `t` y `T` (parte 1 de 2).

```
11
12        // impresión con caracteres de conversión para composiciones de fecha/hora
13        System.out.printf( "%tc\n", fechaHora );
14        System.out.printf( "%tF\n", fechaHora );
15        System.out.printf( "%tD\n", fechaHora );
16        System.out.printf( "%tr\n", fechaHora );
17        System.out.printf( "%tT\n", fechaHora );
18
19        // impresión con caracteres de conversión para fechas
20        System.out.printf( "%1$tA, %1$tB %1$td, %1$tY\n", fechaHora );
21        System.out.printf( "%1$TA, %1$TB %1$Td, %1$TY\n", fechaHora );
22        System.out.printf( "%1$ta, %1$tb %1$te, %1$ty\n", fechaHora );
23
24        // impresión con caracteres de conversión para horas
25        System.out.printf( "%1$tH:%1$tM:%1$tS\n", fechaHora );
26        System.out.printf( "%1$tZ %1$tI:%1$tM:%1$tS %tp", fechaHora );
27     } // fin de main
28  } // fin de la clase PruebaFechaHora
```

```
mié nov 07 11:54:30 CST 2007
2007-11-07
11/07/07
11:54:30 AM
11:54:30
miércoles, noviembre 07, 2007
MIÉRCOLES, NOVIEMBRE 07, 2007
mié, nov 7, 07
11:54:30
CST 11:54:30 AM
```

Fig. G.9 | Aplicación de formato a fechas y horas con los caracteres de conversión `t` y `T` (parte 2 de 2).

sustituyen por los de la clase `Calendar`. En la línea 10 se invoca el método `static` **`getInstance`** de `Calendar` para obtener un calendario con la fecha y hora actuales. En las líneas 13 a 17, 20 a 22 y 25 a 26 se utiliza este objeto `Calendar` en instrucciones `printf` como el valor al que se aplicará formato con el carácter de conversión `t`. En las líneas 20 a 22 y 25 a 26 se utiliza el índice como **argumento opcional** (`"1$"`) para indicar que todos los especificadores de formato en la cadena de formato utilizan el primer argumento después de la cadena de formato en la lista de argumentos. En la sección G.11 aprenderá más acerca de los índices como argumentos. Al usar el índice como argumento, se elimina la necesidad de enumerar repetidas veces el mismo argumento.

G.8 Otros caracteres de conversión

El resto de los caracteres de conversión son **`b`**, **`B`**, **`h`**, **`H`**, **`%`** y **`n`**. Éstos se describen en la figura G.10. En las líneas 9 y 10 de la figura G.11 se utiliza `%b` para imprimir el valor de los valores `boolean` (o `Boolean`) `false` y `true`. En la línea 11 se asocia un objeto `String` a `%b`, el cual devuelve `true` debido a que no es `null`. En la línea 12 se asocia un objeto `null` a `%B`, el cual muestra `FALSE` ya que prueba es `null`. En las líneas 13 y 14 se utiliza `%h` para imprimir las representaciones de cadena de los valores de código hash para las cadenas `"hola"` y `"Hola"`. Estos valores se podrían utilizar para almacenar o colocar las cadenas en un objeto `Hashtable` o `HashMap` (los cuales pueden verse en el capítulo 20 y en inglés en el sitio Web). Los valores de código hash para estas dos cadenas difieren, ya que una cadena empieza con letra

minúscula y la otra con letra mayúscula. En la línea 15 se utiliza `%H` para imprimir `null` en letras mayúsculas. Las últimas dos instrucciones `printf` (líneas 16 y 17) utilizan `%%` para imprimir el carácter `%` en una cadena, y `%n` para imprimir un separador de línea específico de la plataforma.

Carácter de conversión	Descripción
`b` o `B`	Imprime "`true`" o "`false`" para el valor de un `boolean` o `Boolean`. Estos caracteres de conversión también pueden aplicar formato al valor de cualquier referencia. Si la referencia no es `null`, se imprime "`true`"; en caso contrario, se imprime "`false`". Cuando se utiliza el carácter de conversión `B`, la salida se muestra en letras mayúsculas.
`h` o `H`	Imprime la representación de cadena del valor de código hash de un objeto en formato hexadecimal. Si el correspondiente argumento es `null`, se imprime "`null`". Cuando se utiliza el carácter de conversión `H`, la salida se muestra en letras mayúsculas.
`%`	Imprime el carácter de por ciento.
`n`	Imprime el separador de línea específico de la plataforma (por ejemplo, `\r\n` en Windows o `\n` en UNIX/LINUX).

Fig. G.10 | Otros especificadores de conversión.

```
1  // Fig. G.11: OtrasConversiones.java
2  // Uso de los caracteres de conversión b, B, h, H, % y n.
3
4  public class OtrasConversiones
5  {
6     public static void main( String[] args[] )
7     {
8        Object prueba = null;
9        System.out.printf( "%b\n", false );
10       System.out.printf( "%b\n", true );
11       System.out.printf( "%b\n", "Prueba" );
12       System.out.printf( "%B\n", prueba );
13       System.out.printf( "El codigo hash de \"hola\" es %h\n", "hola" );
14       System.out.printf( "El codigo hash de \"Hola\" es %h\n", "Hola" );
15       System.out.printf( "El codigo hash de null es %H\n", prueba );
16       System.out.printf( "Impresion de un %% en una cadena de formato\n" );
17       System.out.printf( "Impresion de una nueva linea %nla siguiente linea empieza
                            aqui" );
18    } // fin de main
19 } // fin de la clase OtrasConversiones
```

```
false
true
true
FALSE
El codigo hash de "hola" es 30f4bc
```

Fig. G.11 | Uso de los caracteres de conversión `b`, `B`, `h`, `H`, `%` y `n` (parte 1 de 2).

```
El codigo hash de "Hola" es 2268dc
El codigo hash de null es NULL
Impresion de un % en una cadena de formato
Impresion de una nueva linea
la siguiente linea empieza aquí
```

Fig. G.11 | Uso de los caracteres de conversión b, B, h, H, % y n (parte 2 de 2).

Error común de programación G.2

Tratar de imprimir un carácter de porcentaje literal mediante el uso de % en vez de %% en la cadena de formato podría provocar un error lógico difícil de detectar. Cuando aparece el % en una cadena de formato, debe ir seguido de un carácter de conversión en la cadena. El signo de por ciento individual podría ir seguido accidentalmente de un carácter de conversión legítimo, con lo cual se produciría un error lógico.

G.9 Impresión con anchuras de campo y precisiones

El tamaño de un campo en el que se imprimen datos se especifica mediante una **anchura de campo**. Si la anchura de campo es mayor que los datos que se van a imprimir, éstos se justificarán a la derecha dentro de ese campo, de manera predeterminada. En la sección G.10 demostraremos la justificación a la izquierda. El programador inserta un entero que representa la anchura de campo entre el signo % y el carácter de conversión (por ejemplo, %4d) en el especificador de formato. En la figura G.12 se imprimen dos grupos de cinco números cada uno, y se justifican a la derecha los números que contienen menos dígitos que la anchura de campo. La anchura de campo se incrementa para imprimir valores más anchos que el campo, y el signo menos para un valor negativo utiliza una posición de carácter en el campo. Además, si no se especifica la anchura del campo, los datos se imprimen en todas las posiciones que sean necesarias. Las anchuras de campo pueden utilizarse con todos los especificadores de formato, excepto el separador de línea (%n).

```
1  // Fig. G.12: PruebaAnchuraCampo.java
2  // Justificación a la derecha de enteros en campos.
3
4  public class PruebaAnchuraCampo
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf( "%4d\n", 1 );
9        System.out.printf( "%4d\n", 12 );
10       System.out.printf( "%4d\n", 123 );
11       System.out.printf( "%4d\n", 1234 );
12       System.out.printf( "%4d\n\n", 12345 ); // datos demasiado extensos
13
14       System.out.printf( "%4d\n", -1 );
15       System.out.printf( "%4d\n", -12 );
16       System.out.printf( "%4d\n", -123 );
17       System.out.printf( "%4d\n", -1234 ); // datos demasiado extensos
18       System.out.printf( "%4d\n", -12345 ); // datos demasiado extensos
19    } // fin de main
20 } // fin de la clase PruebaAnchuraCampo
```

Fig. G.12 | Justificación a la derecha de enteros en campos (parte 1 de 2).

```
    1
   12
  123
 1234
12345

   -1
  -12
 -123
-1234
-12345
```

Fig. G.12 | Justificación a la derecha de enteros en campos (parte 2 de 2).

Error común de programación G.3

Si no se proporciona una anchura de campo suficientemente extensa como para manejar un valor a imprimir, se pueden desplazar los demás datos que se impriman, con lo que se producirán resultados confusos. ¡Debe conocer sus datos!

El método `printf` también proporciona la habilidad de especificar la precisión con la que se van a imprimir los datos. La precisión tiene distintos significados para los diferentes tipos. Cuando se utiliza con los caracteres de conversión de punto flotante `e` y `f`, la precisión es el número de dígitos que aparecen después del punto decimal. Cuando se utiliza con los caracteres de conversión `g`, `a` o `A`, la precisión es el número máximo de dígitos significativos a imprimir. Cuando se utiliza con el carácter de conversión `s`, la precisión es el número máximo de caracteres a escribir de la cadena. Para utilizar la precisión, se debe colocar entre el signo de porcentaje y el especificador de conversión un punto decimal (`.`), seguido de un entero que representa la precisión. En la figura G.13 se demuestra el uso de la precisión en las cadenas de formato. Cuando se imprime un valor de punto flotante con una precisión menor que el número original de posiciones decimales en el valor, éste se redondea. Además, el especificador de formato `%.3g` indica que el número total de dígitos utilizados para mostrar el valor de punto flotante es 3. Como el valor tiene tres dígitos a la izquierda del punto decimal, se redondea a la posición de las unidades.

La anchura de campo y la precisión pueden combinarse, para lo cual se coloca la anchura de campo, seguida de un punto decimal, seguido de una precisión entre el signo de porcentaje y el carácter de conversión, como en la siguiente instrucción:

```
printf( "%9.3f", 123.456789 );
```

la cual muestra `123.457` con tres dígitos a la derecha del punto decimal, y se justifica a la derecha en un campo de nueve dígitos; antes del número se colocarán dos espacios en blanco en su campo.

```
1  // Fig G.13: PruebaPrecision.java
2  // Uso de la precisión para números de punto flotante y cadenas.
3  public class PruebaPrecision
4  {
5     public static void main( String[] args )
6     {
7        double f = 123.94536;
8        String s = "Feliz Cumpleanios";
```

Fig. G.13 | Uso de la precisión para los números de punto flotante y las cadenas (parte 1 de 2).

```
9
10          System.out.printf( "Uso de la precision para numeros de punto flotante\n" );
11          System.out.printf( "\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f );
12
13          System.out.printf( "Uso de la precision para las cadenas\n" );
14          System.out.printf( "\t%.11s\n", s );
15       } // fin de main
16    } // fin de la clase PruebaPrecision
```

```
Uso de la precision para numeros de punto flotante
        123.945
        1.239e+02
        124

Uso de la precision para las cadenas
        Feliz Cumpl
```

Fig. G.13 | Uso de la precisión para los números de punto flotante y las cadenas (parte 2 de 2).

G.10 Uso de banderas en la cadena de formato de `printf`

Pueden usarse varias banderas con el método `printf` para suplementar sus herramientas de formato de salida. Hay siete banderas disponibles para usarlas en las cadenas de formato (figura G.14).

Bandera	Descripción
- (signo negativo)	Justifica a la izquierda la salida dentro del campo especificado.
+ (signo positivo)	Muestra un signo positivo antes de los valores positivos, y un signo negativo antes de los valores negativos.
espacio	Imprime un espacio antes de un valor positivo que no se imprime con la bandera +.
#	Antepone un 0 al valor de salida cuando se utiliza con el carácter de conversión octal o. Antepone 0x al valor de salida cuando se usa con el carácter de conversión hexadecimal x.
0 (cero)	Rellena un campo con ceros a la izquierda.
, (coma)	Usa el separador de miles específico para la configuración regional (es decir, ',' para los EUA), para mostrar números decimales y de punto flotante.
(	Encierra los números negativos entre paréntesis.

Fig. G.14 | Banderas de la cadena de formato.

Para usar una bandera en una cadena de formato, coloque la bandera justo a la derecha del signo de porcentaje. Pueden usarse varias banderas en el mismo especificador de formato. En la figura G.15 se muestra la justificación a la derecha y la justificación a la izquierda de una cadena, un entero, un carácter y un número de punto flotante. La línea 9 sirve como mecanismo de conteo para la salida en la pantalla.

En la figura G.16 se imprime un número positivo y un número negativo, cada uno con y sin la **bandera +**. El signo negativo se muestra en ambos casos, pero el signo positivo se muestra sólo cuando se utiliza la bandera +.

```
1  // Fig G.15: PruebaBanderaMenos.java
2  // Justificación a la derecha y justificación a la izquierda de los valores
3
4  public class PruebaBanderaMenos
5  {
6     public static void main( String[] args )
7     {
8        System.out.println( "Columnas:" );
9        System.out.println( "0123456789012345678901234567890123456789\n" );
10       System.out.printf( "%10s%10d%10c%10f\n\n", "hola", 7, 'a', 1.23 );
11       System.out.printf(
12          "%-10s%-10d%-10c%-10f\n", "hola", 7, 'a', 1.23 );
13    } // fin de main
14 } // fin de la clase PruebaBanderaMenos
```

```
Columnas:
0123456789012345678901234567890123456789

      hola         7         a  1.230000

hola      7         a         1.230000
```

Fig. G.15 | Justificación a la derecha y justificación a la izquierda de los valores.

```
1  // Fig. G.16: PruebaBanderaMas.java
2  // Impresión de números con y sin la bandera +.
3
4  public class PruebaBanderaMas
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf( "%d\t%d\n", 786, -786 );
9        System.out.printf( "%+d\t%+d\n", 786, -786 );
10    } // fin de main
11 } // fin de la clase PruebaBanderaMas
```

```
786     -786
+786    -786
```

Fig. G.16 | Impresión de números con y sin la bandera +.

En la figura G.17 se antepone un espacio al número positivo mediante la **bandera de espacio**. Esto es útil para alinear números positivos y negativos con el mismo número de dígitos. El valor -547 no va precedido por un espacio en la salida, debido a su signo negativo. En la figura G.18 se utiliza la **bandera #** para anteponer un 0 al valor octal, y 0x para el valor hexadecimal.

```
1  // Fig. G.17: PruebaBanderaEspacio.java
2  // Impresión de un espacio antes de valores no negativos.
3
```

Fig. G.17 | Impresión de un espacio antes de valores no negativos (parte 1 de 2).

```
4  public class PruebaBanderaEspacio
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf( "% d\n% d\n", 547, -547 );
9     } // fin de main
10 } // fin de la clase PruebaBanderaEspacio
```

```
 547
-547
```

Fig. G.17 | Impresión de un espacio antes de valores no negativos (parte 2 de 2).

```
1  // Fig. G.18: PruebaBanderaLibras.java
2  // Uso de la bandera # con los caracteres de conversión o y x.
3
4  public class PruebaBanderaLibras
5  {
6     public static void main( String[] args )
7     {
8        int c = 31;       // inicializa c
9
10       System.out.printf( "%#o\n", c );
11       System.out.printf( "%#x\n", c );
12    } // fin de main
13 } // fin de la clase PruebaBanderaLibras
```

```
037
0x1f
```

Fig. G.18 | Uso de la bandera # con los caracteres de conversión o y x.

En la figura G.19 se combinan la bandera +, la **bandera 0** y la bandera de espacio para imprimir 452 en un campo con una anchura de 9, con un signo + y ceros a la izquierda; después se imprime 452 en un campo de anchura 9, usando sólo la bandera 0, y al final se imprime 452 en un campo de anchura 9, usando sólo la bandera de espacio.

```
1  // Fig. G.19: PruebaBanderaCero.java
2  // Impresión con la bandera 0 (cero) para rellenar con ceros a la izquierda.
3
4  public class PruebaBanderaCero
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf( "%+09d\n", 452 );
9        System.out.printf( "%09d\n", 452 );
10       System.out.printf( "% 9d\n", 452 );
11    } // fin de main
12 } // fin de la clase PruebaBanderaCero
```

Fig. G.19 | Impresión con la bandera 0 (cero) para rellenar con ceros a la izquierda (parte 1 de 2).

```
+00000452
000000452
     452
```

Fig. G.19 | Impresión con la bandera 0 (cero) para rellenar con ceros a la izquierda (parte 2 de 2).

En la figura G.20 se utiliza la bandera de coma (,) para mostrar un número decimal y un número de punto flotante con el separador de miles. En la figura G.21 se encierran números negativos entre paréntesis, usando la bandera (. El valor 50 no se encierra entre paréntesis en la salida, ya que es un número positivo.

```
1  // Fig. G.20: PruebaBanderaComa.java
2  // Uso de la bandera de coma (,) para mostrar números con el separador de miles.
3
4  public class PruebaBanderaComa
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf( "%,d\n", 58625 );
9        System.out.printf( "%,.2f\n", 58625.21 );
10       System.out.printf( "%,.2f", 12345678.9 );
11    } // fin de main
12 } // fin de la clase PruebaBanderaComa
```

```
58,625
58,625.21
12,345,678.90
```

Fig. G.20 | Uso de la bandera de coma (,) para mostrar números con el separador de miles.

```
1  // Fig. G.21: PruebaBanderaParentesis.java
2  // Uso de la bandera ( para colocar paréntesis alrededor de números negativos.
3
4  public class PruebaBanderaParentesis
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf( "%(d\n", 50 );
9        System.out.printf( "%(d\n", -50 );
10       System.out.printf( "%(.1e\n", -50.0 );
11    } // fin de main
12 } // fin de la clase PruebaBanderaParentesis
```

```
50
(50)
(5.0e+01)
```

Fig. G.21 | Uso de la bandera (para colocar paréntesis alrededor de números negativos.

G.11 Impresión con índices como argumentos

Un **índice como argumento** es un entero opcional, seguido de un signo de $, el cual indica la posición del argumento en la lista de argumentos. Por ejemplo, en las líneas 20 a 22 y 25 a 26 de la figura G.9 se utiliza el índice como argumento "1$" para indicar que todos los especificadores de formato utilizan el primer argumento en la lista de argumentos. Los índices como argumentos permiten a los programadores reordenar la salida, de manera que los argumentos en la lista de argumentos no necesariamente se encuentren en el orden de sus especificadores de formato correspondientes. Los índices como argumentos también ayudan a evitar argumentos duplicados. En la figura G.22 se imprimen argumentos en la lista de argumentos en orden inverso, mediante el uso del índice como argumento.

```
1  // Fig. G.22: PruebaIndiceArgumento
2  // Reordenamiento de la salida con los índices como argumentos.
3
4  public class PruebaIndiceArgumento
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf(
9           "Lista de parametros sin reordenar: %s %s %s %s\n",
10          "primero", "segundo", "tercero", "cuarto" );
11       System.out.printf(
12          "Lista de parametros despues de reordenar: %4$s %3$s %2$s %1$s\n",
13          "primero", "segundo", "tercero", "cuarto" );
14    } // fin de main
15 } // fin de la clase PruebaIndiceArgumento
```

```
Lista de parametros sin reordenar: primero segundo tercero cuarto
Lista de parametros despues de reordenar: cuarto tercero segundo primero
```

Fig. G.22 | Reordenamiento de la salida con índices como argumentos.

G.12 Impresión de literales y secuencias de escape

La mayoría de los caracteres literales que se imprimen en una instrucción `printf` sólo necesitan incluirse en la cadena de formato. Sin embargo, hay varios caracteres "problemáticos", como el signo de comillas dobles (") que delimita a la cadena de formato en sí. Varios caracteres de control, como el carácter de nueva línea y el de tabulación, deben representarse mediante secuencias de escape. Una secuencia de escape se representa mediante una barra diagonal inversa (\), seguida de un carácter de escape. En la figura G.23 se enumeran las secuencias de escape y las acciones que producen.

Secuencia de escape	Descripción
\' (comilla sencilla)	Imprime el carácter de comilla sencilla (').
\" (doble comilla)	Imprime el carácter de doble comilla (").
\\ (barra diagonal inversa)	Imprime el carácter de barra diagonal inversa (\).
\b (retroceso)	Desplaza el cursor una posición hacia atrás en la línea actual.

Fig. G.23 | Secuencias de escape (parte 1 de 2).

Secuencia de escape	Descripción
\f (nueva página o avance de página)	Desplaza el cursor al principio de la siguiente página lógica.
\n (nueva línea)	Desplaza el cursor al principio de la siguiente línea.
\r (retorno de carro)	Desplaza el cursor al principio de la línea actual.
\t (tabulador horizontal)	Desplaza el cursor hacia la siguiente posición del tabulador horizontal.

Fig. G.23 | Secuencias de escape (parte 2 de 2).

Error común de programación G.4

Tratar de imprimir un carácter de comillas dobles o un carácter de barra diagonal inversa como datos literales en una instrucción `printf`*, sin anteponer al carácter una barra diagonal inversa para formar una secuencia de escape apropiada, podría producir un error de sintaxis.*

G.13 Aplicación de formato a la salida con la clase Formatter

Hasta ahora, hemos visto cómo mostrar salida con formato en el flujo de salida estándar. ¿Qué deberíamos hacer si quisiéramos enviar salidas con formato a otros flujos de salida o dispositivos, como un objeto `JTextArea` o un archivo? La solución recae en la clase `Formatter` (en el paquete `java.util`), la cual proporciona las mismas herramientas de formato que `printf`. `Formatter` es una clase utilitaria que permite a los programadores imprimir datos con formato hacia un destino especificado, como un archivo en el disco. De manera predeterminada, un objeto `Formatter` crea una cadena en la memoria. En la figura G.24 se muestra cómo usar un objeto `Formatter` para crear una cadena con formato, la cual después se muestra en un cuadro de diálogo de mensaje.

En la línea 11 se crea un objeto `Formatter` mediante el uso del constructor predeterminado, por lo que este objeto creará una cadena en la memoria. Se incluyen otros constructores para que el programador pueda especificar el destino hacia el que se deben enviar los datos con formato. Para obtener más detalles, consulte la página `java.sun.com/javase/6/docs/api/java/util/Formatter.html`.

```
1  // Fig. G.24: PruebaFormatter.java
2  // Aplicar formato a la salida con la clase Formatter.
3  import java.util.Formatter;
4  import javax.swing.JOptionPane;
5
6  public class PruebaFormatter
7  {
8     public static void main( String[] args )
9     {
10       // crea un objeto Formatter y aplica formato a la salida
11       Formatter formatter = new Formatter();
12       formatter.format( "%d = %#o = %#X", 10, 10, 10 );
13
14       // muestra la salida en el componente JOptionPane
15       JOptionPane.showMessageDialog( null, formatter.toString() );
16    } // fin de main
17 } // fin de la clase PruebaFormatter
```

Fig. G.24 | Aplicar formato a la salida con la clase `Formatter` (parte 1 de 2).

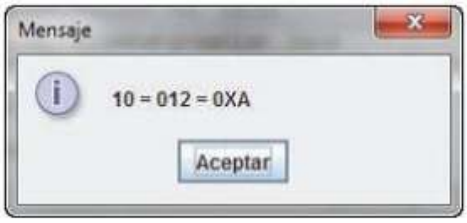


Fig. G.24 | Aplicar formato a la salida con la clase `Formatter` (parte 2 de 2).

En la línea 12 se invoca el método `format` para dar formato a la salida. Al igual que `printf`, el método `format` recibe una cadena de formato y una lista de argumentos. La diferencia es que `printf` envía la salida con formato directamente al flujo de salida estándar, mientras que `format` envía la salida con formato al destino especificado por su constructor (en este programa, una cadena en la memoria). En la línea 15 se invoca el método `toString` de `Formatter` para obtener los datos con formato como una cadena, que luego se muestra en un cuadro de diálogo de mensaje.

La clase `String` también proporciona un método de conveniencia `static` llamado `format`, el cual nos permite crear una cadena en la memoria, sin necesidad de crear primero un objeto `Formatter`. Podríamos haber sustituido las líneas 11 a 12 y la línea 15 de la figura G.24 por:

```
String s = String.format( "%d = %#o = %#x", 10, 10, 10 );
JOptionPane.showMessageDialog( null, s );
```

G.14 Conclusión

En este apéndice vimos un resumen acerca de cómo aplicar formato a la salida mediante los diversos caracteres y banderas de formato. Mostramos números decimales mediante el uso de los caracteres de formato `d`, `o`, `x` y `X`; mostramos números de punto flotante usando los caracteres de formato `e`, `E`, `f`, `g` y `G`; y mostramos tanto fechas como horas en diversos formatos, usando los caracteres de formato `t` y `T` junto con sus caracteres de sufijo de conversión. Aprendió a mostrar la salida con anchuras de campo y precisiones. Presentamos las banderas `+`, `-`, espacio, `#`, `0`, coma y `(` que se utilizan en conjunto con los caracteres de formato para producir la salida. También demostramos cómo aplicar formato a la salida con la clase `Formatter`.

Resumen

Sección G.2 Flujos

- Por lo general, las operaciones de entrada y salida se llevan a cabo con flujos, los cuales son secuencias de bytes.
- Por lo común, el flujo de entrada estándar se conecta al teclado, y el flujo de salida estándar se conecta a la pantalla de la computadora.

Sección G.3 Aplicación de formato a la salida con `printf`

- La cadena de formato de `printf` describe los formatos en los que aparecen los valores de salida. El especificador de formato consiste en un índice como argumento, banderas, anchuras de campo, precisiones y caracteres de conversión.

Sección G.4 Impresión de enteros

- Los enteros se imprimen con los caracteres de conversión `d` para enteros decimales, `o` para enteros en formato octal y `x` (o `X`) para los enteros en formato hexadecimal. `X` muestra letras mayúsculas.

Sección G.5 Impresión de números de punto flotante

- Los valores de punto flotante se imprimen con los caracteres de conversión `e` (o `E`) para la notación exponencial, `f` para la notación de punto flotante regular, y `g` (o `G`) para la notación `e` (o `E`) o `f`. Para el especificador de conversión `g`, se utiliza el carácter de conversión `e` si el valor es menor que 10^{-3}, o mayor o igual a 10^{7}; en caso contrario, se utiliza el carácter de conversión `f`.

Sección G.6 Impresión de cadenas y caracteres

- El carácter de conversión `c` imprime un carácter.
- El carácter de conversión `s` (o `S`) imprime una cadena de caracteres. El carácter de conversión `S` muestra la salida en letras mayúsculas.

Sección G.7 Impresión de fechas y horas

- El carácter de conversión `t` (o `T`), seguido de un carácter de sufijo de conversión, imprime la fecha y la hora en diversos formatos. El carácter de conversión `T` muestra la salida en letras mayúsculas.
- El carácter de conversión `t` (o `T`) requiere que el argumento sea de tipo `long`, `Long`, `Calendar` o `Date`.

Sección G.8 Otros caracteres de conversión

- El carácter de conversión `b` (o `B`) imprime la representación de cadena de un valor `boolean` o `Boolean`. Estos caracteres de conversión también imprimen `"true"` para las referencias que no son `null`, y `"false"` para las referencias `null`. El carácter de conversión `B` muestra la salida en letras mayúsculas.
- El carácter de conversión `h` (o `H`) devuelve `null` para una referencia `null`, y una representación `String` del valor de código hash (en base 16) del objeto. Los códigos de hash se utilizan para almacenar y obtener objetos que se encuentran en objetos `Hashtable` y `HashMap`. El carácter de conversión `H` muestra la salida en letras mayúsculas.
- El carácter de conversión `n` imprime el separador de línea específico de la plataforma.
- El carácter de conversión `%` se utiliza para mostrar un `%` literal.

Sección G.9 Impresión con anchuras de campo y precisiones

- Si la anchura de campo es mayor que el objeto a imprimir, éste se justifica a la derecha en el campo.
- Las anchuras de campo se pueden usar con todos los caracteres de conversión, excepto la conversión con el separador de línea.
- La precisión que se utiliza con los caracteres de conversión de punto flotante `e` y `f` indica el número de dígitos que aparecen después del punto decimal. La precisión que se utiliza con el carácter de conversión de punto flotante `g` indica el número de dígitos significativos que deben aparecer.
- La precisión que se utiliza con el carácter de conversión `s` indica el número de caracteres a imprimir.
- La anchura de campo y la precisión se pueden combinar, para lo cual se coloca la anchura de campo, seguida de un punto decimal, y luego de la precisión entre el signo de porcentaje y el carácter de conversión.

Sección G.10 Uso de banderas en la cadena de formato de `printf`

- La bandera `-` justifica a la izquierda su argumento en un campo.
- La bandera `+` imprime un signo más para los valores positivos, y un signo menos para los valores negativos.
- La bandera de espacio imprime un espacio antes de un valor positivo. La bandera de espacio y la bandera `+` no se pueden utilizar juntas en un carácter de conversión integral.
- La bandera `#` antepone un `0` a los valores octales, y `0x` a los valores hexadecimales.
- La bandera `0` imprime ceros a la izquierda para un valor que no ocupa todo su campo.
- La bandera de coma (`,`) utiliza el separador de miles específico de la configuración regional para mostrar números enteros y de punto flotante.
- La bandera `(` encierra un número negativo entre paréntesis.

Sección G.11 Impresión con índices como argumentos

- Un índice como argumento es un entero decimal opcional, seguido de un signo $ que indica la posición del argumento en la lista de argumentos.
- Los índices como argumentos permiten a los programadores reordenar la salida, de manera que los argumentos en la lista de argumentos no estén necesariamente en el orden de sus correspondientes especificadores de formato. Los índices como argumentos también ayudan a evitar los argumentos duplicados.

Sección G.13 Aplicación de formato a la salida con la clase `Formatter`

- La clase `Formatter` (en el paquete `java.util`) proporciona las mismas herramientas de formato que `printf`. `Formatter` es una clase utilitaria que permite a los programadores imprimir salida con formato hacia varios destinos, incluyendo componentes de GUI, archivos y otros flujos de salida.
- El método `format` de la clase `Formatter` imprime los datos con formato al destino especificado por el constructor de `Formatter`.
- El método `static format` de la clase `String` aplica formato a los datos y devuelve los datos con formato, como un objeto `String`.

Ejercicios de autoevaluación

G.1 Complete los enunciados:

a) Todas las operaciones de entrada y salida se manejan en forma de _____ .

b) El flujo _____ se conecta por lo general al teclado.

c) El flujo _____ se conecta por lo común a la pantalla de la computadora.

d) El método _____ de `System.out` se puede utilizar para aplicar formato al texto que se muestra en la salida estándar.

e) El carácter de conversión _____ puede utilizarse para imprimir en pantalla un entero decimal.

f) Los caracteres de conversión _____ y _____ se utilizan para mostrar enteros en formato octal y hexadecimal, respectivamente.

g) El carácter de conversión _____ se utiliza para mostrar un valor de punto flotante en notación exponencial.

h) Los caracteres de conversión `e` y `f` se muestran con _____ dígitos de precisión a la derecha del punto decimal, si no se especifica una precisión.

i) Los caracteres de conversión _____ y _____ se utilizan para imprimir cadenas y caracteres, respectivamente.

j) El carácter de conversión _____ y el carácter de sufijo de conversión _____ se utilizan para imprimir la hora para el reloj de 24 horas, como `hora:minuto:segundo`.

k) La bandera _____ hace que la salida se justifique a la izquierda en un campo.

l) La bandera _____ hace que los valores se muestren con un signo más o con un signo menos.

m) El índice como argumento _____ corresponde al segundo argumento en la lista de argumentos.

n) La clase _____ tiene la misma capacidad que `printf`, pero permite a los programadores imprimir salida con formato en varios destinos, además del flujo de salida estándar.

G.2 Encuentre el error en cada uno de los siguientes enunciados, y explique cómo se puede corregir.

a) La siguiente instrucción debe imprimir el carácter `'c'`.
```
System.out.printf( "%c\n", "c" );
```
b) La siguiente instrucción debe imprimir `9.375%`.
```
System.out.printf( "%.3f%", 9.375 );
```
c) La siguiente instrucción debe imprimir el tercer argumento en la lista de argumentos:
```
System.out.printf( "%2$s\n", "Lun", "Mar", "Mie", "Jue", "Vie" );
```
d)
```
System.out.printf( ""Una cadena entre comillas"" );
```
e)
```
System.out.printf( %d %d, 12, 20 );
```
f)
```
System.out.printf( "%s\n", 'Richard' );
```

G.3 Escriba una instrucción para cada uno de los siguientes casos:
a) Imprimir `1234` justificado a la derecha, en un campo de 10 dígitos.
b) Imprimir `123.456789` en notación exponencial con un signo (+ o -) y 3 dígitos de precisión.
c) Imprimir `100` en formato octal, precedido por `0`.
d) Dado un objeto calendario de la clase `Calendar`, imprima una fecha con formato de mes/día/año (cada uno con dos dígitos).
e) Dado un objeto `Calendar` llamado `calendario`, imprimir una hora para el reloj de 24 horas como `hora:minuto:segundo` (cada uno con dos dígitos), usando un índice como argumento y caracteres de sufijo de conversión para aplicar formato a la hora.
f) Imprimir `3.333333` con un signo (+ o -) en un campo de 20 caracteres, con una precisión de 3.

Respuestas a los ejercicios de autoevaluación

G.1 a) Flujos. b) de entrada estándar. c) de salida estándar. d) `printf`. e) `d`. f) `o`, `x` o `X`. g) `e` o `E`. h) 6. i) `s` o `S`, `c` o `C`. j) `t`, `T`. k) - (menos). l) + (más). m) `2$`. n) `Formatter`.

G.2 a) Error: el carácter de conversión `c` espera un argumento del tipo primitivo `char`.
Corrección: para imprimir el carácter `'c'`, cambie "`c`" a `'c'`.
b) Error: está tratando de imprimir el carácter literal `%` sin usar el especificador de formato `%%`.
Corrección: use `%%` para imprimir un carácter `%` literal.
c) Error: el índice como argumento no empieza con 0; por ejemplo, el primer argumento es `1$`.
Corrección: para imprimir el tercer argumento, use `3$`.
d) Error: está tratando de imprimir el carácter literal " sin usar la secuencia de escape `\"`.
Corrección: sustituya cada comilla en el conjunto interno de comillas con `\"`.
e) Error: la cadena de formato no va encerrada entre comillas dobles.
Corrección: encierre `%d %d` entre comillas dobles.
f) Error: la cadena a imprimir está encerrada entre comillas sencillas.
Corrección: use dobles comillas en vez de comillas sencillas para representar una cadena.

G.3 a)
```
System.out.printf( "%10d\n", 1234 );
```
b)
```
System.out.printf( "%+.3e\n", 123.456789 );
```
c)
```
System.out.printf( "%#o\n", 100 );
```
d)
```
System.out.printf( "%tD\n", calendario );
```
e)
```
System.out.printf( "%1$tH:%1$tM:%1$tS\n", calendario );
```
f)
```
System.out.printf( "%+20.3f\n", 3.333333 );
```

Ejercicios

G.4 Escriba una o más instrucciones para cada uno de los siguientes casos:
a) Imprimir el entero `40000` justificado a la derecha en un campo de 15 dígitos.
b) Imprimir `200` con y sin un signo.
c) Imprimir `100` en formato hexadecimal, precedido por `0x`.
d) Imprimir `1.234` con tres dígitos de precisión en un campo de 9 dígitos con ceros a la izquierda.

G.5 Muestre lo que se imprime en cada una de las siguientes instrucciones. Si una instrucción es incorrecta, indique por qué.
a)
```
System.out.printf( "%-10d\n", 10000 );
```
b)
```
System.out.printf( "%c\n", "Esta es una cadena" );
```
c)
```
System.out.printf( "%8.3f\n", 1024.987654 );
```
d)
```
System.out.printf( "%#o\n%#X\n", 17, 17 );
```
e)
```
System.out.printf( "% d\n%+d\n", 1000000, 1000000 );
```
f)
```
System.out.printf( "%10.2e\n", 444.93738 );
```
g)
```
System.out.printf( "%d\n", 10.987 );
```

G.6 Encuentre el(los) error(es) en cada uno de los siguientes segmentos de programa. Muestre la instrucción corregida.

a) `System.out.printf( "%s\n", 'Feliz cumpleanios' );`

b) `System.out.printf( "%c\n", 'Hola' );`

c) `System.out.printf( "%c\n", "Esta es una cadena" );`

d) La siguiente instrucción debe imprimir "Buen viaje" con las dobles comillas:
`System.out.printf( ""%s"", "Buen viaje" );`

e) La siguiente instrucción debe imprimir "Hoy es viernes":
`System.out.printf( "Hoy es %s\n", "Lunes", "Viernes" );`

f) `System.out.printf( 'Escriba su nombre: ' );`

g) `System.out.printf( %f, 123.456 );`

h) La siguiente instrucción debe imprimir la hora actual en el formato "hh:mm:ss":
```
Calendar fechaHora = Calendar.getInstance();
System.out.printf( "%1$tk:1$%tl:%1$tS\n", fechaHora );
```

G.7 ***(Impresión de fechas y horas)*** Escriba un programa que imprima fechas y horas en los siguientes formatos:

```
GMT-05:00 04/30/04 09:55:09 AM
GMT-05:00 Abril 30 2004 09:55:09
2004-04-30 dia-del-mes:30
2004-04-30 dia-del-anio:121
Vie Abr 30 09:55:09 GMT-05:00 2004
```

[*Nota:* dependiendo de su ubicación, tal vez tenga una zona horaria distinta de GMT-05:00].

G.8 Escriba un programa para probar los resultados de imprimir el valor entero `12345` y el valor de punto flotante `1.2345` en campos de varios tamaños.

G.9 ***(Redondeo de números)*** Escriba un programa que imprima el valor `100.453267` redondeado al dígito, a la decena, centena, múltiplo de mil y de diez mil más cercanos.

G.10 Escriba un programa que reciba como entrada una palabra del teclado y determine su longitud. Imprima la palabra usando el doble de la longitud como anchura de campo.

G.11 ***(Conversión de temperatura en grados Fahrenheit a Centígrados)*** Escriba un programa que convierta temperaturas enteras en grados Fahrenheit de `0` a `212` grados, a temperaturas en grados centígrados de punto flotante con tres dígitos de precisión. Use la siguiente fórmula:

```
centigrados = 5.0 / 9.0 * ( fahrenheit - 32 );
```

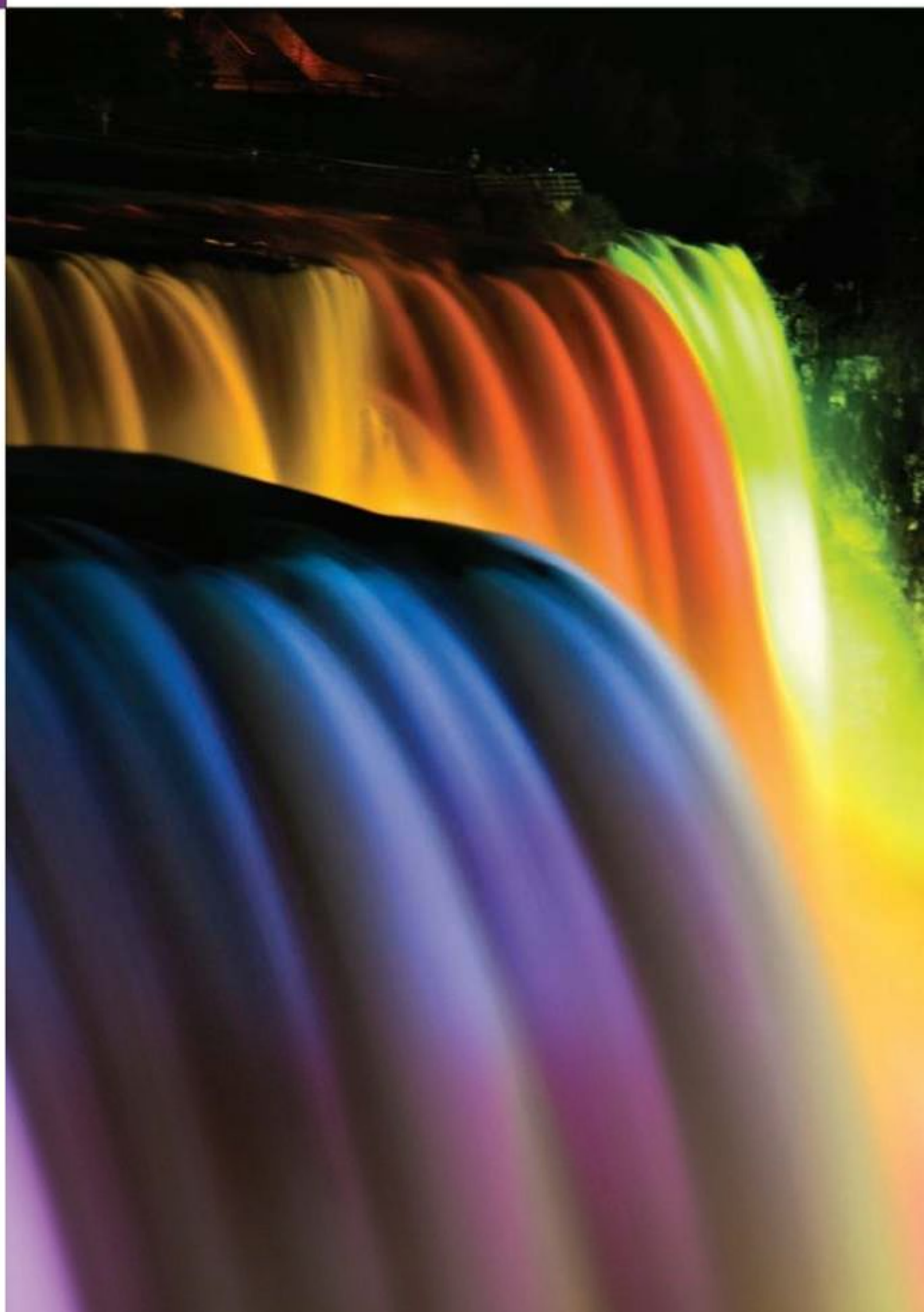
para realizar el cálculo. La salida debe imprimirse en dos columnas justificadas a la derecha de 10 caracteres cada una, y las temperaturas en grados centígrados deben ir precedidas por un signo, tanto para los valores positivos como para los negativos.

G.12 Escriba un programa para probar todas las secuencias de escape en la figura G.23. Para las secuencias de escape que desplazan el cursor, imprima un carácter antes y después de la secuencia de escape, de manera que se pueda ver con claridad a dónde se ha desplazado el cursor.

G.13 Escriba un programa que utilice el carácter de conversión `g` para imprimir el valor `9876.12345`. Imprima el valor con precisiones que varíen de 1 a 9.

H

Sistemas numéricos



Objetivos

En este apéndice aprenderá a:

- Comprender los conceptos básicos acerca de los sistemas numéricos, como base, valor posicional y valor simbólico.
- Trabajar con los números representados en los sistemas numéricos binario, octal y hexadecimal.
- Abreviar los números binarios como octales o hexadecimales.
- Convertir los números octales y hexadecimales en binarios.
- Realizar conversiones hacia y desde números decimales y sus equivalentes en binario, octal y hexadecimal.
- Comprender el funcionamiento de la aritmética binaria y la manera en que se representan los números binarios negativos, utilizando la notación de complemento a dos.

Plan general

H.1 Introducción

En este apéndice presentaremos los sistemas numéricos clave que utilizan los programadores de Java, en especial cuando trabajan en proyectos de software que requieren de una estrecha interacción con el hardware a nivel de máquina. Entre los proyectos de este tipo están los sistemas operativos, el software de redes computacionales, los compiladores, sistemas de bases de datos y aplicaciones que requieren de un alto rendimiento.

Cuando escribimos un entero, como 227 o –63, en un programa de Java, se asume que el número está en el sistema numérico decimal (base 10). Los dígitos en el sistema numérico decimal son 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. El dígito más bajo es el 0 y el más alto es el 9; uno menos que la base, 10. En su interior, las computadoras utilizan el sistema numérico binario (base 2). Este sistema numérico sólo tiene dos dígitos: 0 y 1. El dígito más bajo es el 0 y el más alto es el 1 (uno menos que la base, 2).

Como veremos, los números binarios tienden a ser mucho más extensos que sus equivalentes decimales. Los programadores que trabajan con lenguajes ensambladores y en lenguajes de alto nivel como Java, que les permiten llegar hasta el nivel de máquina, encuentran que es complicado trabajar con números binarios. Por eso existen otros dos sistemas numéricos: el sistema numérico octal (base 8) y el sistema numérico hexadecimal (base 16), que son populares debido a que permiten abreviar los números binarios de una manera conveniente.

En el sistema numérico octal, los dígitos varían del 0 al 7. Debido a que tanto el sistema numérico binario como el octal tienen menos dígitos que el sistema numérico decimal, sus dígitos son los mismos que sus correspondientes en decimal.

El sistema numérico hexadecimal presenta un problema, ya que requiere de 16 dígitos: el dígito más bajo es 0 y el más alto tiene un valor equivalente al 15 decimal (uno menos que la base, 16). Por convención utilizamos las letras de la A a la F para representar los dígitos hexadecimales que corresponden a los valores decimales del 10 al 15. Por lo tanto, en hexadecimal podemos tener números como el 876, que consisten sólo de dígitos similares a los decimales; números como 8A55F que consisten de dígitos y letras; y números como FFE que consisten sólo de letras. En ocasiones, un número hexadecimal puede coincidir con una palabra común como FACE o FEED (en inglés); esto puede parecer extraño para los programadores acostumbrados a trabajar con números. Los dígitos de los sistemas numéricos binario, octal, decimal y hexadecimal se sintetizan en las figuras H.1 y H.2.

Cada uno de estos sistemas numéricos utilizan la notación posicional: cada posición en la que se escribe un dígito tiene un valor posicional distinto. Por ejemplo, en el número decimal 937 (el 9, el 3 y el 7 se conocen como valores simbólicos) decimos que el 7 se escribe en la posición de las unidades; el 3, en la de las decenas; y el 9, en la de las centenas. Cada una de estas posiciones es una potencia de la base (10), y estas potencias empiezan en 0 y aumentan de 1 en 1, a medida que nos desplazamos hacia la izquierda por el número (figura H.3).

Dígito binario	Dígito octal	Dígito decimal	Dígito hexadecimal
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8
		9	9
			A (valor de 10 en decimal)
			B (valor de 11 en decimal)
			C (valor de 12 en decimal)
			D (valor de 13 en decimal)
			E (valor de 14 en decimal)
			F (valor de 15 en decimal)

Fig. H.1 | Dígitos de los sistemas numéricos binario, octal, decimal y hexadecimal.

Atributo	Binario	Octal	Decimal	Hexadecimal
Base	2	8	10	16
Dígito más bajo	0	0	0	0
Dígito más alto	1	7	9	F

Fig. H.2 | Comparación de los sistemas binario, octal, decimal y hexadecimal.

Valores posicionales en el sistema numérico decimal			
Dígito decimal	9	3	7
Nombre de la posición	Centenas	Decenas	Unidades
Valor posicional	100	10	1
Valor posicional como potencia de la base (10)	10^2	10^1	10^0

Fig. H.3 | Valores posicionales en el sistema numérico decimal.

Para números decimales más extensos, las siguientes posiciones a la izquierda serían: de millares (10 a la tercera potencia), de decenas de millares (10 a la cuarta potencia), de centenas de millares (10 a la quinta potencia), de los millones (10 a la sexta potencia), de decenas de millones (10 a la séptima potencia), y así en lo sucesivo.

En el número binario 101, el 1 más a la derecha se escribe en la posición de los unos, el 0 se escribe en la posición de los dos y el 1 de más a la izquierda se escribe en la posición de los cuatros. Cada una de estas posiciones es una potencia de la base (2), y estas potencias empiezan en 0 y aumentan de 1 en 1, a medida que nos desplazamos hacia la izquierda por el número (figura H.4). Por lo tanto, $101 = 2^2 + 2^0 = 4 + 1 = 5$.

Valores posicionales en el sistema numérico binario			
Dígito binario	1	0	1
Nombre de la posición	Cuatros	Dos	Unos
Valor posicional	4	2	1
Valor posicional como potencia de la base (2)	2^2	2^1	2^0

Fig. H.4 | Valores posicionales en el sistema numérico binario.

Para números binarios más extensos, las siguientes posiciones a la izquierda serían la posición de los ochos (2 a la tercera potencia), la posición de los dieciséis (2 a la cuarta potencia), la posición de los treinta y dos (2 a la quinta potencia), la posición de los sesenta y cuatros (2 a la sexta potencia), y así en lo sucesivo.

En el número octal 425, decimos que el 5 se escribe en la posición de los unos, el 2 se escribe en la posición de los ochos y el 4 se escribe en la posición de los sesenta y cuatros. Cada una de estas posiciones es una potencia de la base (8), y estas potencias empiezan en 0 y aumentan de 1 en 1, a medida que nos desplazamos hacia la izquierda por el número (figura H.5).

Valores posicionales en el sistema numérico octal			
Dígito octal	4	2	5
Nombre de la posición	Sesenta y cuatros	Ochos	Unos
Valor posicional	64	8	1
Valor posicional como potencia de la base (8)	8^2	8^1	8^0

Fig. H.5 | Valores posicionales en el sistema numérico octal.

Para números octales más extensos, las siguientes posiciones a la izquierda serían: la posición de los quinientos doces (8 a la tercera potencia), la posición de los cuatro mil noventa y seis (8 a la cuarta potencia), la posición de los treinta y dos mil setecientos sesenta y ochos (8 a la quinta potencia), y así en lo sucesivo.

En el número hexadecimal 3DA, decimos que la A se escribe en la posición de los unos, la D se escribe en la posición de los dieciséis y el 3 se escribe en la posición de los doscientos cincuenta y seis. Cada una de estas posiciones es una potencia de la base (16), y estas potencias empiezan en 0 y aumentan de 1 en 1, a medida que nos desplazamos hacia la izquierda por el número (figura H.6).

Para números hexadecimales más extensos, las siguientes posiciones a la izquierda serían: la posición de los cuatro mil noventa y seis (16 a la tercera potencia), la posición de los sesenta y cinco mil quinientos treinta y seis (16 a la cuarta potencia), y así en lo sucesivo.

Valores posicionales en el sistema numérico hexadecimal			
Dígito hexadecimal	3	D	A
Nombre de la posición	Doscientos cincuenta y seis	Dieciséis	Unos
Valor posicional	256	16	1
Valor posicional como potencia de la base (16)	16^2	16^1	16^0

Fig. H.6 | Valores posicionales en el sistema numérico hexadecimal.

H.2 Abreviatura de los números binarios como números octales y hexadecimales

En computación, el uso principal de los números octales y hexadecimales es para abreviar representaciones binarias demasiado extensas. La figura H.7 muestra que los números binarios extensos pueden expresarse en forma más concisa en los sistemas numéricos, con bases mayores que en el sistema numérico binario.

Número decimal	Representación binaria	Representación octal	Representación hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Fig. H.7 | Equivalentes en decimal, binario, octal y hexadecimal.

Una relación muy importante que tienen tanto el sistema numérico octal como el hexadecimal con el sistema binario es que las bases de los sistemas octal y hexadecimal (8 y 16, respectivamente) son

potencias de la base del sistema numérico binario (base 2). Considere el siguiente número binario de 12 dígitos y sus equivalentes en octal y hexadecimal. Vea si puede determinar cómo esta relación hace que sea conveniente el abreviar los números binarios en octal o hexadecimal. La respuesta sigue después de los números.

Número binario	Equivalente en octal	Equivalente en hexadecimal
100011010001	4321	8D1

Para ver cómo el número binario se convierte con facilidad en octal, sólo divida el número binario de 12 dígitos en grupos de tres bits consecutivos y escriba esos grupos por encima de los dígitos correspondientes del número octal, como se muestra a continuación:

```
100     011     010     001
4       3       2       1
```

El dígito octal que escribió debajo de cada grupo de tres bits corresponde precisamente al equivalente octal de ese número binario de 3 dígitos, como se muestra en la figura H.7.

El mismo tipo de relación puede observarse al convertir números de binario a hexadecimal. Divida el número binario de 12 dígitos en grupos de cuatro bits consecutivos y escriba esos grupos por encima de los dígitos correspondientes del número hexadecimal, como se muestra a continuación:

```
1000    1101    0001
8       D       1
```

Observe que el dígito hexadecimal que escribió debajo de cada grupo de cuatro bits corresponde precisamente al equivalente hexadecimal de ese número binario de 4 dígitos que se muestra en la figura H.7.

H.3 Conversión de números octales y hexadecimales a binarios

En la sección anterior vimos cómo convertir números binarios a sus equivalentes en octal y hexadecimal, formando grupos de dígitos binarios y simplemente volviéndolos a escribir como sus valores equivalentes en dígitos octales o hexadecimales. Este proceso puede utilizarse en forma inversa para producir el equivalente en binario de un número octal o hexadecimal específico.

Por ejemplo, para convertir el número octal 653 en binario sólo hay que escribir el 6 como su equivalente binario de 3 dígitos 110, el 5 como su equivalente binario de 3 dígitos 101 y el 3 como su equivalente binario de 3 dígitos 011, para formar el número binario de 9 dígitos 110101011.

Para convertir el número hexadecimal FAD5 en binario, sólo hay que escribir la F como su equivalente binario de 4 dígitos 1111, la A como su equivalente binario de 4 dígitos 1010, la D como su equivalente binario de 4 dígitos 1101 y el 5 como su equivalente binario de 4 dígitos 0101, para formar el número binario de 16 dígitos 1111101011010101.

H.4 Conversión de un número binario, octal o hexadecimal a decimal

Como estamos acostumbrados a trabajar con el sistema decimal, a menudo es conveniente convertir un número binario, octal o hexadecimal en decimal para tener una idea de lo que "realmente" vale el número. Nuestros diagramas en la sección H.1 expresan los valores posicionales en decimal. Para convertir un número en decimal desde otra base, multiplique el equivalente en decimal de cada dígito por su

valor posicional y sume estos productos. Por ejemplo, el número binario 110101 se convierte en el número 53 decimal, como se muestra en la figura H.8.

Conversión de un número binario en decimal						
Valores posicionales:	32	16	8	4	2	1
Valores simbólicos:	1	1	0	1	0	1
Productos:	1*32=32	1*16=16	0*8=0	1*4=4	0*2=0	1*1=1
Suma:	= 32 + 16 + 0 + 4 + 0s + 1 = 53					

Fig. H.8 | Conversión de un número binario en decimal.

Para convertir el número 7614 octal en el número 3980 decimal utilizamos la misma técnica, esta vez utilizando los valores posicionales apropiados para el sistema octal, como se muestra en la figura H.9.

Conversión de un número octal en decimal				
Valores posicionales:	512	64	8	1
Valores simbólicos:	7	6	1	4
Productos	7*512=3584	6*64=384	1*8=8	4*1=4
Suma:	= 3584 + 384 + 8 + 4 = 3980			

Fig. H.9 | Conversión de un número octal en decimal.

Para convertir el número AD3B hexadecimal en el número 44347 decimal utilizamos la misma técnica, esta vez empleando los valores posicionales apropiados para el sistema hexadecimal, como se muestra en la figura H.10.

Conversión de un número hexadecimal en decimal				
Valores posicionales:	4096	256	16	1
Valores simbólicos:	A	D	3	B
Productos	A*4096=40960	D*256=3328	3*16=48	B*1=11
Suma:	= 40960 + 3328 + 48 + 11 = 44347			

Fig. H.10 | Conversión de un número hexadecimal en decimal.

H.5 Conversión de un número decimal a binario, octal o hexadecimal

Las conversiones de la sección H.4 siguen de manera natural las convenciones de la notación posicional. Las conversiones de decimal a binario, octal o hexadecimal también siguen estas convenciones.

Suponga que queremos convertir el número 57 decimal en binario. Empezamos escribiendo los valores posicionales de las columnas de derecha a izquierda, hasta llegar a una columna cuyo valor posicional sea mayor que el número decimal. Como no necesitamos esa columna, podemos descartarla. Por lo tanto, primero escribimos:

Valores posicionales:	64	32	16	8	4	2	1

Luego descartamos la columna con el valor posicional de 64, dejando:

Valores posicionales:	32	16	8	4	2	1

A continuación, empezamos a trabajar desde la columna más a la izquierda y nos vamos desplazando hacia la derecha. Dividimos 57 entre 32 y observamos que hay un 32 en 57, con un residuo de 25, por lo que escribimos 1 en la columna de los 32. Dividimos 25 entre 16 y observamos que hay un 16 en 25, con un residuo de 9, por lo que escribimos 1 en la columna de los 16. Dividimos 9 entre 8 y observamos que hay un 8 en 9 con un residuo de 1. Las siguientes dos columnas producen el cociente de cero cuando se divide 1 entre sus valores posicionales, por lo que escribimos 0 en las columnas de los 4 y de los 2. Por último, 1 entre 1 es 1, por lo que escribimos 1 en la columna de los 1. Esto nos da:

Valores posicionales:	32	16	8	4	2	1
Valores simbólicos:	1	1	1	0	0	1

y por lo tanto, el 57 decimal es equivalente al 111001 binario.

Para convertir el número decimal 103 en octal, empezamos por escribir los valores posicionales de las columnas hasta llegar a una columna cuyo valor posicional sea mayor que el número decimal. Como no necesitamos esa columna, podemos descartarla. Por lo tanto, primero escribimos:

Valores posicionales:	512	64	8	1

Luego descartamos la columna con el valor posicional de 512, lo que nos da:

Valores posicionales:	64	8	1

A continuación, empezamos a trabajar desde la columna más a la izquierda y nos vamos desplazando hacia la derecha. Dividimos 103 entre 64 y observamos que hay un 64 en 103 con un residuo de 39, por lo que escribimos 1 en la columna de los 64. Dividimos 39 entre 8 y observamos que el 8 cabe cuatro veces en 39 con un residuo de 7, por lo que escribimos 4 en la columna de los 8. Por último, dividimos 7 entre 1 y observamos que el 1 cabe siete veces en 7 y no hay residuo, por lo que escribimos 7 en la columna de los 1. Esto nos da:

Valores posicionales:	64	8	1
Valores simbólicos:	1	4	7

y por lo tanto, el 103 decimal es equivalente al 147 octal.

Para convertir el número decimal 375 en hexadecimal, empezamos por escribir los valores posicionales de las columnas hasta llegar a una columna cuyo valor posicional sea mayor que el número decimal. Como no necesitamos esa columna, podemos descartarla. Por consecuencia, primero escribimos:

Valores posicionales:	4096	256	16	1

Luego descartamos la columna con el valor posicional de 4096, lo que nos da:

Valores posicionales:	256	16	1

A continuación, empezamos a trabajar desde la columna más a la izquierda y nos vamos desplazando hacia la derecha. Dividimos 375 entre 256 y observamos que 256 cabe una vez en 375 con un residuo de 119, por lo que escribimos 1 en la columna de los 256. Dividimos 119 entre 16 y observamos que el 16 cabe siete veces en 119 con un residuo de 7, por lo que escribimos 7 en la columna de los 16.

Por último, dividimos 7 entre 1 y observamos que el 1 cabe siete veces en 7 y no hay residuo, por lo que escribimos 7 en la columna de los 1. Esto produce:

```
Valores posicionales:   256    16    1
Valores simbólicos:     1      7     7
```

y por lo tanto, el 375 decimal es equivalente al 177 hexadecimal.

H.6 Números binarios negativos: notación de complemento a dos

La discusión en este apéndice se ha enfocado hasta ahora en números positivos. En esta sección explicaremos cómo las computadoras representan números negativos mediante el uso de la *notación de complementos a dos*. Primero explicaremos cómo se forma el complemento a dos de un número binario y después mostraremos por qué representa el valor negativo de dicho número binario.

Considere una máquina con enteros de 32 bits. Suponga que se ejecuta la siguiente instrucción:

```
int valor = 13;
```

La representación en 32 bits de `valor` es:

```
00000000 00000000 00000000 00001101
```

Para formar el negativo de `valor`, primero formamos su *complemento a uno* aplicando el operador de complemento a nivel de bits de Java (~):

```
complementoAUnoDeValor = ~valor;
```

Internamente, `~valor` es ahora valor con cada uno de sus bits invertidos; los unos se convierten en ceros y los ceros en unos, como se muestra a continuación:

```
valor:
00000000 00000000 00000000 00001101
~valor (es decir, el complemento a uno de valor):
11111111 11111111 11111111 11110010
```

Para formar el complemento a dos de `valor`, simplemente sumamos uno al complemento a uno de `valor`. Por lo tanto:

```
El complemento a dos de valor es:
11111111 11111111 11111111 11110011
```

Ahora, si esto de hecho es igual a –13, deberíamos poder sumarlo al 13 binario y obtener como resultado 0. Comprobemos esto:

```
 00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
-----------------------------------
 00000000 00000000 00000000 00000000
```

El bit de acarreo que sale de la columna que está más a la izquierda se descarta y, sin duda, obtenemos 0 como resultado. Si sumamos el complemento a uno de un número a ese mismo número, todos los dígitos del resultado serían iguales a 1. La clave para obtener un resultado en el que todos los dígitos sean cero es que el complemento a dos es uno más que el complemento a uno. La suma de 1 hace que el resultado de cada columna sea 0 y se acarrea un 1. El acarreo sigue desplazándose hacia la izquierda hasta que se descarta en el bit que está más a la izquierda, con lo que todos los dígitos del número resultante son iguales a cero.

En realidad, las computadoras realizan una suma como:

```
x = a - valor;
```

mediante la suma del complemento a dos de `valor` con `a`, como se muestra a continuación:

```
x = a + (~valor + 1);
```

Suponga que a es 27 y que `valor` es 13 como en el ejemplo anterior. Si el complemento a dos de `valor` es en realidad el negativo de éste, entonces al sumar el complemento de dos de `valor` con a se produciría el resultado de 14. Comprobemos esto:

```
a (es decir 27)      00000000 00000000 00000000 00011011
+(~valor + 1)       +11111111 11111111 11111111 11110011
                    -----------------------------------
                     00000000 00000000 00000000 00001110
```

lo que ciertamente da como resultado 14.

Resumen

- Cuando escribimos un entero como 19, 227 o –63 en un programa de Java, suponemos que el número se encuentra en el sistema numérico decimal (base 10). Los dígitos en el sistema numérico decimal son 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. El dígito más bajo es el 0 y el más alto es el 9 (uno menos que la base, 10).
- Las computadoras utilizan el sistema numérico binario (base 2). Este sistema numérico sólo tiene dos dígitos: 0 y 1. El dígito más bajo es el 0 y el más alto es el 1 (uno menos que la base, 2).
- El sistema numérico octal (base 8) y el sistema numérico hexadecimal (base 16) son populares en gran parte debido a que permiten abreviar los números binarios de una manera conveniente.
- Los dígitos que se utilizan en el sistema numérico octal son del 0 al 7.
- El sistema numérico hexadecimal presenta un problema, ya que requiere de 16 dígitos: el dígito más bajo es 0 y el más alto tiene un valor equivalente al 15 decimal (uno menos que la base, 16). Por convención utilizamos las letras de la A a la F para representar los dígitos hexadecimales que corresponden a los valores decimales del 10 al 15.
- Cada uno de estos sistemas numéricos utilizan la notación posicional: cada posición en la que se escribe un dígito tiene un distinto valor posicional.
- Una relación muy importante que tienen tanto el sistema numérico octal como el hexadecimal con el sistema binario es que las bases de los sistemas octal y hexadecimal (8 y 16, respectivamente) son potencias de la base del sistema numérico binario (base 2).
- Para convertir un número octal en binario, sustituya cada dígito octal con su equivalente binario de tres dígitos.
- Para convertir un número hexadecimal en binario, simplemente sustituya cada dígito hexadecimal con su equivalente binario de cuatro dígitos.
- Como estamos acostumbrados a trabajar con el sistema decimal, es conveniente convertir un número binario, octal o hexadecimal en decimal para tener una idea de lo que "realmente" vale el número.
- Para convertir un número en decimal desde otra base, multiplique el equivalente en decimal de cada dígito por su valor posicional y sume estos productos.
- Las computadoras representan números negativos mediante el uso de la notación de complementos a dos.
- Para formar el negativo de un valor en binario, primero formamos su complemento a uno aplicando el operador de complemento a nivel de bits de Java (~). Esto invierte los bits del valor. Para formar el complemento a dos de un valor, simplemente sumamos uno al complemento a uno de ese valor.

Ejercicios de autoevaluación

H.1 Las bases de los sistemas numéricos decimal, binario, octal y hexadecimal son _____, _____, _____ y _____, respectivamente.

H.2 En general, las representaciones en decimal, octal y hexadecimal de un número binario dado contienen (más/menos) dígitos de los que contiene el número binario.

H.3 (*Verdadero/falso*) Una de las razones populares de utilizar el sistema numérico decimal es que forma una notación conveniente para abreviar números binarios, en la que tan sólo se sustituye un dígito decimal por cada grupo de cuatro dígitos binarios.

H.4 La representación (octal/hexadecimal/decimal) de un valor binario grande es la más concisa (de las alternativas dadas).

H.5 (*Verdadero/falso*) El dígito de mayor valor en cualquier base es uno más que la base.

H.6 (*Verdadero/falso*) El dígito de menor valor en cualquier base es uno menos que la base.

H.7 El valor posicional del dígito que se encuentra más a la derecha en cualquier número, ya sea binario, octal, decimal o hexadecimal es siempre _____.

H.8 El valor posicional del dígito que está a la izquierda del dígito que se encuentra más a la derecha en cualquier número, ya sea binario, octal, decimal o hexadecimal es siempre igual a _____.

H.9 Complete los valores que faltan en esta tabla de valores posicionales para las cuatro posiciones que están más a la derecha en cada uno de los sistemas numéricos indicados:

decimal	1000	100	10	1
hexadecimal	...	256	...	...
binary	...	...	...	...
octal	512	...	8	...

H.10 Convierta el número binario 110101011000 en octal y en hexadecimal.

H.11 Convierta el número hexadecimal FACE en binario.

H.12 Convierta el número octal 7316 en binario.

H.13 Convierta el número hexadecimal 4FEC en octal. (*Sugerencia:* primero convierta el número 4FEC en binario y después convierta el número resultante en octal).

H.14 Convierta el número binario 1101110 en decimal.

H.15 Convierta el número octal 317 en decimal.

H.16 Convierta el número hexadecimal EFD4 en decimal.

H.17 Convierta el número decimal 177 en binario, en octal y en hexadecimal.

H.18 Muestre la representación binaria del número decimal 417. Después muestre el complemento a uno de 417 y el complemento a dos del mismo número.

H.19 ¿Cuál es el resultado cuando se suma el complemento a dos de un número con ese mismo número?

Respuestas a los ejercicios de autoevaluación

H.1 10, 2, 8, 16.

H.2 Menos.

H.3 Falso. El hexadecimal hace esto.

H.4 Hexadecimal.

H.5 Falso. El dígito de mayor valor en cualquier base es uno menos que la base.

H.6 Falso. El dígito de menor valor en cualquier base es cero.

H.7 1 (La base elevada a la potencia de cero).

H.8 La base del sistema numérico.

H.9 Vea a continuación:

decimal	1000	100	10	1
hexadecimal	4096	256	16	1
binario	8	4	2	1
octal	512	64	8	1

H.10 6530 octal; D58 hexadecimal.

H.11 1111 1010 1100 1110 binario.

H.12 111 011 001 110 binario.

H.13 0 100 111 111 101 100 binario; 47754 octal.

H.14 2+4+8+32+64=110 decimal.

H.15 7+1*8+3*64=7+8+192=207 decimal.

H.16 4+13*16+15*256+14*4096=61396 decimal.

H.17 177 decimal
en binario:

```
256 128 64 32 16 8 4 2 1
128 64 32 16 8 4 2 1
(1*128)+(0*64)+(1*32)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1)
10110001
```

en octal:

```
512 64 8 1
64 8 1
(2*64)+(6*8)+(1*1)
261
```

en hexadecimal:

```
256 16 1
16 1
(11*16)+(1*1)
(B*16)+(1*1)
B1
```

H.18 Binario:

```
512 256 128 64 32 16 8 4 2 1
256 128 64 32 16 8 4 2 1
(1*256)+(1*128)+(0*64)+(1*32)+(0*16)+(0*8)+(0*4)+(0*2)+(1*1)
110100001
```

Complemento a uno: 001011110
Complemento a dos: 001011111
Comprobación: Número binario original + su complemento a dos:

```
110100001
001011111
---------
000000000
```

H.19 Cero.

Ejercicios

H.20 Algunas personas argumentan que muchos de nuestros cálculos se realizarían más fácilmente en el sistema numérico de base 12, ya que el 12 puede dividirse por muchos más números que el 10 (por la base 10). ¿Cuál es el dígito de menor valor en la base 12? ¿Cuál podría ser el símbolo con mayor valor para un dígito en la base 12? ¿Cuáles son los valores posicionales de las cuatro posiciones más a la derecha de cualquier número en el sistema numérico de base 12?

H.21 Complete la siguiente tabla de valores posicionales para las cuatro posiciones más a la derecha en cada uno de los sistemas numéricos indicados:

decimal	1000	100	10	1
base 6	...	...	6	...
base 13	...	169	...	...
base 3	27	...	...	...

H.22 Convierta el número binario 100101111010 en octal y en hexadecimal.

H.23 Convierta el número hexadecimal 3A7D en binario.

H.24 Convierta el número hexadecimal 765F en octal. (*Sugerencia:* primero conviértalo en binario y después convierta el número resultante en octal).

H.25 Convierta el número binario 1011110 en decimal.

H.26 Convierta el número octal 426 en decimal.

H.27 Convierta el número hexadecimal FFFF en decimal.

H.28 Convierta el número decimal 299 en binario, en octal y en hexadecimal.

H.29 Muestre la representación binaria del número decimal 779. Después muestre el complemento a uno de 779 y el complemento a dos del mismo número.

H.30 Muestre el complemento a dos del valor entero –1 en una máquina con enteros de 32 bits.

I

GroupLayout

I.1 Introducción

Java SE 6 incluye un nuevo y poderoso administrador de esquemas llamado **GroupLayout**, el cual es el administrador de esquemas predeterminado en el IDE NetBeans (`www.netbeans.org`). En este apéndice veremos las generalidades acerca de `GroupLayout`, y después demostraremos cómo usar el **diseñador de GUI Matisse** del IDE NetBeans para crear una GUI mediante el uso de `GroupLayout` para posicionar los componentes. NetBeans genera el código de `GroupLayout` por el programador de manera automática. Aunque podemos escribir código de `GroupLayout` en forma manual, en la mayoría de los casos es mejor utilizar una herramienta de diseño de GUI tal como la que proporciona NetBeans, para sacar provecho al poder de `GroupLayout`. Si desea obtener más detalles acerca de `GroupLayout`, consulte la lista de recursos Web al final de este apéndice.

I.2 Fundamentos de GroupLayout

En los capítulos 14 y 25 (el último en el sitio Web del libro) presentamos varios administradores de esquemas que proporcionan herramientas de esquemas de GUI básicas. También vimos cómo combinar administradores de esquemas y varios contenedores para crear esquemas más complejos. La mayoría de los administradores de esquemas no nos proporcionan un control preciso sobre el posicionamiento de los componentes. En el capítulo 25 (en el sitio Web del libro) vimos `GridBagLayout`, que proporciona un control más preciso sobre la posición y el tamaño de los componentes de GUI del programador. Nos permite especificar la posición vertical y horizontal de cada componente, el número de filas y columnas que ocupa cada componente en la cuadrícula, y la forma en que los componentes aumentan y reducen su tamaño, a medida que cambia el tamaño del contenedor. Todo esto se especifica al mismo tiempo con un objeto `GridBagConstraints`. La clase `GroupLayout` es el siguiente paso en la administración de esquemas. `GroupLayout` es más flexible, ya que el programador puede especificar los esquemas horizontal y vertical de sus componentes de manera independiente.

Arreglos en serie y en paralelo

Los componentes se ordenan en secuencia o en paralelo. Los tres objetos `JButton` de la figura I.1 tienen una **orientación horizontal secuencial**: aparecen de izquierda a derecha en secuencia. En sentido vertical, los componentes están ordenados en paralelo, por lo que en cierto sentido, "ocupan el mismo espacio vertical". Los componentes también se pueden ordenar secuencialmente en dirección vertical, y en paralelo en dirección horizontal, como veremos en la sección I.3. Para evitar traslapar los componentes, por lo general los componentes con orientación vertical en paralelo tienen una orientación horizontal secuencial (y viceversa).

Fig. I.1 | Objetos `JButton` ordenados en secuencia para su orientación horizontal, y en paralelo para su orientación vertical.

Grupos y alineación

Para crear interfaces de usuario más complejas, `GroupLayout` nos permite crear **grupos** que contengan elementos secuenciales o en paralelo. Dentro de un grupo, podemos tener componentes de GUI, otros grupos y huecos. Colocar un grupo dentro de otro grupo es similar a crear una GUI usando contenedores anidados, como un objeto `JPanel` que contiene otros objetos `JPanel`, que a su vez contienen componentes de GUI.

Al crear un grupo, podemos especificar la **alineación** de sus elementos. La clase `GroupLayout` contiene cuatro constantes para este fin: `LEADING`, `TRAILING`, `CENTER` y `BASELINE`. La constante `BASELINE` se aplica sólo a orientaciones verticales. En la orientación horizontal, las constantes `LEADING`, `TRAILING` y `CENTER` representan la justificación a la izquierda, justificación a la derecha y centrado, respectivamente. En la orientación vertical, `LEADING`, `TRAILING` y `CENTER` alinean los componentes en su parte central superior, inferior o vertical, respectivamente. Al alinear componentes con `BASELINE` estamos indicando que deben alinearse mediante el uso de la línea base de la fuente para el texto del componente. Para obtener más información acerca de las líneas base, vea la sección 15.4.

Espaciado

`GroupLayout` utiliza de manera predeterminada los lineamientos de diseño de GUI de la plataforma subyacente para aplicar espacio entre un componente y otro. El método **`addGap`** de las clases de `GroupLayout` anidadas **`GroupLayout.Group`**, **`GroupLayout.SequentialGroup`** y **`GroupLayout.ParallelGroup`** nos permite controlar el espaciado entre componentes.

Ajustar el tamaño de los componentes

De manera predeterminada, `GroupLayout` utiliza los métodos `getMinimumSize`, `getMaximumSize` y `getPreferredSize` de cada componente para ayudar a determinar el tamaño del componente. Podemos redefinir la configuración predeterminada.

I.3 Creación de un objeto `SelectorColores`

Ahora vamos a presentar una aplicación llamada `SelectorColores` para demostrar el administrador de esquemas `GroupLayout`. Esta aplicación consiste en tres objetos `JSlider`, cada uno de los cuales representa los valores de `0` a `255` para especificar los valores rojo, verde y azul de un color. Los valores seleccionados para cada objeto `JSlider` se utilizarán para mostrar un rectángulo sólido del color especificado. Vamos a crear esta aplicación usando NetBeans. Para obtener una introducción más detallada acerca de cómo desarrollar aplicaciones de GUI en el IDE NetBeans, vea `www.netbeans.org/kb/trails/matisse.html`.

Cree un nuevo proyecto

Empiece por abrir un nuevo proyecto en NetBeans. Seleccione **Archivo | Proyecto Nuevo...**. En el cuadro de diálogo **Proyecto Nuevo**, seleccione **Java** de la lista **Categorías** y **Aplicación Java** de la lista **Proyectos**; después haga clic en **Siguiente >**. Especifique `SelectorColores` como el nombre del proyecto y desactive la casilla de verificación **Crear clase principal**. También puede especificar la ubicación de su proyecto en el campo **Ubicación del proyecto**. Haga clic en **Terminar** para crear el proyecto.

Agregue una nueva subclase de `JFrame` al proyecto

En la ficha **Proyectos** del IDE, justo debajo del menú **Archivo** y la barra de herramientas (figura I.2), expanda el nodo **Paquetes de fuentes**. Haga clic con el botón derecho del ratón en el nodo **<paquete predeterminado>** que aparece y seleccione **Nuevo > Formulario JFrame**. En el cuadro de diálogo **Nuevo Formulario JFrame**, especifique `SelectorColores` como el nombre de la clase y haga clic en **Terminar**. Esta subclase de `JFrame` mostrará los componentes de la GUI de la aplicación. La ventana de NetBeans ahora deberá ser similar a la figura I.3, mostrando la clase `SelectorColores` en vista de **Diseño**. Los botones **Fuente** y **Diseño** en la parte superior de la ventana `SelectorColores.java` nos permiten alternar entre editar el código fuente y diseñar la GUI.

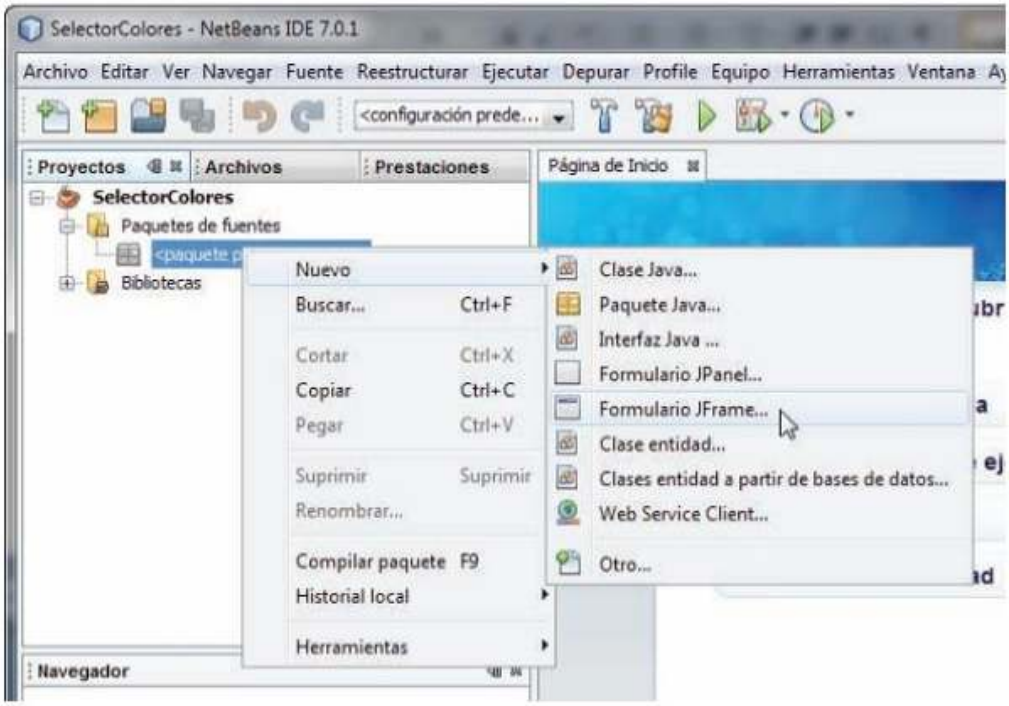


Fig. I.2 | Agregue un nuevo **Formulario JFrame** al proyecto SelectorColores.

La vista **Diseño** sólo muestra el área cliente de `SelectorColores` (es decir, el área que aparecerá dentro de los bordes de la ventana). Para crear una GUI en forma visual, puede arrastrar componentes de GUI desde la ventana **Paleta** hacia el área cliente. Para configurar las propiedades de cada componente, hay que seleccionarlo y después modificar los valores de las propiedades que aparecen en la ventana **Propiedades** (Properties) (figura I.3). Al seleccionar un componente, la ventana **Propiedades** muestra tres botones: **Propiedades**, **Eventos** y **Código** (vea la figura I.4); éstos le permiten configurar varios aspectos del componente.

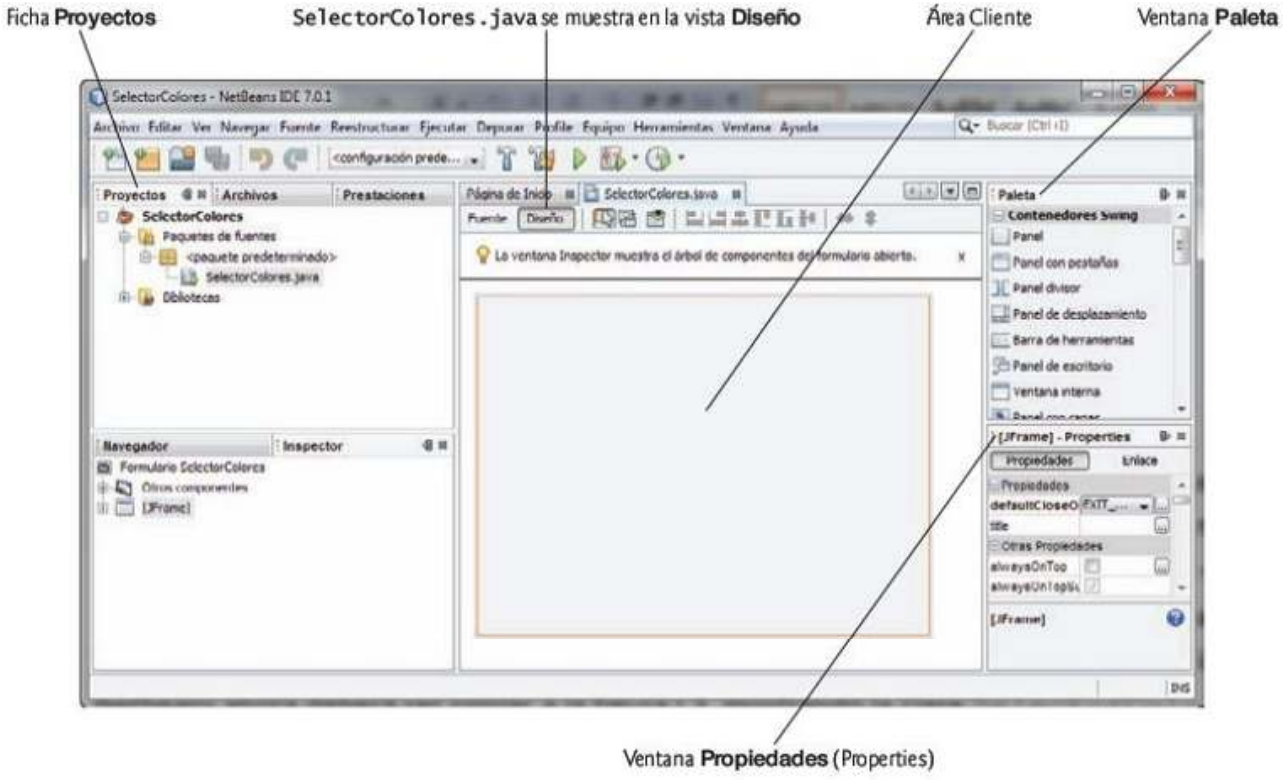


Fig. I.3 | La clase `SelectorColores` se muestra en vista **Diseño** de NetBeans.

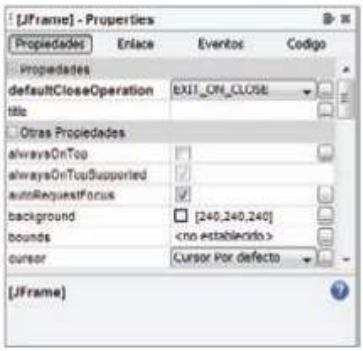


Fig. I.4 | La ventana **Propiedades** con botones que nos permiten configurar diversos aspectos del componente.

Cree la GUI

Arrastre tres componentes **Deslizador** (objetos de la clase `JSlider`) de la **Paleta** hacia el formulario `JFrame` (tal vez necesite desplazarse por la **Paleta**). A medida que arrastramos componentes cerca de los bordes del área cliente, o cerca de otros componentes, NetBeans muestra líneas guía (figura I.5) que indican las distancias y alineaciones recomendadas entre el componente que estamos arrastrando, los bordes del área cliente y los demás componentes. A medida que siga los pasos para crear la GUI, use las líneas guía para ordenar los componentes en tres filas y tres columnas, como en la figura I.6. Después, cambie el nombre de los componentes `JSlider` a `rojoJSlider`, `verdeJSlider` y `azulJSlider`. Para ello, seleccione el primer componente `JSlider`, después haga clic en el botón **Código** de la ventana **Propiedades** y cambie la propiedad **Nombre de variable** a `rojoJSlider`. Repita este

proceso para cambiar el nombre a los otros dos componentes JSlider. Después haga clic en el botón **Propiedades** de la ventana con el mismo nombre, seleccione cada componente JSlider y cambie su propiedad **maximum** a 255, para que produzca valores en el rango de 0 a 255, y cambie su propiedad **value** a 0, de manera que el indicador del componente JSlider se encuentre inicialmente a la izquierda.

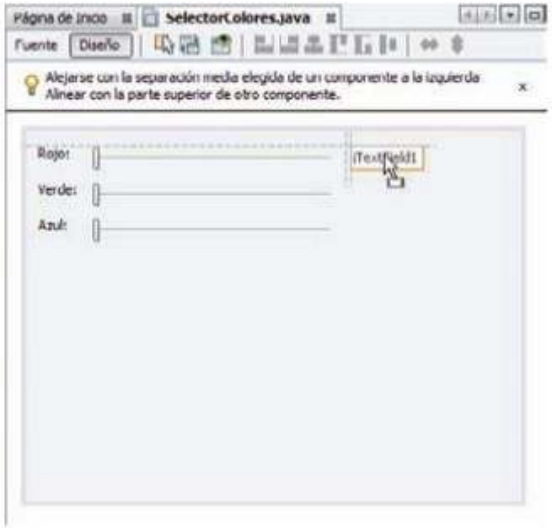


Fig. I.5 | Posicione el primer componente JTextField.

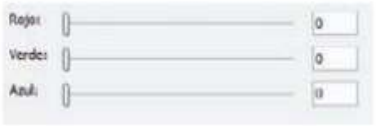


Fig. I.6 | Distribución de los componentes JLabel, JSlider y JTextField.

Arrastre tres componentes **Etiqueta** (objetos de la clase JLabel) de la **paleta** al formulario JFrame para etiquetar cada componente JSlider con el color que representa. Use los nombres rojoJLabel, verdeJLabel y azulJLabel para los componentes JLabel, respectivamente. Cada componente JLabel debe colocarse a la izquierda del componente JSlider correspondiente (figura I.6). Cambie la propiedad **text** de cada componente JLabel, ya sea haciendo doble clic en el componente JLabel y escribiendo el nuevo texto, o seleccionando el componente JLabel y cambiando la propiedad **text** en la ventana **Propiedades**.

Agregue un **Campo de texto** (un objeto de la clase JTextField) a cada uno de los componentes JSlider para mostrar su valor. Use los nombres rojoJTextField, verdeJTextField y azulJTextField, respectivamente. Cambie la propiedad **text** de cada componente JTextField a 0, usando las mismas técnicas que para los componentes JLabel. Cambie la propiedad **columns** de cada componente JTextField a 4. Para alinear cada **Etiqueta**, **Deslizador** y **Campo de texto** en forma apropiada, puede seleccionarlos arrastrando el ratón a través de los tres, y usando los botones de alineación en la parte superior de la ventana Diseño.

Ahora agregue un **Panel** llamado colorJPanel a la derecha de este grupo de componentes. Use las líneas guía como se muestra en la figura I.7 para colocar el componente JPanel. Cambie el color de fondo de este componente (propiedad **background**) a negro (el color RGB seleccionado inicialmente). Por último, arrastre el borde inferior derecho del área cliente hacia la parte superior izquierda del área

Diseño hasta que pueda ver las líneas de ajuste que muestran las medidas de altura recomendadas del área cliente (con base en los componentes que hay en el área cliente), como se muestra en la figura I.8.

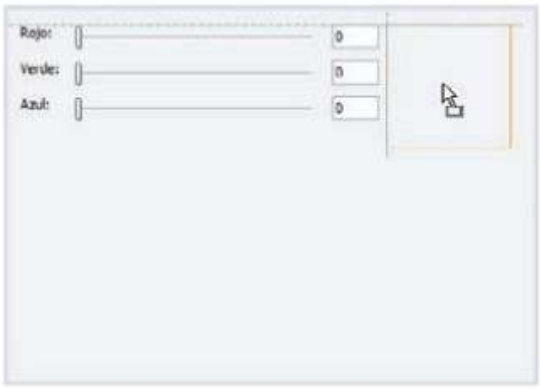


Fig. I.7 | Posicione el componente `JPanel`.

Fig. I.8 | Establezca la altura del área cliente.

Edite el código fuente y agregue manejadores de eventos

El IDE generó de manera automática el código de la GUI, incluyendo métodos para inicializar componentes y alinearlos mediante el administrador de esquemas `GroupLayout`. Debemos agregar la funcionalidad deseada a los manejadores de eventos de los componentes. Para agregar un manejador de eventos para un componente, haga clic con el botón derecho sobre él y coloque el ratón sobre la opción **Eventos** en el menú contextual. A continuación, podrá seleccionar la categoría de evento que desee manejar, y el evento específico dentro de esa categoría. Por ejemplo, para agregar los manejadores de eventos para los componentes `JSlider` en este ejemplo, haga clic en cada componente `JSlider` y seleccione **Eventos > Change > stateChanged**. Al hacer esto, NetBeans agrega un objeto `ChangeListener` al componente `JSlider` y cambia de vista de **Diseño** a vista de **Fuente**, en donde podemos colocar código en el manejador de eventos. Use el botón **Diseño** para regresar a la vista de **Diseño** y repita los pasos anteriores para agregar los manejadores de eventos para los otros dos componentes `JSlider`. Para completar los manejadores de eventos, agregue primero el método de la figura I.9 después del constructor de la clase. En cada manejador de eventos de `JSlider`, establezca el componente `JTextField` correspondiente con el nuevo valor del componente `JSlider`, y después llame al método `cambiarColor`. La figura I.10 muestra la clase `SelectorColores` completa, exactamente como la genera NetBeans. No cambiamos el código para adaptarlo a nuestras convenciones de codificación que hemos utilizado a lo largo del libro. Ahora puede ejecutar el programa para verlo en acción. Arrastre cada deslizador y vea cómo cambia el color de fondo del componente `colorJPanel`.

El método `initComponents` (líneas 39 a 149) fue generado completamente por NetBeans, con base en las interacciones del lector con el diseñador de GUI. Este método contiene el código que crea y da formato a la GUI. En las líneas 41 a 93 se construyen e inicializan los componentes de la GUI. En las líneas 95 a 148 se especifica la distribución de esos componentes mediante el uso de `GroupLayout`.

En las líneas 108 a 123 se especifica el grupo horizontal y en las líneas 124 a 146 se especifica el grupo vertical. Observe la complejidad del código. Cada vez una mayor parte del desarrollo de software se lleva a cabo con herramientas que generan código complicado como éste, lo cual ahorra al lector el tiempo y esfuerzo de hacerlo por sí mismo.

Agregamos en forma manual el método `cambiarColor` en las líneas 25 a 30. Cuando el usuario desplaza el indicador en uno de los componentes `JSlider`, el manejador de eventos de ese componente establece el texto en su correspondiente componente `JTextField` con el nuevo valor del componente `JSlider` (líneas 152, 157 y 162), después llama el método `cambiarColor` (líneas 153, 158 y 163) para actualizar el color de fondo del componente `colorJPanel`. El método `cambiarColor` obtiene el valor actual de cada componente `JSlider` (líneas 28 y 29), y utiliza estos valores como argumentos para el constructor de `Color` y crear un nuevo objeto `Color`.

```
1  // cambia el color de fondo del componente colorJPanel, con base en los valores
2  // actuales de los componentes JSlider
3  public void cambiarColor()
4  {
5     colorJPanel.setBackground( new java.awt.Color(
6        rojoJSlider.getValue(), verdeJSlider.getValue(),
7        azulJSlider.getValue() ) );
8  } // fin del método cambiarColor
```

Fig. I.9 | Método que cambia el color de fondo del componente `colorJPanel`, con base en los valores de los tres componentes `JSlider`.

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5
6  /*
7   * SelectorColores.java
8   *
9   * Created on 04-nov-2011, 10:57:51
10  */
11
12 /**
13  *
14  * @author Paul Deitel
15  */
16 public class SelectorColores extends javax.swing.JFrame {
17
18    /** Creates new form SelectorColores */
19    public SelectorColores() {
20       initComponents();
21    }
22
23    // cambia el color de fondo del componente colorJPanel, con base en los valores
24    // actuales de los componentes JSlider
25    public void cambiarColor()
26    {
```

Fig. I.10 | Método que cambia el color de fondo del componente `colorJPanel`, con base en los valores de los tres componentes `JSlider` (parte 1 de 6).

```
27          colorJPanel.setBackground( new java.awt.Color(
28             rojoJSlider.getValue(), verdeJSlider.getValue(),
29             azulJSlider.getValue() ) );
30       } // fin del método cambiarColor
31
32       /** This method is called from within the constructor to
33        * initialize the form.
34        * WARNING: Do NOT modify this code. The content of this method is
35        * always regenerated by the Form Editor.
36        */
37       @SuppressWarnings("unchecked")
38       // <editor-fold defaultstate="collapsed" desc="Generated Code">
39       private void initComponents() {
40
41          rojoJSlider = new javax.swing.JSlider();
42          verdeJSlider = new javax.swing.JSlider();
43          azulJSlider = new javax.swing.JSlider();
44          rojoJLabel = new javax.swing.JLabel();
45          verdeJLabel = new javax.swing.JLabel();
46          azulJLabel = new javax.swing.JLabel();
47          rojoJTextField = new javax.swing.JTextField();
48          verdeJTextField = new javax.swing.JTextField();
49          azulJTextField = new javax.swing.JTextField();
50          colorJPanel = new javax.swing.JPanel();
51
52          setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
53
54          rojoJSlider.setMaximum(255);
55          rojoJSlider.setValue(0);
56          rojoJSlider.addChangeListener(new javax.swing.event.ChangeListener() {
57             public void stateChanged(javax.swing.event.ChangeEvent evt) {
58                rojoJSliderStateChanged(evt);
59             }
60          });
61
62          verdeJSlider.setMaximum(255);
63          verdeJSlider.setValue(0);
64          verdeJSlider.addChangeListener(new javax.swing.event.ChangeListener() {
65             public void stateChanged(javax.swing.event.ChangeEvent evt) {
66                verdeJSliderStateChanged(evt);
67             }
68          });
69
70          azulJSlider.setMaximum(255);
71          azulJSlider.setValue(0);
72          azulJSlider.addChangeListener(new javax.swing.event.ChangeListener() {
73             public void stateChanged(javax.swing.event.ChangeEvent evt) {
74                azulJSliderStateChanged(evt);
75             }
76          });
```

Fig. I.10 | Método que cambia el color de fondo del componente `colorJPanel`, con base en los valores de los tres componentes `JSlider` (parte 2 de 6).

```
77
78          rojoJLabel.setText("Rojo:");
79
80          verdeJLabel.setText("Verde:");
81
82          azulJLabel.setText("Azul:");
83
84          rojoJTextField.setColumns(4);
85          rojoJTextField.setText("0");
86
87          verdeJTextField.setColumns(4);
88          verdeJTextField.setText("0");
89
90          azulJTextField.setColumns(4);
91          azulJTextField.setText("0");
92
93          colorJPanel.setBackground(new java.awt.Color(0, 0, 0));
94
95          javax.swing.GroupLayout colorJPanelLayout = new javax.swing.
GroupLayout(colorJPanel);
96          colorJPanel.setLayout(colorJPanelLayout);
97          colorJPanelLayout.setHorizontalGroup(
98          colorJPanelLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
99             .addGap(0, 100, Short.MAX_VALUE)
100         );
101         colorJPanelLayout.setVerticalGroup(
102         colorJPanelLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
103            .addGap(0, 100, Short.MAX_VALUE)
104         );
105
106         javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());
107         getContentPane().setLayout(layout);
108         layout.setHorizontalGroup(
109         layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
110            .addGroup(layout.createSequentialGroup()
111               .addContainerGap()
112            .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
LEADING)
113               .addComponent(verdeJLabel)
114               .addComponent(rojoJLabel)
115               .addComponent(azulJLabel))
116            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
117            .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
LEADING)
118            .addGroup(layout.createSequentialGroup()
119            .addComponent(rojoJSlider, javax.swing.GroupLayout.PREFERRED_SIZE, javax.
swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
120            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
121            .addComponent(rojoJTextField, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
122            .addGroup(layout.createSequentialGroup()
```

Fig. I.10 | Método que cambia el color de fondo del componente `colorJPanel`, con base en los valores de los tres componentes `JSlider` (parte 3 de 6).

```
123    .addComponent(verdeJSlider, javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.
GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
124    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
125    .addComponent(verdeJTextField, javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.
GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
126    .addGroup(layout.createSequentialGroup()
127    .addComponent(azulJSlider, javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.
GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
128    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
129    .addComponent(azulJTextField, javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.
GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)))
130    .addGroup(layout.createSequentialGroup()
131    .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
132    .addComponent(rojoJSlider, javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.
GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
133    .addComponent(rojoJLabel)
134    .addComponent(rojoJTextField, javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.
GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
135    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
136    .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
137    .addComponent(verdeJSlider, javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.
GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
138    .addComponent(verdeJLabel)
139    .addComponent(verdeJTextField, javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.
GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
140    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
141    .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
142    .addComponent(azulJTextField, javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.
GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
143    .addComponent(azulJSlider, javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.
GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
144    .addComponent(azulJLabel))))
145    .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
146    );
147
148    pack();
149    }// </editor-fold>
150
151    private void rojoJSliderStateChanged(javax.swing.event.ChangeEvent evt) {
152    rojoJTextField.setText( String.valueOf( rojoJSlider.getValue() ) );
153    cambiarColor();
154    }
155
156    private void verdeJSliderStateChanged(javax.swing.event.ChangeEvent evt) {
157    verdeJTextField.setText( String.valueOf( verdeJSlider.getValue() ) );
158    cambiarColor();
159    }
160
161    private void azulJSliderStateChanged(javax.swing.event.ChangeEvent evt) {
162    azulJTextField.setText( String.valueOf( azulJSlider.getValue() ) );
163    cambiarColor();
164    }
```

Fig. I.10 | Método que cambia el color de fondo del componente `colorJPanel`, con base en los valores de los tres componentes `JSlider` (parte 4 de 6).

```
165
166  /**
167  * @param args the command line arguments
168  */
169  public static void main(String args[]) {
170  /* Set the Nimbus look and feel */
171  //<editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional)
">
172  /* If Nimbus (introduced in Java SE 6) is not available, stay with the default look
and feel.
173  * For details see http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/
plaf.html
174  */
175  try {
176  for (javax.swing.UIManager.LookAndFeelInfo info : javax.swing.UIManager.
getInstalledLookAndFeels()) {
177  if ("Nimbus".equals(info.getName())) {
178  javax.swing.UIManager.setLookAndFeel(info.getClassName());
179  break;
180  }
181  }
182  } catch (ClassNotFoundException ex) {
183  java.util.logging.Logger.getLogger(SelectorColores.class.getName()).log(java.util.
logging.Level.SEVERE, null, ex);
184  } catch (InstantiationException ex) {
185  java.util.logging.Logger.getLogger(SelectorColores.class.getName()).log(java.util.
logging.Level.SEVERE, null, ex);
186  } catch (IllegalAccessException ex) {
187  java.util.logging.Logger.getLogger(SelectorColores.class.getName()).log(java.util.
logging.Level.SEVERE, null, ex);
188  } catch (javax.swing.UnsupportedLookAndFeelException ex)
189  {
190  java.util.logging.Logger.getLogger(SelectorColores.class.getName()).log(java.util.
logging.Level.SEVERE, null, ex);
191  }
192  //</editor-fold>
193
194  /* Create and display the form */
195  java.awt.EventQueue.invokeLater(new Runnable() {
196
197  public void run() {
198  new SelectorColores().setVisible(true);
199  }
200  });
201  }
202  // Variables declaration - do not modify
203  private javax.swing.JLabel azulJLabel;
204  private javax.swing.JSlider azulJSlider;
205  private javax.swing.JTextField azulJTextField;
206  private javax.swing.JPanel colorJPanel;
```

Fig. I.10 | Método que cambia el color de fondo del componente `colorJPanel`, con base en los valores de los tres componentes `JSlider` (parte 5 de 6).

```
207  private javax.swing.JLabel rojoJLabel;
208  private javax.swing.JSlider rojoJSlider;
209  private javax.swing.JTextField rojoJTextField;
210  private javax.swing.JLabel verdeJLabel;
211  private javax.swing.JSlider verdeJSlider;
212  private javax.swing.JTextField verdeJTextField;
213  // End of variables declaration
214  }
```

Fig. I.10 | Método que cambia el color de fondo del componente `colorJPanel`, con base en los valores de los tres componentes `JSlider` (parte 6 de 6).

I.4 Recursos Web sobre GroupLayout

`download.oracle.com/javase/6/docs/api/javax/swing/GroupLayout.html`
Documentación de la API para la clase `GroupLayout`.

`wiki.java.net/bin/view/Javadesktop/GroupLayoutExample`
Proporciona una demostración de una Libreta de direcciones, de una GUI creada en forma manual con `GroupLayout`, con código fuente.

`www.developer.com/java/ent/article.php/3589961`
Tutorial: "Building Java GUIs with Matisse: A Gentle Introduction", por Dick Wall.

J

Componentes de integración Java Desktop

J.1 Introducción

Los **Componentes de integración Java Desktop (JDIC)** son parte de un proyecto de código fuente abierto, orientado a permitir una mejor integración entre las aplicaciones de Java y las plataformas en las que se ejecutan. Algunas características de JDIC son:

- interacción con la plataforma subyacente para iniciar aplicaciones nativas (como navegadores Web y clientes de correo electrónico)
- mostrar una pantalla de inicio cuando una aplicación empieza a ejecutarse para indicar al usuario que se está cargando
- creación de iconos en la bandeja del sistema (también llamada área de estado de la barra de tareas, o área de notificación) para proporcionar acceso a las aplicaciones Java que se ejecutan en segundo plano
- registro de asociaciones de tipos de archivos, para que los archivos de tipos especificados se abran de manera automática en las correspondientes aplicaciones de Java
- creación de paquetes instaladores, y otras cosas más.

La página inicial de JDIC (`jdic.dev.java.net/`) incluye una introducción a JDIC, descargas, documentación, preguntas frecuentes (FAQ), demos, artículos, blogs, anuncios, proyectos de incubadora, una página para el desarrollador, foros, listas de correo y mucho más. Java SE 6 incluye algunas de las características antes mencionadas. Aquí hablaremos sobre varias de estas características.

J.2 Pantallas de inicio

Los usuarios de aplicaciones de Java perciben con frecuencia un problema en el rendimiento, ya que no aparece nada en la pantalla cuando se inicia una aplicación por primera vez. Una manera de mostrar a un usuario que su programa se está cargando es mediante una **pantalla de inicio**: una ventana sin bordes que aparece de manera temporal mientras se inicia una aplicación. Java SE 6 proporciona la nueva opción de línea de comandos **–splash** para que el comando `java` pueda llevar a cabo esta tarea. Esta opción permite al programador especificar una imagen PNG, GIF o JPG que debe aparecer al momento en que una aplicación empieza a cargarse. Para demostrar esta nueva opción, creamos un programa (figura J.1) que permanece inactivo durante 5 segundos (para que el usuario pueda ver la pantalla de inicio) y después muestra un mensaje en la línea de comandos. El directorio para este ejemplo incluye una imagen en formato PNG para utilizarla como pantalla de inicio. Para mostrar la pantalla de inicio a la hora de cargar esta aplicación, use el siguiente comando:

```
java -splash:DeitelBug.png DemoInicio
```

```
1  // Fig. J.1: DemoInicio.java
2  // Demostración de la pantalla de inicio.
3  public class DemoInicio
4  {
5     public static void main( String[] args )
6     {
7        try
8        {
9           Thread.sleep( 5000 );
10       } // fin de try
11       catch ( InterruptedException e )
12       {
13          e.printStackTrace();
14       } // fin de catch
15
16       System.out.println(
17          "Esta fue la demostracion de la pantalla de inicio." );
18    } // fin del método main
19 } // fin de la clase DemoInicio
```

Fig. J.1 | Pantalla de inicio que se muestra mediante la opción `-splash` del comando `java`.

Una vez que haya iniciado la visualización de la pantalla de inicio, podrá interactuar con ésta por medio de programación, mediante la clase `SplashScreen` del paquete `java.awt`. Para ello, puede agregar contenido dinámico a la pantalla de inicio. Para obtener más información acerca de cómo trabajar con las pantallas de inicio, vea los siguientes sitios:

```
java.sun.com/developer/technicalArticles/J2SE/Desktop/javase6/
   splashscreen/
download.oracle.com/javase/6/docs/api/java/awt/SplashScreen.html
```

J.3 La clase Desktop

La clase **Desktop** nos permite especificar un archivo o URI que deseemos abrir, mediante el uso de la aplicación apropiada de la plataforma subyacente. Por ejemplo, si el navegador Web predeterminado de su computadora es Firefox, puede usar el método browse de la clase Desktop para abrir un sitio Web en Firefox. Además, puede abrir una ventana de composición de correo electrónico en el cliente de correo electrónico predeterminado de su sistema, abrir un archivo en su aplicación asociada e imprimir un archivo mediante el uso del comando imprimir de la aplicación asociada. En la figura J.2 se demuestran las primeras tres de estas capacidades.

El manejador de eventos en las líneas 22 a 52 obtiene el número de índice de la tarea que el usuario selecciona en el componente tasksJComboBox (línea 25), y el objeto String que representa el archivo o URI a procesar (línea 26). En la línea 28 se utiliza el método static **isDesktopSupported** de Desktop para determinar si se soportan las características de la clase Desktop en la plataforma en la que se ejecute la aplicación. De ser así, en la línea 32 se utiliza el método static **getDesktop** de Desktop para obtener un objeto Desktop. Si el usuario seleccionó la opción para abrir el navegador Web predeterminado, en la línea 37 se crea un nuevo objeto URI, mediante el uso del objeto String llamado entrada, como el sitio a mostrar en el navegador, y después se pasa el objeto URI al método **browse** de Desktop, el cual invoca al navegador Web predeterminado del sistema y le pasa el URI para que lo muestre. Si el usuario selecciona la opción para abrir un archivo en su programa asociado, en la línea 40 se crea un nuevo objeto File usando el objeto String llamado entrada como el archivo a abrir, y después se pasa este objeto File al método **open** de Desktop, el cual pasa el archivo a la aplicación apropiada para que lo abra. Por último, si el usuario selecciona la opción para componer un correo electrónico, en la línea 43 se crea un nuevo objeto URI usando el objeto String llamado entrada como la dirección de correo a la cual se enviará el correo electrónico, y después se pasa el objeto URI al método **mail** de Desktop, el cual invoca al cliente de correo electrónico predeterminado del sistema y pasa el URI a ese cliente de correo electrónico como el recipiente del mensaje. Para aprender más acerca de la clase Desktop, visite el sitio:

```
download.oracle.com/javase/6/docs/api/java/awt/Desktop.html
```

```
1  // Fig. J.2: DesktopDemo.java
2  // Usa a Desktop para iniciar el navegador predeterminado, abrir un archivo en su
3  // aplicación asociada y componer un email en el cliente de email predeterminado.
4  import java.awt.Desktop;
5  import java.io.File;
6  import java.io.IOException;
7  import java.net.URI;
8
9  public class DesktopDemo extends javax.swing.JFrame
10 {
11    // constructor
12    public DesktopDemo()
13    {
14       initComponents();
15    } // end DesktopDemo constructor
16
17    // Para ahorrar espacio, no mostramos aquí las líneas 20 a 84 del código de GUI
18    // generado por Netbeans de manera automática. El código completo para este ejemplo
19    // se encuentra en el archivo DesktopDemo.java en el directorio de este ejemplo.
20
```

Fig. J.2 | Use a Desktop para iniciar el navegador Web predeterminado, abrir un archivo en su aplicación asociada y componer un correo electrónico en el cliente de correo predeterminado (parte 1 de 3).

```
21    // determina la tarea seleccionada y la lleva a cabo
22    private void doTaskJButtonActionPerformed(
23       java.awt.event.ActionEvent evt)
24    {
25       int indice = tasksJComboBox.getSelectedIndex();
26       String entrada = inputJTextField.getText();
27
28       if ( Desktop.isDesktopSupported() )
29       {
30          try
31          {
32             Desktop escritorio = Desktop.getDesktop();
33
34             switch ( indice )
35             {
36                case 0: // abre el navegador
37                   escritorio.browse( new URI( entrada ) );
38                   break;
39                case 1: // abre archivo
40                   escritorio.open( new File( entrada ) );
41                   break;
42                case 2: // abre la ventana de composición de email
43                   escritorio.mail( new URI( entrada ) );
44                   break;
45             } // fin de switch
46          } // fin de try
47          catch ( Exception e )
48          {
49             e.printStackTrace();
50          } // fin de catch
51       } //fin de if
52    } // fin del método doTaskJButtonActionPerformed
53
54    public static void main(String[] args)
55    {
56       java.awt.EventQueue.invokeLater(
57          new Runnable()
58          {
59             public void run()
60             {
61                new DesktopDemo().setVisible(true);
62             }
63          }
64       );
65    } // fin del método main
66
67    // Declaración de variables - no modificar
68    private javax.swing.JButton doTaskJButton;
69    private javax.swing.JLabel inputJLabel;
70    private javax.swing.JTextField inputJTextField;
71    private javax.swing.JLabel instructionLabel;
```

Fig. J.2 | Use a `Desktop` para iniciar el navegador Web predeterminado, abrir un archivo en su aplicación asociada y componer un correo electrónico en el cliente de correo predeterminado (parte 2 de 3).

```
72      private javax.swing.JComboBox tasksJComboBox;
73      // Termina declaración de variables
74  }
```

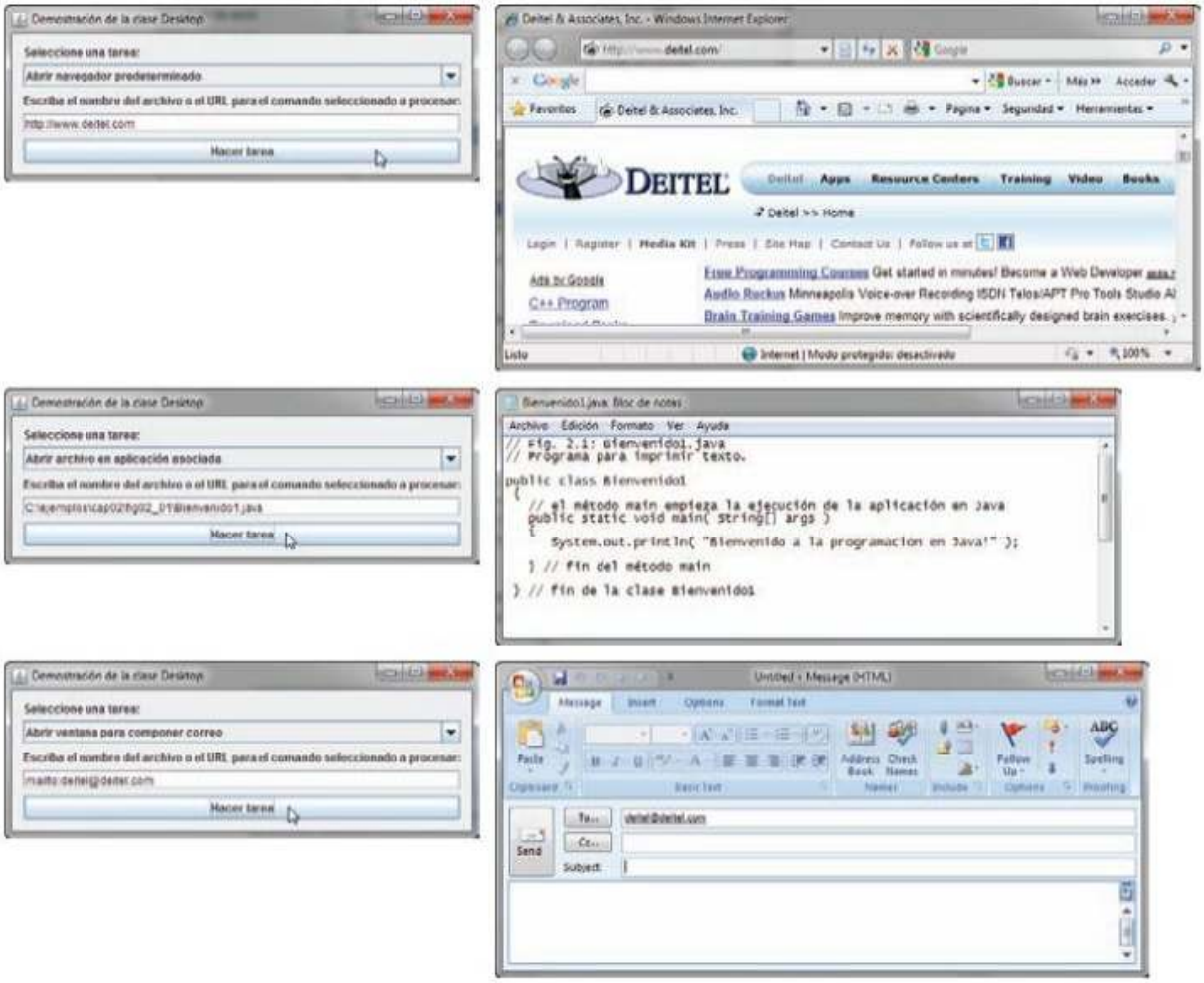


Fig. J.2 | Use a `Desktop` para iniciar el navegador Web predeterminado, abrir un archivo en su aplicación asociada y componer un correo electrónico en el cliente de correo predeterminado (parte 3 de 3).

J.4 Iconos de la bandeja

Los **iconos de la bandeja** aparecen por lo común en la bandeja de nuestro sistema, en el área de estado de la barra de tareas o en el área de notificación. Por lo general, proporcionan un acceso rápido a las aplicaciones que se ejecutan en segundo plano en nuestro sistema. Al posicionar el ratón sobre uno de estos íconos, aparece una barra de herramientas indicando qué aplicación representa el icono. Si hace dic en el icono, aparecerá un menú contextual con opciones para esa aplicación.

Las clases `SystemTray` y `TrayIcon` (ambas del paquete `java.awt`) nos permiten crear y administrar nuestros propios íconos de la bandeja, de una manera independiente a la plataforma. La clase **`SystemTray`** proporciona acceso a la bandeja del sistema de la plataforma subyacente; la clase consiste en tres métodos:

- el método `static` **`getDefaultSystemTray`** devuelve la bandeja del sistema
- el método **`addTrayIcon`** agrega un nuevo objeto `TrayIcon` a la bandeja del sistema
- el método **`removeTrayIcon`** elimina un icono de la bandeja del sistema

La clase `TrayIcon` consiste en varios métodos que permiten a los usuarios especificar un icono, un cuadro de información sobre las herramientas y un menú contextual para el icono. Además, los íconos de la bandeja soportan objetos `ActionListener`, `MouseListener` y `MouseMotionListener`. Para aprender más acerca de las clases `SystemTray` y `TrayIcon`, visite:

```
download.oracle.com/javase/6/docs/api/java/awt/SystemTray.html
download.oracle.com/javase/6/docs/api/java/awt/TrayIcon.html
```

K

Mashups

K.1 Introducción

La creación de mashups de aplicaciones Web es uno de los temas insignia de Web 2.0. El término mashup se originó en el mundo de la música; un mashup es un remix de dos o más canciones para crear una nueva canción. Puede escuchar algunos mashups de música en `www.ccmixter.org/`. Un mashup de aplicación Web combina la funcionalidad complementaria que, por lo general, se utiliza a través de servicios Web (capítulo 31) y transmisiones RSS (`www.deitel.com/rss` y `www.rssbus.com`) de varios sitios Web. Podemos crear poderosas e innovadoras aplicaciones mashup Web 2.0 con mucha más rapidez que si tuviéramos que escribir las aplicaciones desde cero. Por ejemplo, `www.housingmaps.com` combina los listados de apartamentos Craigslist con Google Maps para mostrar en un mapa todos los apartamentos en renta en un vecindario.

K.2 Mashups populares

En la figura K.1 se muestran algunos mashups populares.

URL	API	Descripción
`www.mappr.com/`	Google Maps, FlickR	Búsqueda de fotografías de sitios en Estados Unidos.
`www.housingmaps.com/`	Google Maps, Craigslist	Búsqueda de apartamentos y casas disponibles por vecindario. Incluye precios, fotografías, la dirección y la información de contacto del agente de bienes raíces.
`www.estately.com/`	Google Maps	Busque el valor aproximado de su casa, con base en las ventas recientes de casas en su área.
`www.liveplasma.com/`	Amazon eCommerce	Ingrese el nombre de un músico, banda, actor, director o película. La aplicación muestra los álbumes y películas relacionadas, etcétera. Haga clic en las imágenes para ir a la página en Amazon en donde puede comprar la película o álbum.

Fig. K.1 | Mashups populares (parte 1 de 2).

URL	API	Descripción
www.secretprices.com/default.aspx	Shopping.com, Amazon A9 OpenSearch, Amazon eCommerce	Sitio de comparación de compras que también ofrece promociones y cupones.
www.checkinmania.com/	FourSquare, Gowalla, Google Maps	Rastrea a las personas que se registran en ubicaciones cercanas, usando Foursquare y Gowalla.
appexchange.salesforce.com/listingDetail?listingId=a0330000003z9bdAAA	Facebook, Salesforce.com	Integra la información del perfil de Facebook para sus amigos en su base de datos de administración de relaciones con los clientes (CRM) de Salesforce.

Fig. K.1 | Mashups populares (parte 2 de 2).

Ahora que ha leído la mayor parte de este libro, tal vez esté familiarizado con las categorías de API, incluyendo gráficos, GUI, colecciones, multimedia, bases de datos y muchas más. Casi todas ellas proporcionan una *funcionalidad de cómputo* mejorada. Muchas API de servicios Web proporcionan *funcionalidad comercial*: eBay proporciona herramientas para subastas, Amazon proporciona ventas de libros (y ventas de otros tipos de productos, como CD, DVD, dispositivos electrónicos, y otros más), Google proporciona herramientas de búsqueda, PayPal proporciona servicios de pago, etcétera. Por lo general, estos servicios Web son gratuitos para su uso no comercial; algunos establecen cuotas (por lo general, razonables) para su uso comercial. Esto crea enormes posibilidades para las personas que crean aplicaciones y comercios basados en Internet.

K.3 Algunas API de uso común en mashups

Hemos enfatizado la importancia de la reutilización de software. Los mashups son otra forma más de reutilización de software que nos ahorra tiempo, dinero y esfuerzo; podemos crear con rapidez versiones prototipo de nuestras aplicaciones, integrar funcionalidad de negocios, de búsqueda y mucho más. En la figura 1.19 encontrará algunas API de uso común en mashups.

K.4 Centro de recursos Deitel sobre mashups

Nuestro Centro de recursos sobre mashups, que se encuentra en

```
www.deitel.com/mashups/
```

se enfoca en la enorme cantidad de contenido de mashups gratuito, disponible en línea. Encontrará tutoriales, artículos, documentación, los libros más recientes, blogs, directorios, herramientas, foros, etcétera, que le ayudarán a desarrollar rápidamente aplicaciones de mashups.

- Dé un vistazo a los mashups más recientes y populares, incluyendo decenas de mashups basados en Google Maps, mostrando la ubicación de cines, bienes raíces para venta o renta, propiedades que se han vendido en su área, ¡e incluso las ubicaciones de los sanitarios públicos en San Francisco!.
- Busque mashups en ProgrammableWeb por categoría.

- Dé un vistazo a las API de Flickr para agregar fotografías a sus aplicaciones, actualizar fotografías, reemplazarlas, ver peticiones de ejemplos y enviar en forma asíncrona.
- Lea el artículo "Building Mashups for Non-Programmers".
- Dé un vistazo a la herramienta Smashforce que permite a los usuarios de Salesforce.com usar aplicaciones como Google Maps con sus aplicaciones Multiforce y Sforce empresariales.
- Dé un vistazo a la Herramienta de Mashups empresarial de IBM.
- Dé un vistazo a las API de búsqueda y mapas de Microsoft, Yahoo! y Google que puede usar en sus aplicaciones de mashups.
- Use las API de Technorati para buscar todos los blogs que vinculen a un sitio específico, busque la mención de ciertas palabras en blogs, vea cuáles blogs están vinculados con un blog dado y busque blogs asociados con un sitio Web específico.
- Use la API de Backpack para que le ayude a organizar tareas y eventos, planear su itinerario, colaborar con otros, monitorear a sus competidores en línea, y mucho más.

K.5 Centro de recursos Deitel sobre RSS

Las transmisiones RSS son también fuentes de información populares para los mashups. Para aprender más acerca de las transmisiones RSS, visite nuestro Centro de recursos de RSS en `www.deitel.com/RSS/`.

K.6 Cuestiones de rendimiento y confiabilidad de los mashups

Hay varios retos a vencer al crear aplicaciones de mashups. Sus aplicaciones se hacen susceptibles a los problemas de tráfico y confiabilidad en Internet; circunstancias que, por lo general, están fuera de su control. Las compañías podrían cambiar repentinamente las API que sus aplicaciones utilizan. Su aplicación depende de las herramientas de hardware y software de otras compañías. Además, las compañías podrían establecer estructuras de cuotas para servicios Web que antes eran gratuitos, o podrían incrementar las cuotas existentes.

L

Unicode®

L.1 Introducción

El uso de **codificaciones de caracteres** (es decir, valores numéricos asociados con caracteres) inconsistentes al desarrollar productos de software globales provoca graves problemas, ya que las computadoras procesan la información utilizando números. Por ejemplo, el carácter "a" se convierte en un valor numérico para que una computadora pueda manipular esa pieza de información. Muchos países y corporaciones han desarrollado sus propios sistemas de codificación, los cuales son incompatibles con los sistemas de codificación de otros países y corporaciones. Por ejemplo, el sistema operativo Microsoft Windows asigna el valor 0xC0 al carácter "A con un acento grave" mientras que el sistema operativo Apple Macintosh asigna ese mismo valor a un signo de interrogación de apertura. Esto produce una mala representación y la posible corrupción de los datos.

A falta de un estándar universal de codificación de caracteres, los desarrolladores de software de todo el mundo tuvieron que localizar sus productos en forma exhaustiva, antes de distribuirlos. La **localización** incluye la traducción del lenguaje y la adaptación cultural del contenido. El proceso de localización por lo general incluye modificaciones considerables al código fuente (como la conversión de valores numéricos y las suposiciones subyacentes por parte de los programadores), lo cual produce un aumento en costos y retrasos en la liberación del software. Por ejemplo, un programador de habla inglesa podría diseñar un producto de software global suponiendo que un solo carácter puede ser representado por un byte. Sin embargo, cuando esos productos se localizan en mercados asiáticos las suposiciones del programador ya no son válidas debido a que hay muchos más caracteres asiáticos y, por ende, la mayor parte del (si no es que todo el) código necesita volver a escribirse. La localización es necesaria cada vez que se libera una versión. Para cuando el producto de software se localiza para un mercado específico ya está lista para distribuirse una nueva versión, la cual también necesita localizarse. Como resultado, es muy complicado y costoso producir y distribuir productos de software global en un mercado en el que no existe un estándar universal de codificación de caracteres universal.

En respuesta a esta situación se creó el **Estándar Unicode**, un estándar de codificación que facilita la producción y distribución de software. El estándar Unicode describe una especificación para producir la codificación consistente de los caracteres y símbolos de todo el mundo. Los productos de software que se encargan del texto codificado en el estándar Unicode necesitan localizarse, pero este proceso de localización es más simple y eficiente debido a que los valores numéricos no necesitan convertirse, y las suposiciones de los programadores acerca de la codificación de caracteres son universales. El estándar Unicode es mantenido por una organización sin fines de lucro conocida como el **Consorcio Unicode**, cuyos miembros incluyen a Apple, IBM, Microsoft, Oracle, Sun Microsystems, Sybase y muchos otros.

Cuando el Consorcio ideó y desarrolló el estándar Unicode, querían un sistema de codificación que fuera **universal**, **eficiente**, **uniforme** y **sin ambigüedades**. Un sistema de codificación universal comprende a todos los caracteres de uso común. Un sistema de codificación eficiente permite que los archivos de texto se analicen con rapidez. Un sistema de codificación uniforme asigna valores fijos a todos los caracteres. Un sistema de codificación sin ambigüedades representa a un carácter dado en una manera consistente. A estos cuatro términos se les conoce como la base del diseño del estándar Unicode.

L.2 Formatos de transformación de Unicode

Aunque Unicode incorpora el conjunto de caracteres (es decir, una colección de caracteres) ASCII limitado, comprende un conjunto de caracteres más completo. En ASCII, cada carácter se representa mediante un byte que contiene 0s y 1s. Un byte es capaz de almacenar los números binarios de 0 a 255. A cada carácter se le asigna un número entre 0 y 255, por lo cual los sistemas basados en ASCII sólo soportan 256 caracteres, una minúscula fracción de los caracteres existentes en el mundo. Unicode extiende el conjunto de caracteres ASCII al codificar la gran mayoría de los caracteres de todo el mundo. El estándar Unicode codifica a todos esos caracteres en un espacio numérico uniforme, de 0 a 10FFFF en hexadecimal. Una implementación expresará estos números en uno de varios formatos de transformación, seleccionando el que se adapte mejor a la aplicación específica en consideración.

Hay tres de esos formatos en uso: **UTF-8**, **UTF-16** y **UTF-32**. UTF-8, un formato de codificación de anchura variable, requiere de uno a cuatro bytes para expresar cada uno de los caracteres Unicode. Los datos de UTF-8 consisten en bytes de 8 bits (secuencias de uno, dos, tres o cuatro bytes, dependiendo del carácter que se vaya a codificar) y son bastante adecuados para los sistemas basados en ASCII, cuando predominan los caracteres de un byte (ASCII representa los caracteres como un byte). En la actualidad, UTF-8 se implementa mucho en sistemas UNIX y bases de datos.

El formato de codificación UTF-16 de anchura variable expresa los caracteres Unicode en unidades de 16 bits (es decir, como dos bytes adyacentes, o un entero corto en muchos equipos). La mayoría de los caracteres de Unicode se expresan en una sola unidad de 16 bits. Sin embargo, los caracteres con valores por arriba de FFFF en hexadecimal se expresan mediante un par ordenado de unidades de 16 bits, conocidas como **sustitutos**. Los sustitutos son enteros de 16 bits en el rango de D800 a DFFF, que se utilizan únicamente para el propósito de "escapar" hacia caracteres con una numeración más alta. De esta forma pueden expresarse cerca de un millón de caracteres. Aunque un par de sustitutos requiere de 32 bits para representar caracteres, al utilizar estas unidades de 16 bits se optimiza el uso del espacio. Los sustitutos son caracteres raros en las implementaciones actuales. Muchas implementaciones para el manejo de cadenas se escriben en términos de UTF-16. [*Nota:* los detalles y el código de ejemplo para el manejo de UTF-16 están disponibles en el sitio Web del consorcio Unicode, en `www.unicode.org`].

Las implementaciones que requieren de un uso considerable de caracteres raros o secuencias de comandos completas con una codificación por encima del FFFF hexadecimal deben usar UTF-32, un formato de codificación de 32 bits con anchura fija que, por lo general, requiere del doble de memoria que los caracteres codificados con UTF-16. La principal ventaja del formato de codificación UTF-32 de anchura fija es que expresa de manera uniforme a todos los caracteres, por lo que es fácil de manejar en los arreglos.

Hay unos cuantos lineamientos que indican cuándo utilizar un formato de codificación específico. El mejor formato de codificación a utilizar depende de los sistemas computacionales y los protocolos de negocios, no de la información en sí. En general, debe utilizarse el formato de codificación UTF-8 cuando los sistemas computacionales y los protocolos de negocios requieren que la información se maneje en unidades de 8 bits, especialmente en los sistemas heredados que se están actualizando, ya que esto a menudo simplifica los cambios que deben realizarse en los programas existentes. Por esta razón, UTF-8 se ha convertido en el formato de codificación preferido en Internet. De la misma forma, UTF-16 es el formato de codificación preferido en aplicaciones para Microsoft Windows. Es probable que

UTF-32 se utilice mucho más en el futuro, a medida que se codifiquen más caracteres con valores por encima del FFFF hexadecimal. Además, UTF-32 requiere de un manejo menos sofisticado que UTF-16, debido a la presencia de los pares de sustitutos. En la figura L.1 se muestran las distintas formas en que los tres formatos de codificación manejan la codificación de caracteres.

Carácter	UTF-8	UTF-16	UTF-32
LETRA A MAYÚSCULA EN LATÍN	0x41	0x0041	0x00000041
LETRA ALFA MAYÚSCULA EN GRIEGO	0xCD 0x91	0x0391	0x00000391
IDEOGRAMA UNIFICADO CJK-4E95	0xE4 0xBA 0x95	0x4E95	0x00004E95
LETRA A EN CURSIVA ANTIGUA	0xF0 0x80 0x83 0x80	0xDC00 0xDF00	0x00010300

Fig. L.1 | La correlación entre los tres formatos de codificación.

L.3 Caracteres y glifos

El Estándar Unicode consiste de caracteres: componentes escritos (es decir, alfabetos, números, signos de puntuación, acentos, etcétera) que se pueden representar mediante valores numéricos. Un ejemplo de este tipo de caracteres es: U+0041, LETRA A MAYÚSCULA EN LATÍN. En la primera representación de caracteres, **U+*yyyy*** es un **valor de código**, en donde U+ se refiere a los valores de código de Unicode, a diferencia de los demás valores hexadecimales. Las letras *yyyy* representan un número hexadecimal de cuatro dígitos, perteneciente a un carácter codificado. Los valores de códigos son combinaciones de bits que representan caracteres codificados. Los caracteres se representan utilizando **glifos**: varias figuras, tipos de letra y tamaños para mostrar caracteres. No hay valores de código para los glifos en el estándar Unicode. En la figura L.2 se muestran ejemplos de glifos.

El estándar Unicode comprende los alfabetos, ideogramas, silabarios, signos de puntuación, signos **diacríticos**, operadores matemáticos, y otras características que comprenden los lenguajes escritos y manuscritos del mundo. Un signo diacrítico es un signo especial que se agrega a un carácter para diferenciarlo de otra letra, o para indicar un acento (por ejemplo, en español se utiliza la tilde "~" por encima del carácter "n"). En la actualidad, Unicode proporciona los valores de código para 96,382 representaciones de caracteres, con más de 878,000 valores de código reservados para una expansión en el futuro.

Fig. L.2 | Varios glifos del carácter A.

L.4 Ventajas/Desventajas de Unicode

El estándar Unicode tiene varias ventajas considerables que promueven su uso. Una es el impacto que tiene sobre el rendimiento de la economía internacional. Unicode estandariza los caracteres para los sistemas de escritura mundiales en un modelo uniforme que promueve la transferencia y la compartición de información. Los programas que se desarrollan usando este esquema mantienen su precisión, ya que

cada carácter tiene una sola definición (por ejemplo, *a* es siempre U+0061, *%* es siempre U+0025). Esto permite a las corporaciones administrar las altas demandas de los mercados internacionales, al procesar distintos sistemas de escritura al mismo tiempo. Además, los caracteres pueden administrarse en forma idéntica, evitando así la confusión ocasionada por las distintas arquitecturas de códigos de caracteres. Lo que es más, al administrar los datos de una manera consistente se elimina la corrupción de los mismos, ya que la información puede ordenarse, buscarse y manipularse utilizando un proceso consistente.

Otra ventaja del estándar Unicode es la portabilidad (es decir, software que puede ejecutarse en computadoras distintas o con distintos sistemas operativos). La mayoría de los sistemas operativos, bases de datos, lenguajes de programación (incluyendo Java y los lenguajes Microsoft .NET) y navegadores Web soportan en la actualidad, o planean soportar, Unicode.

Una desventaja del estándar Unicode es la cantidad de memoria que requieren UTF-16 y UTF-32. Los conjuntos de caracteres ASCII tienen una longitud de 8 bits, por lo que requieren de una menor capacidad de almacenamiento que el conjunto de caracteres Unicode predeterminado de 16 bits. El **conjunto de caracteres de doble byte (DBCS)** codifica los caracteres asiáticos con uno o dos bytes por carácter. El **conjunto de caracteres multibyte (MBCS)** codifica los caracteres asiáticos con un número variable de bytes por carácter. En tales casos, pueden usarse los formatos de codificación UTF-16 o UTF-32 con pocas restricciones en cuanto a memoria y rendimiento.

Otra desventaja de Unicode es que, aunque incluye más caracteres que cualquier otro conjunto de caracteres de uso común, todavía no codifica a todos los caracteres escritos del mundo. Además, UTF-8 y UTF-16 son formatos de codificación con anchura variable, por lo que los caracteres ocupan distintas cantidades de memoria.

L.5 Uso de Unicode

Muchos lenguajes de programación (por ejemplo: C, Java, JavaScript, Perl, Visual Basic) proporcionan cierto nivel de soporte para el estándar Unicode. La aplicación que se muestra en las figuras L.3 y L.4 imprime el texto "¡Bienvenido a Unicode!" en ocho idiomas distintos: inglés, ruso, francés, alemán, japonés, portugués, español y chino tradicional.

```
1  // Fig. L.3: UnicodeJFrame.java
2  // Demostración de cómo usar Unicode en programas de Java.
3  import java.awt.GridLayout;
4  import javax.swing.JFrame;
5  import javax.swing.JLabel;
6
7  public class UnicodeJFrame extends JFrame
8  {
9     // el constructor crea componentes JLabel para mostrar Unicode
10    public UnicodeJFrame()
11    {
12       super ( "Demostración de Unicode" );
13
14       setLayout( new GridLayout( 8, 1 ) );  // establece esquema de marco
15
16       // crea componentes JLabel que usan Unicode
17       JLabel inglesJLabel = new JLabel( "\u0057\u0065\u006C\u0063\u006F" +
18          "\u006D\u0065\u0020\u0074\u006F\u0020Unicode\u0021" );
```

Fig. L.3 | Aplicación de Java que utiliza codificación de Unicode (parte 1 de 2).

```
19          inglesJLabel.setToolTipText( "Esto es inglés" );
20          add( inglesJLabel );
21
22          JLabel chinoJLabel = new JLabel( "\u6B22\u8FCE\u4F7F\u7528" +
23             "\u0020\u0020Unicode\u0021" );
24          chinoJLabel.setToolTipText( "Esto es chino tradicional" );
25          add( chinoJLabel );
26
27          JLabel cirilicoJLabel = new JLabel( "\u0414\u043E\u0431\u0440" +
28             "\u043E\u0020\u043F\u043E\u0436\u0430\u043B\u043E\u0432" +
29             "\u0430\u0442\u044A\u0020\u0432\u0020Unicode\u0021" );
30          cirilicoJLabel.setToolTipText( "Esto es ruso" );
31          add( cirilicoJLabel );
32
33          JLabel francesJLabel = new JLabel( "\u0042\u0069\u0065\u006E\u0076" +
34             "\u0065\u006E\u0075\u0065\u0020\u0061\u0075\u0020Unicode\u0021");
35          francesJLabel.setToolTipText( "Esto es francés" );
36          add( francesJLabel );
37
38          JLabel alemanJLabel = new JLabel( "\u0057\u0069\u006C\u006B\u006F" +
39             "\u006D\u006D\u0065\u006E\u0020\u007A\u0075\u0020Unicode\u0021" );
40          alemanJLabel.setToolTipText( "Esto es alemán" );
41          add( alemanJLabel );
42
43          JLabel japonesJLabel = new JLabel( "Unicode\u3078\u3087\u3045" +
44             "\u3053\u305D\u0021" );
45          japonesJLabel.setToolTipText( "Esto es japonés" );
46          add( japonesJLabel );
47
48          JLabel portuguesJLabel = new JLabel( "\u0053\u00E9\u006A\u0061" +
49             "\u0020\u0042\u0065\u006D\u0076\u0069\u006E\u0064\u006F\u0020" +
50             "Unicode\u0021" );
51          portuguesJLabel.setToolTipText( "Esto es portugués" );
52          add( portuguesJLabel );
53
54          JLabel espanolJLabel = new JLabel( "\u0042\u0069\u0065\u006E" +
55             "\u0076\u0065\u006E\u0069\u0064\u0061\u0020\u0061\u0020" +
56             "Unicode\u0021" );
57          espanolJLabel.setToolTipText( "Esto es español" );
58          add( espanolJLabel );
59       } // fin del constructor de Unicode
60    } // fin de la clase UnicodeJFrame
```

Fig. L.3 | Aplicación de Java que utiliza codificación de Unicode (parte 2 de 2).

```
1  // Fig. L.4: Unicode.java
2  // Cómo mostrar Unicode en pantalla.
3  import javax.swing.JFrame;
4
5  public class Unicode
6  {
```

Fig. L.4 | Visualización de Unicode (parte 1 de 2).

```
7     public static void main( String[] args )
8     {
9        UnicodeJFrame unicodeJFrame = new UnicodeJFrame();
10       unicodeJFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       unicodeJFrame.setSize( 350, 250 );
12       unicodeJFrame.setVisible( true );
13    } // fin del método main
14 } // fin de la clase Unicode
```

Fig. L.4 | Visualización de Unicode (parte 2 de 2).

La clase `UnicodeJFrame` (figura L.3) utiliza secuencias de escape para representar caracteres. Una secuencia de escape tiene la forma `\uyyyy`, en donde *yyyy* representa el valor de código hexadecimal de cuatro dígitos. Las líneas 17 y 18 contienen las series de secuencias de escape necesarias para imprimir el mensaje "Bienvenido a Unicode!" en inglés (Welcome to Unicode). La primera secuencia de escape (`\u0057`) corresponde al carácter "W," la segunda secuencia de escape (`\u0065`) corresponde al carácter "e", y así sucesivamente. La secuencia de escape `\u0020` (línea 18) es la codificación para el carácter de espacio. Las secuencias de escape `\u0074` y `\u006F` corresponden a la palabra "to". La palabra "Unicode" no se codifica, ya que es una marca registrada y no tiene traducción equivalente en la mayoría de los lenguajes. La línea 18 también contiene la secuencia de escape `\u0021` para el signo de admiración (!).

Las líneas 22 a 56 contienen las secuencias de escape para los otros siete idiomas. El sitio Web del Consorcio Unicode contiene un vínculo a tablas de códigos que enumeran los valores de código de 16 bits de Unicode. Los caracteres de inglés, francés, alemán, portugués y español se encuentran en el bloque **Basic Latin** (Latín básico), los caracteres japoneses se encuentran en el bloque **Hiragana**, los caracteres rusos se encuentran en el bloque **Cyrillic** (cirílico) y los caracteres de chino tradicional se encuentran en el bloque **CJK Unified Ideographs** (ideogramas unificados CJK). En la siguiente sección veremos estos bloques.

L.6 Rangos de caracteres

El estándar Unicode asigna valores de código, que varían de `0000` (**Latín básico**) a `E007F` (**Marcas**), a los caracteres escritos de todo el mundo. En la actualidad existen valores de código para 96,382 caracteres. Para simplificar la búsqueda de un carácter y su valor de código asociado, el estándar Unicode por lo general agrupa los valores de código por **escritura** y función (es decir, los caracteres en latín se agrupan en un bloque, los operadores matemáticos en otro bloque, etcétera). Como regla, una escritura es un sistema de escritura individual que se utiliza para varios lenguajes (por ejemplo, la escritura del latín se utiliza para el inglés, francés, español, etcétera). La página **Code Charts** (Tablas de códigos) en el sitio Web del Consorcio Unicode enlista todos los bloques definidos y sus respectivos valores de códigos. En la figura L.5 se enlistan algunos bloques (escrituras) del sitio Web y su rango de valores de código.

Escritura	Rango de valores de código
Arábica	U+0600–U+06FF
Latín básico	U+0000–U+007F
Bengalí (La India)	U+0980–U+09FF
Cherokee (Nativo de América)	U+13A0–U+13FF
Ideogramas Unificados CJK (Este de Asia)	U+4E00–U+9FFF
Cirílico (Rusia y Este de Europa)	U+0400–U+04FF
Etiope	U+1200–U+137F
Griego	U+0370–U+03FF
Hangul Jamo (Corea)	U+1100–U+11FF
Hebreo	U+0590–U+05FF
Hiragana (Japón)	U+3040–U+309F
Khmer (Cambodia)	U+1780–U+17FF
Lao (Laos)	U+0E80–U+0EFF
Mongol	U+1800–U+18AF
Myanmar	U+1000–U+109F
Ogham (Irlanda)	U+1680–U+169F
Runic (Alemania y Escandinavia)	U+16A0–U+16FF
Sinhala (Sri Lanka)	U+0D80–U+0DFF
Telugu (La India)	U+0C00–U+0C7F
Thai	U+0E00–U+0E7F

Fig. L.5 | Algunos rangos de caracteres.

Índice

Las referencias de páginas que aparecen en **negritas** es donde se definen los términos.

Símbolos

A

D

E

F

G

Q

R

T

U

V

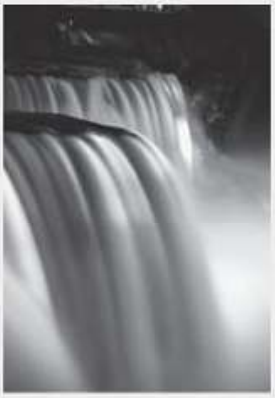
W

Y

Z

Tema de la portada

El tema de portada para la serie DEITEL® CÓMO PROGRAMAR, hace énfasis en aspectos sociales, entre los que destacan fomentar la ecología, la energía limpia, el reciclaje y la sustentabilidad. Además de los ejercicios de programación convencionales hemos incluido otros en un grupo llamado Marcar la diferencia, para crear conciencia sobre cuestiones tales como el calentamiento global, el aumento de la población, los servicios médicos costeables, la accesibilidad y la privacidad de los registros electrónicos. En este libro utilizará Java para programar aplicaciones que se relacionan con estos temas. Esperamos que lo que aprenda en *Cómo programar en Java, novena edición* le ayude a marcar la diferencia.



Noche en las cataratas del Niágara: Ontario y Nueva York

Las cataratas del Niágara se extienden en la frontera entre Ontario, Canadá, y el norte del estado de Nueva York, en Estados Unidos. Las cataratas Horseshoe Falls se encuentran del lado canadiense, y las cataratas American Falls y Bridal Veil Falls están del lado de Estados Unidos.

Las cataratas del Niágara son una atracción turística impresionante de clase mundial, pero además son una fuente importante de energía hidroeléctrica. En promedio, caen casi cuatro millones de pies cúbicos de agua sobre la línea de la cresta cada minuto. Estados Unidos y Canadá aprovechan la energía natural de estas cataratas para generar electricidad limpia y de bajo costo. La primera estación hidroeléctrica en el río Niágara se construyó en 1881. Su electricidad abasteció a los molinos locales y el alumbrado público. La Niagara Redevelopment Act, aprobada por el Congreso de Estados Unidos en 1957, otorgó a la New York Power Authority el derecho de desarrollar plantas de energía hidroeléctrica en el Río Niágara de Estados Unidos. El proyecto hidroeléctrico *Niagara Falls* empezó a operar en 1961. Hasta 375,000 galones de agua por segundo se desvían del río por medio de ductos hacia las plantas de energía. El agua hace girar turbinas que alimentan a los generadores, los cuales convierten la energía mecánica en energía eléctrica. En la actualidad, el proyecto genera 2.4 millones de kilowatts, que pueden alimentar 24 millones de bombillas de 100 watts al mismo tiempo. Para mayor información, visite:

es.wikipedia.org/wiki/Cataratas_del_Ni%C3%Algara
en.wikipedia.org/wiki/Niagra_falls

www.nypa.gov/facilities/niagara.htm
www.niagarafrontier.com/power.html

Acerca de Deitel & Associates, Inc.

Deitel & Associates, Inc. es una organización internacional para la creación de material didáctico y de capacitación corporativa. La compañía ofrece cursos en las instalaciones de sus clientes en todo el mundo, sobre lenguajes de programación y temas relacionados con el software, como Java™, C#®, Visual Basic®, Visual C++®, C++, C, Objective-C®, XML®, Python®, JavaScript, tecnología de objetos, programación en Internet y Web, y desarrollo de aplicaciones para Android y iPhone. Entre sus clientes se incluyen muchas de las empresas más grandes del mundo, así como agencias gubernamentales, ramas del ejército e instituciones académicas. Para conocer más sobre las publicaciones de educación superior de Deitel-Pearson y la capacitación corporativa de la serie Dive Into®, envíe un correo electrónico a deitel@deitel.com o visite www.deitel.com/training/. Puede seguir a Deitel por Facebook® en www.deitel.com/deitelfan/ y por Twitter® en @deitel.

Cómo programar en Java, Novena edición

Bienvenido a *Cómo programar en Java, Novena edición*. Este libro presenta las tecnologías de computación de vanguardia y las mejores prácticas de ingeniería de software para estudiantes, profesores y desarrolladores de software.

Esta nueva edición se basa en el reconocido método de "código activo", donde los conceptos se presentan en un contexto de programas funcionales completos, en el que se aprovechan los conceptos y el código, y la experiencia es semejante a la que se tendrá en el desarrollo profesional.

Este libro ofrece una introducción clara, simple, atractiva y entretenida a la programación en Java.

Entre las características más relevantes de esta edición sobresalen:

- Amplia cobertura de los fundamentos, que incluye dos capítulos sobre instrucciones de control.
- Enfoque en ejemplos reales.
- Un conjunto de ejercicios "marcar la diferencia", con enfoque en el medio ambiente.
- Introducción a las clases y objetos desde los primeros capítulos.
- Manejo de excepciones integrado.
- Secciones modulares opcionales sobre el lenguaje y las características de biblioteca del nuevo Java SE 7.
- Introducción opcional en línea al desarrollo de aplicaciones Android basado en Java.
- Archivos, flujos y serialización de objetos.
- Recursividad, búsqueda, ordenamiento, colecciones genéricas, estructuras de datos, applets, multimedia, multihilos, bases de datos/JDBC™, desarrollo de aplicaciones Web, servicios Web, y un caso de estudio opcional sobre el diseño orientado a objetos con el ATM.

Además, en el sitio web del libro encontrará:

- Capítulos en español para aquellos alumnos que deseen continuar la secuencia del curso.
- Capítulos en inglés para cursos avanzados y profesionales.
- Todo el código en español de los ejemplos utilizados en el libro.
- Todo el código fuente en inglés (disponible también en www.deitel.com/books/jhtp9/).

Para mayor información visite el sitio Web de este libro en:

www.pearsonenespañol.com/deitel.

ISBN 978-607-32-1150-5
90000
9 786073 211505