PAUL DEITEL
HARVEY DEITEL

Java™

CÓMO PROGRAMAR

DÉCIMA EDICIÓN



Pearson

DEITEL®

Java™

CÓMO PROGRAMAR

Décima edición



Java™ CÓMO PROGRAMAR

Décima edición

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

Traducción
Alfonso Vidal Romero Elizondo
Ingeniero en Sistemas Electrónicos
Instituto Tecnológico y de Estudios Superiores de Monterrey - Campus Monterrey

Revisión técnica
Sergio Fuenlabrada Velázquez
Edna Martha Miranda Chávez
Judith Sonck Ledezma
Mario Alberto Sesma Martínez
Mario Oviedo Galdeano
José Luis López Goytia
Departamento de Sistemas
Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas, Instituto Politécnico Nacional, México

Pearson

Datos de catalogación bibliográfica

DEITEL, PAUL y DEITEL, HARVEY

Cómo programar en Java
Décima edición

PEARSON EDUCACIÓN, México, 2016

ISBN: 978-607-32-3802-1
Área: Computación

Formato: 20 × 25.5 cm Páginas: 560

Authorized translation from the English language edition, entitled *Java How to program 10th* Edition, by *Paul Deitel and Harvey Deitel,* published by Pearson Education, Inc., publishing as Prentice Hall, Copyright © 2016. All rights reserved.
ISBN 9780133807806

Traducción autorizada de la edición en idioma inglés titulada *Java How to program 10a* edición, por *Paul Deitel and Harvey Deitel,* publicada por Pearson Education, Inc., publicada como Prentice Hall, Copyright © 2016. Todos los derechos reservados.

Esta edición en español es la única autorizada.

Edición en español

Director General:	Sergio Fonseca Garza
Director de innovación y servicios educativos:	Alan David Palau
Gerencia de contenidos y servicios editoriales:	Jorge Luis Íñiguez Caso
Coordinador de Contenidos, Educación Superior:	Guillermo Domínguez Chávez guillermo.dominguez@pearson.com
Especialista en Contenidos de Aprendizaje:	Luis Miguel Cruz Castillo
Especialista en Desarrollo de Contenidos:	Bernardino Gutiérrez Hernández
Supervisor de Arte y Diseño:	José Dolores Hernández Garduño

DÉCIMA EDICIÓN, 2016

D.R. © 2016 por Pearson Educación de México, S.A. de C.V.
Antonio Dovalí Jaime, núm. 70,
Torre B, Piso 6, Col. Zedec
Ed. Plaza Santa Fe,
Deleg. Álvaro Obregón
C.P. 01210, Ciudad de México

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

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

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

ISBN VERSIÓN IMPRESA: 978-607-32-3802-1
ISBN VERSIÓN E-BOOK: 978-607-32-3803-8

Impreso en México. *Printed in Mexico.*
1 2 3 4 5 6 7 8 9 0 - 19 18 17 16

Pearson

www.pearsonenespañol.com

Para Brian Goetz,
Arquitecto del lenguaje Java de Oracle y
jefe de especificaciones para el proyecto
Lambda de Java SE 8:

Su tutoría nos ayudó a hacer un mejor libro.
Gracias por insistir en que lo hiciéramos bien.

Paul y Harvey Deitel



Contenido

Prólogo xxi

Prefacio xxiii

Antes de empezar xxxvii

1 Introducción a las computadoras, Internet y Java 1

1.1 Introducción 2
1.2 Hardware y software 4
1.2.1 Ley de Moore 4
1.2.2 Organización de la computadora 5
1.3 Jerarquía de datos 6
1.4 Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel 9
1.5 Introducción a la tecnología de los objetos 10
1.5.1 El automóvil como un objeto 10
1.5.2 Métodos y clases 11
1.5.3 Instanciamiento 11
1.5.4 Reutilización 11
1.5.5 Mensajes y llamadas a métodos 11
1.5.6 Atributos y variables de instancia 11
1.5.7 Encapsulamiento y ocultamiento de información 12
1.5.8 Herencia 12
1.5.9 Interfaces 12
1.5.10 Análisis y diseño orientado a objetos (A/DOO) 12
1.5.11 El UML (Lenguaje unificado de modelado) 13
1.6 Sistemas operativos 13
1.6.1 Windows: un sistema operativo propietario 13
1.6.2 Linux: un sistema operativo de código fuente abierto 14
1.6.3 Android 14
1.7 Lenguajes de programación 15
1.8 Java 17
1.9 Un típico entorno de desarrollo en Java 17
1.10 Prueba de una aplicación en Java 21
1.11 Internet y World Wide Web 25
1.11.1 Internet: una red de redes 26
1.11.2 World Wide Web: cómo facilitar el uso de Internet 26
1.11.3 Servicios Web y *mashups* 26

1.11.4 Ajax 27
1.11.5 Internet de las cosas 27
1.12 Tecnologías de software 28
1.13 Cómo estar al día con las tecnologías de información 30

2 Introducción a las aplicaciones en Java: entrada/salida y operadores 34

2.1 Introducción 35
2.2 Su primer programa en Java: impresión de una línea de texto 35
2.3 Edición de su primer programa en Java 41
2.4 Cómo mostrar texto con `printf` 43
2.5 Otra aplicación: suma de enteros 45
2.5.1 Declaraciones `import` 45
2.5.2 Declaración de la clase `Suma` 46
2.5.3 Declaración y creación de un objeto `Scanner` para obtener la entrada del usuario mediante el teclado 46
2.5.4 Declaración de variables para almacenar enteros 47
2.5.5 Cómo pedir la entrada al usuario 48
2.5.6 Cómo obtener un valor `int` como entrada del usuario 48
2.5.7 Cómo pedir e introducir un segundo `int` 49
2.5.8 Uso de variables en un cálculo 49
2.5.9 Cómo mostrar el resultado del cálculo 49
2.5.10 Documentación de la API de Java 49
2.6 Conceptos acerca de la memoria 50
2.7 Aritmética 51
2.8 Toma de decisiones: operadores de igualdad y relacionales 54
2.9 Conclusión 58

3 Introducción a las clases, los objetos, los métodos y las cadenas 69

3.1 Introducción 70
3.2 Variables de instancia, métodos *establecer* y métodos *obtener* 71
3.2.1 La clase `Cuenta` con una variable de instancia, un método *establecer* y un método *obtener* 71
3.2.2 La clase `PruebaCuenta` que crea y usa un objeto de la clase `Cuenta` 74
3.2.3 Compilación y ejecución de una aplicación con varias clases 77
3.2.4 Diagrama de clases en UML de `Cuenta` con una variable de instancia y métodos *establecer* y *obtener* 77
3.2.5 Observaciones adicionales sobre la clase `PruebaCuenta` 78
3.2.6 Ingeniería de software con variables de instancia `private` y métodos *establecer* y *obtener* `public` 79
3.3 Comparación entre tipos primitivos y tipos por referencia 80
3.4 La clase `Cuenta`: inicialización de objetos mediante constructores 81
3.4.1 Declaración de un constructor de `Cuenta` para la inicialización personalizada de objetos 81
3.4.2 La clase `PruebaCuenta`: inicialización de objetos `Cuenta` cuando se crean 82
3.5 La clase `Cuenta` con un saldo: los números de punto flotante 84
3.5.1 La clase `Cuenta` con una variable de instancia llamada `saldo` de tipo `double` 85
3.5.2 La clase `PruebaCuenta` que usa la clase `Cuenta` 86

3.6 (Opcional) Ejemplo práctico de GUI y gráficos: uso de cuadros de diálogo 90
3.7 Conclusión 93

4 Instrucciones de control: parte 1: operadores de asignación, ++ y -- 101

4.1 Introducción 102
4.2 Algoritmos 102
4.3 Seudocódigo 103
4.4 Estructuras de control 103
4.5 Instrucción `if` de selección simple 105
4.6 Instrucción `if...else` de selección doble 106
4.7 Clase `Estudiante`: instrucciones `if...else` anidadas 111
4.8 Instrucción de repetición `while` 113
4.9 Formulación de algoritmos: repetición controlada por un contador 115
4.10 Formulación de algoritmos: repetición controlada por un centinela 119
4.11 Formulación de algoritmos: instrucciones de control anidadas 126
4.12 Operadores de asignación compuestos 131
4.13 Operadores de incremento y decremento 131
4.14 Tipos primitivos 134
4.15 (Opcional) Ejemplo práctico de GUI y gráficos: creación de dibujos simples 135
4.16 Conclusión 139

5 Instrucciones de control: parte 2: operadores lógicos 152

5.1 Introducción 153
5.2 Fundamentos de la repetición controlada por contador 153
5.3 Instrucción de repetición `for` 155
5.4 Ejemplos sobre el uso de la instrucción `for` 159
5.5 Instrucción de repetición `do...while` 163
5.6 Instrucción de selección múltiple `switch` 165
5.7 Ejemplo práctico de la clase `PolizaAuto`: objetos `String` en instrucciones `switch` 171
5.8 Instrucciones `break` y `continue` 174
5.9 Operadores lógicos 176
5.10 Resumen sobre programación estructurada 182
5.11 (Opcional) Ejemplo práctico de GUI y gráficos: dibujo de rectángulos y óvalos 187
5.12 Conclusión 190

6 Métodos: un análisis más detallado 200

6.1 Introducción 201
6.2 Módulos de programas en Java 201
6.3 Métodos `static`, campos `static` y la clase `Math` 203
6.4 Declaración de métodos con múltiples parámetros 205
6.5 Notas sobre cómo declarar y utilizar los métodos 208
6.6 La pila de llamadas a los métodos y los marcos de pila 209
6.7 Promoción y conversión de argumentos 210
6.8 Paquetes de la API de Java 211
6.9 Ejemplo práctico: generación de números aleatorios seguros 213
6.10 Ejemplo práctico: un juego de probabilidad; introducción a los tipos `enum` 218

6.11 Alcance de las declaraciones 222
6.12 Sobrecarga de métodos 225
6.13 (Opcional) Ejemplo práctico de GUI y gráficos: colores y figuras rellenas 227
6.14 Conclusión 230

7 Arreglos y objetos ArrayList 243
7.1 Introducción 244
7.2 Arreglos 245
7.3 Declaración y creación de arreglos 246
7.4 Ejemplos sobre el uso de los arreglos 247
7.4.1 Creación e inicialización de un arreglo 247
7.4.2 Uso de un inicializador de arreglos 248
7.4.3 Cálculo de los valores a almacenar en un arreglo 249
7.4.4 Suma de los elementos de un arreglo 251
7.4.5 Uso de gráficos de barra para mostrar en forma gráfica los datos de un arreglo 251
7.4.6 Uso de los elementos de un arreglo como contadores 253
7.4.7 Uso de arreglos para analizar los resultados de una encuesta 254
7.5 Manejo de excepciones: procesamiento de la respuesta incorrecta 256
7.5.1 La instrucción `try` 256
7.5.2 Ejecución del bloque `catch` 256
7.5.3 El método `toString` del parámetro de excepción 257
7.6 Ejemplo práctico: simulación para barajar y repartir cartas 257
7.7 Instrucción `for` mejorada 262
7.8 Paso de arreglos a los métodos 263
7.9 Comparación entre paso por valor y paso por referencia 265
7.10 Ejemplo práctico: la clase `LibroCalificaciones` que usa un arreglo para almacenar las calificaciones 266
7.11 Arreglos multidimensionales 272
7.12 Ejemplo práctico: la clase `LibroCalificaciones` que usa un arreglo bidimensional 275
7.13 Listas de argumentos de longitud variable 281
7.14 Uso de argumentos de línea de comandos 283
7.15 La clase `Arrays` 285
7.16 Introducción a las colecciones y la clase `ArrayList` 287
7.17 (Opcional) Ejemplo práctico de GUI y gráficos: cómo dibujar arcos 291
7.18 Conclusión 294

8 Clases y objetos: un análisis más detallado 315
8.1 Introducción 316
8.2 Ejemplo práctico de la clase `Tiempo` 316
8.3 Control del acceso a los miembros 321
8.4 Referencias a los miembros del objeto actual mediante la referencia `this` 322
8.5 Ejemplo práctico de la clase `Tiempo`: constructores sobrecargados 324
8.6 Constructores predeterminados y sin argumentos 330
8.7 Observaciones acerca de los métodos *Establecer* y *Obtener* 330
8.8 Composición 332
8.9 Tipos `enum` 335
8.10 Recolección de basura 337
8.11 Miembros de clase `static` 338
8.12 Declaración de importación `static` 342
8.13 Variables de instancia `final` 343

8.14 Acceso a paquetes 344
8.15 Uso de `BigDecimal` para cálculos monetarios precisos 345
8.16 (Opcional) Ejemplo práctico de GUI y gráficos: uso de objetos con gráficos 348
8.17 Conclusión 352

9 Programación orientada a objetos: herencia 360

9.1 Introducción 361
9.2 Superclases y subclases 362
9.3 Miembros `protected` 364
9.4 Relación entre las superclases y las subclases 365
9.4.1 Creación y uso de una clase `EmpleadoPorComision` 365
9.4.2 Creación y uso de una clase `EmpleadoBaseMasComision` 371
9.4.3 Creación de una jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` 376
9.4.4 La jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` mediante el uso de variables de instancia `protected` 379
9.4.5 La jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` mediante el uso de variables de instancia `private` 382
9.5 Los constructores en las subclases 387
9.6 La clase `Object` 387
9.7 (Opcional) Ejemplo práctico de GUI y gráficos: mostrar texto e imágenes usando etiquetas 388
9.8 Conclusión 391

10 Programación orientada a objetos: polimorfismo e interfaces 395

10.1 Introducción 396
10.2 Ejemplos del polimorfismo 398
10.3 Demostración del comportamiento polimórfico 399
10.4 Clases y métodos abstractos 401
10.5 Ejemplo práctico: sistema de nómina utilizando polimorfismo 404
10.5.1 La superclase abstracta `Empleado` 405
10.5.2 La subclase concreta `EmpleadoAsalariado` 407
10.5.3 La subclase concreta `EmpleadoPorHoras` 409
10.5.4 La subclase concreta `EmpleadoPorComision` 411
10.5.5 La subclase concreta indirecta `EmpleadoBaseMasComision` 413
10.5.6 El procesamiento polimórfico, el operador `instanceof` y la conversión descendente 414
10.6 Asignaciones permitidas entre variables de la superclase y la subclase 419
10.7 Métodos y clases `final` 419
10.8 Una explicación más detallada de los problemas con las llamadas a métodos desde los constructores 420
10.9 Creación y uso de interfaces 421
10.9.1 Desarrollo de una jerarquía `PorPagar` 422
10.9.2 La interfaz `PorPagar` 423
10.9.3 La clase `Factura` 424
10.9.4 Modificación de la clase `Empleado` para implementar la interfaz `PorPagar` 426
10.9.5 Modificación de la clase `EmpleadoAsalariado` para usarla en la jerarquía `PorPagar` 428

10.9.6 Uso de la interfaz `PorPagar` para procesar objetos `Factura` y `Empleado` mediante el polimorfismo 430
10.9.7 Algunas interfaces comunes de la API de Java 431
10.10 Mejoras a las interfaces de Java SE 8 432
10.10.1 Métodos `default` de una interfaz 432
10.10.2 Métodos `static` de una interfaz 433
10.10.3 Interfaces funcionales 433
10.11 (Opcional) Ejemplo práctico de GUI y gráficos: realización de dibujos mediante el polimorfismo 433
10.12 Conclusión 436

11 Manejo de excepciones: un análisis más detallado 441
11.1 Introducción 442
11.2 Ejemplo: división entre cero sin manejo de excepciones 443
11.3 Ejemplo: manejo de excepciones tipo `ArithmeticException` e `InputMismatchException` 445
11.4 Cuándo utilizar el manejo de excepciones 451
11.5 Jerarquía de excepciones en Java 451
11.6 Bloque `finally` 454
11.7 Limpieza de la pila y obtención de información de un objeto excepción 459
11.8 Excepciones encadenadas 461
11.9 Declaración de nuevos tipos de excepciones 464
11.10 Precondiciones y poscondiciones 465
11.11 Aserciones 465
11.12 Cláusula `try` con recursos: desasignación automática de recursos 467
11.13 Conclusión 467

A Tabla de precedencia de operadores A-3

B Conjunto de caracteres ASCII A-5

C Palabras clave y palabras reservadas A-6

D Tipos primitivos A-7

E Uso del depurador A-8
E.1 Introducción A-9
E.2 Los puntos de interrupción y los comandos `run`, `stop`, `cont` y `print` A-9
E.3 Los comandos `print` y `set` A-13
E.4 Cómo controlar la ejecución mediante los comandos `step`, `step up` y `next` A-15
E.5 El comando `watch` A-18
E.6 El comando `clear` A-20
E.7 Conclusión A-23

Índice analítico I-1

LOS CAPÍTULOS 12 A 19 SE ENCUENTRAN DISPONIBLES, EN ESPAÑOL, EN EL SITIO WEB DE ESTE LIBRO

12 Componentes de la GUI: parte 1 473

12.1 Introducción 474
12.2 La apariencia visual Nimbus de Java 475
12.3 Entrada/salida simple basada en GUI con `JOptionPane` 476
12.4 Generalidades de los componentes de Swing 479
12.5 Presentación de texto e imágenes en una ventana 481
12.6 Campos de texto y una introducción al manejo de eventos con clases anidadas 485
12.7 Tipos de eventos comunes de la GUI e interfaces de escucha 491
12.8 Cómo funciona el manejo de eventos 493
12.9 `JButton` 495
12.10 Botones que mantienen el estado 498
12.10.1 `JCheckBox` 499
12.10.2 `JRadioButton` 501
12.11 `JComboBox`: uso de una clase interna anónima para el manejo de eventos 504
12.12 `JList` 508
12.13 Listas de selección múltiple 511
12.14 Manejo de eventos de ratón 513
12.15 Clases adaptadoras 518
12.16 Subclase de `JPanel` para dibujar con el ratón 522
12.17 Manejo de eventos de teclas 525
12.18 Introducción a los administradores de esquemas 528
12.18.1 `FlowLayout` 530
12.18.2 `BorderLayout` 532
12.18.3 `GridLayout` 536
12.19 Uso de paneles para administrar esquemas más complejos 538
12.20 `JTextArea` 539
12.21 Conclusión 542

13 Gráficos y Java 2D 555

13.1 Introducción 556
13.2 Contextos y objetos de gráficos 558
13.3 Control de colores 559
13.4 Manipulación de tipos de letra 566
13.5 Dibujo de líneas, rectángulos y óvalos 571
13.6 Dibujo de arcos 575
13.7 Dibujo de polígonos y polilíneas 578
13.8 La API Java 2D 581
13.9 Conclusión 588

14 Cadenas, caracteres y expresiones regulares 596

14.1 Introducción 597
14.2 Fundamentos de los caracteres y las cadenas 597
14.3 La clase `String` 598
14.3.1 Constructores de `String` 598

14.3.2 Métodos `length`, `charAt` y `getChars` de `String` 599
14.3.3 Comparación entre cadenas 600
14.3.4 Localización de caracteres y subcadenas en las cadenas 605
14.3.5 Extracción de subcadenas de las cadenas 607
14.3.6 Concatenación de cadenas 608
14.3.7 Métodos varios de `String` 608
14.3.8 Método `valueOf` de `String` 610
14.4 La clase `StringBuilder` 611
14.4.1 Constructores de `StringBuilder` 612
14.4.2 Métodos `length`, `capacity`, `setLength` y `ensureCapacity` de `StringBuilder` 612
14.4.3 Métodos `charAt`, `setCharAt`, `getChars` y `reverse` de `StringBuilder` 614
14.4.4 Métodos `append` de `StringBuilder` 615
14.4.5 Métodos de inserción y eliminación de `StringBuilder` 617
14.5 La clase `Character` 618
14.6 División de objetos `String` en tokens 623
14.7 Expresiones regulares, la clase `Pattern` y la clase `Matcher` 624
14.8 Conclusión 633

15 Archivos, flujos y serialización de objetos 644

15.1 Introducción 645
15.2 Archivos y flujos 645
15.3 Uso de clases e interfaces NIO para obtener información de archivos y directorios 647
15.4 Archivos de texto de acceso secuencial 651
15.4.1 Creación de un archivo de texto de acceso secuencial 651
15.4.2 Cómo leer datos de un archivo de texto de acceso secuencial 655
15.4.3 Caso de estudio: un programa de solicitud de crédito 657
15.4.4 Actualización de archivos de acceso secuencial 661
15.5 Serialización de objetos 662
15.5.1 Creación de un archivo de acceso secuencial mediante el uso de la serialización de objetos 663
15.5.2 Lectura y deserialización de datos de un archivo de acceso secuencial. 668
15.6 Abrir archivos con `JFileChooser` 670
15.7 (Opcional) Clases adicionales de `java.io` 673
15.7.1 Interfaces y clases para entrada y salida basada en bytes 673
15.7.2 Interfaces y clases para entrada y salida basada en caracteres 675
15.8 Conclusión 676

16 Colecciones de genéricos 684

16.1 Introducción 685
16.2 Generalidades acerca de las colecciones 685
16.3 Clases de envoltura de tipos 687
16.4 Autoboxing y auto-unboxing 687
16.5 La interfaz Collection y la clase Collections 687
16.6 Listas 688
16.6.1 `ArrayList` e `Iterator` 689
16.6.2 `LinkedList` 691
16.7 Métodos de las colecciones 696
16.7.1 El método `sort` 697
16.7.2 El método `shuffle` 700

16.7.3 Los métodos `reverse`, `fill`, `copy`, `max` y `min` 702
16.7.4 El método `binarySearch` 704
16.7.5 Los métodos `addAll`, `frequency` y `disjoint` 706
16.8 La clase `Stack` del paquete `java.util` 708
16.9 La clase `PriorityQueue` y la interfaz `Queue` 710
16.10 Conjuntos 711
16.11 Mapas 714
16.12 La clase `Properties` 718
16.13 Colecciones sincronizadas 721
16.14 Colecciones no modificables 721
16.15 Implementaciones abstractas 722
16.16 Conclusión 722

17 Lambdas y flujos de Java SE 8 729

17.1 Introducción 730
17.2 Generalidades acerca de las tecnologías de programación funcional 731
17.2.1 Interfaces funcionales 732
17.2.2 Expresiones lambda 733
17.2.3 Flujos 734
17.3 Operaciones `IntStream` 736
17.3.1 Creación de un `IntStream` y visualización de sus valores con la operación terminal `forEach` 738
17.3.2 Operaciones terminales `count`, `min`, `max`, `sum` y `average` 739
17.3.3 Operación terminal `reduce` 739
17.3.4 Operaciones intermedias: filtrado y ordenamiento de valores `IntStream` 741
17.3.5 Operación intermedia: asignación (mapping) 742
17.3.6 Creación de flujos de valores `int` con los métodos `range` y `rangeClosed` de `IntStream` 743
17.4 Manipulaciones `Stream<Integer>` 743
17.4.1 Crear un `Stream<Integer>` 744
17.4.2 Ordenar un `Stream` y recolectar los resultados 745
17.4.3 Filtrar un `Stream` y ordenar los resultados para su uso posterior 745
17.4.4 Filtrar y ordenar un `Stream` y recolectar los resultados 745
17.4.5 Ordenar los resultados recolectados previamente 745
17.5 Manipulciones Stream<String> 746
17.5.1 Asignar objetos `String` a mayúsculas mediante una referencia a un método 747
17.5.2 Filtrar objetos `String` y luego ordenarlos en forma ascendente sin distinguir entre mayúsculas y minúsculas 748
17.5.3 Filtrar objetos String y luego ordenarlos en forma descendente sin distinguir entre mayúsculas y minúsculas 748
17.6 Manipulaciones `Stream<Empleado>` 748
17.6.1 Crear y mostrar una `List<Empleado>` 750
17.6.2 Filtrar objetos `Empleado` con salarios en un rango especificado 751
17.6.3 Ordenar objetos `Empleado` con base en varios campos 752
17.6.4 Asignar objetos `Empleado` a objetos `String` con apellidos únicos 754
17.6.5 Agrupar objetos `Empleado` por departamento 755
17.6.6 Contar el número de objetos `Empleado` en cada departamento 756
17.6.7 Sumar y promediar los salarios de los objetos `Empleado` 756
17.7 Crear un objeto `Stream<String>` a partir de un archivo 758

17.8 Generar flujos de valores aleatorios 761
17.9 Manejadores de eventos con lambdas 763
17.10 Observaciones adicionales sobre las interfaces de Java SE 8 763
17.11 Recusos de Java SE 8 y de programación funcional 764
17.12 Conclusión 764

18 Recursividad 776

18.1 Introducción 777
18.2 Conceptos de recursividad 778
18.3 Ejemplo de uso de recursividad: factoriales 779
18.4 Reimplementar la clase `CalculadoraFactorial` mediante la clase `BigInteger` 781
18.5 Ejemplo de uso de recursividad: serie de Fibonacci 783
18.6 La recursividad y la pila de llamadas a métodos 786
18.7 Comparación entre recursividad e iteración 787
18.8 Las torres de Hanoi 789
18.9 Fractales 791
18.9.1 Fractal de curva de Koch 791
18.9.2 (Opcional) Ejemplo práctico: fractal de pluma Lo 792
18.10 "Vuelta atrás" recursiva (backtracking) 801
18.11 Conclusión 802

19 Búsqueda, ordenamiento y Big O 810

19.1 Introducción 811
19.2 Búsqueda lineal 812
19.3 Notación Big O 814
19.3.1 Algoritmos O(1) 814
19.3.2 Algoritmos O(n) 815
19.3.3 Algoritmos O(n^2) 815
19.3.4 El Big O de la búsqueda lineal 816
19.4 Búsqueda binaria 816
19.4.1 Implementación de la búsqueda binaria 817
19.4.2 Eficiencia de la búsqueda binaria 820
19.5 Algoritmos de ordenamiento 820
19.6 Ordenamiento por selección 821
19.6.1 Implementación del ordenamiento por selección 821
19.6.2 Eficiencia del ordenamiento por selección 824
19.7 Ordenamiento por inserción 824
19.7.1 Implementación del ordenamiento por inserción 825
19.7.2 Eficiencia del ordenamiento por inserción 827
19.8 Ordenamiento por combinación 827
19.8.1 Implementación del ordenamiento por combinación 828
19.8.2 Eficiencia del ordenamiento por combinación 832
19.9 Resumen de Big O para los algoritmos de búsqueda y ordenamiento de este capítulo 833
19.10 Conclusión 834

Los capítulos 20 a 34 y los apéndices F a N se encuentran disponibles, en idioma inglés, en el sitio web de este libro

20 Generic Classes and Methods 839

20.1 Introduction 840
20.2 Motivation for Generic Methods 840
20.3 Generic Methods: Implementation and Compile-Time Translation 842
20.4 Additional Compile-Time Translation Issues: Methods That Use a Type Parameter as the Return Type 845
20.5 Overloading Generic Methods 848
20.6 Generic Classes 849
20.7 Raw Types 856
20.8 Wildcards in Methods That Accept Type Parameters 860
20.9 Wrap-Up 864

21 Custom Generic Data Structures 869

21.1 Introduction 870
21.2 Self-Referential Classes 871
21.3 Dynamic Memory Allocation 871
21.4 Linked Lists 872
21.4.1 Singly Linked Lists 872
21.4.2 Implementing a Generic `List` Class 873
21.4.3 Generic Classes `ListNode` and `List` 878
21.4.4 Class `ListTest` 878
21.4.5 `List` Method `insertAtFront` 878
21.4.6 `List` Method `insertAtBack` 879
21.4.7 `List` Method `removeFromFront` 880
21.4.8 `List` Method `removeFromBack` 881
21.4.9 `List` Method `print` 882
21.4.10 Creating Your Own Packages 882
21.5 Stacks 886
21.6 Queues 890
21.7 Trees 893
21.8 Wrap-Up 900

22 GUI Components: Part 2 911

22.1 Introduction 912
22.2 `JSlider` 912
22.3 Understanding Windows in Java 916
22.4 Using Menus with Frames 917
22.5 `JPopupMenu` 925
22.6 Pluggable Look-and-Feel 928

22.7 JDesktopPane and JInternalFrame 933
22.8 JTabbedPane 936
22.9 BoxLayout Layout Manager 938
22.10 GridBagLayout Layout Manager 942
22.11 Wrap-Up 952

23 Concurrency 957
23.1 Introduction 958
23.2 Thread States and Life Cycle 960
23.2.1 New and Runnable States 961
23.2.2 Waiting State 961
23.2.3 Timed Waiting State 961
23.2.4 Blocked State 961
23.2.5 Terminated State 961
23.2.6 Operating-System View of the Runnable State 962
23.2.7 Thread Priorities and Thread Scheduling 962
23.2.8 Indefinite Postponement and Deadlock 963
23.3 Creating and Executing Threads with the Executor Framework 963
23.4 Thread Synchronization 967
23.4.1 Immutable Data 968
23.4.2 Monitors 968
23.4.3 Unsynchronized Mutable Data Sharing 969
23.4.4 Synchronized Mutable Data Sharing—Making Operations Atomic 974
23.5 Producer/Consumer Relationship without Synchronization 976
23.6 Producer/Consumer Relationship: ArrayBlockingQueue 984
23.7 (Advanced) Producer/Consumer Relationship with synchronized, wait, notify and notifyAll 987
23.8 (Advanced) Producer/Consumer Relationship: Bounded Buffers 994
23.9 (Advanced) Producer/Consumer Relationship: The Lock and Condition Interfaces 1002
23.10 Concurrent Collections 1009
23.11 Multithreading with GUI: SwingWorker 1011
23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers 1012
23.11.2 Processing Intermediate Results: Sieve of Eratosthenes 1018
23.12 sort and parallelSort Timings with the Java SE 8 Date/Time API 1025
23.13 Java SE 8: Sequential vs. Parallel Streams 1027
23.14 (Advanced) Interfaces Callable and Future 1030
23.15 (Advanced) Fork/Join Framework 1034
23.16 Wrap-Up 1034

24 Accessing Databases with JDBC 1045
24.1 Introduction 1046
24.2 Relational Databases 1047
24.3 A books Database 1048
24.4 SQL 1052
24.4.1 Basic SELECT Query 1052
24.4.2 WHERE Clause 1053
24.4.3 ORDER BY Clause 1055
24.4.4 Merging Data from Multiple Tables: INNER JOIN 1056
24.4.5 INSERT Statement 1058

24.4.6 `UPDATE` Statement 1059
24.4.7 `DELETE` Statement 1060
24.5 Setting up a Java DB Database 1060
24.5.1 Creating the Chapter's Databases on Windows 1061
24.5.2 Creating the Chapter's Databases on Mac OS X 1062
24.5.3 Creating the Chapter's Databases on Linux 1063
24.6 Manipulating Databases with JDBC 1063
24.6.1 Connecting to and Querying a Database 1063
24.6.2 Querying the `books` Database 1067
24.7 `RowSet` Interface 1080
24.8 `PreparedStatements` 1082
24.9 Stored Procedures 1098
24.10 Transaction Processing 1098
24.11 Wrap-Up 1099

25 JavaFX GUI: Part 1 1107
25.1 Introduction 1108
25.2 JavaFX Scene Builder and the NetBeans IDE 1109
25.3 JavaFX App Window Structure 1110
25.4 **Welcome** App—Displaying Text and an Image 1111
25.4.1 Creating the App's Project 1111
25.4.2 NetBeans **Projects** Window—Viewing the Project Contents 1113
25.4.3 Adding an Image to the Project 1114
25.4.4 Opening JavaFX Scene Builder from NetBeans 1114
25.4.5 Changing to a `VBox` Layout Container 1115
25.4.6 Configuring the `VBox` Layout Container 1116
25.4.7 Adding and Configuring a `Label` 1116
25.4.8 Adding and Configuring an `ImageView` 1116
25.4.9 Running the `Welcome` App 1117
25.5 **Tip Calculator** App—Introduction to Event Handling 1118
25.5.1 Test-Driving the **Tip Calculator** App 1119
25.5.2 Technologies Overview 1119
25.5.3 Building the App's GUI 1122
25.5.4 `TipCalculator` Class 1126
25.5.5 `TipCalculatorController` Class 1128
25.6 Features Covered in the Online JavaFX Chapters 1133
25.7 Wrap-Up 1134

26 JavaFX GUI: Part 2

27 JavaFX Graphics and Multimedia

28 Networking

29 Java Persistence API (JPA)

30 JavaServer™ Faces Web Apps: Part 1

31 JavaServer™ Faces Web Apps: Part 2

32 REST-Based Web Services

33 (Optional) ATM Case Study, Part 1: Object-Oriented Design with the UML

34 (Optional) ATM Case Study, Part 2: Implementing an Object-Oriented Design

F Using the Java API Documentation

G Creating Documentation with `javadoc`

H Unicode®

I Formatted Output

J Number Systems

K Bit Manipulation

L Labeled `break` and `continue` Statements

M UML 2: Additional Diagram Types

N Design Patterns



Prólogo

He estado enamorado de Java incluso antes de que se publicara su versión 1.0 en 1995. Desde entonces he sido desarrollador de Java, autor, ponente, maestro y embajador de tecnología Java (Oracle Java Technology Ambassador). En esta aventura he tenido el privilegio de llamar colega a Paul Deitel, además de aprovechar y recomendar con frecuencia su libro *Cómo programar en Java*. En sus muchas ediciones, este libro ha demostrado ser un excelente texto para cursos universitarios y profesionales que otros y yo hemos desarrollado para enseñar el lenguaje de programación Java.

Una de las cualidades que hacen de este libro un excelente recurso es su cobertura detallada y perspicaz de los conceptos de Java, incluyendo los que se introdujeron recientemente en Java SE 8. Otra cualidad útil es su tratamiento de los conceptos y las prácticas esenciales para un desarrollo de software efectivo.

Como admirador incondicional de este libro, me gustaría señalar algunas de las características que más me han impresionado de esta décima edición:

- Un nuevo y ambicioso capítulo sobre las expresiones lambda y los flujos en Java. Este capítulo comienza con una introducción básica sobre la programación funcional, donde presenta las expresiones lambda de Java y cómo usar los flujos para realizar tareas de programación funcionales en las colecciones.
- Aunque desde la primera edición de este libro se ha lidiado con la concurrencia, es cada vez más importante debido a las arquitecturas multinúcleo. En el capítulo de concurrencia hay ejemplos de sincronización (donde se usan las nuevas clases de la API de fecha/hora que se introdujo en Java SE 8) que muestran las mejoras de rendimiento de la tecnología multinúcleo en comparación con el uso de un solo núcleo.
- JavaFX es la tecnología de GUI/gráficos/multimedia de Java en progreso, por lo que es agradable ver un tratamiento de tres capítulos de JavaFX en el estilo pedagógico de código activo de Deitel. Uno de estos capítulos es parte del libro impreso y los otros dos están en el sitio Web.

Les pido que se unan conmigo para felicitar a Paul y Harvey Deitel por su más reciente edición de un maravilloso recurso, tanto para los estudiantes de ciencias computacionales como para los desarrolladores de software.

James L. Weaver
Java Technology Ambassador
Oracle Corporation



Prefacio

"El principal mérito del lenguaje es la claridad…"
—Galen

Bienvenido al lenguaje de programación Java y al libro *Cómo programar en Java*, *décima edición*. Este libro presenta las tecnologías computacionales de vanguardia para estudiantes, instructores y desarrolladores de software. Es apropiado para cursos académicos introductorios y para cursos profesionales basados 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", en el cual los conceptos se presentan en el contexto de programas funcionales completos que se ejecutan en las versiones recientes de Windows®, OS X® y Linux®, en vez de hacerlo a través de fragmentos separados de código. Cada ejemplo de código completo viene acompañado de ejemplos de ejecuciones reales.

Cómo contactar a los autores

Si surge alguna duda o pregunta a medida que lea este libro, envíe un correo electrónico a

```
deitel@deitel.com
```

y le responderemos a la brevedad. Para obtener actualizaciones sobre este libro, visite

```
http://www.deitel.com/books/jhtp10
```

también puede suscribirse al boletín de correo electrónico *Deitel® Buzz Online* en

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

y puede unirse a las comunidades de redes sociales de Deitel en

- Facebook® (`http://www.deitel.com/deitelfan`)
- Twitter® (`@deitel`)
- Google+™ (`http://google.com/+DeitelFan`)
- YouTube® (`http://youtube.com/DeitelTV`)
- LinkedIn® (`http://linkedin.com/company/deitel-&-associates`)

Código fuente

Encontrará todo el código fuente utilizado en esta edición en español en:

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

Notas de video (VideoNotes)

Encontrará los videos indicados en el libro (en inglés) en:

```
http://www.deitel.com/books/jhtp10
```

lo invitamos a que visite el sitio Web del libro en:

```
http://www.pearsonenespañol.com/deitel
```

Organización modular

Cómo programar en Java, 10 edición es apropiado para cursos de programación en diversos niveles, en especial los de ciencias computacionales CS 1 y CS 2, además de aquellos introductorios en disciplinas relacionadas. La estructura modular del libro ayuda a los instructores a organizar sus planes de estudios:

Introducción

- Capítulo 1, Introducción a las computadoras, Internet y Java
- Capítulo 2, Introducción a las aplicaciones en Java: entrada/salida y operadores
- Capítulo 3, Introducción a las clases, los objetos, los métodos y las cadenas

Fundamentos adicionales de programación

- Capítulo 4, Instrucciones de control: parte 1: operadores de asignación, ++ y –
- Capítulo 5, Instrucciones de control: parte 2: operadores lógicos
- Capítulo 6, Métodos: un análisis más detallado
- Capítulo 7, Arreglos y objetos `ArrayList`
- Capítulo 14 (en línea), Cadenas, caracteres y expresiones regulares
- Capítulo 15 (en línea), Archivos, flujos y serialización de objetos

Programación orientada a objetos y diseño orientado a objetos

- Capítulo 8, Clases y objetos: un análisis más detallado
- Capítulo 9, Programación orientada a objetos: herencia
- Capítulo 10, Programación orientada a objetos: polimorfismo e interfaces
- Capítulo 11, Manejo de excepciones: un análisis más detallado
- Capítulo 33 (en línea), Ejemplo práctico opcional del cajero automático (ATM), parte 1: Diseño orientado a objetos con el UML (en inglés)
- Capítulo 34 (en línea), Ejemplo práctico opcional del cajero automático (ATM), parte 2: Implementación de un diseño orientado a objetos (en inglés)

Interfaces gráficas de usuario de Swing y gráficos de Java 2D

- Capítulo 12 (en línea), Componentes de la GUI: parte 1
- Capítulo 13 (en línea), Gráficos y Java 2D
- Capítulo 22 (en línea), Componentes de GUI: parte 2 (en inglés)

Estructuras de datos, colecciones, lambdas y flujos

- Capítulo 16 (en línea), Colecciones de genéricos
- Capítulo 17 (en línea), Lambdas y flujos de Java SE 8
- Capítulo 18 (en línea), Recursividad
- Capítulo 19 (en línea), Búsqueda, ordenamiento y Big O
- Capítulo 20 (en línea), Clases y métodos genéricos (en inglés)
- Capítulo 21 (en línea), Estructuras de datos genéricas personalizadas (en inglés)

Concurrencia: redes

- Capítulo 23 (en línea), Concurrencia (en inglés)
- Capítulo 28 (en línea), Redes (en inglés)

Interfaces gráficas de usuario, gráficos y multimedia de JavaFX

- Capítulo 25 (en línea), GUI JavaFX: parte 1
- Capítulo 26 (en línea), GUI JavaFX: parte 2
- Capítulo 27 (en línea), Gráficos y multimedia de JavaFX

Desarrollo Web y de escritorio orientado a bases de datos

- Capítulo 24 (en línea), Acceso a bases de datos con JDBC
- Capítulo 29 (en línea), API de persistencia de Java (JPA)
- Capítulo 30 (en línea), Aplicaciones Web de JavaServer™ Faces: parte 1
- Capítulo 31 (en línea), Aplicaciones Web de JavaServer™ Faces: parte 2
- Capítulo 32 (en línea), Servicios Web basados en REST

Características nuevas y mejoradas del libro

He aquí las actualizaciones que realizamos a *Cómo programar en Java, décima edición*:

Java Standard Edition: Java SE 7 y el nuevo Java SE 8

- ***Fácil de usar con Java SE 7 o Java SE 8.*** Para satisfacer las necesidades de nuestros usuarios, diseñamos el libro para cursos tanto universitarios como profesionales que tengan como base Java SE 7, Java SE 8 o una mezcla de ambos. Las características de Java SE 8 se cubren en secciones opcionales que pueden incluirse u omitirse con facilidad. Las nuevas herramientas de Java SE 8 pueden mejorar notablemente el proceso de programación. La figura 1 muestra algunas de las nuevas características que cubrimos de Java SE 8.

Características de Java SE 8
Expresiones lambda
Mejoras en la inferencia de tipos
Anotación `@FunctionalInterface`
Ordenamiento de arreglos en paralelo
Operaciones de datos a granel para colecciones de Java: `filter`, `map` y `reduce`
Mejoras en la biblioteca para el soporte de lambdas (por ejemplo, `java.util.stream`, `java.util.function`)
API de fecha y hora (`java.time`)
Mejoras en la API de concurrencia de Java
Métodos `static` y `default` en interfaces
Interfaces funcionales: interfaces que definen sólo un método `abstract` y pueden incluir métodos `static` y `default`
Mejoras en JavaFX

Fig. 1 | Algunas características nuevas de Java SE 8.

- ***Lambdas, flujos e interfaces de Java SE 8 con métodos `default` y `static`.*** Las características nuevas más importantes en Java SE 8 son las lambdas y las tecnologías complementarias, que cubrimos con detalle en el capítulo 17 opcional y en las secciones opcionales identificadas como "Java SE 8" en capítulos posteriores. En el capítulo 17 (en línea) verá que la programación funcional con lambdas y flujos puede ayudarle a escribir programas de manera más rápida, concisa y simple, con menos errores y que sean más fáciles de paralelizar (para obtener mejoras de rendimiento en sistemas multinúcleo) que los programas escritos con las técnicas anteriores. Descubrirá que la programación funcional complementa a la programación orientada a objetos. Después de que lea el capítulo 17, podrá reimplementar de manera inteligente muchos de los ejemplos de Java SE 7 del libro (figura 2).

Temas anteriores a Java SE 8	Explicaciones y ejemplos correspondientes de Java SE 8
Capítulo 7, Arreglos y objetos `ArrayList`	Las secciones 17.3 a 17.4 presentan herramientas básicas de lambdas y flujos que procesan arreglos unidimensionales.
Capítulo 10, Programación orientada a objetos: polimorfismo e interfaces	La sección 10.10 presenta las nuevas características de interfaces de Java SE 8 (métodos `default`, métodos `static` y el concepto de interfaces funcionales) que dan soporte a la programación funcional con lambdas y flujos.
Capítulos 12 y 22, Componentes de GUI: parte 1 y parte 2, respectivamente	La sección 17.9 muestra cómo usar una expresión lambda para implementar una interfaz funcional de componente de escucha de eventos de Swing.
Capítulo 14, Cadenas, caracteres y expresiones regulares	La sección 17.5 muestra cómo usar lambdas y flujos para procesar colecciones de objetos `String`.
Capítulo 15, Archivos, flujos y serialización de objetos	La sección 17.7 muestra cómo usar lambdas y flujos para procesar líneas de texto de un archivo.
Capítulo 23, Concurrencia	Muestra que los programas funcionales son más fáciles de paralelizar para que puedan aprovechar las arquitecturas multinúcleo y mejorar el rendimiento. Demuestra el procesamiento de flujos en paralelo. Muestra que el método `parallelSort` de `Arrays` mejora el rendimiento en arquitecturas multinúcleo al ordenar arreglos de gran tamaño.
Capítulo 25, GUI JavaFX: parte 1	La sección 25.5.5 muestra cómo usar una expresión lambda para implementar una interfaz funcional para un componente de escucha de eventos de JavaFX.

Fig. 2 | Explicaciones y ejemplos de lambdas y flujos de Java SE 8.

- ***La instrucción `try` con recursos y la interfaz `AutoClosable` de Java SE 7.*** Los objetos `AutoClosable` reducen la probabilidad de que haya fugas de recursos si los usa con la instrucción `try` con recursos, la cual cierra de manera automática los objetos `AutoClosable`. En esta edición usamos `try` con recursos y objetos `AutoClosable` según sea apropiado a partir del capítulo 15, Archivos, flujos y serialización de objetos.
- ***Seguridad de Java.*** Auditamos nuestro libro conforme al estándar CERT de codificación segura de Oracle para Java según lo apropiado para un libro de texto introductorio. Consulte la sección Programación segura en Java de este prefacio para obtener más información sobre CERT.
- ***API NIO de Java.*** Actualizamos los ejemplos de procesamiento de archivos en el capítulo 15 para usar las características de la API NIO (nueva IO) de Java.

- ***Documentación de Java.*** A lo largo del libro le proporcionamos vínculos hacia la documentación de Java, donde aprenderá más sobre los diversos temas que presentamos. Para la documentación de Java SE 7, los vínculos empiezan con

```
http://docs.oracle.com/javase/7/
```

y para la documentación de Java SE 8, los vínculos comienzan con

```
http://download.java.net/jdk8/
```

Estos vínculos pueden cambiar cuando Oracle libere Java SE 8; *es posible* que los vínculos comiencen con

```
http://docs.oracle.com/javase/8/
```

Si hay vínculos que cambien después de publicar el libro, las actualizaciones estarán disponibles en

```
http://www.deitel.com/books/jhtp10
```

GUI Swing y JavaFX, gráficos y multimedia

- ***GUI Swing y gráficos de Java 2D.*** Hablaremos sobre la GUI Swing de Java en las secciones opcionales de GUI y gráficos de los capítulos 3 al 10, y en los capítulos 12 y 22. Ahora Swing se encuentra en modo de mantenimiento: Oracle detuvo el desarrollo y proporcionará sólo corrección de errores de aquí en adelante, pero seguirá siendo parte de Java y aún se utiliza ampliamente. El capítulo 13 habla de los gráficos de Java 2D.
- ***GUI JavaFX, gráficos y multimedia.*** La API de GUI, gráficos y multimedia de Java que continuará progresando es JavaFX. En el capítulo 25 usamos JavaFX 2.2 (que se liberó en 2012) con Java SE 7. Nuestros capítulos 26 y 27 en línea, que se encuentran en el sitio Web complementario del libro (vea la portada interior del libro), presentan características adicionales de la GUI JavaFX e introducen los gráficos y multimedia de JavaFX en el contexto de Java FX 8 y Java SE 8. En los capítulos 25 a 27 usamos Scene Builder, una herramienta de arrastrar y soltar para crear interfaces GUI de JavaFX en forma rápida y conveniente. Es una herramienta independiente que puede usar por separado o con cualquiera de los IDE de Java.
- ***Presentación escalable de GUI y gráficos.*** Los profesores que dictan cursos introductorios tienen una gran variedad de opciones en cuanto a la cantidad por cubrir de GUI, gráficos y multimedia: desde nada en lo absoluto, pasando por las secciones de introducción opcionales en los primeros capítulos, hasta un análisis detallado de la GUI Swing y los gráficos de Java 2D en los capítulos 12, 13 y 22, o incluso un análisis detallado de la GUI Java FX, gráficos y multimedia en el capítulo 25 y los capítulos en línea 26 y 27.

Concurrencia

- ***Concurrencia para un óptimo rendimiento con multinúcleo.*** En esta edición tenemos el privilegio de tener como revisor a Brian Goetz, coautor de *Java Concurrency in Practice* (Addison-Wesley). Actualizamos el capítulo 23 (en línea) con la tecnología y el idioma de Java SE 8. Agregamos un ejemplo de comparación entre `parallelSort` y `sort` que usa la API de hora/fecha de Java SE 8 para sincronizar cada operación y demostrar que `parallelSort` tiene un mejor rendimiento en un sistema multinúcleo. Incluimos un ejemplo de procesamiento de comparación entre flujos paralelos y secuenciales de Java SE 8, usando de nuevo la API de hora/fecha para mostrar las mejoras de rendimiento. Por último, agregamos un ejemplo de `CompletableFuture` de Java SE 8 que demuestra la ejecución secuencial y paralela de cálculos extensos.

- ***Clase `SwingWorker`.*** Usamos la clase `SwingWorker` para crear interfaces de usuario multihilo. En el capítulo 26 (en línea), mostramos cómo es que JavaFX maneja la concurrencia.
- ***La concurrencia es desafiante.*** La programación de aplicaciones concurrentes es difícil y propensa a errores. Hay una gran variedad de características de concurrencia. Señalamos las que deberían usarse más y mencionamos las que deben dejarse a los expertos.

Cómo calcular bien las cantidades monetarias

- ***Cantidades monetarias.*** En los primeros capítulos usamos por conveniencia el tipo `double` para representar cantidades monetarias. Debido a la posibilidad de realizar cálculos monetarios incorrectos con el tipo `double`, hay que usar la clase `BigDecimal` (que es un poco más compleja) para representar las cantidades monetarias. Demostramos el uso de `BigDecimal` en los capítulos 8 y 25.

Tecnología de objetos

- ***Programación y diseño orientados a objetos.*** Usamos la metodología de tratar los *objetos en los primeros capítulos*, donde presentamos la terminología y los conceptos básicos de la tecnología de objetos en el capítulo 1. Los estudiantes desarrollan sus primeras clases y objetos personalizados en el capítulo 3. Al presentar los objetos y las clases en los primeros capítulos, hacemos que los estudiantes "piensen en objetos" de inmediato y que dominen estos conceptos con más profundidad [para 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, 10ª edición*].
- ***Ejemplos prácticos reales sobre el uso de objetos en los primeros capítulos.*** La presentación de clases y objetos en los primeros capítulos incluye ejemplos prácticos con las clases `Cuenta`, `Estudiante`, `AutoPolicy`, `Tiempo`, `Empleado`, `LibroCalificaciones` y un ejemplo práctico sobre barajar y repartir cartas con la clase `Carta`, introduciendo de manera gradual los conceptos de OO más complejos.
- ***Herencia, interfaces, polimorfismo y composición.*** Usamos una serie de ejemplos prácticos reales para ilustrar cada uno de estos conceptos de OO y explicamos situaciones en las que sea conveniente usar cada uno de ellos para crear aplicaciones industriales.
- ***Manejo de excepciones.*** Integramos el manejo básico de excepciones en los primeros capítulos del libro y luego presentamos un análisis más detallado en el capítulo 11. El manejo de excepciones es importante para crear aplicaciones de "misión crítica" e "imprescindibles para los negocios". Los programadores necesitan lidiar con las siguientes dudas: "¿Qué ocurre cuando el componente que invoco para realizar un trabajo experimenta dificultades? ¿Cómo indicará ese componente que tuvo un problema?". Para usar un componente de Java no sólo hay que saber cómo se comporta ese componente cuando "las cosas salen bien", sino también las excepciones que ese componente "lanza" cuando "las cosas salen mal".
- ***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. La extensa selección de ejercicios del capítulo incluye un proyecto importante sobre cómo crear su propia computadora mediante la técnica de simulación por software. El capítulo 21 (en línea) incluye un proyecto de seguimiento sobre la creación de su propio compilador que puede compilar programas en lenguaje de alto nivel a código de lenguaje máquina, y que se ejecutará en su simulador de computadora.
- ***Ejemplo práctico opcional: desarrollo de un diseño orientado a objetos y una implementación en Java de un cajero automático (ATM).*** Los capítulos 33 y 34 (en línea) incluyen un ejemplo práctico *opcional* sobre el diseño orientado a objetos mediante el uso del UML (Lenguaje Unificado de Modelado™): el lenguaje gráfico estándar en la industria para modelar sistemas orientados a objetos. 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, producimos 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.

Estructuras de datos y colecciones genéricas

- ***Presentación de estructuras de datos.*** Empezamos con la clase genérica `ArrayList` en el capítulo 7. Nuestros análisis posteriores sobre las estructuras de datos (capítulos 16 al 21) ofrecen un tratamiento más detallado de las colecciones de genéricos, ya que enseñan a utilizar las colecciones integradas de la API de Java. Hablamos sobre la recursividad, que es importante para implementar las clases de estructuras de datos tipo árbol. Hablamos sobre los algoritmos populares de búsqueda y ordenamiento para manipular el contenido de las colecciones y proporcionamos una introducción amigable a Big O: un medio para describir qué tan duro tendría que trabajar un algoritmo para resolver un problema. Luego mostramos cómo implementar los métodos y las clases genéricas, además de las estructuras de datos genéricas *personalizadas* (esto es un tema para las materias de especialidad en ciencias computacionales; la mayoría de los programadores deben usar las colecciones de genéricos prefabricadas). Las lambdas y los flujos (que se presentan en el capítulo 17) son especialmente útiles para trabajar con colecciones de genéricos.

Base de datos

- ***JDBC.*** El capítulo 24 (en línea) trata sobre JDBC y usa el sistema de administración de bases de datos Java DB. El capítulo presenta el Lenguaje estructurado de consulta (SQL) y un ejemplo práctico de OO sobre cómo desarrollar una libreta de direcciones controlada por una base de datos en donde se demuestran las instrucciones preparadas.
- ***API de persistencia de Java.*** El nuevo capítulo 29 (en línea) cubre la API de persistencia de Java (JPA): un estándar para el mapeo objeto-relacional (ORM) que usa JDBC "tras bambalinas". Las herramientas de ORM pueden analizar el esquema de una base de datos y generar un conjunto de clases que nos permitan interactuar con esta base de datos sin tener que usar JDBC y SQL de manera directa. Esto agiliza el desarrollo de las aplicaciones de bases de datos, reduce los errores y produce código más portable.

Desarrollo de aplicaciones Web

- ***Java Server Faces (JSF).*** Los capítulos 30 y 31 (en línea) se actualizaron para introducir la tecnología JavaServer Faces (JSF) más reciente, que simplifica en gran medida la creación de aplicaciones Web con JSF. El capítulo 30 incluye ejemplos sobre la creación de interfaces GUI para aplicaciones Web, la validación de formularios y el rastreo de sesiones. El capítulo 31 habla sobre las aplicaciones JSF controladas por datos y habilitadas para Ajax. El capítulo incluye una libreta de direcciones Web multinivel controlada por una base de datos, la cual permite a los usuarios agregar y buscar contactos.
- ***Servicios Web.*** El capítulo 32 (en línea) se concentra ahora en la creación y el consumo de servicios Web basados en REST. Ahora la gran mayoría de los servicios Web de la actualidad usan REST.

Programación segura en Java

Es difícil crear sistemas industriales que resistan a los ataques de virus, gusanos y otras formas de "malware". En la actualidad, debido al uso de Internet, esos ataques pueden ser instantáneos y de un alcance global. Al integrar la seguridad en el software desde el principio del ciclo del desarrollo es posible reducir, en gran

medida, las vulnerabilidades. En nuestros análisis y ejemplos de código incorporamos varias prácticas de codificación seguras en Java (en la medida apropiada para un libro de texto introductorio).

El Centro de coordinación del CERT® (`www.cert.org`) se creó para analizar y responder de manera oportuna a los ataques. El CERT (Equipo de respuesta ante emergencias informáticas) es una organización financiada por el gobierno de los Estados Unidos dentro de Carnegie Mellon University Software Engineering Institute™. CERT publica y promueve estándares de codificación segura para diversos lenguajes de programación populares con el fin de ayudar a los desarrolladores de software a implementar sistemas industriales que eviten las prácticas de programación que dejan los sistemas abiertos a los ataques.

Nos gustaría agradecer a Robert C. Seacord, gerente de codificación segura en CERT y profesor adjunto en Carnegie Mellon University School of Computer Science. El Sr. Seacord fue revisor técnico de nuestro libro *Cómo programar en C, séptima edición,* en donde escudriñó nuestros programas en C desde el punto de vista de la seguridad y nos recomendó adherirnos al *estándar de codificación segura en C del CERT*. Esta experiencia también influyó en nuestras prácticas de codificación en *Cómo programar en C++, novena edición,* y en *Cómo programar en Java, décima edición*.

Ejemplo práctico opcional de GUI y gráficos

A los estudiantes les gusta crear aplicaciones de GUI y gráficos. Para los cursos en los que se presentan los temas de GUI y gráficos en las primeras clases, integramos una introducción opcional de 10 segmentos para crear gráficos e interfaces gráficas de usuario (GUI) basadas en Swing. El objetivo de este ejemplo práctico es crear una aplicación polimórfica simple de dibujo en la que el usuario pueda elegir una forma para dibujar, seleccionar las características de esa forma (como su color) y usar el ratón para dibujarla. El ejemplo práctico se desarrolla en forma gradual para llegar a ese objetivo, en donde el lector implementa el dibujo polimórfico en el capítulo 10, agrega una GUI orientada a eventos en el capítulo 12 y mejora las capacidades de dibujo en el capítulo 13 con Java 2D.

- Sección 3.6: Uso de cuadros de diálogo
- Sección 4.15: Creación de dibujos simples
- Sección 5.11: Dibujo de rectángulos y óvalos
- Sección 6.13: Colores y figuras rellenas
- Sección 7.17: Cómo dibujar arcos
- Sección 8.16: Uso de objetos con gráficos
- Sección 9.7: Mostrar texto e imágenes usando etiquetas
- Sección 10.11: Realización de dibujos mediante el polimorfismo
- Ejercicio 12.17: Expansión de la interfaz
- Ejercicio 13.31: Incorporación de Java2D

Métodos de enseñanza

Cómo programar en Java, 10ª edición contiene cientos de ejemplos funcionales completos. Hacemos hincapié en la claridad de los programas y nos concentramos en crear software bien diseñado.

Notas de video (VideoNotes). El sitio Web complementario incluye muchas notas en inglés de video en las que el coautor Paul Deitel explica con detalle la mayoría de los programas de los capítulos básicos del libro. A los estudiantes les gusta ver las notas de video para reforzar los conceptos básicos y para obtener información adicional.

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

Uso de fuentes para dar énfasis. Para facilitar su identificación, ponemos en **negritas** los términos clave y la referencia de la página del índice para cada definición. 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. Todos los ejemplos de código fuente se pueden descargar de:

```
http://www.deitel.com/books/jhtp10
http://www.pearsonenespañol.com/deitel
```

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

Ilustraciones y figuras. Incluimos una gran cantidad de tablas, dibujos lineales, diagramas de 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.



Buena práctica de programación

Las buenas prácticas de programación *llaman la atención hacia técnicas que le ayudarán a producir programas más claros, comprensibles y fáciles de mantener.*



Error común de programación

Al poner atención en estos Errores comunes de programación *se reduce la probabilidad de que usted pueda cometer los mismos errores.*



Tip para prevenir errores

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



Tip de rendimiento

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



Tip de portabilidad

Los tips de portabilidad *le ayudan a escribir código que pueda ejecutarse en una variedad de plataformas.*



Observación de ingeniería de software

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



Observación de apariencia visual

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

Viñetas de resumen. Presentamos un resumen detallado de cada sección del capítulo en un estilo de lista con viñetas. Para facilitar la referencia, incluimos el número de página de cada definición de los términos clave del libro.

Ejercicios de autoevaluación y respuestas. Se incluyen diversos ejercicios de autoevaluación *y sus* respuestas, para que los estudiantes practiquen por su cuenta. Todos los ejercicios en el ejemplo práctico opcional sobre el ATM están totalmente resueltos.

Ejercicios. Los ejercicios de los capítulos incluyen:

- 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, hay ejercicios "Marcando la diferencia" que animan al lector a usar las computadoras e Internet para investigar y resolver problemas sociales importantes.

Los ejercicios que son únicamente para la versión SE 8 de Java se identifican como tales. Consulte nuestro Centro de recursos de proyectos de programación, donde encontrará muchos ejercicios adicionales y posibilidades de proyectos (`www.deitel.com/ProgrammingProjects/`).

Índice. Incluimos un índice extenso. Las entradas que corresponden a las definiciones de los términos clave están resaltadas **en negritas** junto con su número de página. El índice del libro incluye todos los términos del libro impreso y de los capítulos en español disponibles en el sitio web; es decir, incluye los términos de los capítulos 1 a 19.

Software utilizado en *Cómo programar en Java 10ª edición*

Podrá descargar a través de Internet, y sin costo, todo el software necesario para este libro . En la sección Antes de empezar que se encuentra después de este Prefacio, encontrará los 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) 7. Para los módulos opcionales de Java SE 8, utilizamos la versión JDK 8 de prueba de OpenJDK. En el capítulo 25 (en línea) y en varios capítulos también utilizamos el IDE Netbeans. 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 registrados en el Centro de recursos para el profesor de Pearson (`www.pearsonenespañol.com/deitel`):

- ***Diapositivas de PowerPoint®*** que contienen todo el código y las figuras del texto, además de elementos en viñetas que sintetizan los puntos clave.
- ***Banco de exámenes*** con preguntas de opción múltiple (aproximadamente dos por cada sección del libro).
- ***Manual de soluciones*** con soluciones para la gran mayoría de los ejercicios que aparecen al final de capítulo. **Antes de asignar un ejercicio de tarea, los profesores deberán consultar el IRC para asegurarse de que incluya la solución.**

Por favor, no escriba a los autores para solicitar acceso al Centro de recursos para el profesor de Pearson. El acceso está limitado estrictamente a profesores que utilicen este libro como texto en sus cursos y estén registrados en nuestro sistema. Los profesores sólo pueden obtener acceso a través de los representantes de Pearson. No se proveen soluciones para los ejercicios de «proyectos».

Si no es un profesor registrado, póngase en contacto con su representante de Pearson o visite `www.pearsonhighered.com/educator/replocator/`.

Contenido de este libro

Este libro contiene los primeros 11 capítulos mencionados (impresos); adicionalmente, dentro de la página Web www.pearsonenespañol.com/deitel, encontrará, en español, los capítulos 12 a 19; y en inglés los capítulos 20 a 34, así como los apéndices F a N.

Reconocimientos

Queremos agradecer a Abbey Deitel y a Barbara Deitel por las largas horas que dedicaron a este proyecto. Somos afortunados al haber trabajado en este proyecto con un dedicado equipo de editores profesionales de Pearson. Apreciamos la orientación, inteligencia y energía de Tracy Johnson, editora en jefe de ciencias computacionales. Tracy y su equipo se encargan de todos nuestros libros de texto académicos. Carole Snyder reclutó a los revisores técnicos del libro y se hizo cargo del proceso de revisión. Bob Engelhardt se hizo cargo de la publicación del libro. Nosotros seleccionamos la imagen de portada y Laura Gardner diseñó la portada.

Revisores

Queremos agradecer los esfuerzos de nuestros revisores más recientes: un grupo distinguido de profesores, miembros del equipo de Oracle Java, Oracle Java Champions y otros profesionales de la industria. Ellos examinaron minuciosamente el texto y los programas, proporcionando innumerables sugerencias para mejorar la presentación.

Apreciamos la asesoría tanto de Jim Weaver y de Johan Vos (coautores de *Pro JavaFX 2*), así como de Simon Ritter en los tres capítulos sobre JavaFX.

Revisores de la décima edición: Lance Andersen (Oracle Corporation), Dr. Danny Coward (Oracle Corporation), Brian Goetz (Oracle Corporation), Evan Golub (University of Maryland), Dr. Huiwei Guan (profesor del departamento de ciencias computacionales y de la información en North Shore Community College), Manfred Riem (Java Champion), Simon Ritter (Oracle Corporation), Robert C. Seacord (CERT, Software Engineering Institute, Carnegie Mellon University), Khallai Taylor (profesora asistente, Triton College y profesora adjunta, Lonestar College: Kinwood), Jorge Vargas (Yumbling y Java Champion), Johan Vos (LodgON y Oracle Java Champion) y James L. Weaver (Oracle Corporation y autor de *Pro JavaFX 2*).

Revisores de ediciones anteriores: 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), 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 en Austin), Peter Pilgrim (Consultor), Manjeet Rege, Ph.D. (Rochester Institute of Technology), 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).

Un agradecimiento especial a Brian Goetz

Tuvimos el privilegio de que Brian Goetz (arquitecto del lenguaje Java de Oracle, líder de especificaciones para el proyecto Lambda de Java SE 8 y coautor de *Java Concurrency in Practice*) realizara una revisión detallada de todo el libro. Examinó minuciosamente cada capítulo y proporcionó muchas ideas y comentarios constructivos bastante útiles. Cualquier otra falla en el libro es culpa nuestra.

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

```
deitel@deitel.com
```

Le responderemos oportunamente. Esperamos que disfrute trabajando con este libro tanto como nosotros al escribirlo.

Paul y Harvey Deitel

Acerca de los autores



Paul J. Deitel, CEO y Director Técnico de Deitel & Associates, Inc., es egresado del MIT, donde estudió Tecnologías de la Información. Posee las designaciones Java Certified Programmer y Java Certified Developer, además de ser Oracle Java Champion. A través de Deitel & Associates, Inc., ha impartido cientos de cursos de programación para clientes de todo el mundo, incluyendo Cisco, IBM, Siemens, Sun Microsystems, Dell, Fidelity, NASA en el Centro Espacial Kennedy, el National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, SunGard Higher Education, Nortel Networks, Puma, iRobot, Invensys y muchos más. Él y su coautor, el Dr. Harvey M. Deitel, son autores de los libros de texto, libros profesionales y videos sobre lenguajes de programación más vendidos en el mundo.

Dr. Harvey M. Deitel, Presidente y Consejero de Estrategia de Deitel & Associates, Inc., tiene más de 50 años de experiencia en el campo de la computación. El Dr. Deitel obtuvo una licenciatura y una maestría en ingeniería eléctrica del MIT y un doctorado en Matemáticas de la Universidad de Boston. Tiene muchos años de experiencia como profesor universitario, lo cual incluye un puesto vitalicio y haber sido presidente del departamento de Ciencias de la computación en Boston College antes de fundar, Deitel & Associates, Inc. en 1991 junto con su hijo Paul. Los libros de los Deitel se han ganado el reconocimiento internacional y han sido traducidos al japonés, alemán, ruso, español, coreano, francés, polaco, italiano, chino simplificado, chino tradicional, coreano, 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.

Acerca de Deitel & Associates, Inc.

Deitel & Associates, Inc., fundada por Paul Deitel y Harvey Deitel, es una empresa reconocida a nivel mundial dedicada a la capacitación corporativa y la creación de contenidos que se especializa en lenguajes de programación computacionales, tecnología de objetos, desarrollo de aplicaciones móviles y tecnología de software de Internet y Web. Sus clientes de capacitación incluyen muchas de las empresas más grandes del mundo, agencias gubernamentales, sectores del ejército e instituciones académicas. La empresa proporciona cursos presenciales en todo el mundo, sobre la mayoría de los lenguajes y plataformas

de programación, como Java™, desarrollo de aplicaciones para Android™, desarrollo de aplicaciones de Objective-C e iOS, C++, C, Visual C#®, Visual Basic®, Visual C++®, Python®, tecnología de objetos, programación en Internet y Web, y una lista cada vez mayor de cursos adicionales de programación y desarrollo de software.

A lo largo de su sociedad editorial de más de 39 años con Pearson/Prentice Hall, Deitel & Associates, Inc. ha publicado libros de texto de vanguardia sobre programación y libros profesionales tanto impresos como en un amplio rango de formatos de libros electrónicos, además de cursos de video *LiveLessons*. Puede contactarse con Deitel & Associates, Inc. y con los autores por medio de correo electrónico:

```
deitel@deitel.com
```

Para conocer más acerca de la oferta de la Serie de Capacitación Corporativa *Dive Into*® de Deitel, visite:

```
http://www.deitel.com/training/
```

Para solicitar una cotización de capacitación presencial en su organización, envíe un correo a `deitel@deitel.com`.

Las personas que deseen comprar libros de Deitel y cursos de capacitación *LiveLessons* pueden hacerlo a través de `www.deitel.com`. Las empresas, gobiernos, instituciones militares y académicas que deseen realizar pedidos al mayoreo deben hacerlo directamente con Pearson. Para obtener más información, visite

```
http://www.informit.com/store/sales.aspx
```



Antes de empezar

Esta sección contiene información que debe revisar antes de usar este libro. Cualquier actualización a la información que se presenta aquí, será publicada en:

```
http://www.deitel.com/books/jhtp10
```

Convenciones de tipos de letra y nomenclatura

Utilizamos tipos de letra distintos para diferenciar entre los componentes de la pantalla (como los nombres de menús y los elementos de los mismos) y el código o los comandos en Java. Nuestra convención es hacer hincapié en los componentes en pantalla en una fuente **Helvetica** sans-serif en negritas (por ejemplo, el menú **Archivo**) y enfatizar el código y los comandos de Java en una fuente `Lucida` sans-serif (por ejemplo, `System.out.println()`). Hemos omitido intencionalmente el uso de acentos y caracteres especiales en los segmentos de código donde estos caracteres podrían presentar alguna dificultad para la correcta ejecución del programa.

Software a utilizar en el libro

Podrá descargar sin costo todo el software necesario para este libro a través de la Web. Con excepción de los ejemplos específicos para Java SE 8, todos los ejemplos se probaron con los kits de desarrollo Java Standard Edition Development Kits (JDK) de las ediciones Java SE 7 y Java SE 8.

Kit de desarrollo de software Java Standard Edition 7 (JDK 7)
Las plataformas de JDK 7 para Windows, OS X y Linux están disponibles en:

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

Kit de desarrollo de software Java Standard Edition 8 (JDK 8)
Al momento de publicar este libro, la versión más reciente de JDK 8 para las plataformas Windows, OS X y Linux estaba disponible en:

```
http://jdk8.java.net/download.html
```

Una vez que se libere la versión final de JDK 8, estará disponible en:

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

Instrucciones de instalación del JDK

Después de descargar el instalador del JDK, asegúrese de seguir con cuidado las instrucciones de instalación para su plataforma, las cuales están disponibles en:

```
http://docs.oracle.com/javase/7/docs/webnotes/install/index.html
```

Aunque estas instrucciones son para el JDK 7, también se aplican al JDK 8; sólo tendrá que actualizar el número de versión del JDK en las instrucciones específicas de cualquier otra versión.

Cómo establecer la variable de entorno PATH

La variable de entorno PATH en su computadora determina los directorios en los que la computadora debe buscar aplicaciones, como las que le permiten compilar y ejecutar sus aplicaciones de Java (llamadas javac y java, respectivamente). *Para lograr establecer correctamente la variable de entorno PATH, siga con cuidado las instrucciones de instalación de Java en su plataforma.* Los pasos para establecer las variables de entorno difieren según el sistema operativo, y algunas veces dependen también de la versión del mismo (por ejemplo, Windows 7 o Windows 8). Las instrucciones para diversas plataformas se encuentran en:

```
http://www.java.com/en/download/help/path.xml
```

Si no establece la variable PATH de manera correcta en Windows, y en algunas instalaciones de Linux, recibirá un mensaje como éste cuando utilice las herramientas del JDK:

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

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.

Directorio de instalación del JDK y el subdirectorio ***bin***

El directorio de instalación del JDK varía según la plataforma que se utilice. Los directorios que se listan a continuación son para el JDK 7 actualización 51 de Oracle:

- JDK de 32 bits en Windows:
  `C:\Program Files (x86)\Java\jdk1.7.0_51`
- JDK de 64 bits en Windows:
  `C:\Program Files\Java\jdk1.7.0_51`
- Mac OS X:
  `/Library/Java/JavaVirtualMachines/jdk1.7.0_51.jdk/Contents/Home`
- Linux Ubuntu:
  `/usr/lib/jvm/java-7-oracle`

Dependiendo de la plataforma que utilice y del idioma que tenga configurado, el nombre de la carpeta de instalación del JDK podría diferir si usa una actualización distinta del JDK 7 o si usa el JDK 8. En el caso de Linux, la ubicación de instalación depende del instalador que utilice y posiblemente de su versión de Linux. Nosotros usamos Linux Ubuntu. La variable de entorno PATH debe apuntar al subdirectorio ***bin*** del directorio de instalación JDK.

Al establecer PATH, asegúrese de usar el nombre del directorio de instalación del JDK correcto para la versión específica del JDK que instaló; a medida que haya versiones disponibles de JDK más recientes, el nombre del directorio de instalación del JDK cambia para incluir un *número de versión de actualización*. Por ejemplo, al momento de escribir este libro la versión del JDK 7 más reciente era la actualización 51. Para esta versión, el nombre del directorio de instalación del JDK termina con "_51".

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 modificarse. Para corregir el error anterior, 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 (:).

Cómo establecer la variable de entorno `JAVA_HOME`

El software de bases de datos Java DB que usará en el capítulo 24 y en varios de los capítulos en línea, requiere que establezca la variable de entorno `JAVA_HOME` en su directorio de instalación del JDK. Puede usar los mismos pasos que utilizó al establecer la variable `PATH` para establecer las demás variables de entorno, como `JAVA_HOME`.

Entornos integrados de desarrollo (IDE) de Java

Hay muchos entornos integrados de desarrollo de Java que usted puede usar para programar en este lenguaje. Por esta razón, para la mayoría de los ejemplos del libro usaremos sólo las herramientas de línea de comandos del JDK. Proporcionamos videos Dive-Into® que muestran cómo descargar, instalar y usar tres IDE populares: NetBeans, Eclipse e IntelliJ IDEA. Usaremos NetBeans en el capítulo 25 y en varios de los capítulos en línea del libro.

Descargas de NetBeans
Puede descargar el paquete JDK/NetBeans en:

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

La versión de NetBeans incluida con el JDK es para el desarrollo con Java SE. Los capítulos en línea de JavaServer Faces (JSF) y de los servicios Web, usan la versión Java Enterprise Edition (Java EE) de NetBeans, que puede descargar en:

```
https://netbeans.org/downloads/
```

Esta versión soporta el desarrollo con Java SE y Java EE.

Descargas de Eclipse
Puede descargar el IDE Eclipse en:

```
https://www.eclipse.org/downloads/
```

Para el desarrollo con Java SE, seleccione el IDE Eclipse para desarrolladores de Java. Para el desarrollo con Java Enterprise Edition (Java EE) (como JSF y los servicios Web), seleccione el IDE Eclipse para desarrolladores de Java EE; esta versión soporta el desarrollo con Java SE y Java EE.

Descargas de IntelliJ IDEA Community Edition
Puede descargar el entorno IntelliJ IDEA Community Edition gratuito en:

```
http://www.jetbrains.com/idea/
```

La versión gratuita soporta solamente el desarrollo con Java SE.

Cómo obtener los ejemplos de código

Los ejemplos de este libro están disponibles para descargarlos sin costo en

```
http://www.deitel.com/books/jhtp10/
```

bajo el encabezado **Download Code Examples and Other Premium Content** (Descargar ejemplos de código y contenido especial adicional). Los ejemplos también están disponibles en

```
http://www.pearsonhighered.com/deitel
```

Cuando descargue el archivo ZIP en su computadora, anote la ubicación en donde eligió guardarlo.

Extraiga el contenido del archivo examples.zip; utilice una herramienta como 7-Sip (`www.7-zip-org`), WinZip (`www.winzip.com`) o las herramientas integradas de su sistema operativo. Las instrucciones en el libro asumen que los ejemplos se encuentran en:

- `C:\ejemplos en Windows`
- la subcarpeta `ejemplos` de la carpeta de inicio de su cuenta de usuario en Linux
- la subcarpeta `ejemplos` de la carpeta Documentos en Mac OS X

Apariencia visual Nimbus de Java

Java incluye una apariencia visual multiplataforma conocida como Nimbus. En los programas con interfaces gráficas de usuario Swing (como en los capítulos 12 y 22), hemos configurado nuestros equipos de prueba 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, visite `http://docs.oracle.com/javase/7/docs/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, 10ª edición*. ¡Esperamos que disfrute el libro!

Introducción a las computadoras, Internet y 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

Objetivos

En este capítulo aprenderá sobre:

- Los excitantes y recientes desarrollos 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.
- La importancia de Java y los otros lenguajes de programación líderes.
- Los fundamentos de la programación orientada a objetos.
- La importancia de Internet y Web.
- Un entorno de desarrollo típico de programas en Java.
- Cómo probar una aplicación en Java.
- Algunas de las recientes tecnologías de software clave.
- Cómo mantenerse actualizado con las tecnologías de la información.

Plan general

1.1 Introducción

1.2 Hardware y software
- 1.2.1 Ley de Moore
- 1.2.2 Organización de la computadora

1.3 Jerarquía de datos

1.4 Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel

1.5 Introducción a la tecnología de los objetos
- 1.5.1 El automóvil como un objeto
- 1.5.2 Métodos y clases
- 1.5.3 Instanciación
- 1.5.4 Reutilización
- 1.5.5 Mensajes y llamadas a métodos
- 1.5.6 Atributos y variables de instancia
- 1.5.7 Encapsulamiento y ocultamiento de información
- 1.5.8 Herencia
- 1.5.9 Interfaces
- 1.5.10 Análisis y diseño orientado a objetos (A/DOO)
- 1.5.11 El UML (Lenguaje unificado de modelado)

1.6 Sistemas operativos
- 1.6.1 Windows: un sistema operativo propietario
- 1.6.2 Linux: un sistema operativo de código fuente abierto
- 1.6.3 Android

1.7 Lenguajes de programación

1.8 Java

1.9 Un típico entorno de desarrollo en Java

1.10 Prueba de una aplicación en Java

1.11 Internet y World Wide Web
- 1.11.1 Internet: una red de redes
- 1.11.2 World Wide Web: cómo facilitar el uso de Internet
- 1.11.3 Servicios Web y *mashups*
- 1.11.4 Ajax
- 1.11.5 Internet de las cosas

1.12 Tecnologías de software

1.13 Cómo estar al día con las tecnologías de información

Ejercicios de autoevaluación | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Marcando la diferencia*

1.1 Introducción

Bienvenido a Java, uno de los lenguajes de programación de computadoras más utilizados en el mundo. Usted ya está familiarizado con las poderosas tareas que realizan las computadoras. Mediante este libro de texto, podrá escribir instrucciones que ordenen a las computadoras que realicen esos tipos de tareas. El **software** (es decir, las instrucciones que usted escribe) controla el **hardware** (es decir, las computadoras).

Usted aprenderá sobre la *programación orientada a objetos*: la principal metodología de programación en 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.

Forrester Research pronostica que habrá más de dos mil millones de computadoras personales en uso para 2015.[1] De acuerdo con Oracle, el 97% de los equipos de escritorio empresariales, el 89% de las PC de escritorio, tres mil millones de dispositivos (figura 1.1) y el 100% de todos los reproductores Blu-Ray Disc™ ejecutan Java, y hay más de 9 millones de desarrolladores de este lenguaje.[2]

De acuerdo con un estudio realizado por Gartner, los dispositivos móviles seguirán superando a las PC como los dispositivos primarios de los usuarios: se estima que se entregarán alrededor de 2 mil millones de teléfonos inteligentes y 388 millones de tabletas en 2015; 8.7 veces el número de equipos PC.[3] Para 2018,

1 http://www.worldometers.info/computers.
2 http://www.oracle.com/technetwork/articles/java/javaone12review-1863742.html.
3 http://www.gartner.com/newsroom/id/2645115.

Dispositivos		
Sistemas de aeroplanos	Cajeros automáticos (ATM)	Sistemas de infoentretenimiento de automóviles
Reproductores Blu-Ray Disc™	Decodificadores de TV por cable	Copiadoras
Tarjetas de crédito	Escáneres para TC	Computadoras de escritorio
Lectores de libros electrónicos	Consolas de juegos	Sistemas de navegación GPS
Electrodomésticos	Sistemas de seguridad para el hogar	Interruptores de luz
Terminales de lotería	Dispositivos médicos	Teléfonos móviles
MRI	Estaciones de pago de estacionamiento	Impresoras
Pases de transporte	Robots	Enrutadores
Tarjetas inteligentes	Medidores inteligentes	Plumas inteligentes
Teléfonos inteligentes	Tabletas	Televisiones
Decodificadores para televisores	Termostatos	Sistemas de diagnóstico vehicular

Fig. 1.1 | Algunos dispositivos que utilizan Java.

se espera que el mercado de las aplicaciones móviles (apps) llegue a $92 mil millones.[4] Esto genera oportunidades profesionales importantes para las personas que programan aplicaciones móviles, muchas de las cuales se programan en Java (vea la sección 1.6.3).

Java Standard Edition

Java ha evolucionado con tanta rapidez que esta décima edición de *Cómo programar en Java* —basada en **Java Standard Edition 7 (Java SE 7)** y **Java Standard Edition 8 (Java SE 8)**— se publicó tan sólo 17 años después de la primera edición. Java Standard Edition contiene las herramientas necesarias para desarrollar aplicaciones de escritorio y de servidor. El libro puede usarse *ya sea* con Java SE 7 *o* Java SE 8. Todas las características de Java SE 8 se analizan en secciones modulares que son fáciles de incluir u omitir a lo largo del libro.

Antes de Java SE 8, Java soportaba tres paradigmas de programación: *programación por procedimientos*, *programación orientada a objetos* y *programación genérica*. Java SE 8 agrega la *programación funcional*. En el capítulo 17 le mostraremos cómo usar la programación funcional para escribir programas en forma más rápida y concisa, con menos errores y que sean fáciles de *paralelizar* (es decir, que puedan realizar varios cálculos al mismo tiempo) para aprovechar las arquitecturas de hardware multinúcleo actuales que mejoran el rendimiento de una aplicación.

Java Enterprise Edition

Java se utiliza en un espectro tan amplio de aplicaciones que tiene otras dos ediciones. **Java Enterprise Edition (Java EE)** está orientada hacia el desarrollo de aplicaciones de red distribuidas, de gran escala, y aplicaciones basadas en Web. En el pasado, la mayoría de las aplicaciones de computadora se ejecutaban en computadoras "independientes" (que no estaban conectadas en red). En la actualidad se pueden escribir aplicaciones que se comuniquen con computadoras en todo el mundo por medio de Internet y Web. Más adelante hablaremos sobre cómo crear dichas aplicaciones basadas en Web con Java.

4 https://www.abiresearch.com/press/tablets-will-generate-35-of-this-years-25-billion-.

Java Micro Edition
Java Micro Edition (Java ME) (un subconjunto de Java SE) está orientada hacia el desarrollo de aplicaciones para pequeños dispositivos incrustados con una capacidad limitada de memoria, como los relojes inteligentes, los reproductores MP3, los decodificadores para televisión, los medidores inteligentes (para monitorear el uso de la energía eléctrica) y más.

1.2 Hardware y software

Las computadoras pueden realizar cálculos y tomar decisiones lógicas con una rapidez increíblemente mayor que los humanos. Muchas de las computadoras personales actuales 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 millones* de instrucciones por segundo! ¡La supercomputadora Tianhe-2 de la National University of Defense Technology de China puede realizar más de 33 mil billones de cálculos por segundo (33.86 *petaflops*)![5] Para ilustrar este ejemplo, *¡la supercomputadora Tianhe-2 puede ejecutar en un segundo el equivalente a cerca de 3 millones de cálculos por cada habitante del planeta!* Y estos "límites máximos" están aumentando con rapidez.

Las computadoras procesan datos bajo el control de secuencias de instrucciones conocidas como **programas de computadora**. Estos programas guían a la computadora a través de acciones ordenadas especificadas por gente conocida como **programadores** de computadoras. En este libro aprenderá una metodología de programación clave que está mejorando la productividad del programador, con lo cual se reducen los costos de desarrollo del software: la *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 espacios y que costaban millones de dólares hace algunas décadas, ahora pueden grabarse en 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 un ingrediente de la arena común). La tecnología de los chips de silicio se ha vuelto tan económica que las computadoras se han convertido en un producto básico.

1.2.1 Ley de Moore

Es probable que cada año espere pagar al menos un poco más por la mayoría de los productos y servicios que utiliza. En 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 en 1960: Gordon Moore, cofundador de Intel (uno de los principales fabricantes de procesadores para las computadoras y los sistemas incrustados (embebidos) de la actualidad). La Ley de Moore y las observaciones relacionadas son especialmente ciertas para la cantidad de memoria que tienen destinada las computadoras para programas, para la cantidad de almacenamiento secundario (como el almacenamiento en disco) que tienen para guardar los programas y datos durante periodos extendidos de tiempo, y para las velocidades de sus procesadores (las velocidades con que las computadoras *ejecutan* sus programas y 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 (es decir, la

5 http://www.top500.org/.

capacidad de transmisión de información) atrae una competencia intensa. No conocemos otros campos 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*.

1.2.2 Organización de la computadora

Sin importar las diferencias en la apariencia física, es posible percibir una segmentación de las computadoras en varias **unidades lógicas** o secciones (figura 1.2).

Unidad lógica	Descripción
Unidad de entrada	Esta sección de "recepción" 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, las pantallas táctiles y los ratones. La información también puede introducirse de muchas otras formas, como a través de comandos de voz, la digitalización de imágenes y códigos de barras, por medio de dispositivos de almacenamiento secundario (como discos duros, unidades de DVD, Blu-ray Disc™ y memorias Flash USB —también conocidas como "*thumb drives*" o "*memory sticks*"), mediante la recepción de video de una cámara Web y al recibir 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: la lectura de datos de geolocalización a través de un dispositivo GPS, y la información sobre el movimiento y la orientación mediante un *acelerómetro* (un dispositivo que responde a la aceleración hacia arriba o abajo, a la derecha o izquierda y hacia delante o atrás) en un teléfono inteligente o un controlador de juegos (Como Microsoft® Kinect® y Xbox®, Wii™ Remote y Sony® PlayStation® Move).
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 muestra en pantallas (incluyendo pantallas táctiles), se imprime en papel (lo cual no es muy bueno para el medio ambiente), se reproduce como audio o video en equipos PC y reproductores de medios (como los 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". La información también se envía por lo general a dispositivos de almacenamiento secundarios, como discos duros, unidades DVD y unidades Flash USB. Una forma popular reciente de salida es la vibración de los teléfonos 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 pueda estar disponible de manera inmediata y procesarla 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*, ya que por lo general se pierde cuando se apaga la computadora. Con frecuencia, a la unidad de memoria se le conoce como **memoria**, **memoria principal** o **RAM** (memoria de acceso aleatorio). Las típicas memorias principales en las computadoras de escritorio y portátiles pueden contener hasta 128 GB de RAM. GB se refiere a gigabytes; un gigabyte equivale aproximadamente a mil millones de bytes. Un **byte** equivale a ocho bits. Un bit puede ser un `0` o un `1`.

Fig. 1.2 | Unidades lógicas de una computadora (parte 1 de 2).

Unidad lógica	Descripción
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 secciones. La CPU le indica a la unidad de entrada cuándo debe colocarse 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 con frecuencia las demás unidades se colocan por lo general en dispositivos de almacenamiento secundario (por ejemplo, el *disco duro*) hasta que se requieran de nuevo, lo cual puede llegar a ser horas, días, meses o incluso años después. La información en los dispositivos de almacenamiento secundario es *persistente*, lo que significa que se mantiene aun 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 disco duro, DVD y Flash USB son ejemplos de dispositivos de almacenamiento secundario, los cuales pueden contener hasta 2 TB (TB se refiere a terabytes; un terabyte equivale aproximadamente a un billón de bytes). Los discos duros típicos en las computadoras de escritorio y portátiles pueden contener hasta 2 TB y algunas unidades de disco duro de escritorio pueden contener hasta 4 TB.

Fig. 1.2 | Unidades lógicas de una computadora (parte 2 de 2).

1.3 Jerarquía de datos

Los elementos de datos que procesan las computadoras forman una **jerarquía de datos** cuya estructura se vuelve cada vez más grande y compleja, a medida que pasamos de los elementos de datos más simples (conocidos como "bits") a los más complejos, como los caracteres y los campos. La figura 1.3 ilustra una porción de la jerarquía de datos.

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 increíble que todas las impresionantes funciones que realizan las computadoras impliquen sólo las manipulaciones más simples de los dígitos `0` y `1`, como *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`).

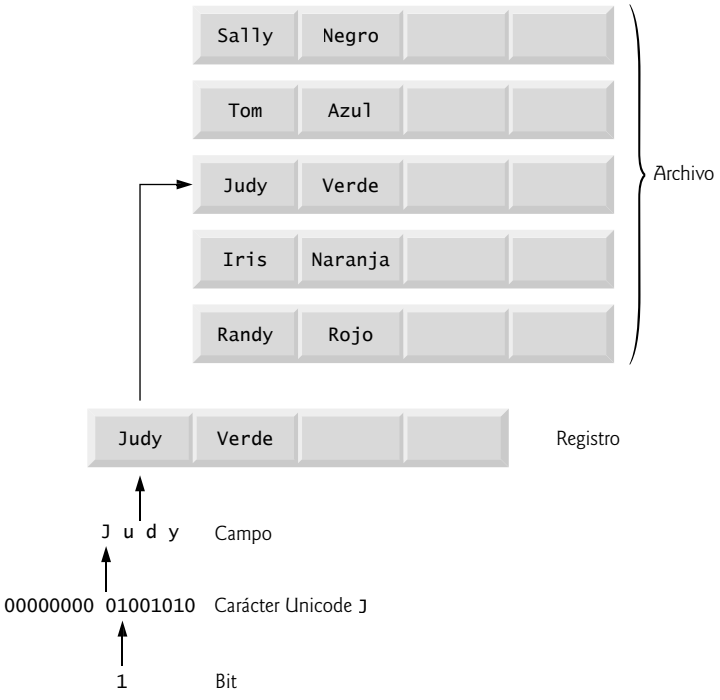


Fig. 1.3 | Jerarquía de datos.

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 caracteres que se utilizan para escribir programas y representar elementos de datos. Las computadoras sólo procesan unos y ceros, por lo que el conjunto de caracteres de una computadora representa a cada carácter como un patrón de unos y ceros. Java usa caracteres **Unicode®** que están compuestos de uno, dos o cuatro bytes (8, 16 o 32 bits). Unicode contiene caracteres para muchos de los idiomas que se usan en el mundo. En el apéndice H obtendrá más información sobre Unicode. En el apéndice B obtendrá más información sobre el conjunto de caracteres **ASCII (Código estándar estadounidense para el intercambio de información)**, que es el popular subconjunto de Unicode que representa las letras mayúsculas y minúsculas, los dígitos y algunos caracteres especiales comunes.

Campos
Así como los caracteres están compuestos de bits, los **campos** están compuestos de 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 un campo compuesto de dígitos decimales podría representar la edad de esa persona.

Registros
Se pueden usar varios campos relacionados para componer un **registro** (el cual se implementa como una clase [`class`] en Java). Por ejemplo, en un sistema de nómina, el registro de un empleado podría consistir en los siguientes campos (los posibles tipos para estos campos 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)

Por lo tanto, 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 byte* y 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. En el capítulo 15 verá cómo se hace eso]. 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.

Base de datos

Una **base de datos** es una colección de datos organizados para facilitar su acceso y manipulación. El modelo más popular es la *base de datos relacional*, en la que los datos se almacenan en simples *tablas*. Una tabla incluye *registros* y *campos*. Por ejemplo, una tabla de estudiantes podría incluir los campos nombre, apellido, especialidad, año, número de identificación (ID) del estudiante y promedio de calificaciones. Los datos para cada estudiante constituyen un registro y las piezas individuales de información en cada registro son los campos. Puede *buscar*, *ordenar* y manipular de otras formas los datos con base en la relación que tienen con otras tablas o bases de datos. Por ejemplo, una universidad podría utilizar datos de la base de datos de los estudiantes en combinación con los de bases de datos de cursos, alojamiento en el campus, planes alimenticios, etc. En el capítulo 24 hablaremos sobre las bases de datos.

Big Data

La cantidad de datos que se produce a nivel mundial es enorme y aumenta con rapidez. De acuerdo con IBM, cada día se generan alrededor de 2.5 trillones (2.5 *exabytes*) de datos y el 90% de los datos en el mundo se crearon ¡tan sólo en los últimos dos años![6] De acuerdo con un estudio de Digital Universe, en 2012 el suministro de datos globales llegó a 2.8 *zettabytes* (lo que equivale a 2.8 billones de gigabytes).[7] La figura 1.4 muestra algunas mediciones comunes de bytes. Las aplicaciones de **Big Data** lidian con dichas cantidades masivas de datos y este campo crece con rapidez, lo que genera muchas oportunidades para los desarrolladores de software. De acuerdo con un estudio por parte de Gartner Group, para 2015 más de 4 millones de empleos de TI a nivel mundial darán soporte a los grandes volúmenes de datos (Big Data).[8]

---

6 http://www-01.ibm.com/software/data/bigdata/.

7 http://www.guardian.co.uk/news/datablog/2012/dec/19/big-data-study-digital-universe-global-volume.

8 http://tech.fortune.cnn.com/2013/09/04/big-data-employment-boom/.

Unidad	Bytes	Lo que equivale aproximadamente a
1 kilobyte (KB)	1024 bytes	10^3 (1024 bytes exactamente)
1 megabyte (MB)	1024 kilobytes	10^6 (1,000,000 bytes)
1 gigabyte (GB)	1024 megabytes	10^9 (1,000,000,000 bytes)
1 terabyte (TB)	1024 gigabytes	10^{12} (1,000,000,000,000 bytes)
1 petabyte (PB)	1024 terabytes	10^{15} (1,000,000,000,000,000 bytes)
1 exabyte (EB)	1024 petabytes	10^{18} (1,000,000,000,000,000,000 bytes)
1 zettabyte (ZB)	1024 exabytes	10^{21} (1,000,000,000,000,000,000,000 bytes)

Fig. 1.4 | Mediciones de bytes.

1.4 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

Lenguajes máquina

Cualquier computadora sólo puede entender de manera directa 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 unos y ceros) 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 una sección de uno de los primeros programas de nómina 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
```

Lenguajes ensambladores y ensambladores

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.

Lenguajes de alto nivel y compiladores
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 los que 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 uno de los lenguajes de alto nivel más utilizados.

Intérpretes
El proceso de compilación de un programa extenso escrito en lenguaje de alto nivel a un lenguaje máquina, puede tardar un tiempo considerable en la computadora. Los programas *intérpretes*, que se desarrollaron para ejecutar de manera directa programas en lenguaje de alto nivel, evitan el retraso de la compilación, aunque se ejecutan con más lentitud que 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.

1.5 Introducción a la tecnología de los objetos

Ahora que la demanda de software nuevo y más poderoso va en aumento, crear software en forma rápida, correcta y económica sigue siendo un objetivo difícil de alcanzar. Los *objetos*, o dicho en forma más precisa, 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, etc. 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 grupos de desarrollado de software pueden usar una metodología de diseño e implementación orientada a objetos y modular para ser mucho 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.

1.5.1 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 presionar 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* al conductor 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 ningún conocimiento sobre cómo funcionan los motores, los frenos y los mecanismos de la dirección puedan conducir un auto con facilidad.

Así como no es posible cocinar en la cocina que está en 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 aun así no es suficiente; el auto no acelerará por su propia cuenta (¡esperemos que así sea!), así que el conductor debe *presionar* el pedal para acelerar el auto.

1.5.2 Métodos y clases

Ahora vamos a utilizar nuestro ejemplo del auto para presentar algunos conceptos clave de la programación orientada a objetos. Para realizar una tarea en una aplicación se requiere un **método**. Ese método aloja las instrucciones del programa que se encargan de realizar sus tareas. El método oculta al usuario estas instrucciones, 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* dinero de una cuenta y un tercero para *solicitar* el saldo actual de la cuenta. 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.

1.5.3 Instanciación

Así como alguien tiene que *construir un auto* a partir de sus dibujos de ingeniería para que alguien más pueda conducirlo 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 *instanciación*. Por lo tanto, un objeto viene siendo una **instancia** de su clase.

1.5.4 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, ya 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 del software incitada por la tecnología de objetos.



Observación de ingeniería de software 1.1

Use un método de construcción en bloques para crear sus 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.*

1.5.5 Mensajes y llamadas a métodos

Cuando usted conduce un auto, al presionar 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 para aumentar el saldo de esa cuenta.

1.5.6 Atributos y variables de instancia

Además de tener capacidades para realizar tareas, un auto también tiene *atributos*: color, número de puertas, capacidad de gasolina en el tanque, velocidad actual y registro del total de kilómetros recorridos (es

decir, la lectura de su odómetro). Al igual que sus capacidades, los atributos del auto se representan como parte de su diseño en sus diagramas de ingeniería (que, por ejemplo, incluyen un velocímetro y un indicador de combustible). Al conducir un auto real, estos atributos se llevan junto con el auto. Cada auto mantiene 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 del objeto de esa clase. 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.

1.5.7 Encapsulamiento y ocultamiento de información

Las clases y sus objetos **encapsulan** (envuelven) sus atributos y métodos. Los atributos y métodos de una clase (y sus objetos) 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.

1.5.8 Herencia

Mediante la **herencia** es posible crear con rapidez y de manera conveniente una nueva clase de objetos. La nueva clase (conocida como **subclase**) comienza con las características de una clase existente (conocida como **superclase**), 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 toldo puede ponerse o quitarse.

1.5.9 Interfaces

Java también soporta las **interfaces**: colecciones de métodos relacionados que por lo general nos permiten indicar a los objetos *qué* hacer, pero no *cómo* hacerlo (en Java SE 8 veremos una excepción a esto). En la analogía del auto, una interfaz con "capacidades básicas de conducción" que consista de un volante de dirección, un pedal acelerador y un pedal del freno, permitiría a un conductor indicar al auto *qué* debe hacer. Una vez que sepa cómo usar esta interfaz para dar vuelta, acelerar y frenar, podrá conducir muchos tipos de autos, incluso aunque los fabricantes puedan *implementar* estos sistemas de manera *diferente*.

Una clase **implementa** cero o más interfaces, cada una de las cuales puede tener uno o más métodos, al igual que un auto implementa interfaces separadas para las funciones básicas de conducción, para el control del radio, el control de los sistemas de calefacción y aire acondicionado, etcétera. Así como los fabricantes de automóviles implementan las capacidades de manera *diferente*, las clases pueden implementar los métodos de una interfaz de manera *diferente*. Por ejemplo, un sistema de software puede incluir una interfaz de "respaldo" que ofrezca los métodos *guardar* y *restaurar*. Las clases pueden implementar esos métodos de manera distinta, dependiendo de los tipos de cosas que se vayan a respaldar, como programas, textos, archivos de audio, videos, etc., y los tipos de dispositivos en donde se vayan a almacenar estos elementos.

1.5.10 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 ¿qué tal 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 revisen su diseño) 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.

1.5.11 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 33 y 34 presentamos un subconjunto simple de las características del UML, mientras lo guiamos por una experiencia de diseño orientada a objetos.

1.6 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, eficiente y *concurrente* (es decir, en paralelo) con otras aplicaciones. Al software que contiene los componentes básicos del sistema operativo se denomina **kernel**. Los sistemas operativos de escritorio populares son: Linux, Windows y Mac OS X. Los sistemas operativos móviles populares que se utilizan en teléfonos inteligentes y tabletas son: Android de Google, iOS de Apple (para sus dispositivos iPhone, iPad e iPod Touch), Windows Phone 8 y BlackBerry OS.

1.6.1 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 con el que los usuarios interactuaban tecleando 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 10 es el sistema operativo más reciente de Microsoft; sus características incluyen soporte para equipos PC y tabletas, una interfaz de usuario basada en mosaicos, mejoras en la seguridad, soporte para pantallas táctiles y multitáctiles, entre otras cosas más. Windows es un sistema operativo *propietario*; está bajo el control exclusivo de Microsoft. Es por mucho el sistema operativo más utilizado en el mundo.

1.6.2 Linux: un sistema operativo de código fuente abierto

El sistema operativo **Linux** (muy popular en servidores, computadoras personales y sistemas incrustados) 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* (que se utiliza, por ejemplo, con Windows de Microsoft y Mac OS X de Apple). Con el desarrollo de código fuente abierto, individuos y compañías (por lo general a nivel mundial) suman sus esfuerzos para desarrollar, mantener y evolucionar el software. Cualquiera lo puede usar y personalizar para sus propios fines, por lo general sin costo. Ahora el Kit de desarrollo de Java y muchas de las tecnologías de Java relacionadas son de código fuente abierto.

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 entrega páginas Web a través de Internet* en respuesta a las solicitudes de los navegadores Web) y *GitHub* y *SourceForge* (que proporcionan las *herramientas para administrar proyectos de código fuente abierto*).

Las rápidas mejoras en la computación y las comunicaciones, la reducción en costos y el software de código fuente abierto han logrado que en la actualidad sea mucho más fácil y económico crear un negocio basado en software de lo que era hace unas cuantas décadas. Facebook, que se inició desde un dormitorio universitario, se creó con software de código fuente abierto.[9]

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

1.6.3 Android

Android —el sistema operativo con mayor crecimiento a la fecha para dispositivos móviles y teléfonos inteligentes— 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) —que ahora cuenta con 87 compañías miembro a nivel mundial (`http://www.openhandsetalliance.com/oha_members.html`)—, para continuar con el desarrollo, mantenimiento y evolución de Android, impulsando la innovación en la tecnología móvil y mejorando la experiencia del usuario, reduciendo al mismo tiempo los costos. Al mes de abril de 2013, se activaban a *diario* más de 1.5 millones de dispositivos con Android (teléfonos inteligentes, tabletas, etc.).[10] Para octubre de 2013, un informe de Strategy Analytics mostró que Android tenía el 81.3% de la participación global en el mercado de *teléfonos inteligentes*, en comparación con el 13.4% de Apple, el 4.1% de Microsoft y el 1% de BlackBerry.[11] Ahora los dispositivos Android incluyen teléfonos inteligentes, tabletas, lectores electrónicos, robots, motores de jet, satélites de la NASA, consolas de juegos, refrigeradores, televisiones, cámaras, dispositivos para el cuidado de la salud, relojes inteligentes, sistemas de infoentretenimiento en vehículos (para controlar el radio, GPS, llamadas telefónicas, termostato, etc.) y más.[12]

---

9 `http://developers.facebook.com/opensource`.

10 `http://www.technobuffalo.com/2013/04/16/google-daily-android-activations-1-5-million`.

11 `http://www.cnet.com/news/android-shipments-exceed-1-billion-for-first-time-in-2014/`.

12 `http://www.businessweek.com/articles/2013-05-29/behind-the-internet-of-things-is-android-and-its-everywhere`.

Los teléfonos inteligentes Android incluyen la funcionalidad de un teléfono móvil, un cliente de Internet (para navegar en Web y comunicarse a través de Internet), un reproductor de MP3, una consola de juegos, una cámara digital y demás. Estos dispositivos portátiles cuentan con *pantallas multitáctiles* a todo color, que 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 de Google Play y de otros mercados de aplicaciones. Al momento de escribir este libro había más de un millón de aplicaciones en **Google Play**; esta cifra aumenta con rapidez.[13]

En nuestro libro de texto *Android How to Program*, *segunda edición*, y en nuestro libro profesional, *Android for Programmers: An App-Driven Approach, segunda edición*, presentamos una introducción al desarrollo de aplicaciones para Android. Después de que aprenda Java, descubrirá que no es tan complicado empezar a desarrollar y ejecutar aplicaciones Android. Puede colocar sus aplicaciones en Google Play (`play.google.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.

1.7 Lenguajes de programación

En esta sección comentaremos brevemente algunos 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 soportan la programación orientada 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, que también abogó por la estandarización internacional de los lenguajes de programación. 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, fáciles de probar y depurar, y más fáciles de modificar que los programas extensos producidos con técnicas anteriores. Un resultado de esta investigación fue el desarrollo del lenguaje de programación Pascal en 1971, el cual 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 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 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). Ada también soporta la programación orientada a objetos.

Fig. 1.5 | Otros lenguajes de programación (parte 1 de 3).

13 `http://en.wikipedia.org/wiki/Google_Play`.

Lenguaje de programación	Descripción
Basic	Basic se desarrolló en la década de 1960 en el Dartmouth College, para familiarizar a los principiantes con las técnicas de programación. Muchas de sus versiones más recientes son orientadas a objetos.
C	C fue desarrollado a principios de la década de 1970 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++.
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 las 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 adquirida por Apple. Se ha convertido en el lenguaje de programación clave para el sistema operativo 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 orientados a objetos de Microsoft son Visual Basic (basado en el Basic original), Visual C++ (basado en C++) y Visual 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, el cual recibe soporte por medio de una comunidad de usuarios y desarrolladores; se utiliza en millones de sitios Web. 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, incluyendo la popular MySQL de código fuente abierto.
Perl	Perl (Lenguaje práctico de extracción y reporte), uno de los lenguajes de secuencia de comandos orientados a objetos más utilizados para la programación Web, fue desarrollado en 1987 por Larry Wall. Cuenta con capacidades complejas de procesamiento de textos.
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", lo que significa que 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 comportamiento dinámico a las páginas Web; por ejemplo, animaciones e interactividad mejorada con el usuario. Se incluye en todos los principales navegadores Web.

Fig. 1.5 | Otros lenguajes de programación (parte 2 de 3).

Lenguaje de programación	Descripción
Ruby on Rails	Ruby, que se creó a mediados de la década de 1990, 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 desarrollar aplicaciones Web que trabajan de manera intensiva con bases de datos.

Fig. 1.5 | Otros lenguajes de programación (parte 3 de 3).

1.8 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 han tenido 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 dio como resultado 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 explotó 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 atrajo la atención de la comunidad de negocios debido al fenomenal interés en el servicio 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 navegadores 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. Java también es el lenguaje clave para desarrollar aplicaciones para teléfonos inteligentes y tabletas de Android. En 2010, Oracle adquirió Sun Microsystems.

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 y métodos 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 eficiencia. Esta técnica también reduce el tiempo de desarrollo de los programas.

1.9 Un típico entorno de desarrollo en Java

Ahora explicaremos los pasos típicos utilizados para crear y ejecutar una aplicación en Java. Por lo general hay cinco fases: edición, compilación, carga, verificación y ejecución. Hablaremos sobre estos conceptos

en el contexto del Kit de desarrollo de Java (JDK) SE 8. Lea la sección *Antes de empezar de este libro para obtener información acerca de cómo descargar e instalar el JDK en Windows, Linux y OS X.*

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). A través del editor, usted escribe un programa en Java (a lo cual se le conoce por lo general como **código fuente**), realiza las correcciones necesarias y guarda el programa en un dispositivo de almacenamiento secundario, como su disco duro. Los archivos de código fuente de Java reciben un nombre que termina con la **extensión `.java`**, lo que indica que éste contiene código fuente en Java.

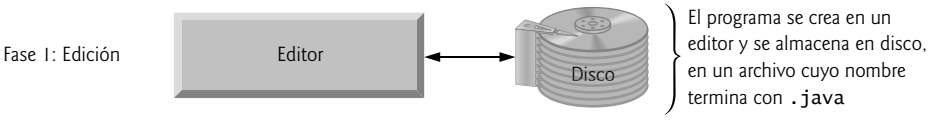


Fig. 1.6 | Entorno de desarrollo típico de Java: fase de edición.

Dos de los editores muy utilizados en sistemas Linux son `vi` y `emacs`. Windows cuenta con el **`Bloc de Notas`**. OS X ofrece **`TextEdit`**. También hay muchos editores de freeware y shareware disponibles en línea, como Notepad++ (`notepad-plus-plus.org`), Edit-Plus (`www.editplus.com`), TextPad (`www.textpad.com`) y jEdit (`www.jedit.org`).

Los **entornos de desarrollo integrados (IDE)** proporcionan herramientas que dan soporte al proceso de desarrollo del software, entre las que se incluyen editores, depuradores para localizar **errores lógicos** (errores que provocan que los programas se ejecuten en forma incorrecta) y más. Hay muchos IDE de Java populares, como

- Eclipse (`www.eclipse.org`)
- NetBeans (`www.netbeans.org`)
- IntelliJ IDEA (`www.jetbrains.com`)

En el sitio Web del libro en

```
www.deitel.com/books/jhtp10
```

proporcionamos videos Dive-Into® que le muestran cómo ejecutar las aplicaciones de Java de este libro y cómo desarrollar nuevas aplicaciones de Java con Eclipse, NetBeans e IntelliJ IDEA.

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** de Windows, o la aplicación **Terminal** en OS X) o en un shell de Linux (que también se conoce como **Terminal** en algunas versiones de Linux). Si el programa se compila, el compilador produce un archivo **`.class`** llamado `Bienvenido.class` que contiene la versión compilada del programa. Por lo general los IDE proveen un elemento de menú, como **Build** (Generar) o **Make** (Crear), que invoca el comando `javac` por usted. Si el compilador detecta errores, tendrá que ir a la fase 1 y corregirlos. En el capítulo 2 hablaremos más sobre los tipos de errores que el compilador puede detectar.

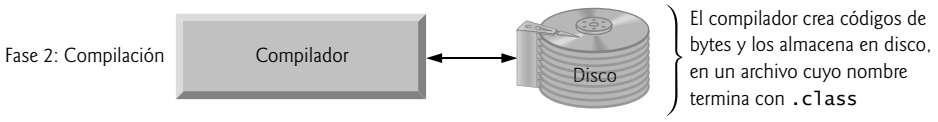


Fig. 1.7 | Entorno de desarrollo típico de Java: fase de compilación.

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 escritas para ese tipo de VM 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 de las instrucciones de lenguaje máquina, que *dependen de la plataforma* (es decir, dependen del hardware de una computadora específica), los códigos de bytes son instrucciones *independientes de la plataforma*. Por lo tanto, 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 incluya 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. Por lo general los IDE proporcionan un elemento de menú, como **`Run`** (**`Ejecutar`**), que invoca el comando `java` por usted.

Fase 3: Carga de un programa en memoria

En la fase 3, la JVM coloca el programa en memoria para ejecutarlo; a esto se le conoce como **carga** (figura 1.8). El **cargador de clases** de la JVM 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).

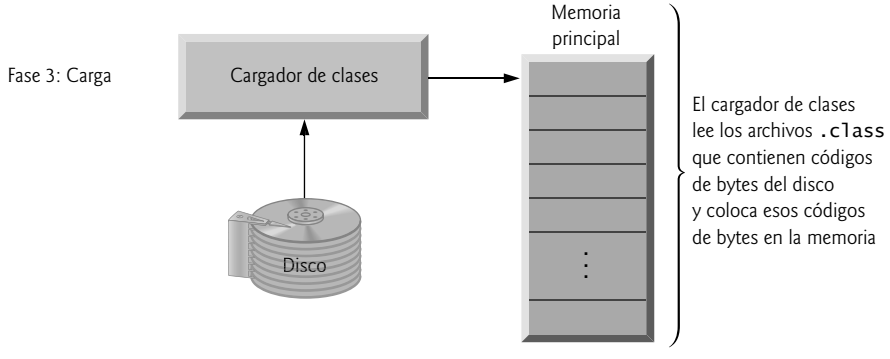


Fig. 1.8 | Entorno de desarrollo típico de Java: fase de carga.

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).

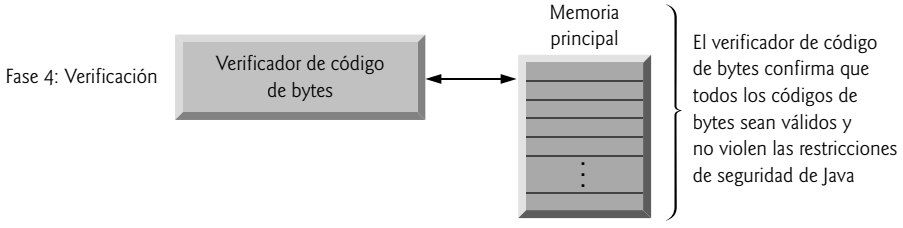


Fig. 1.9 | Entorno de desarrollo típico de Java: fase de verificación.

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 bytes 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)**, como el **compilador HotSpot™ de Java** de Oracle, traduce los códigos de bytes al lenguaje máquina correspondiente de la computadora. Cuando la JVM vuelve a encontrar estas partes compiladas, 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.

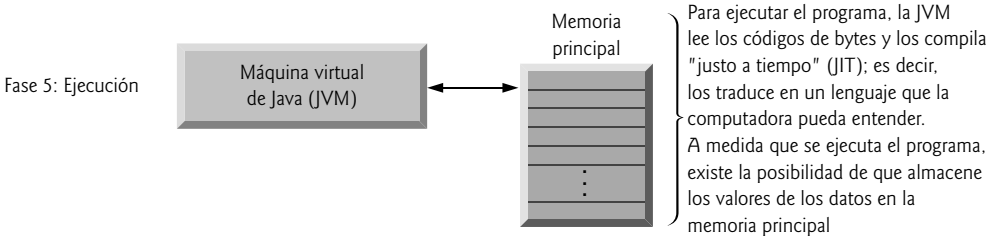


Fig. 1.10 | Entorno de desarrollo típico de Java: fase de ejecución.

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 libro. 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 mostrara un mensaje de error. Si esto ocurre, tendría que regresar a la fase de edición, hacer las correcciones necesarias y proseguir de nuevo con las fases restantes, para determinar que las correcciones hayan resuelto el o los problemas [*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 muestra ese mensaje 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. La aplicación **Painter**, que creará en el transcurso de varios ejercicios, le permite arrastrar el ratón para "dibujar". Los elementos y la funcionalidad que podemos ver en esta aplicación son típicos de lo que aprenderá a programar en este libro. Mediante el uso de la interfaz gráficos de usuario (GUI) de **Painter**, usted puede controlar el color de dibujo, la forma a dibujar (línea, rectángulo u óvalo) y si la forma se debe llenar o no con un color. También puede deshacer la última forma que agregó al dibujo o borrarlo todo [*nota:* utilizamos fuentes para diferenciar las diversas características. 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 negritas, y enfatizar los elementos que no son de la pantalla, como los nombres de archivo, código del programa o los datos de entrada (como `NombrePrograma.java`) en una fuente `Lucida sans-serif`].

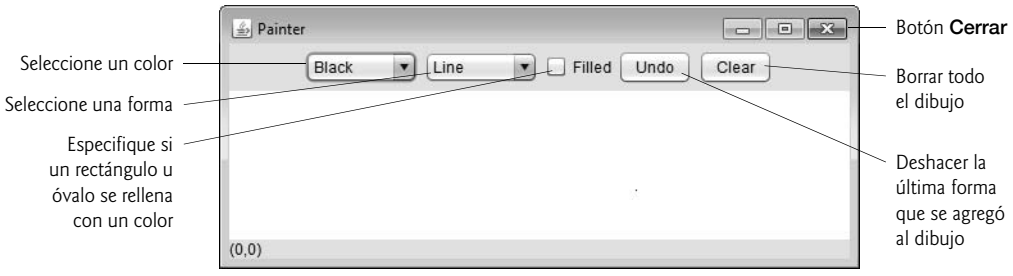
Los pasos en esta sección le muestran cómo ejecutar la aplicación **Painter** desde una ventana **Símbolo del sistema** (Windows), **Terminal** (OS X) o shell (Linux) de su sistema. A lo largo del libro nos referiremos a estas ventanas simplemente como *ventanas de comandos*. Realice los siguientes pasos para usar la aplicación **Painter** para dibujar una cara sonriente:

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

2. ***Cambie al directorio de la aplicación completa.*** Abra una ventana de comandos y use el comando `cd` para cambiar al directorio (también conocido como *carpeta*) de la aplicación **Painter**. Vamos a suponer que los ejemplos del libro se encuentran en `C:\ejemplos` en Windows o en la carpeta `Documents/ejemplos` en Linux o en OS X. En Windows escriba `cd C:\ejemplos\cap01\painter` y después oprima *Intro*. En Linux u OS X escriba `cd ~/Documents/ejemplos/cap01/painter` y después oprima *Intro*.

3. ***Ejecute la aplicación Painter.*** Recuerde que el comando `java`, seguido del nombre del archivo `.class` de la aplicación (en este caso, `Painter`), ejecuta la aplicación. Escriba el comando `java Painter` y oprima *Intro* para ejecutar la aplicación. La figura 1.11 muestra la aplicación en ejecución en Windows, Linux y OS X, respectivamente; redujimos el tamaño de las ventanas para ahorrar espacio.

a) La aplicación **Painter** ejecutándose en Windows.



b) La aplicación **Painter** ejecutándose en Linux.



c) La aplicación **Painter** ejecutándose en OS X.



Fig. 1.11 | La aplicación **Painter** ejecutándose en Windows 7, Linux y OS X.

[*Nota:* los comandos en Java son *sensibles a mayúsculas/minúsculas*. Es importante escribir el nombre de esta aplicación como `Painter` con `P` mayúscula. De lo contrario, la aplicación *no* se ejecutará. Si se especifica la extensión `.class` al usar el comando `java` se produce un error. Ademas, si recibe el mensaje de error "`Exception in thread "main" java.lang.NoClassDefFoundError: Painter`", entonces su sistema tiene un problema con `CLASSPATH`. Consulte la sección Antes de empezar del libro para obtener instrucciones sobre cómo corregir este problema].

4. ***Dibuje un óvalo relleno de color amarillo para el rostro.*** Seleccione **Yellow** (**Amarillo**) como el color de dibujo, **Oval** (**Óvalo**) como la forma y marque la casilla de verificación **Filled** (**Relleno**); luego arrastre el ratón para dibujar un óvalo más grande (figura 1.12).

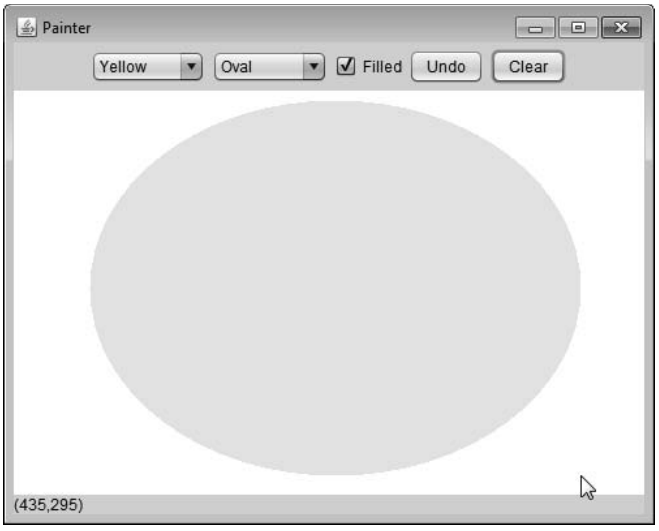


Fig. 1.12 | Dibuje un óvalo relleno de color amarillo para el rostro.

5. ***Dibuje los ojos azules.*** Seleccione **Blue** (**Azul**) como el color de dibujo y luego dibuje dos óvalos pequeños como los ojos (figura 1.13).

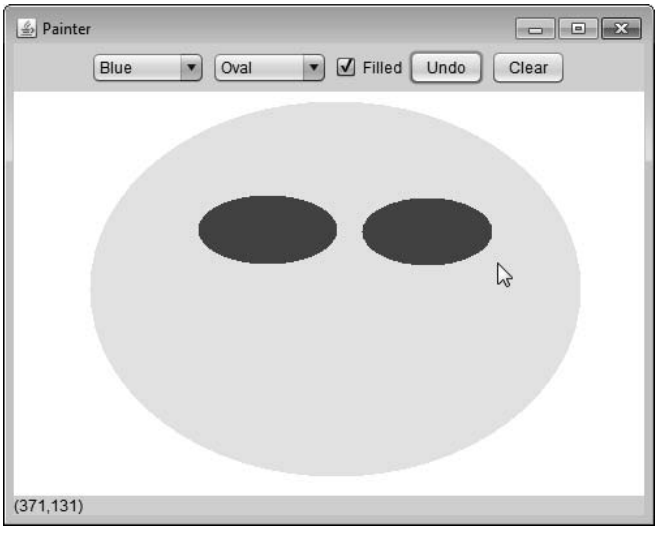


Fig. 1.13 | Dibuje los ojos azules.

6. ***Dibuje cejas negras y una nariz.*** Seleccione **Black** (**Negro**) como el color de dibujo y **Line** (**Línea**) como la forma; después dibuje cejas y una nariz (figura 1.14). Las líneas no tienen relleno, por lo que si se deja la casilla de verificación **Filled** (**Relleno**) marcada, esto no tendrá efecto cuando se dibujen las líneas.

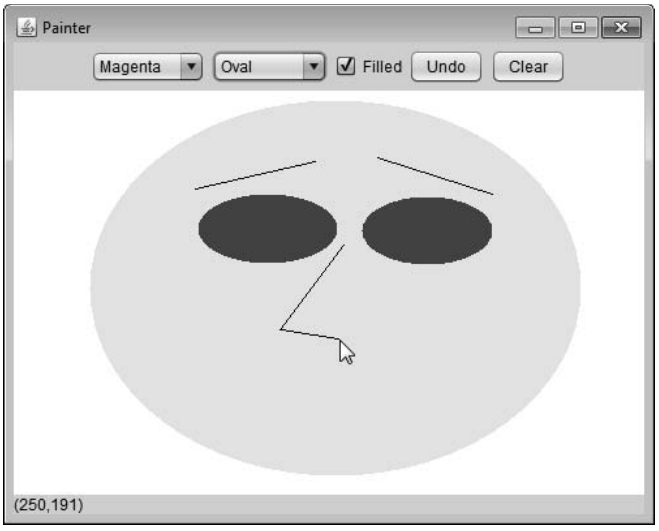


Fig. 1.14 | Dibuje cejas negras y una nariz.

7. ***Dibuje una boca color magenta.*** Seleccione **Magenta** como el color de dibujo y **Oval** (**Óvalo**) como la forma; después dibuje una boca (figura 1.15).

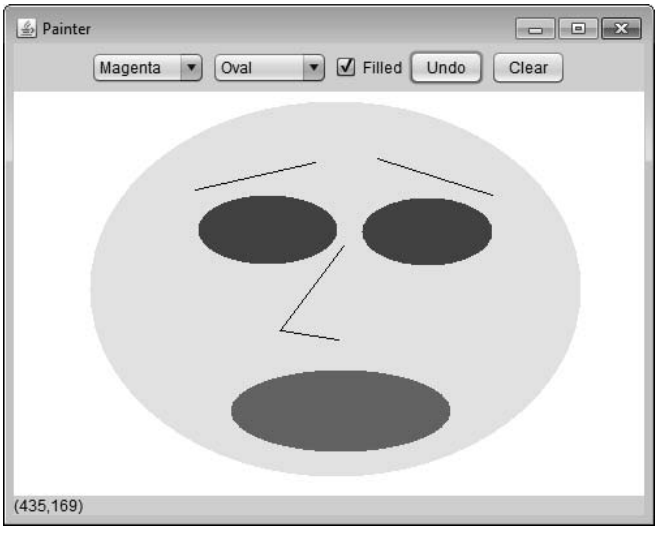


Fig. 1.15 | Dibuje una boca color magenta.

8. ***Dibuje un óvalo amarillo en la boca para crear una sonrisa.*** Seleccione **Yellow** (**Amarillo**) como el color de dibujo y luego dibuje un óvalo para convertir el óvalo color magenta en una sonrisa (figura 1.16).

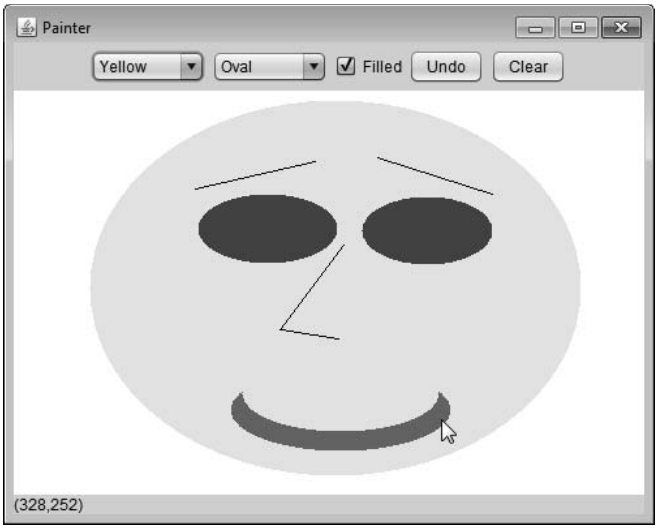


Fig. 1.16 | Dibuje un óvalo amarillo en la boca para crear una sonrisa.

9. ***Salga de la aplicación Painter.*** Para salir de la aplicación **Painter**, haga clic en el botón **Cerrar** (en la esquina superior derecha de la ventana en Windows y en la esquina superior izquierda en Linux y OS X). Al cerrar la ventana terminará la ejecución de la aplicación **Painter**.

1.11 Internet y World Wide Web

A finales de la década de 1960, la ARPA (Agencia de proyectos avanzados de investigación del Departamento de Defensa de Estados Unidos) implementó los planes para conectar en red los principales sistemas de cómputo de aproximadamente una docena de universidades e instituciones de investigación financiadas por la ARPA. Las computadoras se iban a conectar con líneas de comunicación que operaban a velocidades en el orden de 50,000 bits por segundo, una tasa impresionante en una época en que la mayoría de las personas (de los pocos que incluso tenían acceso a redes) se conectaban a través de líneas telefónicas a las computadoras a una tasa de 110 bits por segundo. La investigación académica estaba a punto de dar un gran paso hacia delante. La ARPA procedió a implementar lo que rápidamente se conoció como ARPANET, precursora de la red Internet actual. Las velocidades de Internet más rápidas de la actualidad están en el orden de miles de millones de bits por segundo y pronto estarán disponibles las velocidades de billones de bits por segundo.

Las cosas funcionaron de manera distinta al plan original. Aunque la ARPANET permitió que los investigadores conectaran en red sus computadoras, su principal beneficio demostró ser la capacidad de comunicarse con rapidez y facilidad a través de lo que se denominó correo electrónico (e-mail). Esto es cierto incluso en la red Internet actual, en donde el correo electrónico, la mensajería instantánea, la transferencia de archivos y los medios sociales como Facebook y Twitter permiten que miles de millones de personas en todo el mundo se comuniquen de una manera rápida y sencilla.

El protocolo (conjunto de reglas) para comunicarse a través de la ARPANET se denominó **Protocolo de control de transmisión (TCP)**. Este protocolo se aseguraba de que los mensajes, que consistían en piezas numeradas en forma secuencial conocidas como *paquetes*, se enrutaran correctamente del emisor al receptor, que llegaran intactos y se ensamblaran en el orden correcto.

1.11.1 Internet: una red de redes

En paralelo con la evolución de Internet en sus primeras etapas, las organizaciones de todo el mundo estaban implementando sus propias redes para comunicarse tanto dentro de la organización como entre varias organizaciones. En esta época apareció una enorme variedad de hardware y software de red. Un desafío era permitir que esas distintas redes se comunicaran entre sí. La ARPA logró esto al desarrollar el Protocolo Internet (IP), que creó una verdadera "red de redes", la arquitectura actual de Internet. Al conjunto combinado de protocolos se le conoce ahora como **TCP/IP**.

Las empresas descubrieron rápidamente que al usar Internet podían mejorar sus operaciones además de ofrecer nuevos y mejores servicios a sus clientes. Las compañías comenzaron a invertir grandes cantidades de dinero para desarrollar y mejorar su presencia en Internet. Esto generó una feroz competencia entre las operadoras de comunicaciones y los proveedores de hardware y software para satisfacer la mayor demanda de infraestructura. Como resultado, el **ancho de banda** (la capacidad que tiene las líneas de comunicación para transportar información) en Internet se incrementó de manera considerable, mientras que los costos del hardware se desplomaron.

1.11.2 World Wide Web: cómo facilitar el uso de Internet

World Wide Web (conocida simplemente como "Web") es una colección de hardware y software asociado con Internet que permite a los usuarios de computadora localizar y ver documentos basados en multimedia (documentos con diversas combinaciones de texto, gráficos, animaciones, sonido y videos) sobre casi cualquier tema. La introducción de Web fue un evento relativamente reciente. En 1989, Tim Berners-Lee de CERN (la Organización europea de investigación nuclear) comenzó a desarrollar una tecnología para compartir información a través de documentos de texto con "hipervínculos". Berners-Lee llamó a su invención el **Lenguaje de marcado de hipertexto (HTML)**. También escribió protocolos de comunicaciones como el **Protocolo de transferencia de hipertexto (HTTP)** para formar la espina dorsal de su nuevo sistema de información de hipertexto, al cual denominó World Wide Web.

En 1994, Berners-Lee fundó una organización conocida como **Consorcio World Wide Web (W3C**, www.w3.org), dedicada al desarrollo de tecnologías Web. Uno de los principales objetivos del W3C es que Web sea accesible en forma universal para todos, sin importar las discapacidades, el idioma o la cultura. En este libro usted usará Java para crear aplicaciones basadas en Web.

1.11.3 Servicios Web y *mashups*

En el capítulo 32 incluimos un tratamiento detallado sobre los servicios Web (figura 1.17). La metodología de desarrollo de aplicaciones conocida como *mashups* le permite desarrollar rápidamente poderosas y asombrosas aplicaciones de software, al combinar servicios Web complementarios (a menudo gratuitos) y otras fuentes de información. Uno de los primeros mashups combinaba los listados de bienes raíces proporcionados por `www.craigslist.org` con las capacidades de generación de mapas de *Google Maps* para ofrecer mapas que mostraran las ubicaciones de las casas en venta o renta dentro de cierta área.

Fuente de servicios Web	Cómo se utiiza
Google Maps	Servicios de mapas
Twitter	Microblogs

Fig. 1.17 | Algunos servicios Web populares (`http://www.programmableweb.com/category/all/apis`) (parte 1 de 2).

Fuente de servicios Web	Cómo se utiiza
YouTube	Búsquedas de videos
Facebook	Redes sociales
Instagram	Compartir fotografías
Foursquare	Registros de visitas (check-ins) móviles
LinkedIn	Redes sociales para negocios
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 la relación con los clientes (CRM)
Skype	Telefonía por Internet
Microsoft Bing	Búsqueda
Flickr	Compartir fotografías
Zillow	Precios de bienes raíces
Yahoo Search	Búsqueda
WeatherBug	Clima

Fig. 1.17 | Algunos servicios Web populares (`http://www.programmableweb.com/category/all/apis`) (parte 2 de 2).

1.11.4 Ajax

Ajax 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 se su apariencia visual se asemeja a las aplicaciones de escritorio. Aunque en este libro no hablaremos sobre la programación "pura" con Ajax (que es bastante compleja), en el capítulo 31 le mostraremos cómo crear aplicaciones habilitadas para Ajax mediante el uso de los componentes de JavaServer Faces (JSF) habilitados para Ajax.

1.11.5 Internet de las cosas

Internet ya no sólo es una red de computadoras: es una **Internet de las cosas**. Una *cosa* es cualquier objeto con una dirección IP y la habilidad de enviar datos de manera automática a través de una red (por ejemplo, un auto con un transpondedor para pagar peaje, un monitor cardiaco implantado en un humano, un medidor inteligente que reporta el uso de energía, aplicaciones móviles que pueden rastrear su movimiento y ubicación, y termostatos inteligentes que ajustan las temperaturas del cuarto con base en los pronósticos del clima y la actividad en el hogar). En el capítulo 28 en línea usted usará direcciones IP para crear aplicaciones en red.

1.12 Tecnologías de software

La figura 1.18 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
Desarrollo ágil de software	El **desarrollo ágil de software** es un conjunto de metodologías que tratan de implementar software con más rapidez y mediante el uso de menos recursos. 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 programas para hacerlos más claros y fáciles de mantener, al tiempo que se conserva su funcionalidad e integridad. Es muy utilizado con las metodologías de desarrollo ágil. Muchos IDE contienen *herramientas de refactorización* integradas para realizar de manera automática la mayor parte del proceso de refactorización.
Patrones de diseño	Los **patrones de diseño** son arquitecturas comprobadas para construir software orientado a objetos flexible y que pueda mantenerse. El campo de los patrones de diseño trata de enumerar esos 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 N analizaremos los patrones de diseño de Java.
LAMP	**LAMP** es un acrónimo para las tecnologías de código fuente abierto que muchos desarrolladores usan 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). MySQL es un sistema manejador de base de datos de código abierto. PHP es el lenguaje servidor de "secuencias de comandos" de código fuente abierto más popular para el desarrollo de aplicaciones Web. Apache es el software servidor Web más popular. El equivalente para el desarrollo en Windows es WAMP: *Windows*, Apache, MySQL y PHP.
Software como un servicio (SaaS)	Por lo general, el software siempre se ha visto como un producto; la mayoría del software aún se ofrece de esta forma. Si desea ejecutar una aplicación, tiene que comprarla a un distribuidor de software: a menudo un CD, DVD o descarga Web. Después instala ese software en la computadora y lo ejecuta cuando sea necesario. A medida que aparecen nuevas versiones, debe actualizar el software, lo cual genera con frecuencia un costo considerable en tiempo y dinero. 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)**, el software se ejecuta en servidores ubicados en cualquier parte de Internet. Cuando se actualizan esos servidores, los clientes en todo el mundo ven las nuevas capacidades; no se requiere 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 (PaaS)	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. Algunos proveedores de PaaS son: Google App Engine, Amazon EC2 y Windows Azure™.

Fig. 1.18 | Tecnologías de software (parte 1 de 2).

Tecnología	Descripción
Computación en la nube	SaaS y PaaS son ejemplos de **computación en la nube**. Puede usar el software y los datos almacenados en la "nube" (es decir, se accede a éstos mediante computadoras remotas o servidores, a través de Internet y están disponibles bajo demanda) en vez de tenerlos almacenados en su computadora de escritorio, notebook o dispositivo móvil. Esto le permite aumentar o disminuir los recursos de cómputo para satisfacer sus necesidades en cualquier momento dado, lo cual es más efectivo en costo que comprar hardware para ofrecer suficiente almacenamiento y poder de procesamiento para satisfacer las demandas pico ocasionales. La computación en la nube también ahorra dinero al transferir al proveedor de servicios la carga de administrar estas aplicaciones.
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. 1.18 | Tecnologías de software (parte 2 de 2).

El software es complejo. Las aplicaciones de software extensas que se usan en el mundo real pueden tardar muchos meses, o incluso años, en diseñarse e implementarse. Cuando hay grandes productos de software en desarrollo, por lo general se ponen a disponibilidad de las comunidades de usuarios como una serie de versiones, cada una más completa y pulida que la anterior (figura 1.19).

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, están incompletas y son inestables; además se liberan a un pequeño número de desarrolladores para que evalúen las nuevas características, para obtener retroalimentación lo más pronto posible, etcétera.
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	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).
Liberación de versión final	Cualquier error que aparezca en el candidato para liberación 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 se aloja en la *nube* (no se instala en su computadora) y evoluciona de manera constante, de modo que los usuarios siempre dispongan de la versión más reciente.

Fig. 1.19 | Terminología de liberación de versiones de productos de software.

1.13 Cómo estar al día con las tecnologías de información

La figura 1.20 muestra una lista de las publicaciones técnicas y comerciales clave que le ayudarán a mantenerse 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 `www.deitel.com/ResourceCenters.html`.

Publicación	URL
AllThingsD	`allthingsd.com`
Bloomberg BusinessWeek	`www.businessweek.com`
CNET	`news.cnet.com`
Comunications of the ACM	`cacm.acm.org`
Computerworld	`www.computerworld.com`
Engadget	`www.engadget.com`
eWeek	`www.eweek.com`
Fast Company	`www.fastcompany.com/`
Fortune	`money.cnn.com/magazines/fortune/`
GigaOM	`gigaom.com`
Hacker News	`news.ycombinator.com`
IEEE Computer Magazine	`www.computer.org/portal/web/computingnow/computer`
InfoWorld	`www.infoworld.com`
Mashable	`mashable.com`
PCWorld	`www.pcworld.com`
SD Times	`www.sdtimes.com`
Slashdot	`slashdot.org/`
Technology Review	`technologyreview.com`
Techcrunch	`techcrunch.com`
The Next Web	`thenextweb.com`
The Verge	`www.theverge.com`
Wired	`www.wired.com`

Fig. 1.20 | Publicaciones técnicas y comerciales.

Ejercicios de autoevaluación

1.1 Complete las siguientes oraciones:

a) Las computadoras procesan datos bajo el control de conjuntos de instrucciones conocidas como _____ .

b) Las unidades lógicas clave de la computadora son _____, _____, _____, _____, _____ y _____ .

c) Los tres tipos de lenguajes descritos en este capítulo son _____, _____ y _____ .

d) Los programas que traducen programas en lenguaje de alto nivel a lenguaje máquina se denominan _____ .

e) _____ es un sistema operativo para dispositivos móviles, basado en el kernel de Linux y en Java.
f) En general, el software _____ tiene todas sus características completas, está (supuestamente) libre de errores y listo para que la comunidad lo utilice.
g) 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 código fuente 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 de archivo _____ .

1.3 Complete las siguientes oraciones (con base en la sección 1.5):
a) Los objetos permiten la práctica de diseño para _____ ; 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) Es posible crear una nueva clase de objetos convenientemente mediante _____ , la nueva clase (subclase) comienza con las características de una clase existente (la superclase), posiblemente personalizándolas y agregando sus propias características únicas.
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) programas. b) unidad de entrada, unidad de salida, unidad de memoria, unidad central de procesamiento, unidad aritmética y lógica, unidad de almacenamiento secundario. c) lenguajes máquina, lenguajes ensambladores, lenguajes de alto nivel. d) compiladores. e) Android. f) candidato de liberación. g) acelerómetro.

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

1.3 a) ocultar información. b) clases. c) análisis y diseño orientados a objetos (A/DOO). d) la 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 misma para que ésta la utilice se llama _____ .
b) Al 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 a varios dispositivos, de manera que la información 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) El lenguaje de programación _____ 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) El (la) _____ se asegura de que los mensajes, que consisten en piezas numeradas en forma secuencial conocidas como bytes, se enruten de manera apropiada del emisor hasta el receptor, que lleguen intactos y se ensamblen en el orden correcto.
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 ella.
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 Uno de los objetos más comunes en el mundo es el reloj de pulsera. 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), modelado, mensajes, encapsulamiento, interfaz y ocultamiento de información.

Marcando la diferencia

Hemos incluido en este libro ejercicios Marcando la diferencia, en los que le pediremos que trabaje con problemas que son de verdad importantes para individuos, comunidades, países y para el mundo. Estos ejercicios le serán muy útiles en la práctica profesional.

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 carbono. Las organizaciones y los individuos se preocupan cada vez más por el "impacto ambiental debido al 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. Pruebe estas calculadoras 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 calcular el impacto ambiental del carbono.

1.10 ***(Prueba práctica: calculadora del índice de masa corporal)*** La obesidad provoca aumentos considerables en las 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 `http://www.nhlbi.nih.gov/guidelines/obesity/BMI/bmicalc.htm`. Úsela para calcular su propio IMC. Un próximo ejercicio le pedirá que programe su propia calculadora del IMC. Como preparación, le sugerimos usar la Web para 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 vehículos híbridos se están volviendo cada vez más populares, puesto que por lo general pueden ofrecer mucho más kilometraje que los vehículos 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, etc.).

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 "esposo" y "esposa" por "cónyuge", "hombre" y "mujer" por "persona", "hija" e "hijo" por "descendiente"), explique el procedimiento que utilizaría para leer un párrafo de texto y realizar estos reemplazos en forma manual. ¿Cómo podría su procedimiento hacer las adaptaciones de género de los artículos que acompañan a las palabras reemplazadas? Pronto aprenderá que un término más formal para "procedimiento" es "algoritmo", y que un algoritmo especifica los pasos a realizar, además del orden en el que se deben llevar a cabo. Le mostraremos cómo desarrollar algoritmos y luego convertirlos en programas de Java que se puedan ejecutar en computadoras.

2

Introducción a las aplicaciones en Java: entrada/salida y operadores



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

—William Shakespeare

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

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

2.3 Edición de su primer programa en Java

2.4 Cómo mostrar texto con `printf`

2.5 Otra aplicación: suma de enteros

- 2.5.1 Declaraciones `import`
- 2.5.2 Declaración de la clase `Suma`
- 2.5.3 Declaración y creación de un objeto `Scanner` para obtener la entrada del usuario mediante el teclado
- 2.5.4 Declaración de variables para almacenar enteros
- 2.5.5 Cómo pedir la entrada al usuario
- 2.5.6 Cómo obtener un valor `int` como entrada del usuario
- 2.5.7 Cómo pedir e introducir un segundo `int`
- 2.5.8 Uso de variables en un cálculo
- 2.5.9 Cómo mostrar el resultado del cálculo
- 2.5.10 Documentación de la API de Java

2.6 Conceptos acerca de la memoria

2.7 Aritmética

2.8 Toma de decisiones: operadores de igualdad y relacionales

2.9 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Marcando la diferencia*

2.1 Introducción

En este capítulo le presentaremos la programación de aplicaciones en Java. Empezaremos con ejemplos de programas que muestran (dan salida a) mensajes en la pantalla. Después veremos un programa que obtiene (da entrada a) dos números de un usuario, calcula la suma y muestra el resultado. Usted aprenderá cómo ordenar a la computadora que realice cálculos aritméticos y guardar sus resultados para usarlos más adelante. El último ejemplo demuestra cómo tomar decisiones. La aplicación compara dos números y después muestra mensajes con los resultados de la comparación. Usará las herramientas de la línea de comandos del JDK para compilar y ejecutar los programas de este capítulo. Si prefiere usar un entorno de desarrollo integrado (IDE), también publicamos videos Dive Into® en `http://www.deitel.com/books/jhtp10/` para Eclipse, NetBeans e IntelliJ IDEA.

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

Una **aplicación** en 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.

```
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
```

Fig. 2.1 | Programa para imprimir texto (parte 1 de 2).

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

Fig. 2.1 | Programa para imprimir texto (parte 2 de 2). *Nota*: En los códigos se han omitido los acentos para evitar la incompatibilidad al momento de ejecutarse en distintos entornos.

El programa incluye números de línea, que incluimos para fines académicos; *no* son parte de un programa en Java. Este ejemplo ilustra varias características importantes de Java. Pronto veremos que la línea 9 se encarga del verdadero trabajo: mostrar la frase `Bienvenido a la programacion en Java!` en la pantalla.

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**, el cual termina al final de la línea en la que aparecen los caracteres `//`. Un comentario de fin de línea no necesita empezar una línea; también puede estar en medio de ella y continuar hasta el final (como en las líneas 6, 10 y 11). La línea 2

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

según nuestra convención, 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 comentarios de fin de línea de los lenguajes de programación C y C++, respectivamente. Nosotros preferimos usar los comentarios con `//`.

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. Dichos comentarios son el formato preferido en la industria. El **programa de utilería `javadoc`** (parte del JDK) lee esos comentarios y los utiliza para preparar la documentación de su programa, en formato HTML. En el apéndice G en línea, 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 un 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 compilación del programa. Al encontrar un error de sintaxis, el compilador genera un mensaje de error. Debe eliminar todos los errores de compilación para que su programa se compile de manera correcta.*



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.



Tip para prevenir errores 2.1

Cuando escriba nuevos programas o modifique alguno que ya exista, mantenga sus comentarios actualizados con el código. A menudo los programadores tendrán que realizar cambios en el código existente para corregir errores o mejorar capacidades. Al actualizar sus comentarios, ayuda a asegurar que éstos reflejen con precisión lo que el código hace. Esto facilitará la comprensión y edición de su programa en el futuro. Los programadores que usan o actualizan código con comentarios obsoletos podrían realizar suposiciones incorrectas sobre el código, lo cual podría conducir a errores o incluso infracciones de seguridad.

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 compilador ignora el espacio en blanco.



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, la cual debe ir seguida de inmediato por el **nombre de la clase** (`Bienvenido1`). Las **palabras clave** (también conocidas como **palabras reservadas**) se reservan 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.

En los capítulos 2 al 7, todas las clases que definimos comienzan con la palabra clave `public`. Por ahora, sólo recuerde que debemos usar `public`. En el capítulo 8 aprenderá más sobre las clases `public` y las que no son `public`.

Nombre de archivo para una clase `public`

Una clase `public` *debe* colocarse en un archivo que tenga el nombre de la forma *NombreClase*`.java`. Por lo tanto, la clase `Bienvenido1` se almacenará en el archivo `Bienvenido1.java`.



Error común de programación 2.2

Ocurrirá un error de compilación si el nombre de archivo de una clase `public` no es exactamente el mismo nombre que el de la clase (tanto por su escritura como por el uso de mayúsculas y minúsculas) seguido de la extensión `.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**, es decir, una serie de caracteres que pueden ser letras, dígitos, guiones bajos (_) y signos de moneda ($), que *no* comiencen con un dígito *ni* tengan espacios. Algunos identificadores válidos son `Bienvenido1`, `$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).

Cuerpo de la clase
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.



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. Este formato enfatiza la estructura de la declaración de la clase, y facilita su lectura. Usamos tres espacios para formar un nivel de sangría; muchos programadores prefieren dos o cuatro espacios. Sea cual sea el formato que utilice, hágalo de manera consistente.



Tip para prevenir errores 2.2
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 la llave derecha de cierre por usted cuando escribe la llave izquierda de apertura.



Error común de programación 2.3
Es un error de sintaxis no utilizar las llaves por pares.



Buena práctica de programación 2.4
Por lo general los IDE insertan por usted la sangría en el código. También puede usar la tecla Tab *para aplicar sangría al código. Puede configurar cada IDE para especificar el número de espacios insertados cuando 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 éstas hayan concluido. En la sección 3.2.5 explicaremos el propósito de la palabra clave `static`. 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 los caracteres contenidos entre los signos de comillas dobles (las comillas dobles *no* se muestran en la salida). En conjunto, las comillas dobles y los caracteres entre ellas son una **cadena**, lo que también se conoce como **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.

`System.out` (que usted predefinió) se conoce como el **objeto de salida estándar**. Permite a una aplicación de Java mostrar información en la **ventana de comandos** desde la cual se ejecuta. 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. Cuando el método `System.out.println` completa su tarea, coloca el cursor de salida (la ubicación donde se mostrará el siguiente carácter) al principio de la siguiente línea de 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**. 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.

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. Estos mensajes no siempre indican el problema exacto en el código. Cuando se encuentre con un mensaje de error, éste le dará una idea de qué fue lo que ocasionó el error [intente quitando un punto y coma o una llave en el programa de la figura 2.1, y vuelva a compilarlo para 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 no encuentra errores en esa línea, verifique varias de las líneas anteriores.

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

Como ayuda para los principiantes en la programación, 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 termina la declaración de una clase. Por ejemplo, la línea 10

```
} // fin del método main
```

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

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

indica la llave de cierre de la clase `Bienvenido1`. Cada comentario indica el método o la clase que termina con esa llave derecha. Después de este capítulo omitiremos estos comentarios finales.

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. Para ayudarlos a compilar y ejecutar sus programas en un IDE, proporcionamos videos Dive Into® para los IDE populares Eclipse, NetBeans e IntelliJ IDEA. Estos videos se encuentran en el sitio Web del libro:

```
http://www.deitel.com/books/jhtp10
```

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 compilación, este comando 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.



Error común de programación 2.4

Cuando use `javac`, 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. Esto indica que la variable de entorno `PATH` del sistema no se estableció de manera apropiada. Consulte las instrucciones de instalación en la sección Antes de empezar de 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.

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.



Error común de programación 2.5

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.

Ejecución de la aplicación `Bienvenido1`

Las siguientes instrucciones asumen que los ejemplos del libro se encuentran en `C:\ejemplos` en Windows o en la carpeta `Documents/ejemplos` de su cuenta de usuario en Linux u OS X. Para ejecutar este programa en una ventana de comandos cambie al directorio que contiene `Bienvenido1.java` (`C:\ejemplos\cap02\fig02_01` en Microsoft Windows o `~/Documents/ejemplos/cap02/fig02_01` en Linux/OS X). A continuación, escriba

```
java Bienvenido1
```

y oprima *Intro*. Este comando inicia la JVM, la cual carga el archivo `Bienvenido1.class`. El comando *omite* la extensión `.class` del nombre de archivo; de lo contrario, la JVM *no* ejecutará el programa. La JVM llama al método `main` de `Bienvenido1`. A continuación, la instrucción de la línea 9 de `main` muestra "`Bienvenido a la programacion en Java!`". La figura 2.2 muestra el programa ejecutándose en una ventana de **Símbolo del sistema** de Microsoft Windows [*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].



Tip para prevenir errores 2.4

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 las instrucciones de instalación en la sección Antes de empezar de este libro. En algunos sistemas, tal vez necesite reiniciar su equipo o abrir un nuevo símbolo del sistema después de configurar la variable CLASSPATH.*

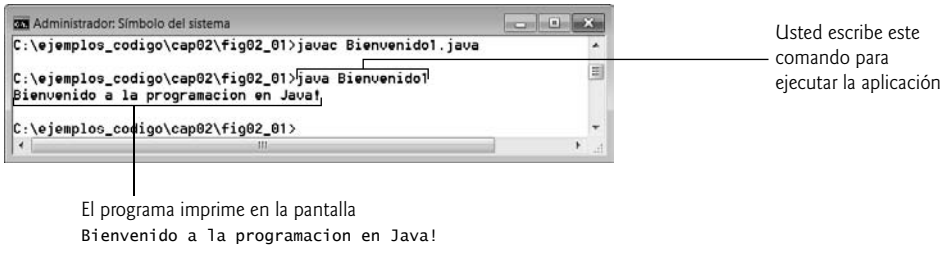


Fig. 2.2 | Ejecución de `Bienvenido1` desde el **Símbolo del sistema**.

2.3 Edición de su 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].

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

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

```
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 | 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 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 líneas 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     {
```

Fig. 2.4 | Imprimir varias líneas de texto con una sola instrucción (parte 1 de 2).

```
 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 | Imprimir varias líneas de texto con una sola instrucción (parte 2 de 2).

La línea 9

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

muestra cuatro líneas 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 la combina con el siguiente carácter 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 listan 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

```
http://docs.oracle.com/javase/specs/jls/se7/html/jls-3.html#jls-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\"");` muestra "`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 en dos líneas las cadenas "`Bienvenido a`" y "`la programacion en Java!`".

```
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

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

llaman al método `System.out.println` 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**. Al proceso de llamar a un método también se le conoce como **invocar** un método.



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.

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 2.6

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**. El método `printf` 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 desplegar. Los especificadores de formato también pueden incluir información de formato opcional.

Los especificadores de formato 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 desplegar dos cadenas, y que a cada cadena 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.

Cabe mencionar que, en vez de usar la secuencia de escape `\n`, usamos el especificador de formato **`%n`**, el cual es un separador de línea *portable* entre distintos sistemas operativos. No puede usar `%n` en el argu-

mento para `System.out.print` o `System.out.println`; sin embargo, el separador de línea que produce `System.out.println` *después* de mostrar su argumento, *es* portable entre distintos sistemas operativos. El apéndice I en línea presenta más detalles sobre cómo dar formato a la salida con `printf`.

2.5 Otra aplicación: 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 esos datos a través de elementos del programa 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**). Como en los programas anteriores, las líneas 1 y 2 indican el número de figura, nombre de archivo y propósito del programa.

```
1  // Fig. 2.7: Suma.java
2  // Programa que recibe dos números y muestra la suma.
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
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 recibe dos números y muestra la suma.

2.5.1 Declaraciones `import`

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`. Así, el compilador se asegura de que utilice la clase en forma correcta.



Error común de programación 2.7

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 la declaración de la misma, es un error de sintaxis.



Error común de programación 2.8

Si olvida incluir una declaración `import` para una clase que debe importarse, se produce un error de compilación que contiene un mensaje tal como "`cannot find symbol`". Cuando esto ocurra, verifique que haya proporcionado las declaraciones `import` apropiadas y que los nombres en las mismas estén escritos correctamente, incluyendo el uso apropiado de las letras mayúsculas y minúsculas.



Observación de ingeniería de software 2.1

*En cada nueva versión de Java, por lo general las API contienen nuevas herramientas que corrigen errores, mejoran el rendimiento u ofrecen mejores formas de realizar tareas. Las correspondientes versiones anteriores ya no son necesarias, por lo que no deben usarse. Se dice que dichas API están **obsoletas** y podrían retirarse de versiones posteriores de Java.*

A menudo se encontrará con versiones obsoletas de API cuando explore la documentación de las API. El compilador le advertirá cuando compile código que utilice API obsoletas. Si compila su código con `javac` mediante el uso del argumento de línea de comandos `–deprecation`, el compilador le indicará las características obsoletas que está usando. Para cada una, la documentación en línea (`http://docs.oracle.com/javase/7/docs/api/`) indica la característica obsoleta y por lo general contiene vínculos hacia la nueva característica que la sustituye.

2.5.2 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. Al método `main` se le aplica un nivel de sangría en el cuerpo de la clase `Suma`, y al código en el cuerpo de `main` se le aplica otro nivel para mejorar la legibilidad.

2.5.3 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 una variable permite al programa acceder al *valor* de la variable en memoria. El

nombre de una variable puede ser cualquier identificador válido; de nuevo, una serie de caracteres compuestos por letras, dígitos, guiones bajos (_) y signos de moneda ($) que *no* comiencen con un dígito y *no* contengan espacios. El *tipo* de una variable especifica el tipo 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 los datos.

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 leer los *bytes* de datos escritos por el usuario. El objeto `Scanner` traduce estos bytes en tipos (como `int`) que se pueden usar en un programa.

2.5.4 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` que use en un programa 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 (Vg. `A`), un dígito (Vg. `7`), un carácter especial (Vg. `*` o `%`) o una secuencia de escape (como el carácter de tabulación, `\t`). 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, separando con comas los nombres de las variables (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 su propia declaración. Este formato permite insertar un comentario descriptivo enseguida 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 con sólo leerlo, en lugar de leer la documentación asociada o crear y 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 del nombre que van después de la primera, deben empezar con una letra mayúscula. Por ejemplo, el identificador `primerNumero` *tiene una* `N` *mayúscula en su segunda palabra,* `Numero`*. A esta convención de nombres se le conoce como* `CamelCase`*, ya que las mayúsculas sobresalen de forma similar a la joroba de un camello.*

2.5.5 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. 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 2.2

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`*.*

2.5.6 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

Coloque espacios en ambos lados de un operador binario para mejorar la legibilidad del programa.

2.5.7 Cómo pedir e introducir un segundo `int`

La línea 20

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

indica 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`.

2.5.8 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`". 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.

2.5.9 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 de 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` son opcionales; se incluyen para enfatizar que el valor de *toda* la expresión se imprime en la posición del especificador de formato `%d`. Se dice que dichos paréntesis son **redundantes**.

2.5.10 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

```
http://docs.oracle.com/javase/7/docs/api/index.html
```

También puede descargar esta documentación de la sección Additional Resources (Recursos adicionales) en

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

El apéndice F 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, por lo que se dice que este proceso es *destructivo*.



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.

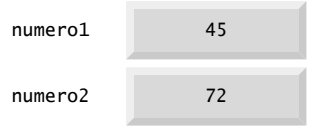


Fig. 2.9 | Ubicaciones de memoria, después de almacenar valores para `numero1` y `numero2`.

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. 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*.

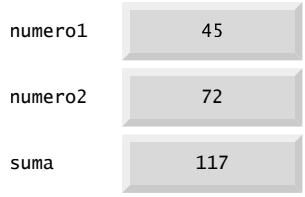


Fig. 2.10 | Ubicaciones de memoria, después de almacenar la suma de `numero1` y `numero2`.

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 `+` junto con 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 $x \div y$	`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* (es decir, 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 la de 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 no es generalmente 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 dentro de expresiones aritméticas en una secuencia precisa, determinada por las siguientes **reglas de precedencia de operadores**, que por lo general 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 operadores se asocian de derecha a izquierda. La figura 2.12 sintetiza estas reglas de precedencia de operadores. En el apéndice A se incluye 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.

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

1 Utilizamos ejemplos simples para explicar el *orden de evaluación* de las expresiones. Existen situaciones sutiles que se presentan 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* (`http://docs.oracle.com/javase/specs/jls/se7/html/index.html`).

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

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 la 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 los exponentes, x^2 se representa como `x * x`. La sección 5.4 muestra una alternativa para los exponentes). 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.

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;
```

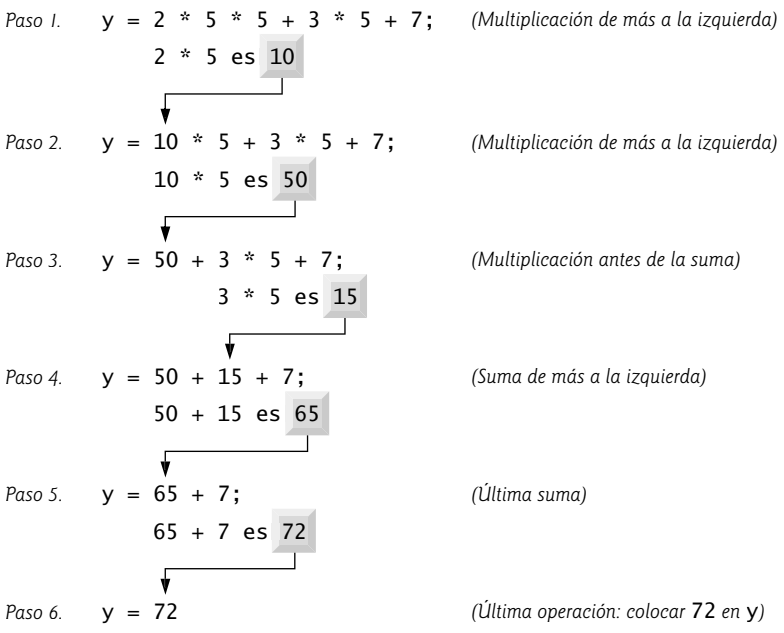


Fig. 2.13 | Orden en el cual se evalúa un polinomio de segundo grado.

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 "la 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*, se ejecuta el cuerpo de la instrucción `if`. Si la condición es *falsa*, no se ejecuta el cuerpo. 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 tipos de 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*.

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

Fig. 2.14 | Operadores de igualdad y relacionales (parte 1 de 2).

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 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 (parte 2 de 2).

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.

```
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);
```

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

```
31
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, del 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);
```

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` usando 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 2.9

Confundir el operador de igualdad (==) con el de asignación (=) puede producir un error lógico o de compilación. El operador de igualdad debe leerse como "es igual a", 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`. Dicho punto y coma 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);
```

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 ejecutaría, *sin importar* que la condición sea verdadera o falsa, ya que la instrucción de salida no forma parte de la instrucción `if`.

Espacio en blanco

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 pueden espaciarse de acuerdo con las preferencias del programador, sin afectar el significado de un programa. Es incorrecto dividir identificadores y cadenas. Idealmente las instrucciones deben mantenerse lo más reducidas que sea posible, pero no siempre se puede hacer esto.



Tip para prevenir errores 2.5

Una instrucción larga puede repartirse en varias líneas. Si una sola instrucción debe dividirse en 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 en dos o más líneas, aplique sangría a todas las líneas subsecuentes hasta el final de la instrucción.

Operadores descritos hasta ahora

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, `=`, se asocia de *derecha a izquierda*. El valor de una expresión de asignación es el que se haya asignado a la variable del lado izquierdo del operador `=` (por ejemplo, el valor de la expresión `x = 7` es `7`). Entonces, 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`.

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.



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 forzarlo, en la misma forma que lo haría con las expresiones algebraicas.

2.9 Conclusión

En este capítulo aprendió sobre muchas características importantes de Java, como mostrar datos en la pantalla en un `Símbolo del sistema`, introducir datos mediante el teclado, realizar cálculos y tomar decisiones. Mediante las aplicaciones que vimos en este capítulo, le presentamos muchos 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`, ya que 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 esas clases en las aplicaciones.

Resumen

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

- Una aplicación de Java (pág. 35) se ejecuta cuando utilizamos el comando `java` para iniciar la JVM.
- Los comentarios (pág. 36) 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. 36) se pueden distribuir en varias líneas; están delimitados por `/*` y `*/`.
- Los comentarios Javadoc (pág. 36) 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. 36; 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. 37). El espacio en blanco mejora la legibilidad de los programas y el compilador lo ignora.
- Las palabras clave (pág. 37) están reservadas para el uso exclusivo de Java, y siempre se escriben con letras minúsculas.
- La palabra clave `class` (pág. 37) 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; es decir, 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 y minúsculas (pág. 38); lo que significa que las letras mayúsculas y las minúsculas son distintas.
- El cuerpo de todas las declaraciones de clases (pág. 38) debe estar delimitado por llaves, `{` y `}`.
- La declaración de una clase `public` (pág. 37) 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. 38) es el punto de inicio de toda aplicación en Java, y debe empezar con:

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

  de lo contrario, la JVM no ejecutará la aplicación.
- Los métodos pueden realizar tareas y devolver información cuando completan estas tareas. La palabra clave `void` (pág. 38) 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. 39) entre comillas dobles se le conoce como cadena de caracteres o literal de cadena.
- El objeto de salida estándar (`System.out`; pág. 39) muestra caracteres en la ventana de comandos.
- El método `System.out.println` (pág. 39) muestra su argumento (pág. 39) 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. 40) 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` seguida del nombre de la clase que contiene a `main`.

Sección 2.3 Edición de su primer programa en Java

- `System.out.print` (pág. 42) 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. 43). Java lo combina con el siguiente carácter para formar una secuencia de escape (pág. 43). La secuencia de escape `\n` (pág. 43) representa el carácter de nueva línea.

Sección 2.4 Cómo mostrar texto con `printf`

- El método `System.out.printf` (pág. 43; `f` se refiere a "formato") muestra datos con formato.
- El primer argumento del método `printf` es una cadena de formato (pág. 44) que contiene texto fijo o especificadores de formato. Cada especificador de formato (pág. 44) 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. 44) es un receptáculo para una cadena.
- El especificador de formato `%n` (pág. 44) es un separador de línea portable. No puede usar `%n` en el argumento de `System.out.print` o `System.out.println`; sin embargo, el separador de línea que muestra `System.out.println` después de su argumento, es portable entre diversos sistemas operativos.

Sección 2.5 Otra aplicación: suma de enteros

- Una declaración `import` (pág. 46) ayuda al compilador a localizar una clase que se utiliza en un programa.
- El extenso conjunto de clases predefinidas de Java se agrupa en paquetes (pág. 45) denominados grupos de clases. A éstos se les conoce como la biblioteca de clases de Java (pág. 46), o la Interfaz de Programación de Aplicaciones de Java (API de Java).
- Una variable (pág. 46) es una ubicación en la memoria de la computadora, en la cual se puede guardar un valor para usarlo posteriormente 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 al valor de la variable en memoria.
- Un objeto `Scanner` (paquete `java.util`; pág. 47) 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.
- Para poder usarlas en un programa, las variables deben inicializarse (pág. 47).
- La expresión `new Scanner(System.in)` crea un objeto `Scanner` que lee datos desde el objeto de entrada estándar (`System.in`; pág. 47); por lo general es el teclado.
- El tipo de datos `int` (pág. 47) 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. 47) especifican números reales con puntos decimales, como `3.4` y `-11.19`.
- Las variables de tipo `char` (pág. 47) representan caracteres individuales, como una letra mayúscula (por ejemplo, `A`), un dígito (Vg. `7`), un carácter especial (Vg. `*` o `%`) o una secuencia de escape (Vg. tab, `\t`).
- Los tipos como `int`, `float`, `double` y `char` son tipos primitivos (pág. 47). Los nombres de los tipos primitivos son palabras clave; por ende, deben aparecer escritos sólo con letras minúsculas.
- Un indicador (pág. 48) 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. 48), permite al programa dar un valor a una variable. Se llama operador binario (pág. 48), ya que tiene dos operandos.
- Las partes de las instrucciones que tienen valores se llaman expresiones (pág. 49).
- El especificador de formato `%d` (pág. 49) es un receptáculo para un valor `int`.

Sección 2.6 Conceptos acerca de la memoria

- Los nombres de las variables (pág. 50) 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. 51) son + (suma), – (resta), * (multiplicación), / (división) y % (residuo).
- La división de enteros (pág. 51) produce un cociente entero.
- El operador residuo, % (pág. 51), produce el residuo después de la división.
- Las expresiones aritméticas deben escribirse en formato de línea recta (pág. 51).
- Si una expresión contiene paréntesis anidados (pág. 52), el conjunto de paréntesis más interno se evalúa primero.
- Java aplica los operadores dentro de las expresiones aritméticas en una secuencia precisa, la cual se determina mediante las reglas de precedencia de los operadores (pág. 52).
- Cuando decimos que los operadores se aplican de izquierda a derecha, nos referimos a su asociatividad (pág. 52). Algunos operadores se asocian de derecha a izquierda.
- Los paréntesis redundantes (pág. 53) pueden hacer que una expresión sea más clara.

Sección 2.8 Toma de decisiones: operadores de igualdad y relacionales

- La instrucción `if` (pág. 54) 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. 54).
- 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. 57) es una instrucción que no realiza ninguna 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 al ejecutarse el programa, la computadora imprima el texto que va después de los caracteres `//` en la pantalla.
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.printf` y dos especificadores de formato `%s`.
f) Si la variable `numero` no es igual a `7`, mostrar "`La variable numero no es igual a 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) Usar `System.out.printf` para 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.printf("%s%n%s%n", "Este es un", "programa en Java");
```
f)
```
if (numero != 7)
   System.out.println("La variable numero no es igual a 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 `=>` por `>=`.

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 de autoevaluación 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) En un programa en Java puede tomarse una decisión usando 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 $y = ax3+ 7$, ¿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 números, 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 número más grande, seguido de las palabras "`es más grande`". Si los números 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 dar como resultado 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.printf("*%n**%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 cálculo como el valor que se imprimirá en una instrucción `System.out.printf`. Los valores producidos por los cálculos del área y de 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 en una considerable variedad de símbolos especiales. Cada carácter tiene su correspondiente representación entera. El conjunto de caracteres que utiliza una computadora, junto con 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 encerrar ese carácter entre comillas sencillas, como en `'A'`.

Usted puede determinar el equivalente entero de un carácter si antepone a ese carácter 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. 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 escribe un número con más de cinco dígitos? ¿Qué ocurre cuando 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.

```
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.

Marcando 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 la reducción 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, los objetos, los métodos y las cadenas

3



Sus servidores públicos le sirven bien.

—Adlai E. Stevenson

Nada puede tener valor sin ser un objeto de utilidad.

—Karl Marx

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.
- Llamar a los métodos de un objeto para hacer que realicen sus tareas.
- Identificar cuáles son las variables locales de un método y qué diferencia tienen de las variables de instancia.
- Determinar cuáles son los tipos primitivos y los tipos por referencia.
- Utilizar un constructor para inicializar los datos de un objeto.
- Representar y usar números que contengan puntos decimales.

Plan general

3.1 Introducción

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

- 3.2.1 La clase `Cuenta` con una variable de instancia, un método *establecer* y un método *obtener*
- 3.2.2 La clase `PruebaCuenta` que crea y usa un objeto de la clase `Cuenta`
- 3.2.3 Compilación y ejecución de una aplicación con varias clases
- 3.2.4 Diagrama de clases en UML de `Cuenta` con una variable de instancia y métodos *establecer* y *obtener*
- 3.2.5 Observaciones adicionales sobre la clase `PruebaCuenta`
- 3.2.6 Ingeniería de software con variables de instancia `private` y métodos *establecer* y *obtener* `public`

3.3 Comparación entre tipos primitivos y tipos por referencia

3.4 La clase `Cuenta`: inicialización de objetos mediante constructores

- 3.4.1 Declaración de un constructor de `Cuenta` para la inicialización personalizada de objetos
- 3.4.2 La clase `PruebaCuenta`: inicialización de objetos `Cuenta` cuando se crean

3.5 La clase `Cuenta` con un saldo: los números de punto flotante

- 3.5.1 La clase `Cuenta` con una variable de instancia llamada `saldo` de tipo `double`
- 3.5.2 La clase `PruebaCuenta` que usa la clase `Cuenta`

3.6 (Opcional) Ejemplo práctico de GUI y gráficos: uso de cuadros de diálogo

3.7 Conclusión

Resumen | Ejercicios de autoevaluación | Respuestas a los ejercicios de autoevaluación | Ejercicios | Marcando la diferencia

3.1 Introducción

[*Nota*: este capítulo depende de la terminología y los conceptos de la programación orientada a objetos que le presentamos en la sección 1.5, Introducción a la tecnología de los objetos].

En el capítulo 2 trabajó con clases, objetos y métodos *existentes*. Usó el objeto de salida estándar *predefinido* `System.out`, *invocó* a sus métodos `print`, `println` y `printf` para mostrar información en la pantalla. Usó el objeto de la clase *existente* `Scanner` para crear un objeto que lee y coloca en memoria datos enteros introducidos por el usuario mediante el teclado. A lo largo del libro usará muchas más clases y objetos *preexistentes*: ésta es una de las grandes ventajas de Java como lenguaje de programación orientado a objetos.

En este capítulo aprenderá a crear sus propias clases y métodos. Cada *nueva* clase que cree se convertirá en un nuevo *tipo* que podrá usarse para declarar variables y crear objetos. Puede declarar nuevas clases según sea necesario; ésta es una razón por la que Java se conoce como lenguaje *extensible*.

Vamos a presentar un ejemplo práctico sobre la creación y el uso de una clase de cuenta bancaria real: `Cuenta`. Dicha clase debe mantener como *variables de instancia* atributos tales como su `nombre` y `saldo`, además de proporcionar *métodos* para tareas tales como consultar el saldo (`obtenerSaldo`), realizar depósitos que incrementen el saldo (`depositar`) y realizar retiros que disminuyan el saldo (`retirar`). En los ejemplos del capítulo agregaremos los métodos `obtenerSaldo` y `depositar` a la clase, así como el método `retirar` en los ejercicios.

En el capítulo 2 usamos el tipo de datos `int` para representar enteros. En este capítulo introduciremos el tipo de datos `double` para representar el saldo de una cuenta como un número que puede contener un *punto decimal*; a dichos números se les conoce como *números de punto flotante* [en el capítulo 8, cuando profundicemos un poco más en la tecnología de objetos, comenzaremos a representar las cantidades monetarias en forma precisa con la clase `BigDecimal` (paquete `java.math`), como se debe hacer a la hora de escribir aplicaciones monetarias de uso industrial].

Por lo general, las aplicaciones que usted desarrolle en este libro consistirán en dos o más clases. Si se integra a un equipo de desarrollo en la industria, tal vez trabaje con aplicaciones que contengan cientos o incluso miles de clases.

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

En esta sección creará dos clases: `Cuenta` (figura 3.1) y `PruebaCuenta` (figura 3.2). La clase `PruebaCuenta` es una *clase de aplicación* en la que el método `main` creará y usará un objeto `Cuenta` para demostrar las capacidades de la clase `Cuenta`.

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

Por lo general las distintas cuentas tienen nombres diferentes. Por esta razón, la clase `Cuenta` (figura 3.1) contiene una *variable de instancia* llamada `nombre`. Las variables de instancia de una clase mantienen los datos para cada objeto (es decir, cada instancia) de la clase. Más adelante en el capítulo agregaremos una variable de instancia llamada `saldo` para poder llevar el registro de cuánto dinero hay en la cuenta. La clase `Cuenta` contiene dos métodos: el método `establecerNombre` almacena un nombre en un objeto `Cuenta` y el método `obtenerNombre` obtiene el nombre de un objeto `Cuenta`.

```
1  // Fig. 3.1: Cuenta.java
2  // Clase Cuenta que contiene una variable de instancia llamada nombre
3  // y métodos para establecer y obtener su valor.
4
5  public class Cuenta
6  {
7     private String nombre; // variable de instancia
8
9     // método para establecer el nombre en el objeto
10    public void establecerNombre(String nombre)
11    {
12       this.nombre = nombre; // almacenar el nombre
13    }
14
15    // método para obtener el nombre del objeto
16    public String obtenerNombre()
17    {
18       return nombre; // devuelve el valor de nombre a quien lo invocó
19    }
20 } // fin de la clase Cuenta
```

Fig. 3.1 | La clase `Cuenta` que contiene una variable de instancia llamada `nombre` y métodos para *establecer* y *obtener* su valor.

Declaración de la clase
La *declaración de la clase* empieza en la línea 5:

```
public class Cuenta
```

La palabra clave `public` (que el capítulo 8 explica con detalle) es un **modificador de acceso**. Por ahora, simplemente declararemos todas las clases como `public`. Cada declaración de clase `public` debe almacenarse en un archivo de texto que tenga el *mismo* nombre que la clase y termine con la extensión de nombre

de archivo `.java`; de lo contrario, ocurrirá un error de compilación. Por ende, las clases `Cuenta` y `PruebaCuenta` (figura 3.2) *deben* declararse en los archivos *independientes* `Cuenta.java` y `PruebaCuenta.java`, respectivamente.

Cada declaración de clase contiene la palabra clave `class`, seguida inmediatamente por el nombre de la clase; en este caso, `Cuenta`. El cuerpo de toda clase se encierra entre un par de llaves, una izquierda y una derecha, como en las líneas 6 y 20 de figura 3.1.

Los identificadores y la nomenclatura de lomo de camello (CamelCase)

Los nombres de las clases, los métodos y las variables son todos *identificadores* y, por convención, usan el mismo esquema de nomenclatura *lomo de camello* que vimos en el capítulo 2. Además, por convención los nombres de las clases comienzan con una letra *mayúscula*, y los nombres de los métodos y variables comienzan con una letra en *minúscula*.

La variable de instancia `nombre`

En la sección 1.5 vimos que un objeto tiene atributos, los cuales se implementan como variables de instancia y los lleva consigo durante su vida útil. Las variables de instancia existen antes de invocar los métodos en un objeto, durante su ejecución y después de que éstos terminan su ejecución. Cada objeto (instancia) de la clase tiene su *propia* copia de las variables de instancia de la clase. Por lo general una clase contiene uno o más métodos que manipulan a las variables de instancia que pertenecen a objetos específicos de la clase.

Las variables de instancia se declaran *dentro* de la declaración de una clase pero *fuera* de los cuerpos de los métodos de la misma. La línea 7

```
private String nombre; // variable de instancia
```

declara la variable de instancia de tipo `String` *fuera* de los cuerpos de los métodos `establecerNombre` (líneas 10 a 13) y `obtenerNombre` (líneas 16 a 19). Las variables `String` pueden contener valores de cadenas de caracteres tales como "`Jane Green`". Si hay muchos objetos `Cuenta`, cada uno tiene su propio `nombre`. Puesto que `nombre` es una variable de instancia, cada uno de los métodos de la clase puede manipularla.



Buena práctica de programación 3.1

Preferimos listar primero las variables de instancia de una clase en el cuerpo de la misma, para que usted pueda ver los nombres y tipos de las variables antes de usarlas en los métodos de la clase. Es posible listar las variables de instancia 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 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. Por lo tanto, la variable `nombre` sólo puede utilizarse en los métodos de cada objeto `Cuenta` (`establecerNombre` y `obtenerNombre` en este caso). Pronto verá que esto presenta poderosas oportunidades de ingeniería de software.

El método `establecerNombre` *de la clase* `Cuenta`

Vamos a recorrer el código de la declaración del método `establecerNombre` (líneas 10 a 13):

```
public void establecerNombre(String nombre) —— Esta línea es el encabezado del método
{
        this.nombre = nombre; // almacenar el nombre
}
```

Nos referimos a la primera línea de la declaración de cada método (línea 10 en este caso) como el *encabezado del método*. El **tipo de valor de retorno** del método (que aparece antes del nombre del método) especifica el tipo de datos que el método devuelve a *quien lo llamó* después de realizar su tarea. El tipo de valor de retorno `void` (línea 10) indica que `establecerNombre` realizará una tarea pero *no* regresará (es decir, no devolverá) ninguna información a quien lo invocó. En capítulo 2 usó métodos que devuelven información; por ejemplo, usó el método `Scanner` `nextInt` para recibir como entrada un entero escrito por el usuario mediante el teclado. Cuando `nextInt` lee un valor del usuario, *devuelve* ese valor para usarlo en el programa. Como pronto veremos, el método `obtenerNombre` de `Cuenta` devuelve un valor.

El método `establecerNombre` recibe el *parámetro* `nombre` de tipo `String`, el cual representa el nombre que se pasará al método como un *argumento*. Cuando hablemos sobre la llamada al método en la línea 21 de la figura 3.2, podrá ver cómo trabajan juntos los parámetros y los argumentos.

Los parámetros se declaran en una **lista de parámetros**, la cual se encuentra dentro de los paréntesis que van después del nombre del método en el encabezado del mismo. Cuando hay varios parámetros, cada uno va separado del siguiente mediante una coma. Cada parámetro *debe* especificar un tipo (en este caso, `String`) seguido de un nombre de variable (en este caso, `nombre`).

Los parámetros son variables locales

En el capítulo 2 declaramos todas las variables de una aplicación en el método `main`. Las variables declaradas en el cuerpo de un método específico (como `main`) son variables locales, las cuales pueden usarse *sólo* en ese método. Cada método puede acceder sólo a sus propias variables locales y no a las de los otros métodos. Cuando un método termina, se *pierden* los valores de sus variables locales. Los parámetros de un método también son variables locales del mismo.

El cuerpo del método `establecerNombre`

Cada *cuerpo de método* está delimitado por un par de *llaves* (como en las líneas 11 y 13 de la figura 3.1), las cuales contienen una o más instrucciones que realizan las tareas del método. En este caso, el cuerpo del método contiene una sola instrucción (línea 12) que asigna el valor del *parámetro* `nombre` (un objeto `String`) a la *variable de instancia* `nombre` de la clase, con lo cual almacena el nombre de la cuenta en el objeto.

Si un método contiene una variable local con el *mismo* nombre que una variable de instancia (como en las líneas 10 y 7, respectivamente), el cuerpo de ese método se referirá a la variable local en vez de a la variable de instancia. En este caso, se dice que la variable local *oculta* a la variable de instancia en el cuerpo del método, el cual puede usar la palabra clave `this` para referirse de manera explícita a la variable de instancia oculta, como se muestra en el lado izquierdo de la asignación en la línea 12.



Buena práctica de programación 3.2

Podríamos haber evitado el uso de la palabra clave `this` *aquí si eligiéramos un nombre diferente para el parámetro en la línea 10, pero usar la palabra clave* `this` *como se muestra en la línea 12 es una práctica aceptada ampliamente para minimizar la proliferación de nombres de identificadores.*

Una vez que se ejecuta la línea 12, el método completa su tarea por lo que regresa a *quien lo llamó*. Como pronto veremos, la instrucción en la línea 21 de `main` (figura 3.2) llama al método `establecerNombre`.

El método `obtenerNombre` *de la clase* `Cuenta`

El método `obtenerNombre` (líneas 16 a 19)

```
public String obtenerNombre()
{
    return nombre; // devuelve el valor de nombre a quien lo invocó
}
```

La palabra clave `return` devuelve el nombre de `String` a quien invocó al método.

devuelve el `nombre` de un objeto `Cuenta` específico a quien lo llamó. El método tiene una lista de parámetros *vacía*, por lo que *no* requiere información adicional para realizar su tarea. El método devuelve un objeto `String`. Cuando un método que especifica un tipo de valor de retorno *distinto* de `void` se invoca y completa su tarea, *debe* devolver un resultado a quien lo llamó. Una instrucción que llama al método `obtenerNombre` en un objeto `Cuenta` (como los de las líneas 16 y 26 de la figura 3.2) espera recibir el nombre de ese objeto `Cuenta`, es decir, 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 de la figura 3.1 regresa el valor `String` de la variable de instancia `nombre` a quien hizo la llamada. Por ejemplo, cuando el valor se devuelve a la instrucción en las líneas 25 y 26 de la figura 3.2, la instrucción usa ese valor para mostrar el nombre en pantalla.

3.2.2 La clase `PruebaCuenta` que crea y usa un objeto de la clase `Cuenta`

A continuación, nos gustaría utilizar la clase `Cuenta` en una aplicación y *llamar* a cada uno de sus métodos. Una clase que contiene el método `main` empieza la ejecución de una aplicación de Java. La clase `Cuenta` *no* se puede ejecutar a sí misma ya que *no* contiene un método `main`. Si escribe `java Cuenta` en el símbolo del sistema, obtendrá el siguiente error: "`Main method not found in class Cuenta`". Para corregir este problema, debe declarar una clase *separada* que contenga un método `main` o colocar un método `main` en la clase `Cuenta`.

La clase controladora `PruebaCuenta`

Para ayudarlo a prepararse para los programas extensos que encontrará más adelante en este libro y en la industria, utilizamos una clase separada `PruebaCuenta` (figura 3.2) que contiene el método `main` para probar la clase `Cuenta`. Una vez que `main` comienza a ejecutarse, puede llamar a otros métodos en ésta y otras clases; a su vez, estos métodos pueden llamar a otros métodos, y así en lo sucesivo. El método `main` de la clase `PruebaCuenta` crea un objeto `Cuenta` y llama a sus métodos `obtenerNombre` y `establecerNombre`. A este tipo de clases se le conoce algunas veces como *clase controladora*. Así como un objeto `Persona` conduce un objeto `Auto` diciéndole lo que debe hacer (ir más rápido o más lento, girar a la izquierda o a la derecha, etc.), la clase `PruebaCuenta` conduce un objeto `Cuenta` y llama a sus métodos para decirle lo que debe hacer.

```
1  // Fig. 3.2: PruebaCuenta.java
2  // Crear y manipular un objeto Cuenta.
3  import java.util.Scanner;
4
5  public class PruebaCuenta
6  {
7     public static void main(String[] args)
8     {
9        // crea un objeto Scanner para obtener la entrada desde el símbolo del sistema
10       Scanner entrada = new Scanner(System.in);
11
12       // crea un objeto Cuenta y lo asigna a miCuenta
13       Cuenta miCuenta = new Cuenta();
14
15       // muestra el valor inicial del nombre (null)
16       System.out.printf("El nombre inicial es: %s%n%n", miCuenta.obtenerNombre());
17
```

Fig. 3.2 | Creación y manipulación de un objeto `Cuenta` (parte 1 de 2).

```
18      // pide y lee el nombre
19      System.out.println("Introduzca el nombre:");
20      String elNombre = entrada.nextLine(); // lee una línea de texto
21      miCuenta.establecerNombre(elNombre); // coloca elNombre en miCuenta
22      System.out.println(); // imprime una línea en blanco
23
24      // muestra el nombre almacenado en el objeto miCuenta
25      System.out.printf("El nombre en el objeto miCuenta es:%n%s%n",
26          miCuenta.obtenerNombre());
27    }
28  } // fin de la clase PruebaCuenta
```

```
El nombre inicial es: null

Introduzca el nombre:
Alfonso Romero

El nombre en el objeto miCuenta es:
Alfonso Romero
```

Fig. 3.2 | Creación y manipulación de un objeto `Cuenta` (parte 2 de 2).

Objeto `Scanner` *para recibir entrada del usuario*

La línea 10 crea un objeto `Scanner` llamado `input` para recibir como entrada el nombre del usuario. La línea 19 pide al usuario que introduzca un nombre. La línea 20 usa el método **`nextLine`** del objeto `Scanner` para leer el nombre del usuario y asignarlo a la variable *local* `elNombre`. Usted escribe el nombre y oprime *Intro* para enviarlo al programa. Al oprimir *Intro* se inserta un carácter de nueva línea después de los caracteres que escribió. El método `nextLine` lee caracteres (incluyendo caracteres de espacio en blanco, como el espacio en "`Alfonso Romero`") hasta encontrar la nueva línea, la cual *descarta*.

La clase `Scanner` proporciona varios métodos de entrada más, como veremos a lo largo del libro. Hay un método similar a `nextLine` (llamado **`next`**) que lee la *siguiente palabra*. Cuando oprime *Intro* después de escribir algo de texto, el método `next` lee caracteres hasta encontrar un *carácter de espacio en blanco* (como un espacio, un tabulador o una nueva línea) y luego devuelve un objeto `String` que contiene los caracteres hasta (*sin* incluir) el carácter de espacio en blanco, el cual se *descarta*. La información que está después del primer carácter de espacio en blanco *no se pierde*: puede leerse en otras instrucciones que llamen a los métodos de `Scanner` posteriormente en el programa.

Instanciación de un objeto: la palabra clave **`new`** *y los constructores*

La línea 13 crea un objeto `Cuenta` y lo asigna a la variable `miCuenta` de tipo `Cuenta`. La variable `miCuenta` se inicializa con el resultado de la **expresión de creación de instancia de clase** `new Cuenta()`. La palabra clave **`new`** crea un nuevo objeto de la clase especificada; en este caso, `Cuenta`. Los paréntesis a la derecha de `Cuenta` son *obligatorios*. Como veremos en la sección 3.4, esos paréntesis en combinación con el nombre de una clase representan una llamada a un **constructor**, que es *similar* a un método pero es invocado de manera implícita por el operador `new` para *inicializar* las variables de instancia de un objeto al momento de *crearlo*. En la sección 3.4 veremos cómo colocar un *argumento* en los paréntesis para especificar un *valor inicial* para la variable de instancia `nombre` de un objeto `Cuenta`; para poder hacer esto usted mejorará la clase `Cuenta`. Por ahora sólo dejaremos los paréntesis *vacíos*. La línea 10 contiene una expresión de creación de instancia de clase para un objeto `Scanner`. La expresión inicializa el objeto `Scanner` con `System.in`, que indica a `Scanner` de dónde debe leer la entrada (por ejemplo, del teclado).

Llamada al método `obtenerNombre` de la clase `Cuenta`

La línea 16 muestra el nombre *inicial*, que se obtiene mediante una llamada al método `obtenerNombre` del objeto. Así como podemos usar el objeto `System.out` para llamar a sus métodos `print`, `printf` y `println`, también podemos usar el objeto `miCuenta` para llamar a sus métodos `obtenerNombre` y `establecerNombre`. La línea 16 llama al método `obtenerNombre` usando el objeto `miCuenta` creado en la línea 13, seguido tanto de un **separador punto** (.), como del nombre del método `obtenerNombre` y de un conjunto *vacío* de paréntesis ya que no se pasan argumentos. Cuando se hace la llamada a `obtenerNombre`:

1. La aplicación transfiere la ejecución del programa de la llamada (línea 16 en `main`) a la declaración del método `obtenerNombre` (líneas 16 a 19 de la figura 3.1). Como la llamada a `obtenerNombre` fue a través del objeto `miCuenta`, `obtenerNombre` "sabe" qué variable de instancia del objeto manipular.

2. A continuación, el método `obtenerNombre` realiza su tarea; es decir, *devuelve* el `nombre` (línea 18 de la figura 3.1). Cuando se ejecuta la instrucción `return`, la ejecución del programa continúa a partir de donde se hizo la llamada a `obtenerNombre` (línea 16 de la figura 3.2).

3. `System.out.printf` muestra el objeto `String` devuelto por el método `obtenerNombre` y luego el programa continúa su ejecución en la línea 19 de `main`.



Tip para prevenir errores 3.1

Nunca use como control de formato una cadena que el usuario haya introducido. Cuando el método `System.out.printf` evalúa la cadena de control de formato en su primer argumento, el método realiza las tareas con base en los especificadores de conversión de esa cadena. Si la cadena de control de formato se obtuviera del usuario, un usuario malicioso podría proveer especificadores de conversión que `System.out.printf` ejecutara, lo que probablemente generará una infracción de seguridad.

`null`: el valor inicial predeterminado de las variables `String`

La primera línea de la salida muestra el nombre "`null`". A diferencia de las variables locales, que *no* se inicializan de manera automática, *cada variable de instancia tiene un* **valor inicial predeterminado**, que es un valor que Java proporciona cuando el programador *no* especifica el valor inicial de la variable de instancia. Por ende, *no* se requiere que las *variables de instancia* se inicialicen de manera explícita antes de usarlas en un programa, a menos que deban inicializarse con valores *distintos de* los predeterminados. El valor predeterminado para una variable de instancia de tipo `String` (como `nombre` en este ejemplo) es `null`, de lo cual hablaremos con más detalle en la sección 3.3, cuando consideremos los *tipos por referencia*.

Llamada al método `establecerNombre` de la clase `Cuenta`

La línea 21 llama al método `establecerNombre` de la clase `miCuenta`. La llamada a un método puede proveer *argumentos* cuyos *valores* se asignen a los parámetros correspondientes del método. En este caso, el valor de la variable local entre paréntesis `elNombre` de `main` es el *argumento* que se pasa a `establecerNombre` para que el método pueda realizar su tarea. Cuando se hace la llamada a `establecerNombre`:

1. La aplicación transfiere la ejecución del programa de la línea 21 en `main` a la declaración del método `establecerNombre` (líneas 10 a 13 de la figura 3.1) y el *valor del argumento* en los paréntesis de la llamada (`elNombre`) se asigna al *parámetro* correspondiente (`nombre`) en el encabezado del método (línea 10 de la figura 3.1). Puesto que `establecerNombre` se llamó a través del objeto `miCuenta`, `establecerNombre` "sabe" qué variable de instancia del objeto manipular.

2. A continuación, el método `establecerNombre` realiza su tarea; es decir, asigna el valor del parámetro `nombre` a la variable de instancia `nombre` (línea 12 de la figura 3.1).

3. Cuando la ejecución del programa llega a la llave derecha de cierre de `establecerNombre`, regresa hasta donde se hizo la llamada a `establecerNombre` (línea 21 de la figura 3.2) y continúa en la línea 22 de la figura 3.2.

El número de *argumentos* en la llamada a un método *debe coincidir* con el número de *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 tipos de los parámetros correspondientes en la declaración del método (como veremos en el capítulo 6, *no* se requiere que el tipo de un argumento y el tipo de su correspondiente parámetro sean *idénticos*). En nuestro ejemplo, la llamada al método pasa un argumento de tipo `String` (`elNombre`) y la declaración del método especifica un parámetro de tipo `String` (`nombre`, declarado en la línea 10 de la figura 3.1). 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.

Obtención del nombre que el usuario introdujo
La línea 22 de la figura 3.2 muestra una línea en blanco. Cuando se ejecuta la segunda llamada al método `obtenerNombre` (línea 26), se muestra el nombre introducido por el usuario en la línea 20. Cuando la instrucción en las líneas 25 y 26 termina de ejecutarse, se llega al final del método `main`, por lo que el programa termina.

3.2.3 Compilación y ejecución de una aplicación con varias clases

Debe compilar las clases de las figuras 3.1 y 3.2 antes de poder *ejecutar* la aplicación. Ésta es la primera vez que crea una aplicación con *varias* clases. La clase `PruebaCuenta` tiene un método `main`; la clase `Cuenta` no. Para compilar esta aplicación, primero cambie al directorio que contiene los archivos de código fuente de la aplicación. Después, escriba el comando

```
javac Cuenta.java PruebaCuenta.java
```

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

```
javac *.java
```

El asterisco (*) en `*.java` indica que deben compilarse *todos* los archivos del directorio *actual* que terminen con la extensión de nombre de archivo "`.java`". Si ambas clases se compilan en forma correcta (es decir, que no aparezcan errores de compilación), podrá entonces ejecutar la aplicación con el comando

```
java PruebaCuenta
```

3.2.4 Diagrama de clases en UML de `Cuenta` con una variable de instancia y métodos *establecer* y *obtener*

Usaremos con frecuencia los diagramas de clases en UML para sintetizar los *atributos* y las *operaciones* de una clase. En la industria, los diagramas de UML ayudan a los diseñadores de sistemas a especificar un sistema de una forma concisa, gráfica e independiente del lenguaje de programación, antes de que los programadores implementen el sistema en un lenguaje de programación específico. La figura 3.3 presenta un **diagrama de clases de UML** para la clase `Cuenta` de la figura 3.1.

Compartimiento superior
En UML, cada clase se modela en un diagrama de clases en forma de un rectángulo con tres compartimientos. En este diagrama el compartimiento *superior* contiene el *nombre de la clase* `Cuenta`, centrado horizontalmente y en negrita.

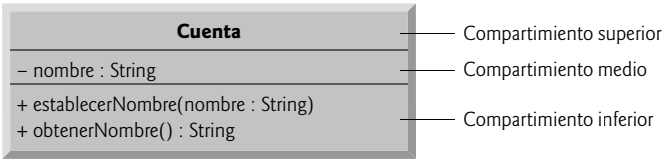


Fig. 3.3 | Diagrama de clases de UML para la clase `Cuenta` de la figura 3.1.

Compartimiento medio
El compartimiento *medio* contiene el *atributo de la clase* `nombre`, que corresponde a la variable de instancia del mismo nombre en Java. La variable de instancia `nombre` es `private` en Java, por lo que el diagrama de clases en UML muestra un *modificador de acceso de signo menos (-)* antes del nombre del atributo. Después del nombre del atributo hay un *signo de dos puntos* y el *tipo del atributo*; en este caso `String`.

Compartimiento inferior
El compartimiento *inferior* contiene las **operaciones** de la clase, `establecerNombre` y `obtenerNombre`, que en Java corresponden a los métodos de los mismos nombres. Para modelar las operaciones, UML lista el nombre de la operación precedido por un *modificador de acceso*; en este caso, `+ obtenerNombre`. Este signo más (+) indica que `obtenerNombre` es una operación pública (`public`) en UML (porque es un método `public` en Java). La operación `obtenerNombre` *no* tiene parámetros, por lo que los paréntesis después del nombre de la operación en el diagrama de clases están *vacíos*, así como en la declaración del método en la línea 16 de la figura 3.1. La operación `establecerNombre`, que también es pública, tiene un parámetro `String` llamado `nombre`.

Tipos de valores de retorno
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 `obtenerNombre` de la clase `Cuenta` (figura 3.1) tiene un tipo de valor de retorno `String`. El método `establecerNombre` *no* devuelve un valor (porque devuelve `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 esta operación.

Parámetros
La forma en que UML modela un parámetro es un poco distinta a la de Java, ya que lista el nombre del parámetro, seguido de dos puntos y del tipo del parámetro entre paréntesis después del nombre de la operación. UML tiene sus propios tipos de datos, que son similares a los de Java, pero por cuestión de simpleza usaremos los tipos de datos de Java. El método `establecerNombre` de `Cuenta` (figura 3.1) tiene un parámetro `String` llamado `nombre`, por lo que en la figura 3.3 se lista a `nombre : String` entre los paréntesis que van después del nombre del método.

3.2.5 Observaciones adicionales sobre la clase `PruebaCuenta`

El método `static main`
En el capítulo 2, cada clase que declaramos tenía un método llamado `main`. 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. Es necesario llamar a la mayoría de los otros métodos para decirles de manera *explícita* que realicen sus tareas.

Las líneas 7 a la 27 de la figura 3.2 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 este caso, la clase `PruebaCuenta`. En el capítulo 6 analizaremos los métodos `static` con detalle.

Observaciones sobre las declaraciones `import`

Observe la declaración `import` en la figura 3.2 (línea 3), la cual indica al compilador que el programa utiliza la clase `Scanner`. Como vimos en el capítulo 2, 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 del disco, como las clases `Cuenta` y `PruebaCuenta`. 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 del 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 la clase `PruebaCuenta` utiliza a la clase `Cuenta`.

La declaración `import` en la línea 3 *no* es obligatoria si hacemos referencia a la clase `Scanner` en este archivo como `java.util.Scanner`, que incluye el *nombre completo del paquete y de la clase*. Esto se conoce como el **nombre de clase completamente calificado**. Por ejemplo, la línea 10 de la figura 3.2 también 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 el nombre de clase completamente calificado se especifica cada vez que se utilice el nombre de una clase. La mayoría de los programadores de Java prefieren usar el estilo de programación más conciso mediante las declaraciones* `import`.

3.2.6 Ingeniería de software con variables de instancia `private` y métodos *establecer* y *obtener* `public`

Como veremos, por medio de los métodos *establecer* y *obtener* es posible *validar* el intento de modificaciones a los datos `private` y controlar cómo se presentan esos datos al que hace la llamada; éstos son beneficios de ingeniería de software convincentes. En la sección 3.5 hablaremos sobre esto con más detalle.

Si la variable de instancia fuera `public`, cualquier **cliente** de la clase (es decir, cualquier otra clase que llama a los métodos de la clase) podría ver los datos y hacer lo que quisiera con ellos, incluyendo establecer un valor *no válido*.

Podría pensar que, aun cuando un cliente de la clase no pueda acceder de manera directa a una variable de instancia `private`, el cliente puede hacer lo que quiera con la variable a través de métodos *establecer* y *obtener* `public`. Cabría pensar que podemos echar un vistazo a los datos `private` en cualquier momento con el método *obtener* `public` y que podemos modificar los datos `private` por medio del método *establecer* `public`. Pero los métodos *establecer* pueden programarse para *validar* sus argumentos y rechazar los intentos de *establecer* datos con valores incorrectos, como una temperatura corporal negativa, un día en marzo fuera del rango 1 a 31, un código de producto que no está en el catálogo de productos de la compañía, etc. Además, un método *obtener* puede presentar los datos en un formato diferente. Por ejemplo, una clase `Calificacion` podría almacenar una calificación como un `int` entre 0 y 100, pero un método `obtenerCalificacion` podría devolver una calificación de letra como un objeto `String`; por ejemplo, "`A`" para las calificaciones entre 90 y 100, "`B`" para las calificaciones entre 80 y 89, etc.

Un control estricto del acceso y la presentación de los datos `private` puede reducir los errores de manera considerable, e incrementar al mismo tiempo la robustez y seguridad de sus programas.

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 `Cuenta`, la variable `nombre` se *encapsula* (oculta) en el objeto, y sólo está accesible para los métodos de la clase de ese objeto.



Observación de ingeniería de software 3.2

Es necesario colocar un modificador de acceso antes de cada declaración de una variable de instancia y de un método. Por lo general, las variables de instancia deben declararse como `private` *y los métodos como* `public`*. Más adelante en el libro hablaremos del por qué sería conveniente declarar un método como* `private`*.*

Vista conceptual de un objeto `Cuenta` *con datos encapsulados*

Podemos pensar en un objeto `Cuenta` como se muestra en la figura 3.4. El nombre de la variable de instancia `private` está *oculto dentro* del objeto (se representa mediante el círculo interno que contiene `nombre`) y *protegido por una capa exterior* de métodos `public` (representados por el círculo exterior que contiene `obtenerNombre` y `establecerNombre`). Cualquier código cliente que necesite interactuar con el objeto `Cuenta` puede hacerlo *solamente* a través de llamadas a los métodos `public` de la capa exterior protectora.

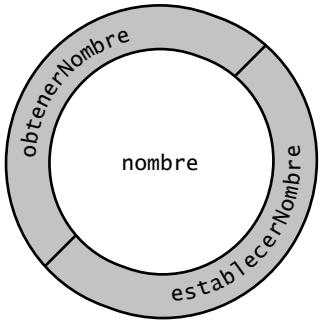


Fig. 3.4 | Vista conceptual de un objeto `Cuenta` con su variable de instancia `private` llamada `nombre` encapsulada y con la capa protectora de métodos `public`.

3.3 Comparación entre tipos primitivos y tipos por referencia

Los tipos en Java se dividen en tipos primitivos y **tipos por referencia**. En el capítulo 2 trabajó con variables de tipo `int`: uno de los tipos primitivos. Los otros tipos primitivos son `boolean`, `byte`, `char`, `short`, `long`, `float` y `double`, cada uno de los cuales veremos en este libro. En el apéndice D hay un resumen de ellos. Todos los tipos no primitivos son *tipos por referencia*, por lo que 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 entero a la vez. Cuando se asigna otro valor a esa variable, el nuevo valor sustituye su valor inicial, el cual se *pierde*.

Cabe mencionar que las variables locales *no* se inicializan de manera predeterminada. Las variables de instancia de tipo primitivo *se inicializan* de manera predeterminada; las variables de los tipos `byte`, `char`, `short`, `int`, `long`, `float` y `double` se inicializan con `0`, y las variables de tipo `boolean` se inicializan con

`false`. Usted puede especificar su propio valor inicial para una variable de tipo primitivo, al asignar a esa variable un valor en su declaración, como en

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

Los programas utilizan variables de tipo por referencia (que por lo general se llaman **referencias**) para almacenar las *direcciones* de los objetos en la memoria de la computadora. Se dice que dicha variable **hace referencia a un objeto** en el programa. Los *objetos* a los que se hace referencia pueden contener *muchas* variables de instancia. La línea 10 de la figura 3.2:

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

crea un objeto de la clase `Scanner` y luego asigna a la variable entrada una *referencia* a ese objeto `Scanner`. La línea 13 de la figura 3.2:

```
Cuenta miCuenta = new Cuenta();
```

crea un objeto de la clase `Cuenta` y luego asigna a la variable `miCuenta` una *referencia* a ese objeto `Cuenta`. *Las variables de instancia de tipo por referencia, si no se inicializan de manera explícita, lo hacen de manera predeterminada con el valor* `null` (una palabra reservada que representa una "referencia a nada"). Esto explica por qué la primera llamada a `obtenerNombre` en la línea 16 de la figura 3.2 devuelve `null`; *no* se había establecido todavía el valor de `nombre`, por lo que se devolvía el *valor inicial predeterminado* `null`.

Para llamar a los métodos de un objeto, necesita una referencia a ese objeto. En la figura 3.2, las instrucciones en el método `main` usan la variable `miCuenta` para llamar a los métodos `obtenerNombre` (líneas 16 y 26) y `establecerNombre` (línea 21) con el fin de interactuar con el objeto `Cuenta`. Las variables de tipos primitivos *no hacen* referencia a objetos, por lo que dichas variables *no pueden* usarse para llamar métodos.

3.4 La clase Cuenta: inicialización de objetos mediante constructores

Como mencionamos en la sección 3.2, cuando se crea un objeto de la clase `Cuenta` (figura 3.1), su variable de instancia `String` llamada `nombre` se inicializa de manera *predeterminada* con `null`. Pero ¿qué pasa si usted desea proporcionar un nombre a la hora de *crear* un objeto `Cuenta`?

Cada clase que usted declare puede proporcionar de manera opcional un *constructor* con parámetros que pueden utilizarse para inicializar un objeto de una clase al momento de crear ese objeto. Java *requiere* una llamada al constructor para *cada* objeto que se crea, por lo que éste es el punto ideal para inicializar las variables de instancia de un objeto. El siguiente ejemplo mejora la clase `Cuenta` (figura 3.5) con un constructor que puede recibir un nombre y usarlo para inicializar la variable de instancia `nombre` al momento de crear un objeto `Cuenta` (figura 3.6).

3.4.1 Declaración de un constructor de Cuenta para la inicialización personalizada de objetos

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 para un objeto `Cuenta` al momento de crear este objeto, como en la línea 10 de la figura 3.6:

```
Cuenta cuenta1 = new Cuenta("Jane Green");
```

En este caso, el argumento String "`Jane Green`" se pasa al constructor del objeto `Cuenta` y se utiliza para inicializar la variable de instancia `nombre`. La instrucción anterior requiere que la clase proporcione un constructor que sólo reciba un parámetro `String`. La figura 3.5 contiene una clase `Cuenta` modificada con dicho constructor.

```
1  // Fig. 3.5: Cuenta.java
2  // Clase Cuenta con un constructor que inicializa el nombre.
3
4  public class Cuenta
5  {
6     private String nombre;// variable de instancia
7
8     // el constructor inicializa nombre con el parámetro nombre
9     public Cuenta(String nombre)// el nombre del constructor es el nombre de la clase
10    {
11        this.nombre = nombre;
12    }
13
14    // método para establecer el nombre
15    public void establecerNombre(String nombre)
16    {
17        this.nombre = nombre;
18    }
19
20    // métodos para recuperar el nombre
21    public String obtenerNombre()
22    {
23        return nombre;
24    }
25 } // fin de la clase Cuenta
```

Fig. 3.5 | La clase `Cuenta` con un constructor que inicializa el `nombre`.

Declaración del constructor de `Cuenta`

Las líneas 9 a la 12 de la figura 3.5 declaran el constructor de `Cuenta`. Un constructor *debe* tener el *mismo nombre* que la clase. La *lista de parámetros* del constructor especifica que éste requiere una o más datos para realizar su tarea. La línea 9 indica que el constructor tiene un parámetro `String` llamado `nombre`. Cuando usted crea un nuevo objeto `Cuenta` (como veremos en la figura 3.6), pasa el nombre de una persona al constructor, el cual recibe ese nombre en el parámetro `nombre`. Después el constructor asigna `nombre` a la *variable de instancia* `nombre` en la línea 11.



Tip para prevenir errores 3.2

Aun cuando es posible hacerlo, no se recomienda llamar métodos desde los constructores. En el capítulo 10, Programación orientada a objetos: polimorfismo e interfaces, explicaremos esto.

El parámetro `nombre` *del constructor de la clase* `Cuenta` *y el método* `establecerNombre`

En la sección 3.2.1 vimos que los parámetros de un método son variables locales. En la figura 3.5, tanto el constructor como el método `establecerNombre` tienen un parámetro llamado `nombre`. Aunque estos parámetros tienen el mismo identificador (`nombre`), el parámetro en la línea 9 es una variable local del constructor que *no* está visible para el método `establecerNombre`, y el de la línea 15 es una variable local de `establecerNombre` que *no* está visible para el constructor.

3.4.2 La clase `PruebaCuenta`: inicialización de objetos `Cuenta` cuando se crean

El programa `PruebaCuenta` (figura 3.6) inicializa dos objetos `Cuenta` mediante el constructor. La línea 10 crea e inicializa el objeto `cuenta1` de la clase `Cuenta`. La palabra clave `new` solicita memoria del sistema para alma-

cenar el objeto Cuenta y luego llama de manera implícita al constructor de la clase para *inicializar* el objeto. La llamada se indica mediante los paréntesis después del nombre de la clase, los cuales contienen el *argumento* "Jane Green" que se usa para inicializar el nombre del nuevo objeto. La expresión de creación de la instancia de la clase en la línea 10 devuelve una *referencia* al nuevo objeto, el cual se asigna a la variable cuenta1. La línea 11 repite este proceso, pero esta vez se pasa el argumento "John Blue" para inicializar el nombre para cuenta2. En cada una de las líneas 14 y 15 se utiliza el método obtenerNombre para obtener los nombres y mostrar que se inicializaron en el momento en el que se *crearon* los objetos. La salida muestra nombres *diferentes*, lo que confirma que cada objeto Cuenta mantiene su *propia copia* de la variable de instancia nombre.

```
1  // Fig. 3.6: PruebaCuenta.java
2  // Uso del constructor de Cuenta para inicializar la variable de instancia
3  // nombre al momento de crear el objeto Cuenta.
4
5  public class PruebaCuenta
6  {
7     public static void main(String[] args)
8     {
9        // crear dos objetos Cuenta
10       Cuenta cuenta1 = new Cuenta("Jane Green");
11       Cuenta cuenta2 = new Cuenta("John Blue");
12
13       // muestra el valor inicial de nombre para cada Cuenta
14       System.out.printf("El nombre de cuenta1 es: %s%n", cuenta1.obtenerNombre());
15       System.out.printf("El nombre de cuenta2 es: %s%n", cuenta2.obtenerNombre());
16    }
17 } // fin de la clase PruebaCuenta
```

```
El nombre de cuenta1 es: Jane Green
El nombre de cuenta2 es: John Blue
```

Fig. 3.6 | Uso del constructor de Cuenta para inicializar la variable de instancia nombre al momento de crear el objeto Cuenta.

Los constructores no pueden devolver valores

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; más adelante en el libro explicaremos cuándo usar constructores private.

Constructor predeterminado

Recuerde que la línea 13 de la figura 3.2

```
Cuenta miCuenta = new Cuenta();
```

usó new para crear un objeto Cuenta. Los paréntesis *vacíos* después de "new Cuenta" indican una llamada al **constructor predeterminado de la clase**. En cualquier clase que *no* declare de manera explícita a un constructor, el compilador proporciona un constructor predeterminado (que nunca tiene parámetros). Cuando una clase sólo tiene el constructor predeterminado, sus variables de instancia se inicializan con sus *valores predeterminados*. En la sección 8.5 veremos que las clases pueden tener varios constructores.

No hay constructor predeterminado en una clase que declara a un constructor
Si usted declara un constructor para una clase, el compilador *no* creará un *constructor predeterminado* para esa clase. En ese caso no podrá crear un objeto `Cuenta` con la expresión de creación de instancia de clase `new Cuenta()` como lo hicimos en la figura 3.2, a menos que el constructor predeterminado que declare *no* reciba parámetros.



Observación de ingeniería de software 3.3
A menos que sea aceptable la inicialización predeterminada de las variables de instancia de su clase, usted deberá proporcionar un constructor para asegurarse que sus variables de instancia se inicialicen en forma apropiada con valores significativos a la hora de crear cada nuevo objeto de su clase.

Agregar el constructor al diagrama de clases de UML de la clase `Cuenta`
El diagrama de clases de UML de la figura 3.7 modela la clase `Cuenta` de la figura 3.5, la cual tiene un constructor con un parámetro llamado `nombre`, de tipo `String`. Al igual que las operaciones, UML modela a los constructores en el *tercer* compartimiento de un diagrama de clases. 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.

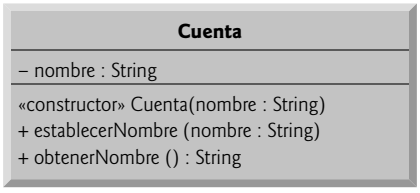


Fig. 3.7 | Diagrama de clases de UML para la clase `Cuenta` de la figura 3.5.

3.5 La clase Cuenta con un saldo: los números de punto flotante y el tipo double

Ahora vamos a declarar una clase `Cuenta` que mantiene el *saldo* de una cuenta bancaria además del nombre. La mayoría de los saldos de las cuentas no son números enteros. Por esta razón, la clase `Cuenta` representa el saldo de las cuentas como un **número de punto flotante**: un número con un *punto decimal*, como 43.95, 0.0, –129.8873. [En el capítulo 8 comenzaremos a representar las cantidades monetarias en forma precisa con la clase `BigDecimal`, como debería hacerse al escribir aplicaciones monetarias para la industria].

Java cuenta con dos tipos primitivos para almacenar números de punto flotante en la memoria: `float` y `double`. 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**. Éstas requieren el *doble* de memoria que las variables `float` y pueden contener hasta *15 dígitos significativos*; aproximadamente el *doble* de precisión de las variables `float`.

La mayoría de los programadores representan los números de punto flotante con el tipo `double`. De hecho, Java trata de manera predeterminada 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`. Dichos valores en el código fuente se conocen como **literales de punto flotante**. En el apéndice D, Tipos primitivos, podrá consultar los rangos precisos de los valores para los tipos `float` y `double`.

3.5.1 La clase Cuenta con una variable de instancia llamada saldo de tipo double

Nuestra siguiente aplicación contiene una versión de la clase Cuenta (figura 3.8) que mantiene como variables de instancia el nombre *y* 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 8 declara una variable de instancia de tipo double llamada saldo. Cada instancia (es decir, objeto) de la clase Cuenta contiene sus *propias* copias de nombre y saldo.

```
1  // Fig. 3.8: Cuenta.java
2  // La clase Cuenta con una variable de instancia double llamada saldo y un constructor
3  // además de un método llamado deposito que realiza validación.
4
5  public class Cuenta
6  {
7     private String nombre; // variable de instancia
8     private double saldo; // variable de instancia
9
10    // Constructor de Cuenta que recibe dos parámetros
11    public Cuenta(String nombre, double saldo)
12    {
13       this.nombre = nombre; // asigna nombre a la variable de instancia nombre
14
15       // valida que el saldo sea mayor que 0.0; de lo contrario,
16       // la variable de instancia saldo mantiene su valor inicial predeterminado de 0.0
17       if (saldo > 0.0) // si el saldo es válido
18          this.saldo = saldo; // lo asigna a la variable de instancia saldo
19    }
20
21    // método que deposita (suma) sólo una cantidad válida al saldo
22    public void depositar(double montoDeposito)
23    {
24       if (montoDeposito > 0.0) // si el montoDeposito es válido
25        saldo = saldo + montoDeposito; // lo suma al saldo
26    }
27
28    // método que devuelve el saldo de la cuenta
29    public double obtenerSaldo()
30    {
31       return saldo;
32    }
33
34    // método que establece el nombre
35    public void establecerNombre(String nombre)
36    {
37       this.nombre = nombre;
38    }
39
40    // método que devuelve el nombre
```

Fig. 3.8 | La clase Cuenta con una variable de instancia double llamada saldo y un constructor además de un método llamado depósito que realiza la validación (parte 1 de 2).

```
41    public String obtenerNombre()
42    {
43       return nombre; //devuelve el valor de name a quien lo invocó
44    } // fin del método obtenerNombre
45 } // fin de la clase Cuenta
```

Fig. 3.8 | La clase `Cuenta` con una variable de instancia `double` llamada saldo y un constructor además de un método llamado `depósito` que realiza la validación (parte 2 de 2).

Constructor de la clase **`Cuenta`** *con dos parámetros*

La clase tiene un *constructor* y cuatro *métodos*. Puesto que es común que alguien abra una cuenta para depositar dinero de inmediato, el constructor (líneas 11 a 19) recibe ahora un segundo parámetro llamado `saldoInicial` de tipo `double`, el cual representa el *saldo inicial* de la cuenta. Las líneas 17 y 18 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 **`depositar`** *de la clase* **`Cuenta`**

El método `depositar` (líneas 22 a la 26) *no* devuelve datos cuando completa su tarea, por lo que su tipo de valor de retorno es `void`. El método recibe un parámetro llamado `montoDeposito`: un valor `double` que se *sumará* al saldo *sólo* si el valor del parámetro es *válido* (es decir, mayor que cero). La línea 25 primero suma el valor actual de `saldo` al `montoDeposito` para formar una suma *temporal* que *después* se asigna a `saldo`, con lo cual se *sustituye* su valor anterior (recuerde que la suma tiene *mayor* precedencia que la asignación). Es importante entender que el cálculo del lado derecho del operador de asignación en la línea 25 *no* modifica el saldo; ésta es la razón por la que es necesaria la asignación.

El método **`obtenerSaldo`** *de la clase* **`Cuenta`**

El método `obtenerSaldo` (líneas 29 a la 32) permite a los *clientes* de la clase (es decir, otras clases cuyos métodos llamen a los métodos de 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*.

Todos los métodos de **`Cuenta`** *pueden usar la variable* **`saldo`**

Observe una vez más que las instrucciones en las líneas 18, 25 y 31 utilizan la variable de instancia `saldo`, aun 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.

3.5.2 La clase `PruebaCuenta` que usa la clase `Cuenta`

La clase `PruebaCuenta` (figura 3.9) crea dos objetos `Cuenta` (líneas 9 y 10) y los inicializa con un saldo *válido* de `50.00` y un saldo *no válido* de `-7.53`, respectivamente; para los fines de nuestros ejemplos vamos a suponer que los saldos deben ser mayores o iguales a cero. Las llamadas al método `System.out.printf` en las líneas 13 a 16 imprimen los nombres y saldos de la cuenta, que se obtienen mediante una llamada a los métodos `obtenerNombre` y `obtenerSaldo` de cada `Cuenta`.

```
1 // Fig. 3.9: PruebaCuenta.java
2 // Entrada y salida de números de punto flotante con objetos Cuenta.
3 import java.util.Scanner;
```

Fig. 3.9 | Entrada y salida de números de punto flotante con objetos `Cuenta` (parte 1 de 3).

```
4
5  public class PruebaCuenta
6  {
7     public static void main(String[] args)
8     {
9        Cuenta cuenta1 = new Cuenta("Jane Green", 50.00);
10       Cuenta cuenta2 = new Cuenta("John Blue", -7.53);
11
12       // muestra el saldo inicial de cada objeto
13       System.out.printf("Saldo de %s: $%.2f%n",
14          cuenta1.obtenerNombre(), cuenta1.obtenerSaldo());
15       System.out.printf("Saldo de %s: $%.2f%n%n",
16          cuenta2.obtenerNombre(), cuenta2.obtenerSaldo());
17
18       // crea un objeto Scanner para obtener la entrada de la ventana de comandos
19       Scanner entrada = new Scanner(System.in);
20
21       System.out.print("Escriba el monto a depositar para cuenta1: "); // indicador (promt)
22       double montoDeposito = entrada.nextDouble(); // obtiene entrada del usuario
23       System.out.printf("%nsumando %.2f al saldo de cuenta1%n%n",
24          montoDeposito);
25       cuenta1.depositar(montoDeposito); // suma al saldo de cuenta1
26
27       // muestra los saldos
28       System.out.printf("Saldo de %s: $%.2f%n",
29          cuenta1.obtenerNombre(), cuenta1.obtenerSaldo());
30       System.out.printf("Saldo de %s: $%.2f%n%n",
31          cuenta2.obtenerNombre(), cuenta2.obtenerSaldo());
32
33       System.out.print("Escriba el monto a depositar para cuenta2: "); // indicador (promt)
34       montoDeposito = entrada.nextDouble(); // obtiene entrada del usuario
35       System.out.printf("%nsumando %.2f al saldo de cuenta2%n%n",
36          montoDeposito);
37       cuenta2.depositar(montoDeposito); // suma al saldo de cuenta2
38
39       // muestra los saldos
40       System.out.printf("Saldo de %s: $%.2f%n",
41          cuenta1.obtenerNombre(), cuenta1.obtenerSaldo());
42       System.out.printf("Saldo de %s: $%.2f%n%n",
43          cuenta2.obtenerNombre(), cuenta2.obtenerSaldo());
44    } // fin de main
45 } // fin de la clase PruebaCuenta
```

```
Saldo de Jane Green: $50.00
Saldo de John Blue: $0.00

Escriba el monto a depositar para cuenta1: 25.53

sumando 25.53 al saldo de cuenta1

Saldo de Jane Green: $75.53
Saldo de John Blue: $0.00
```

Fig. 3.9 | Entrada y salida de números de punto flotante con objetos Cuenta (parte 2 de 3).

```
Escriba el monto a depositar para cuenta2: 123.45

sumando 123.45 al saldo de cuenta2

Saldo de Jane Green: $75.53
Saldo de John Blue: $123.45
```

Fig. 3.9 | Entrada y salida de números de punto flotante con objetos `Cuenta` (parte 3 de 3).

Cómo mostrar los saldos iniciales de los objetos `Cuenta`

Cuando se hace una llamada al método `obtenerSaldo` para `cuenta1` en la línea 14, se devuelve el valor del `saldo` de `cuenta1` de la línea 31 de la figura 3.8, y se imprime en pantalla mediante la instrucción `System.out.printf` (figura 3.9, líneas 13 y 14). De manera similar, cuando se hace la llamada al método `obtenerSaldo` para `cuenta2` en la línea 16, se devuelve el valor del `saldo` de `cuenta2` de la línea 31 en la figura 3.8, y se imprime en pantalla mediante la instrucción `System.out.printf` (figura 3.9, líneas 15 y 16). Al principio el saldo de `cuenta2` es `0.00`, ya que el constructor rechazó el intento de empezar `cuenta2` con un saldo negativo, por lo que el saldo conserva su valor inicial predeterminado.

Formato de los números de punto flotante para visualizarlos en pantalla

Cada uno de los saldos 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 y 27.33379 se redondearía a 27.33.

Cómo leer un valor de punto flotante del usuario y realizar un depósito

La línea 21 (figura 3.9) pide al usuario que introduzca un monto de depósito para `cuenta1`. La línea 22 declara la variable *local* `montoDeposito` para almacenar cada monto de depósito introducido por el usuario. A diferencia de las variables de *instancia* (como `nombre` y `saldo` en la clase `Cuenta`), las variables *locales* (como `montoDeposito` en `main`) *no* se inicializan de manera predeterminada, por lo que normalmente deben inicializarse de manera explícita. Como veremos en un momento, el valor inicial de la variable `montoDeposito` se determinará usando el valor de entrada del usuario.



Error común de programación 3.1

Si usted intenta usar el valor de una variable local sin inicializar, el compilador de Java generará un error de compilación. Esto le ayudará a evitar los peligrosos errores lógicos en tiempo de ejecución. Siempre es mejor detectar los errores de sus programas en tiempo de compilación, en vez de hacerlo en tiempo de ejecución.

La línea 22 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 23 y 24 muestran el `montoDeposito`. La línea 25 llama al método `depositar` del objeto `cuenta1` y le suministra `montoDeposito` como *argumento*. Cuando se hace la llamada al método, el valor del argumento se asigna al parámetro `montoDeposito` del método `depositar` (línea 22 de la figura 3.8); después el método `depositar` suma ese valor al `saldo`. Las líneas 28 a 31 (figura 3.9) imprimen en pantalla los nombres y saldos de ambos objetos `Cuenta` *otra vez*, para mostrar que *sólo* se modificó el `saldo` de `cuenta1`.

La línea 33 pide al usuario que escriba un monto a depositar para `cuenta2`. La línea 34 obtiene la entrada del usuario, para lo cual invoca al método `nextDouble` del objeto `Scanner` llamado `entrada`. Las líneas 35 y 36 muestran el `montoDeposito`. La línea 37 llama al método `depositar` del objeto `cuenta2` y le suministra `montoDeposito` como *argumento*; después, el método `depositar` suma ese valor al `saldo`. Por último, las líneas 40 a 43 imprimen en pantalla los nombres y saldos de ambos objetos `Cuenta` *otra vez*, para mostrar que *sólo* se modificó el saldo de `cuenta2`.

Código duplicado en el método `main`

Las seis instrucciones en las líneas 13-14, 15-16, 28-29, 30-31, 40-41 y 42-43 son casi *idénticas*. Cada una imprime en pantalla el `nombre` y `saldo` de un objeto `Cuenta`. Sólo difieren en el nombre del objeto `Cuenta` (`cuenta1` o `cuenta2`). Este tipo de código duplicado puede crear *problemas de mantenimiento del código* cuando hay que actualizarlo; si las *seis* copias del mismo código tienen el mismo error que hay que corregir o todas deben actualizarse, hay que realizar ese cambio *seis* veces, *sin cometer errores*. El ejercicio 3.15 le pide que modifique la figura 3.9 para incluir un método `mostrarCuenta` que reciba como parámetro un objeto `Cuenta` e imprima en pantalla el `nombre` y `saldo` del objeto. Después sustituirá las instrucciones duplicadas en `main` con seis llamadas a `mostrarCuenta`, con lo cual reducirá el tamaño de su programa y mejorará su facilidad de mantenimiento al tener *una* copia del código que muestra el `nombre` y `saldo` de una `Cuenta`.



Observación de ingeniería de software 3.4

Sustituir código duplicado con llamadas a un método que contiene una sola copia de ese código puede reducir el tamaño de su programa y facilitar su mantenimiento.

Diagrama de clases de UML para la clase `Cuenta`

El diagrama de clases de UML en la figura 3.10 modela de manera concisa la clase `Cuenta` de la figura 3.8. El diagrama modela en su *segundo* compartimiento los atributos `private` `nombre` de tipo `String` y `saldo` de tipo `double`.

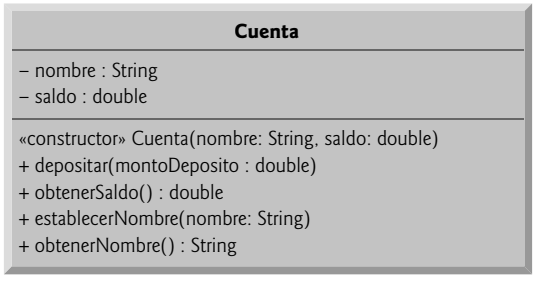


Fig. 3.10 | Diagrama de clases de UML para la clase `Cuenta` de la figura 3.8.

El diagrama modela el *constructor* de la clase `Cuenta` en el *tercer* compartimiento con los parámetros `nombre` de tipo `String` y `saldo` de tipo `double`. Los cuatro métodos `public` de la clase se modelan también en el *tercer* compartimiento: la operación `depositar` con un parámetro `montoDeposito` de tipo `double`, la operación `obtenerSaldo` con un tipo de valor de retorno `double`, la operación `establecerNombre` con un parámetro `nombre` de tipo `String` y la operación `obtenerNombre` con un tipo de valor de retorno `String`.

3.6 (Opcional) Ejemplo práctico de GUI y gráficos: uso de cuadros de diálogo

Este ejemplo práctico opcional está diseñado para quienes desean empezar a conocer las poderosas herramientas de Java para crear interfaces gráficas de usuario (GUI) y gráficos compatibles, antes de las explicaciones más detalladas de estos temas que se presentan más adelante en el libro. Este ejemplo práctico incluye la tecnología Swing consolidada de Java, que al momento de escribir este libro sigue siendo un poco más popular que la tecnología JavaFX más reciente que presentaremos en capítulos posteriores.

Este ejemplo práctico de GUI y gráficos aparece en 10 secciones breves (vea la figura 3.11). Cada sección presenta nuevos conceptos y proporciona ejemplos con capturas de pantalla que muestran interacciones de ejemplo y resultados. En las primeras secciones, usted creará sus primeras aplicaciones gráficas. En las secciones posteriores, utilizará los conceptos de programación orientada a objetos para crear una aplicación que dibuja una variedad de figuras. Cuando presentemos de manera formal a las GUI en el capítulo 12, utilizaremos el ratón para elegir con exactitud qué figuras dibujar y en dónde dibujarlas. En el capítulo 13 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 3.6	Uso de cuadros de diálogo: entrada y salida básica con cuadros de diálogo
Sección 4.15	Creación de dibujos simples: mostrar y dibujar líneas en la pantalla
Sección 5.11	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 7.17	Dibujo de arcos: dibujar espirales con arcos
Sección 8.16	Uso de objetos con gráficos: almacenar figuras como objetos
Sección 9.7	Mostrar texto e imágenes mediante el uso de etiquetas: proporcionar información de estado
Sección 10.10	Dibujo con polimorfismo: identificar las similitudes entre figuras
Ejercicio 12.17	Expansión de la interfaz: uso de componentes de la GUI y manejo de eventos
Ejercicio 13.31	Caso de estudio de GUI y gráficos: Agregar Java 2D

Fig. 3.11 | Resumen del ejemplo práctico 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 Chrome, Firefox, Internet Explorer, Safari y Opera 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 3.12 muestra el objeto `String` "`Bienvenido a Java`" en un diálogo de mensaje.

```
1  // Fig. 3.12: Dialogo1.java
2  // Uso de JOptionPane para mostrar varias líneas en un cuadro de diálogo.
3  import javax.swing.JOptionPane;
4
5  public class Dialogo1
6  {
7     public static void main(String[] args)
8     {
9        // muestra un diálogo con un mensaje
10       JOptionPane.showMessageDialog(null, "Bienvenido a Java");
11    }
12 } // fin de la clase Dialogo1
```



Fig. 3.12 | Uso de `JOptionPane` para mostrar varias líneas en un cuadro de diálogo.

El método `static` `showMessageDialog` de la clase `JOptionPane`

La línea 3 indica que el programa utiliza la clase `JOptionPane` del paquete **`javax.swing`**. Este paquete 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, además de 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 la aplicación de 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 "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.

Introducción de texto en un cuadro de diálogo

La figura 3.13 utiliza otro cuadro de diálogo `JOptionPane` predefinido, conocido como **diálogo de entrada**, el cual permite al usuario *introducir datos* en un programa. El programa pide el nombre del usuario y responde con un diálogo de mensaje que contiene un saludo y el nombre introducido por el usuario.

```
1  // Fig. 3.13: 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 =JOptionPane.showInputDialog("Cual es su nombre?");
11
12       // crea el mensaje
13       String mensaje =
14          String.format("Bienvenido, %s, a la programacion en Java!", nombre);
15
16       // muestra el mensaje para dar la bienvenida al usuario por su nombre
17       JOptionPane.showMessageDialog(null, mensaje);
18    } // fin de main
19 } // fin de la clase DialogoNombre
```



Fig. 3.13 | Cómo obtener la entrada del usuario mediante un cuadro de diálogo.

El método `static` `showInputDialog` *de la clase* `JOptionPane`

La línea 10 utiliza el método **`showInputDialog`** de `JOptionPane` para mostrar un diálogo de entrada que contiene un indicador (promt) y un *campo* (conocido como **campo de texto**), en el cual 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` devuelve un objeto `String` que contiene los caracteres escritos por el usuario. Almacenamos el objeto `String` en la variable `nombre`. Si oprime el botón **Cancelar** en el cuadro de diálogo u oprime *Esc*, el método devuelve `null` y el programa muestra la palabra "`null`" como el nombre.

El método `static` `format` *de la clase* `String`

Las líneas 13 y 14 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 17 muestra el saludo en un cuadro de diálogo de mensaje, como hicimos en la figura 3.12.

Ejercicio del ejemplo práctico de GUI y gráficos

3.1 Modifique el programa de suma que aparece en la figura 2.7 para usar la entrada y salida en un 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 `parseInt`

es un método `static` de la clase `Integer` (del paquete `java.lang`) que recibe un argumento `String` que representa a un entero y devuelve el valor como `int`. Si el objeto `String` no contiene un entero válido, el programa terminará con un error.

3.7 Conclusión

En este capítulo aprendió a crear sus propias clases y métodos, a crear objetos de esas clases y a llamar métodos de esos objetos para realizar acciones útiles. Declaró variables de instancia de una clase para mantener los datos de cada objeto de la clase, y declaró sus propios métodos para operar sobre esos datos. Aprendió cómo llamar a un método para decirle que realice su tarea y cómo pasar información a un método en forma de argumentos cuyos valores se asignan a los parámetros del mismo. 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. Vio cómo crear diagramas de clases en UML que modelen los métodos, atributos y constructores de las clases. Por último, aprendió acerca de los números de punto flotante (números con puntos decimales); aprendió 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 capítulo 8 comenzaremos a representar las cantidades monetarias en forma precisa con la clase `BigDecimal`]. Tal vez también ya haya comenzado el ejemplo práctico opcional de GUI y gráficos, con el que aprenderá a escribir sus primeras aplicaciones de GUI. 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 el orden en el que deben realizar sus tareas.

Resumen

Sección 3.2 Variables de instancia, métodos **establecer** *y métodos* **obtener**

- Cada clase que cree se convierte en un nuevo tipo que puede usarse para declarar variables y crear objetos.
- Puede declarar nuevas clases según sea necesario; ésta es una razón por la que Java se conoce como lenguaje extensible.

Sección 3.2.1 La clase **`Cuenta`** *con una variable de instancia, un método* **establecer** *y un método* **obtener**

- Cada declaración de clase que empieza con el modificador de acceso `public` (pág. 71) 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.
- Los nombres de clases, métodos y variables son identificadores. Por convención todos usan nombres con la nomenclatura de lomo de camello. Los nombres de las clases comienzan con letra mayúscula y los nombres tanto de los métodos como de las variables comienzan con letra minúscula.
- Un objeto tiene atributos, los cuales se implementan como variables de instancia (pág. 72) y los lleva consigo durante su vida útil.
- Las variables de instancia existen antes de que se invoquen los métodos en un objeto, durante la ejecución de los métodos y después de que éstos terminan su ejecución.
- Por lo general una clase contiene uno o más métodos que manipulan a las variables de instancia que pertenecen a objetos específicos de la clase.
- Las variables de instancia se declaran dentro de la declaración de una clase pero fuera de los cuerpos de las declaraciones de los métodos de la clase.
- Cada objeto (instancia) de la clase tiene su propia copia de las variables de instancia de la clase.
- La mayoría de las declaraciones de variables de instancia van precedidas por la palabra clave `private` (pág. 72), que 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.

- Los parámetros se declaran en una lista de parámetros separada por comas (pág. 73), la cual se encuentra dentro de los paréntesis que van después del nombre del método en la declaración del mismo. Si hay varios parámetros, cada uno va separado del siguiente mediante una coma. Cada parámetro debe especificar un tipo seguido de un nombre de variable.
- Las variables declaradas en el cuerpo de un método específico son variables locales, las cuales pueden usarse sólo en ese método. Cuando un método termina, se pierden los valores de sus variables locales. Los parámetros de un método también son variables locales del mismo.
- El cuerpo de todos los métodos está delimitado por llaves izquierda y derecha (`{` y `}`).
- El cuerpo de cada método contiene una o más instrucciones que realizan la o las tareas de éste.
- El tipo de valor de retorno del método especifica el tipo de datos que devuelve a quien lo llamó. La palabra clave `void` indica que un método realizará una tarea pero no regresará ninguna información.
- Los paréntesis vacíos que van después del nombre de un método indican que éste no requiere parámetros para realizar su tarea.
- Cuando se llama a un método que especifica un tipo de valor de retorno (pág. 73) distinto de `void` y completa su tarea, el método debe devolver un resultado al método que lo llamó.
- La instrucción `return` (pág. 74) pasa un valor de un método que se invocó y lo devuelve a quien hizo la llamada.
- A menudo, las clases proporcionan métodos `public` para permitir que los clientes de la clase *establezcan* u *obtengan* variables de instancia `private`. Los nombres de estos métodos no necesitan comenzar con *establecer* u *obtener*, pero se recomienda esta convención de nomenclatura.

Sección 3.2.2 La clase `PruebaCuenta` *que crea y usa un objeto de la clase* `Cuenta`

- Una clase que crea un objeto de otra clase y luego llama a los métodos de ese objeto es una clase controladora.
- El método `nextLine` de `Scanner` (pág. 75) lee caracteres hasta encontrar una nueva línea, y después devuelve los caracteres en forma de un objeto `String`.
- El método `next` de `Scanner` (pág. 75) lee caracteres hasta encontrar cualquier carácter de espacio en blanco, y después devuelve los caracteres que leyó en forma de un objeto `String`.
- Una expresión de creación de instancia de clase (pág. 75) empieza con la palabra clave `new` y crea un nuevo objeto.
- Un constructor es similar a un método, sólo que el operador `new` lo llama de manera implícita para inicializar las variables de instancia de un objeto al momento de su creación.
- Para llamar a un método de un objeto, después del nombre de ese objeto se pone un separador punto (pág. 76), el nombre del método y un conjunto de paréntesis que contienen los argumentos del método.
- Las variables locales no se inicializan de manera predeterminada. Cada variable de instancia tiene un valor inicial predeterminado: un valor que Java proporciona cuando no especificamos el valor inicial de la variable de instancia.
- El valor predeterminado para una variable de instancia de tipo `String` es `null`.
- Una llamada a un método proporciona valores (conocidos como argumentos) para cada uno de los parámetros del método. El valor de cada argumento se asigna al parámetro correspondiente en el encabezado del método.
- El número de argumentos en la llamada a un método debe coincidir con el número de parámetros en la lista de parámetros de la declaración del método.
- Los tipos de los argumentos en la llamada al método deben ser consistentes con los tipos de los parámetros correspondientes en la declaración del método.

Sección 3.2.3 Compilación y ejecución de una aplicación con varias clases

- El comando `javac` puede compilar varias clases a la vez. Sólo debe listar los nombres de archivo del código fuente después del comando, separando cada nombre de archivo del siguiente mediante un espacio. Si el directorio que contiene la aplicación incluye sólo los archivos de esa aplicación, puede compilar todas sus clases 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`".

Sección 3.2.4 Diagrama de clases en UML de `Cuenta` *con una variable de instancia y métodos* establecer *y* obtener

- En UML, cada clase se modela en un diagrama de clases (pág. 77) 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 las variables de instancia en Java. El compartimiento inferior contiene las operaciones de la clase (pág. 78), que corresponden a los métodos y los constructores en Java.
- 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 modelar las operaciones, UML lista 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 pública en UML (es decir, un método `public` en Java).
- 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.
- 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 no especifican tipos de valores de retorno para las operaciones que no devuelven valores.
- Al proceso de declarar variables de instancia como `private` se le conoce como ocultamiento de datos o de información.

Sección 3.2.5 Observaciones adicionales sobre la clase `PruebaCuenta`

- Debe llamar a la mayoría de los métodos distintos de `main` de manera explícita para decirles que realicen sus tareas.
- 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`, la cual indica que `main` es un método `static` que puede llamarse sin tener que crear primero un objeto de la clase en la cual se declara ese método.
- La mayoría de las 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. 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 demás clases del mismo paquete. No se requiere una declaración `import` cuando una clase en un paquete usa otra clase en el mismo paquete.
- No se requiere una declaración `import` si siempre hace referencia a una clase con su nombre de clase completamente calificado, que incluye tanto el nombre de su paquete como de su clase.

Sección 3.2.6 Ingeniería de software con variables de instancia `private` *y métodos* establecer *y* obtener `public`

- Al proceso de declarar variables de instancia como `private` se le conoce como ocultamiento de datos o de información.

Sección 3.3 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 en un momento dado almacenar exactamente un valor de su tipo declarado.
- 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. 81) 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.
- Las variables de instancia del tipo por referencia se inicializan de manera predeterminada con el valor `null`.

- Para invocar a los métodos de un objeto, se requiere una referencia a éste (pág. 81). Una variable de tipo primitivo no hace referencia a un objeto, por lo cual no puede usarse para invocar a un método.

Sección 3.4 La clase `Cuenta`*: inicialización de objetos mediante constructores*

- Cada clase que declare puede proporcionar de manera opcional un constructor con parámetros, el cual puede usarse para inicializar un objeto de una clase al momento de crear ese objeto.
- Java requiere la llamada a un constructor por cada objeto que se crea.
- Los constructores pueden especificar parámetros, pero no tipos de valores de retorno.
- Si una clase no define constructores, el compilador proporciona un constructor predeterminado (pág. 83) sin parámetros, y las variables de instancia de la clase se inicializan con sus valores predeterminados.
- Si declara un constructor para una clase, el compilador *no* creará un *constructor predeterminado* para esa clase.
- UML modela a los constructores en el tercer compartimiento de un diagrama de clases. Para diferenciar a un constructor de las operaciones de una clase, UML coloca la palabra "constructor" entre los signos « y » (pág. 84) antes del nombre del constructor.

Sección 3.5 La clase `Cuenta` *con un saldo: los números de punto flotante*

- Un número de punto flotante (pág. 84) es un número con un punto decimal. Java proporciona dos tipos primitivos para almacenar números de punto flotante en la memoria: `float` y `double` (pág. 84).
- 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. Éstas 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. 84) son de tipo `double` de manera predeterminada.
- El método `nextDouble` de `Scanner` (pág. 88) devuelve un valor `double`.
- El especificador de formato `%f` (pág. 88) se utiliza para mostrar valores de tipo `float` o `double`. El especificador de formato `%.2f` especifica que se deben mostrar dos dígitos de precisión a la derecha del punto decimal (pág. 88), en el número de punto flotante.
- El valor predeterminado para una variable de instancia de tipo `double` es `0.0`, y el valor predeterminado para una variable de instancia de tipo `int` es `0`.

Ejercicios de autoevaluación

3.1 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 inicializar ese objeto.

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) Java proporciona dos tipos primitivos para almacenar números de punto flotante en la memoria: _____ y _____.

g) Las variables de tipo `double` representan a los números de punto flotante _____.

h) El método _____ de la clase `Scanner` devuelve un valor `double`.

i) La palabra clave `public` es un _____ de acceso.
j) El tipo de valor de retorno _____ indica que un método no devolverá un valor.
k) El método _____ de `Scanner` lee caracteres hasta encontrar una nueva línea, y después devuelve esos caracteres como un objeto `String`.
l) La clase `String` está en el paquete _____.
m) No se requiere un(a) _____ si siempre hacemos referencia a una clase con su nombre de clase completamente calificado.
n) Un(a) _____ es un número con un punto decimal, como 7.33, 0.0975 o 1000.12345.
o) Las variables de tipo `float` representan números de punto flotante de precisión _____.
p) El especificador de formato _____ se utiliza para mostrar valores de tipo `float` o `double`.
q) Los tipos en Java se dividen en dos categorías: tipos _____ y tipos _____.

3.2 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 y todas las palabras subsiguientes en el nombre empiezan con la primera letra en mayúscula.
b) Una declaración `import` no es obligatoria cuando una clase en un paquete utiliza a otra clase en el mismo paquete.
c) Los paréntesis vacíos que van después del nombre de un método en la declaración del mismo indican que éste no requiere parámetros para realizar su tarea.
d) Una variable de tipo primitivo puede usarse para invocar un método.
e) 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.
f) El cuerpo de cada método está delimitado por llaves izquierda y derecha (`{` y `}`).
g) Las variables locales de tipo primitivo se inicializan de manera predeterminada.
h) Las variables de instancia de tipo por referencia se inicializan de manera predeterminada con el valor `null`.
i) Cualquier clase que contenga `public static void main(String[] args)` puede usarse para ejecutar una aplicación.
j) El número de argumentos en la llamada a un método debe coincidir con el número de parámetros en la lista de parámetros de la declaración del método.
k) 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.

3.3 ¿Cuál es la diferencia entre una variable local y una variable de instancia?

3.4 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) `float`, `double`. g) de precisión doble. h) `nextDouble`. i) modificador. j) `void`. k) `nextLine`. l) `java.lang`. m) declaración `import`. n) número de punto flotante. o) simple. p) `%f`. q) 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 en el nombre empiezan con una letra en mayúscula. b) Verdadero. c) Verdadero. d) Falso. Una variable de tipo primitivo no puede usarse para invocar a un método; se requiere una referencia a un objeto para invocar a los métodos de ese objeto. e) Falso. Dichas variables se llaman variables locales, y sólo se pueden utilizar en el método en el que están declaradas. f) Verdadero. g) 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. h) Verdadero. i) Verdadero. j) Verdadero. k) 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 a partir del punto en el que se declaró, hasta el final de la declaración del método. Una variable de instancia se declara en una clase, pero no en el cuerpo de alguno de los métodos de la clase. Además, las variables de instancia están accesibles para todos los métodos de la clase (en el capítulo 8 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 los parámetros correspondientes del método, 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 ***(Uso de clases sin importarlas)*** La mayoría de las clases necesitan importarse antes de poder ser usadas en una aplicación ¿Por qué cualquier aplicación puede utilizar las clases `System` y `String` sin tener que importarlas primero?

3.9 ***(Uso de 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 `Cuenta` modificada)*** Modifique la clase `Cuenta` (figura 3.8) para proporcionar un método llamado `retirar`, que retire dinero de un objeto `Cuenta`. Asegúrese de que el monto a retirar 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 retirar excede el saldo de la cuenta`". Modifique la clase `PruebaCuenta` (figura 3.9) para probar el método `retirar`.

3.12 ***(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 un método *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 de artículos por el precio de cada uno) y después devuelva ese monto como un valor `double`. Si la cantidad no es positiva, debe establecerse 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`.

3.13 ***(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.

3.14 ***(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`.

3.15 ***(Eliminar código duplicado en el método `main`)*** En la clase `PruebaCuenta` de la figura 3.9, el método `main` contiene seis instrucciones (líneas 13-14, 15-16, 28-29, 30-31, 40-41 y 42-43), cada una de las cuales muestra en pantalla el `nombre` y `saldo` de un objeto `Cuenta`. Estudie estas instrucciones y notará que difieren sólo en el objeto `Cuenta` que se está manipulando: `cuenta1` o `cuenta2`. En este ejercicio definirá un nuevo método `mostrarCuenta` que

contiene *una* copia de esa instrucción de salida. El parámetro del método será un objeto `Cuenta` y el método imprimirá en pantalla el `nombre` y `saldo` de ese objeto. Después usted sustituirá las seis instrucciones duplicadas en `main` con llamadas a `mostrarCuenta`, pasando como argumento el objeto `Cuenta` específico a mostrar en pantalla.

Modifique la clase `PruebaCuenta` de la figura 3.9 para declarar el siguiente método `mostrarCuenta` *después* de la llave derecha de cierre de `main` y *antes* de la llave derecha de cierre de la clase `PruebaCuenta`:

```
public static void mostrarCuenta(Cuenta cuentaAMostrar)
{
   // coloque aquí la instrucción que muestra en pantalla
   // el nombre y el saldo de cuentaAMostrar
}
```

Sustituya el comentario en el cuerpo del método con una instrucción que muestre el `nombre` y el `saldo` de `cuentaAMostrar`.

Recuerde que `main` es un método `static`, por lo que puede llamarse sin tener que crear primero un objeto de la clase en la que se declara `main`. También declaramos el método `mostrarCuenta` como un método `static`. Cuando `main` necesita llamar a otro método en la misma clase sin tener que crear primero un objeto de esa clase, el otro método *también* debe declararse como `static`.

Una vez que complete la declaración de `mostrarCuenta`, modifique `main` para reemplazar las instrucciones que muestran el `nombre` y `saldo` de cada `Cuenta` con llamadas a `mostrarCuenta`; cada una debe recibir como argumento el objeto `cuenta1` o `cuenta2`, según sea apropiado. Luego, pruebe la clase `PruebaCuenta` actualizada para asegurarse de que produzca la misma salida que se muestra en la figura 3.9.

Marcando 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/presenter.jhtml?identifier=4736`), la fórmula para calcular su *frecuencia cardiaca máxima* en pulsos por minuto es 220 menos su edad en años. Su *frecuencia cardiaca esperada* tiene 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 con base en 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 independientes 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), un método que calcule y devuelva la frecuencia cardiaca máxima de esa persona, y otro método 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 (incluyendo 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 tema 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, podría ayudar a salvar vidas. En este ejercicio usted diseñará una clase "inicial" llamada `PerfilMedico` para una

persona. Los atributos de la clase deben incluir 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 incluir 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 (incluyendo 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.

Instrucciones de control: parte 1: operadores de asignación, ++ y --

4



Desplacémonos un lugar.
—Lewis Carroll

¡Cuántas manzanas tuvieron que caer en la cabeza de Newton antes de que entendiera el suceso!
—Robert Frost

Objetivos

En este capítulo aprenderá a:

- Utilizar 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 dos distintas acciones alternativas.
- Utilizar la estructura de repetición `while` para ejecutar instrucciones de manera repetitiva dentro de un programa.
- Utilizar la repetición controlada por un contador y la repetición controlada por un centinela.
- Utilizar los operadores de asignación compuestos, de incremento y decremento.
- Conocer la portabilidad de los tipos de datos primitivos.

Plan general

4.1 Introducción
4.2 Algoritmos
4.3 Seudocódigo
4.4 Estructuras de control
4.5 Instrucción `if` de selección simple
4.6 Instrucción `if...else` de selección doble
4.7 Clase `Estudiante`: instrucciones `if...else` anidadas
4.8 Instrucción de repetición `while`
4.9 Formulación de algoritmos: repetición controlada por un contador
4.10 Formulación de algoritmos: repetición controlada por un centinela
4.11 Formulación de algoritmos: instrucciones de control anidadas
4.12 Operadores de asignación compuestos
4.13 Operadores de incremento y decremento
4.14 Tipos primitivos
4.15 (Opcional) Ejemplo práctico de GUI y gráficos: creación de dibujos simples
4.16 Conclusión

Resumen | Ejercicios de autoevaluación | Respuestas a los ejercicios de autoevaluación | Ejercicios | Marcando la diferencia

4.1 Introducción

Antes de escribir un programa que dé solución a un problema, usted necesita entender en detalle el problema, además de tener una metodología cuidadosamente planeada para resolverlo. Al escribir un programa, es igualmente esencial comprender los tipos de bloques de construcción disponibles, y emplear las técnicas comprobadas para construir programas. En este capítulo y en el siguiente, hablaremos sobre estas cuestiones cuando presentemos la teoría y los principios de la programación estructurada. Los conceptos presentados aquí son imprescindibles para crear clases y manipular objetos. Hablaremos sobre la instrucción `if` de Java con más detalle y le presentaremos las instrucciones `if...else` y `while`; todos estos bloques de construcción le permiten especificar la lógica requerida para que los métodos realicen sus tareas. También presentaremos el operador de asignación compuesto y los operadores de incremento y decremento. Por último, analizaremos 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 logra que el ejecutivo llegue al trabajo bien preparado para tomar decisiones críticas. 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 programa**. 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 que usamos en este libro 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 4.7 verá un algoritmo escrito en seudocódigo. Desde luego que puede usar sus propios idiomas nativos para desarrollar su propio 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, utilizando cualquier programa editor de texto. Un seudocódigo preparado cuidadosamente puede convertirse sin problema 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 de ejecutarlo en una computadora. Dichas acciones podrían incluir la *entrada, salida o cálculos*. Por lo general *no* incluimos las declaraciones de variables en nuestro seudocódigo, pero algunos programadores optan por listar las variables y mencionar sus propósitos.

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 descubrió 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. [*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`". El término **programación estructurada** se hizo casi un sinónimo de la "eliminación del `goto`". No fue sino hasta la década de 1970 cuando los programadores comenzaron a tomar 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 a tiempo de sistemas 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 de modificar, y había más probabilidad de que estuvieran libres de errores desde el principio.

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

1 C. Bohm y G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, mayo de 1966, páginas 336-371.

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 de 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 queramos en una estructura de secuencia. Como veremos pronto, en donde quiera que se coloque una sola acción, podrán colocarse varias acciones en secuencia.

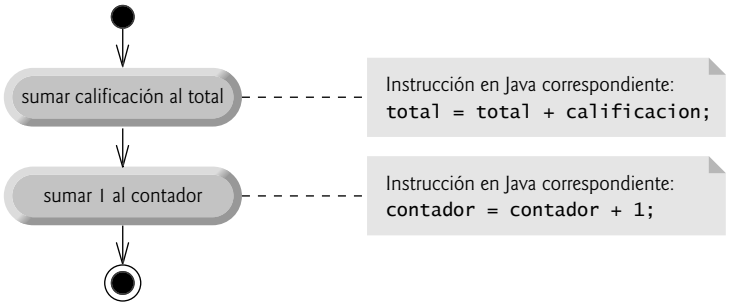


Fig. 4.1 | Diagrama de actividad de una estructura de secuencia.

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, 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. Los diagramas de actividad muestran con claridad cómo operan las estructuras de control. En este capítulo y en el 5 usaremos el UML para mostrar el flujo de control en las instrucciones de control. En los capítulos 33 y 34 en línea usaremos el UML en un ejemplo práctico verdadero de un cajero automático real.

Considere el diagrama de actividad para la estructura de secuencia de la figura 4.1. Este diagrama contiene dos **estados de acción**, cada uno de los cuales 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**; es decir, 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), y son 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 ejemplo práctico opcional de diseño orientado a objetos (capítulos 33 y 34) 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 la condición 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 *dos grupos de acciones*). La instrucción `switch` es una **estructura de selección múltiple**, ya que selecciona entre *diversas acciones distintas* (o *grupos de acciones*).

Instrucciones de repetición en Java

Java cuenta con tres **instrucciones de repetición** (también llamadas **instrucciones de iteración** o **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`, `for` y las instrucciones `for` mejoradas (en el capítulo 5 presentamos las instrucciones `do...while` y `for`, y en el capítulo 7 presentaremos la instrucción `for` mejorada). 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 un 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**. Más adelante 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 colocarla 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.

Diagrama de actividad en UML para una instrucción `if`

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.

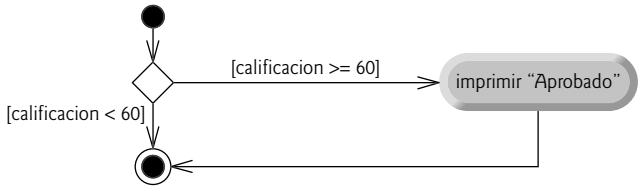


Fig. 4.2 | Diagrama de actividad en UML de la instrucción `if` de selección simple.

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 especificar una acción a realizar cuando la condición es *verdadera*, y otra distinta cuando la condición 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 *si...entonces* en seudocódigo 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.

Diagrama de actividad en UML para una instrucción `if...else`

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.

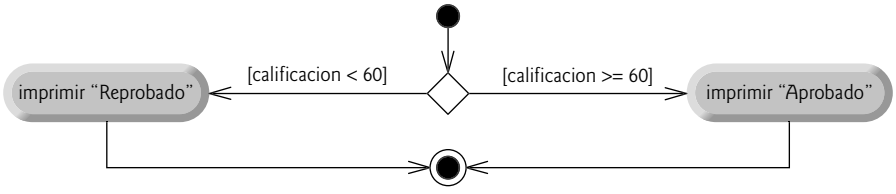


Fig. 4.3 | Diagrama de actividad de UML de la instrucción `if`...`else` de selección doble.

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");
```



Tip para prevenir errores 4.1

En una instrucción `if...else` *anidada, debe asegurarse de evaluar todos los posibles casos.*

Si la variable `calificacionEstudiante` es mayor o igual a 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 y 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 y 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 y 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 y 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`. Peor aún, 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 y y son > 5");
}
else
   System.out.println("x es <= 5");
```

Las llaves indican que el segundo `if` se encuentra en el cuerpo del primero 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. Las instrucciones contenidas dentro de un par de llaves (como el cuerpo de un

método) forman un **bloque**. Un bloque puede colocarse en cualquier parte de un método 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`. Estas llaves 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 detecta 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 donde pueda colocarse una instrucción individual, también es posible no tener instrucción alguna. En la sección 2.8 vimos que la instrucción vacía se representa colocando 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).

Operador condicional (?:)

Java cuenta con el **operador condicional** (`?:`), que en ocasiones puede utilizarse en lugar de una instrucción `if...else`. Este operador puede hacer su código más corto y claro. 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* (`true`), 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`.



Tip para prevenir errores 4.2

Para evitar algunos errores sutiles, use expresiones del mismo tipo para el segundo y tercer operandos del operador `?:`.

4.7 Clase `Estudiante`: instrucciones `if...else` anidadas

El ejemplo de las figuras 4.4 y 4.5 demuestra una instrucción `if...else` anidada que determina la calificación en letras de un estudiante con base en el promedio del estudiante en un curso.

La clase `Estudiante`

La clase `Estudiante` (figura 4.4) tiene características similares a las de la clase `Cuenta` (que vimos en el capítulo 3). La clase `Estudiante` almacena el nombre y promedio de un estudiante, además de proporcionar métodos para manipular esos valores. La clase contiene:

- la variable de instancia `nombre` de tipo `String` (línea 5) para almacenar el nombre de un `Estudiante`
- la variable de instancia `promedio` de tipo `double` (línea 6) para almacenar el promedio de un `Estudiante` en un curso
- un constructor (líneas 9 a 18) que inicializa el `nombre` y el `promedio`; en la sección 5.9 aprenderá a expresar las líneas 15-16 y 37-38 de manera más concisa con operadores lógicos que puedan evaluar múltiples condiciones
- los métodos `establecerNombre` y `obtenerNombre` (líneas 21 a 30) para *establecer* y *obtener* el `nombre` del `Estudiante`
- los métodos `establecerPromedio` y `obtenerPromedio` (líneas 33 a 46) para *establecer* y *obtener* el `promedio` del `Estudiante`
- el método `obtenerCalificacionEstudiante` (líneas 49 a 65) que usa *instrucciones* `if...else` *anidadas* para determinar la *calificación en letras* del `Estudiante`, con base en su promedio.

El constructor y el método `establecerPromedio` usan *instrucciones* `if` *anidadas* (líneas 15 a 17 y 37 a 39) para *validar* el valor que se utiliza para establecer el `promedio`. Estas instrucciones aseguran que el valor sea mayor que `0.0` *y* menor o igual a `100.0`; de lo contrario, el valor de `promedio` permanece *sin cambios*. Cada instrucción `if` contiene una condición *simple*. Si la condición en la línea 15 es *verdadera*, sólo entonces se evaluará la condición en la línea 16 y *sólo* si las condiciones en las líneas 15 *y* 16 son *verdaderas* se ejecutará la instrucción en la línea 17.



Observación de ingeniería de software 4.1

En el capítulo 3 vimos que no debe hacer llamadas a métodos desde los constructores (en el capítulo 10, Programación orientada a objetos: polimorfismo e interfaces le explicaremos por qué). Por esta razón, hay código de validación duplicado en las líneas 15 a 17 y 37 a 39 de la figura 4.4 y en los ejemplos subsiguientes.

```
1  // Fig. 4.4: Estudiante.java
2  // Clase Estudiante que almacena el nombre y promedio de un estudiante.
3  public class Estudiante
4  {
5      private String nombre;
6      private double promedio;
7
8      // el constructor inicializa las variables de instancia
9      public Estudiante(String nombre, double promedio)
10     {
11       this.nombre = nombre;
12
13       // valida que promedio sea > 0.0 y <= 100.0; de lo contrario,
14       // mantiene el valor predeterminado de la variable de instancia promedio (0.0)
15       if (promedio > 0.0)
16          if (promedio <= 100.0)
17             this.promedio = promedio; // asigna a la variable de instancia
18     {
19
20     // establece el nombre del Estudiante
21     public void establecerNombre(String nombre)
22     {
23       this.nombre = nombre;
24     }
25
26     // recupera el nombre del Estudiante
27     public String obtenerNombre()
28     {
29        return nombre;
30     }
31
32     // establece el promedio del Estudiante
33     public void establecerPromedio(double promedio)
34     {
35        // valida que promedio sea > 0.0 y <= 100.0; de lo contrario,
36       // mantiene el valor actual de la variable de instancia promedio
37       if (promedio > 0.0)
38          if (promedio <= 100.0)
39             this.promedio = promedio; // asigna a la variable de instancia
40    }
41
42    // recupera el promedio del Estudiante
43    public double obtenerPromedio()
44    {
45       return promedio;
46    }
47
48    // determina y devuelve la calificación en letras del Estudiante
49    public String obtenerCalificacionEstudiante()
50    {
51      String calificacionEstudiante = ""; // se inicializa con objeto String vacío
52
```

Fig. 4.4 | Clase `Estudiante` que almacena el nombre y promedio de un estudiante (parte 1 de 2).

```
53          if (promedio >= 90.0)
54             calificacionEstudiante = "A";
55          else if (promedio >= 80.0)
56             calificacionEstudiante = "B";
57          else if (promedio >= 70.0)
58             calificacionEstudiante = "C";
59          else if (promedio >= 60.0)
60             calificacionEstudiante = "D";
61          else
62             calificacionEstudiante = "F";
63
64          return calificacionEstudiante;
65       }
66    } // fin de la clase Estudiante
```

Fig. 4.4 | Clase `Estudiante` que almacena el nombre y promedio de un estudiante (parte 2 de 2).

La clase ***PruebaEstudiante***

Para demostrar las instrucciones `if...else` anidadas en el método `obtenerCalificacionEstudiante` de `Estudiante`, el método `main` de la clase `PruebaEstudiante` (figura 4.5) crea dos objetos `Estudiante` (líneas 7 y 8). A continuación, las líneas 10 a 13 muestran el nombre y la calificación en letra de cada `Estudiante` mediante llamadas a los métodos `obtenerNombre` y `obtenerCalificacionEstudiante`, respectivamente.

```
1  // Fig. 4.5: PruebaEstudiante.java
2  // Crea y prueba objetos Estudiante.
3  public class PruebaEstudiante
4  {
5     public static void main(String[] args)
6     {
7        Estudiante cuenta1 = new Estudiante("Jane Green", 93.5);
8        Estudiante cuenta2 = new Estudiante("John Blue", 72.75);
9
10       System.out.printf("La calificacion en letra de %s es: %s%n",
11          cuenta1.obtenerNombre(), cuenta1.obtenerCalificacionEstudiante());
12       System.out.printf("La calificacion en letra de %s es: %s%n",
13          cuenta2.obtenerNombre(), cuenta2.obtenerCalificacionEstudiante());
14    }
15 } // fin de la clase PruebaEstudiante
```

```
La calificacion en letra de Jane Green es: A
La calificacion en letra de John Blue es: C
```

Fig. 4.5 | Crear y probar objetos `Estudiante`.

4.8 Instrucción de repetición `while`

Una instrucción de repetición 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 durante una salida 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 de la lista). 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 a 100. Suponga que la variable `producto` de tipo `int` se inicializa en `3`. Cuando la siguiente instrucción `while` termine de ejecutarse, `producto` contendrá el resultado:

```
while (producto <= 100)
    producto = 3 * producto;
```

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 `producto <= 100)` 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 en el cuerpo de una instrucción `while` no se proporciona 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).*

Diagrama de actividad de UML para una instrucción `while`

El diagrama de actividad de UML de la figura 4.6 muestra el flujo de control que corresponde a la instrucción `while` anterior. Una vez más los símbolos en el diagrama (con excepción del estado inicial, las flechas de transición, un estado final y tres notas) representan un estado de acción y una decisión. Este diagrama introduce el **símbolo de fusión** de UML, que 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).

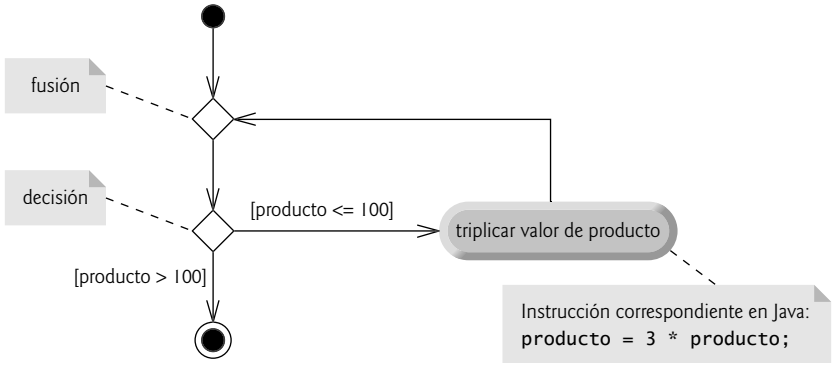


Fig. 4.6 | Diagrama de actividad de UML de la instrucción de repetición `while`.

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 flechas de transición 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 flecha de transición que apunta hacia fuera del rombo, 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.

La figura 4.6 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. Este ciclo 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.

4.9 Formulación de algoritmos: repetición controlada por un contador

Para ilustrar la forma en que se desarrollan los algoritmos, resolveremos dos variantes de un problema que promedia las calificaciones de algunos 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 a 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 el promedio e imprimir el resultado.

Algoritmo de seudocódigo con repetición controlada por un contador

Emplearemos seudocódigo para listar las acciones a ejecutar 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 4.7) completamente desarrollado y su correspondiente programa en Java (figura 4.8) que implementa el algoritmo. En la sección 4.10 demostraremos cómo utilizar el seudocódigo para desarrollar dicho algoritmo desde cero.

Observe las referencias en el algoritmo de la figura 4.7 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.



Observación de ingeniería de software 4.2

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. 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.

```
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. 4.7 | 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 figura 4.8, el método `main` de la clase `PromedioClase` (líneas 7 a 31) implementa el algoritmo para obtener el promedio de la clase, descrito por el seudocódigo de la figura 4.7. Este permite que el usuario introduzca 10 calificaciones para luego calcular y mostrar el promedio en pantalla.

```
1  // Fig. 4.8: PromedioClase.java
2  // Cómo solucionar el problema del promedio de la clase mediante la repetición
   controlada por contador.
3  import java.util.Scanner; // el programa usa la clase Scanner
4
5   public class PromedioClase
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        // fase de inicialización
13        int total = 0; // inicializa la suma de calificaciones introducidas por el
                         usuario
14        int contadorCalificaciones = 1; // inicializa # de calificación a
                                           introducir a continuación
15
16        // la fase de procesamiento usa la repetición controlada por contador
17        while (contadorCalificaciones <= 10) // itera 10 veces
18        {
19           System.out.print("Escriba la calificacion: "); // indicador
20           int calificacion = entrada.nextInt(); // recibe siguiente calificación
                                                      de entrada
21           total = total + calificacion; // suma calificación al total
22           contadorCalificaciones = contadorCalificaciones + 1; // incrementa el
                                                                     contador en 1"
23        }
24
```

Fig. 4.8 | Cómo solucionar el problema del promedio de la clase mediante la repetición controlada por contador (parte 1 de 2).

```
25        // fase de terminación
26        int promedio = total / 10; // la división de enteros produce resultado entero
27
28        // muestra el total y el promedio de las calificaciones
29        System.out.printf("%nEl total de las 10 calificaciones es %d%n", total);
30        System.out.printf("El promedio de la clase es %d%n", promedio);
31     } // fin de  main
32  } // fin de la clase PromedioClase
```

```
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. 4.8 | Cómo solucionar el problema del promedio de la clase mediante la repetición controlada por contador (parte 2 de 2).

Variables locales en el método `main`

La línea 10 declara e inicializa la variable `entrada` de tipo `Scanner`, que se utiliza para leer los valores introducidos por el usuario. Las líneas 13, 14, 20 y 26 declaran las variables locales respectivas `total`, `contadorCalificaciones`, `calificacion` y `promedio` de tipo `int`. La variable `calificacion` almacena la entrada del usuario.

Estas declaraciones aparecen en el cuerpo del método `main`. 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 la variable se utilice en ese método. Una variable local no puede utilizarse fuera del método en el que se declara. La variable `calificacion`, que se declara en el cuerpo del ciclo `while`, puede usarse sólo en ese bloque.

Fase de inicialización: inicializar las variables `total` *y* `contadorCalificacion`

Las asignaciones (en las líneas 13 y 14) inicializan `total` a `0` y `contadorCalificaciones` a `1`. Estas inicializaciones ocurren *antes* de que se utilicen las variables en los cálculos.



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 4.3

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).

Fase de procesamiento: leer 10 calificaciones del usuario

La línea 17 indica que la instrucción `while` debe continuar ejecutando el ciclo (lo que también se conoce como **iterar**), siempre y cuando el valor de `contadorCalificaciones` sea menor o igual a 10. Mientras esta condición sea *verdadera*, la instrucción `while` ejecutará en forma repetida las instrucciones que están entre las llaves que delimitan su cuerpo (líneas 18 a la 23).

La línea 19 muestra el indicador (la palabra indicador también es conocido como promt) "`Escriba la calificacion:`". La línea 20 lee la calificación introducida por el usuario y la asigna a la variable `calificacion`. Después, la línea 21 suma la nueva `calificacion` escrita por el usuario al `total`, y asigna el resultado a `total`, que sustituye su valor anterior.

La línea 22 suma `1` a `contadorCalificaciones` para indicar que el programa ha procesado una calificación y está listo para recibir la siguiente calificación del usuario. Al incrementar a `contadorCalificaciones` en cada iteración, en un momento dado su valor excederá a 10. En ese momento, el ciclo termina debido a que su condición (línea 17) se vuelve *falsa*.

Fase de terminación: calcular y mostrar el promedio de la clase

Cuando el ciclo termina, la línea 26 realiza el cálculo del promedio y asigna su resultado a la variable `promedio`. La línea 29 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 30 utiliza a `printf` para mostrar el texto "`El promedio de la clase es `", seguido del valor de la variable `promedio`. Cuando la ejecución llega a la línea 31, el programa termina.

Cabe mencionar que este ejemplo contiene sólo una clase, en donde el método `main` realiza todo el trabajo. En este capítulo y el anterior hemos visto ejemplos que consisten de dos clases: una que contiene variables de instancia y métodos que realizan tareas usando esas variables, y otra que contiene el método `main`, el cual 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.

Observaciones acerca de la división de enteros y el truncamiento

El cálculo del promedio realizado por el método `main` 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 26 de la figura 4.8) es el entero 84, ya que `total` y `10` son enteros. Al dividir dos enteros se produce una **división entera**: se **trunca** cualquier parte fraccionaria del cálculo (es decir, se *pierde*). 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.

Una observación sobre el desbordamiento aritmético

En la figura 4.8, la línea 21

```
total = total + calificacion; // suma calificación al total
```

suma cada calificación introducida por el usuario al total. Incluso esta instrucción simple tiene un problema *potencial*: sumar los enteros podría producir un valor que sea *demasiado grande* para almacenarse en una variable `int`. Esto se conoce como **desbordamiento aritmético** y provoca un *comportamiento indefinido*, el cual puede producir resultados inesperados (`http://en.wikipedia.org/wiki/Integer_overflow#Security_ramifications`). El programa `Suma` de la figura 2.7 tenía el mismo problema en la línea 23, que calculaba la suma de dos valores `int` escritos por el usuario:

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

Los valores máximo y mínimo que se pueden almacenar en una variable `int` se representan mediante las constantes `MIN_VALUE` y `MAX_VALUE`, respectivamente, las cuales se definen en la clase `Integer`. Hay

constantes similares para los otros tipos integrales y para los tipos de punto flotante. Cada tipo primitivo tiene su correspondiente tipo de clase en el paquete `java.lang`. Puede ver los valores de estas constantes en la documentación en línea de cada clase. La documentación en línea para la clase Integer se encuentra en:

```
http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html
```

Antes de realizar cálculos aritméticos como los de la línea 21 en la figura 4.8 y de la línea 23 en la figura 2.7, se considera una buena práctica asegurarse que éstos *no* se vayan a desbordar. El código para hacer esto se muestra en el sitio web de CERT `www.securecoding.cert.org`; sólo busque el lineamiento "NUM00-J". El código usa los operadores `&&` (AND lógico) y `||` (OR lógico), que se presentarán en el capítulo 5. En el código para uso industrial, es necesario realizar comprobaciones como éstas para *todos* los cálculos.

Un análisis más detallado del proceso de recibir la entrada del usuario

Cada vez que un programa recibe entrada del usuario, pueden ocurrir varios problemas. Por ejemplo, en la línea 20 de la figura 4.8

```
int calificacion = entrada.nextInt(); // recibe siguiente calificación de entrada
```

asumimos que el usuario introducirá una calificación entera en el rango de 0 a 100. Sin embargo, la persona que introduzca una calificación podría introducir un entero menor a 0, un entero mayor a 100, un entero fuera del rango de valores que pueden almacenarse en una variable `int`, un número que contenga un punto decimal, o un valor que contenga letras o símbolos especiales que ni siquiera sea entero.

Para asegurar que las entradas sean válidas, los programas de uso industrial deben probar todos los posibles casos erróneos. Un programa que reciba calificaciones debe **validar** estas calificaciones mediante la **comprobación de rangos** para asegurarse de que sean valores entre 0 y 100. Luego puede pedir al usuario que vuelva a introducir cualquier valor que esté fuera del rango. Si un programa requiere entradas de un conjunto específico de valores (por ejemplo, códigos de productos no secuenciales), puede asegurarse de que cada entrada coincida con un valor en el conjunto.

4.10 Formulación de algoritmos: repetición controlada por un centinela

Generalicemos el problema, de la sección 4.9, 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. El programa 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 error** 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 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: la cima y el primer refinamiento

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, la cima 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 listaremos 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 4.3

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 4.4

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 *calificación* (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* 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 a la vez. Después de introducir la última calificación, el usuario introduce el valor centinela. El programa evalúa el valor centinela después de la introducción de cada calificación, y termina el ciclo cuando se 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 que están debajo de ésta para mostrar que pertenecen a esa estructura *Mientras*. 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 a 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*: 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 4.9.



Tip para prevenir errores 4.4

Al realizar cálculos de división (/) o residuo (%) en donde el operando derecho 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
```

Fig. 4.9 | Algoritmo en seudocódigo del problema para promediar una clase, con una repetición controlada por centinela (parte 1 de 2).

```
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. 4.9 | Algoritmo en seudocódigo del problema para promediar una clase, con una repetición controlada por centinela (parte 2 de 2).

En las figuras 4.7 y 4.9 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 4.9 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 niveles.



Observación de ingeniería de software 4.5

Puede terminar 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 4.6

Algunos programadores no utilizan herramientas de desarrollo de programas como el seudocódigo. Ellos 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 esto podría 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 figura 4.10, el método `main` (líneas 7 a 46) implementa el algoritmo de la figura 4.9 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. También veremos que las estructuras de control pueden *apilarse* una encima de otra (en secuencia). La instrucción `while` (líneas 22 a 30) va seguida por una instrucción `if...else` (líneas 34 a 45) en secuencia. La mayor parte del código en este programa es idéntico al código de la figura 4.8, por lo que nos concentraremos en los nuevos conceptos.

```
1  // Fig. 4.10: PromedioClase.java
2  // Cómo resolver el problema del promedio de una clase mediante la repetición
   controlada por centinela.
3  import java.util.Scanner; // el programa usa la clase Scanner
```

Fig. 4.10 | Cómo resolver el problema del promedio de una clase mediante la repetición controlada por centinela (parte 1 de 2).

```
4
5  public class PromedioClase
6  {
7     public static void main(String[] args)
8     {
9        // crea objeto Scanner para obtener entrada de la ventana de comandos
10       Scanner entrada = new Scanner(System.in);
11
12       // fase de inicialización
13       int total = 0; // inicializa la suma de calificaciones
14       int contadorCalificaciones = 0; // inicializa # de calificaciones introducidas
                                            hasta ahora
15
16        // fase de procesamiento
17       // pide entrada y lee calificación del usuario
18       System.out.print("Escriba la calificacion o -1 para terminar: ");
19       int calificacion = entrada.nextInt();
20
21       // itera hasta recibir el valor centinela del usuario
22       while (calificacion != -1)
23       {
24          total = total + calificacion; // suma calificacion al total
25            contadorCalificaciones = contadorCalificaciones + 1; // incrementa el
                                                                      contador
26
27            // pide entrada y lee la siguiente calificación del usuario
28            System.out.print("Escriba la calificacion o -1 para terminar: ");
29            calificacion = entrada.nextInt();
30       {
31
32         // fase de terminación
33         // si el usuario introdujo al menos una calificación...
34         if (contadorCalificaciones != 0)
35         {
36            // usa número con punto decimal para calcular promedio de calificaciones
37            double promedio = (double) total / contadorCalificaciones;
38
39             // muestra total y promedio (con dos dígitos de precisión)
40            System.out.printf("%nEl total de las %d calificaciones introducidas es %d%n",
41               contadorCalificaciones, total);
42             System.out.printf("El promedio de la clase es %.2f%n", promedio);
43         }
44         else // no se introdujeron calificaciones, por lo que se muestra el mensaje
                   apropiado
45            System.out.println("No se introdujeron calificaciones");
46      }
47  } // fin de la clase PromedioClase
```

```
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 | Cómo resolver el problema del promedio de una clase mediante la repetición controlada por centinela (parte 2 de 2.)

Comparación entre la lógica del programa para la repetición controlada por centinela, y la repetición controlada por contador

La línea 37 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 14 inicializa `contadorCalificaciones` 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 `contadorCalificaciones` sólo cuando el usuario introduce una calificación válida.

Compare la lógica de esta aplicación para la repetición controlada por centinela con la repetición controlada por contador en la figura 4.8. En la repetición controlada por contador, cada iteración de la instrucción `while` (líneas 17 a 23 de la figura 4.8) lee un valor del usuario, para el número especificado de iteraciones. En la repetición controlada por centinela, el programa lee el primer valor (líneas 18 y 19 de la figura 4.10) 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` además de incrementar el `contadorCalificaciones` (líneas 24 y 25). Después, las líneas 28 y 29 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 30, por lo que la ejecución continúa con la evaluación de la condición del `while` (línea 22). 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 al usuario sobre el centinela.

Una vez que termina el ciclo se ejecuta la instrucción `if...else` en las líneas 34 a 45. La condición en la línea 34 determina si se introdujeron calificaciones o no. Si no se introdujo ninguna, se ejecuta la parte del `else` (líneas 44 y 45) de la instrucción `if...else` y muestra el mensaje "`No se introdujeron calificaciones`", y el método devuelve el control al método que lo llamó.

Llaves en una instrucción `while`

Observe el *bloque* de la instrucción `while` en la figura 4.10 (líneas 23 a 30). 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, ocasionando que la computadora interpretara el código de manera incorrecta, como se muestra a continuación:

```
while (calificacion != -1)
   total = total + calificacion; // suma calificacion al total
contadorCalificaciones = contadorCalificaciones + 1; // incrementa el contador

// pide entrada y lee la siguiente calificación del usuario
System.out.print("Escriba la 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 19 (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 37 de la figura 4.10 calcula el promedio de las calificaciones. En la figura 4.8 vimos que la división entre enteros produce un resultado entero. Aun y cuando la variable `promedio` se declara como `double`, si hubiéramos escrito el cálculo del promedio como

```
double promedio = total / contadorCalificaciones;
```

descartaría la parte fraccionaria del cociente *antes* de asignar el resultado de la división a `promedio`. Esto ocurre debido a que `total` y `contadorCalificaciones` son *ambos* enteros, y la división entera produce un resultado entero.

La mayoría de los promedios no son números enteros (por ejemplo, 0, –22 y 1024). Por esta razón, calculamos el promedio de la clase en este ejemplo como número de punto flotante. 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 37 de la figura 4.10 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 en un valor de punto flotante (la copia `double` temporal de `total`) dividido entre el entero `contadorCalificaciones`. Java puede evaluar sólo expresiones aritméticas en las que los tipos de los operandos sean *idénticos*. Para asegurar esto, 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, el valor de `contadorCalificaciones` se promueve 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 e interfaces). La conversión al tipo incorrecto puede ocasionar errores de compilación o errores en tiempo de ejecución.

Un operador de conversión de tipo 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.

La línea 42 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` 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 (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`.

Precisión de números de punto flotante

Los números de punto flotante no siempre son 100% precisos, pero tienen numerosas aplicaciones. Por ejemplo, cuando hablamos de una temperatura corporal "normal" de 36.8 grados, no necesitamos una precisión con gran cantidad de dígitos. Cuando leemos la temperatura en un termómetro como 36.8, en realidad podría ser 36.7999999473210643. Considerar este número simplemente como 36.8 está bien para la mayoría de las aplicaciones relacionadas con temperaturas corporales.

A menudo los números de punto flotante surgen como resultado de la división, como en el cálculo del promedio de la clase de este ejemplo. En la aritmética convencional, cuando dividimos 10 entre 3, el resultado es 3.3333333…, en donde la secuencia de dígitos 3 se repite de manera indefinida. La computadora asigna sólo una cantidad fija de espacio para guardar dicho valor, por lo que sin duda el valor de punto flotante almacenado puede ser sólo una aproximación.

Debido a la naturaleza imperfecta de los números de punto flotante, se prefiere el tipo `double` en vez del tipo `float` ya que las variables double pueden representar los números de punto flotante en forma más precisa. Por esta razón, usamos principalmente el tipo `double` a lo largo del libro. En algunas aplicaciones, la precisión de las variables `float` y `double` será inadecuada. Para números de punto flotante precisos (como los requeridos por los cálculos monetarios), Java cuenta con la clase `BigDecimal` (paquete `java.math`) que veremos en el capítulo 8.



Error común de programación 4.7

Usar números de punto flotante de una manera que asuma que están representados en forma precisa puede conducir a resultados incorrectos.

4.11 Formulación de 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 ejemplo práctico examinaremos la otra forma estructurada en la que pueden conectarse las instrucciones de control; es decir, 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 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 cuidadosamente el enunciado del programa, 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 refinarse de la siguiente manera:

```
Inicializar aprobados en cero
Inicializar reprobados en cero
Inicializar contador de estudiantes en uno
```

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 refinarse 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.

```
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
```

Fig. 4.11 | El seudocódigo para el problema de los resultados del examen (parte 1 de 2).

```
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 (parte 2 de 2).

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, 14, 15 y 22 de la clase `main` declaran las variables que se utilizan para procesar los resultados del examen.



Tip para prevenir errores 4.5

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.

```
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
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;
14        int reprobados = 0;
15        int contadorEstudiantes = 1;
16
17        // procesa 10 estudiantes, usando ciclo controlado por contador
18        while (contadorEstudiantes <= 10)
19        {
20           // pide al usuario la entrada y obtiene el valor
21           System.out.print("Escriba el resultado (1 = aprobado, 2 = reprobado): ");
22           int resultado = entrada.nextInt();
23
24           // if...else anidado en la instrucción while
25           if (resultado == 1)
26              aprobados = aprobados + 1;
27           else
28              reprobados = reprobados + 1;
29
```

Fig. 4.12 | Análisis de los resultados de un examen, utilizando instrucciones de control anidadas (parte 1 de 2).

```
30            // incrementa contadorEstudiantes, para que el ciclo termine en un
                 momento dado
31            contadorEstudiantes = contadorEstudiantes + 1;
32         }
33
34         // fase de terminación; prepara y muestra los resultados
35         System.out.printf("Aprobados: %d%nReprobados: %d%n", aprobados, reprobados);
36
37         // determina si más de 8 estudiantes aprobaron
38         if (aprobados > 8)
39            System.out.println( "Bono para el instructor!" );
40      }
41   } // 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!
```

```
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 de un examen, utilizando instrucciones de control anidadas (parte 2 de 2).

La instrucción `while` (líneas 18 a 32) itera 10 veces. Durante cada iteración, el ciclo recibe y procesa un resultado del examen. Observe que la instrucción `if...else` (líneas 25 a 28) se *anida* en la instrucción `while` para procesar cada resultado. Si el `resultado` es `1`, la instrucción `if...else` incrementa a `aprobados`; en caso contrario, asume que el `resultado` es `2` e incrementa `reprobados`. La línea 31 incrementa `contadorEstudiantes` antes de que se evalúe otra vez la condición del ciclo, en la línea 18. Después de introducir 10 valores, el ciclo termina y la línea 35 muestra el número de `aprobados` y de `reprobados`. La instrucción `if` de las líneas 38 y 39 determina si más de ocho estudiantes aprobaron el examen y, de ser así, imprime el mensaje "`Bono para el instructor!`".

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 38 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.

4.12 Operadores de asignación compuestos

Los **operadores de asignación compuestos** abrevian las expresiones de asignación. Las instrucciones 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 mismo 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.

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.

4.13 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 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 como prefijo (antes) de una variable se le llama **operador de preincremento** o **predecremento**, respectivamente. A un operador de incremento o decremento que se coloca como postfijo (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 predecremento) 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 postincremento (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 la variable 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.

Diferencia entre los operadores de preincremento y postincremento

La figura 4.15 demuestra la diferencia entre la versión de preincremento y la versión de postincremento del operador de incremento ++. El operador de decremento (--) funciona de manera similar.

La línea 9 inicializa la variable `c` con `5`, y la línea 10 imprime el valor inicial de `c`. La línea 11 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 11 imprime el valor inicial de `c` (`5`) otra vez. La línea 12 imprime el nuevo valor de `c` (`6`) para demostrar que en efecto se incrementó el valor de la variable en la línea 11.

La línea 17 restablece el valor de `c` a `5`, y la línea 18 imprime el valor de `c`. La línea 19 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 20 imprime el valor de `c` otra vez, para mostrar que sigue siendo `6` después de que se ejecuta la línea 19.

```
1  // Fig. 4.15: Incremento.java
2  // Operadores de preincremento y postincremento.
3
4  public class Incremento
5  {
```

Fig. 4.15 | Operadores de preincremento y postincremento (parte 1 de 2).

```
6      public static void main(String[] args)
7      {
8        // demuestra el operador de postincremento
9        int c;= 5;
10       System.out.printf("c antes del postincremento: %d%n", c);   // imprime 5
11       System.out.printf("         postincremento de c: %d%n", c++); // imprime 5
12       System.out.printf("  c despues del postincremento: %d%n", c);   // imprime 6
13
14       System.out.println(); // omite una línea
15
16       // demuestra el operador de preincremento
17       c = 5;
18       System.out.printf("c antes del preincremento: %d%n", c);   // imprime 5
19       System.out.printf("         preincremento de c: %d%n", ++c); // imprime 6
20       System.out.printf("  c despues del preincremento: %d%n", c);   // imprime 6
21       }
22   } // fin de la clase Incremento
```

```
   c antes del postincremento: 5
          postincremento de c: 5
 c despues del postincremento: 6

    c antes del preincremento: 5
           preincremento de c: 6
  c despues del preincremento: 6
```

Fig. 4.15 | Operadores de preincremento y postincremento (parte 2 de 2).

Simplificar instrucciones con los operadores de asignación compuestos aritméticos, de incremento y postdecremento

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 26, 28 y 31)

```
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 postincremento tienen distintos efectos (y lo mismo se aplica a los operadores de predecremento y postdecremento).



Error común de programación 4.8

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.

Precedencia y asociatividad de operadores

La figura 4.16 muestra la precedencia y la asociatividad de los operadores que presentamos. 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.



Buena práctica de programación 4.5

Al escribir expresiones que contengan muchos operadores, consulte la tabla de precedencia y asociatividad de operadores (apéndice A). Confirme que los operadores en la expresión se procesen en el orden esperado. Si no está seguro sobre el orden de evaluación en una expresión compleja, descomponga la expresión en instrucciones más pequeñas o use paréntesis para forzar el orden de evaluación, de la misma forma en que lo haría con una expresión algebraica. No olvide que algunos operadores, como el de asignación (=), se asocian de derecha a izquierda en vez de izquierda a derecha.

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.

4.14 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 tipiado**.

En C y C++, los programadores con frecuencia tienen que escribir 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), así como con 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 3.2 vimos que a las variables de tipos primitivos que se declaran fuera de un método como variables de instancia de una clase se les *asignan valores predeterminados de manera automática, 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`.

4.15 (Opcional) Ejemplo práctico 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 sobre cómo crear una ventana para mostrar un dibujo en la pantalla de la computadora.

El sistema de coordenadas de Java

Para dibujar en Java, primero 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 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*** indica 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.

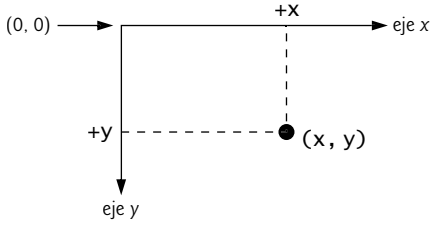


Fig. 4.17 | Sistema de coordenadas de Java. Las unidades están en pixeles.

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    }
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  // Crear un objeto JFrame para mostrar un objeto PanelDibujo.
3  import javax.swing.JFrame;
4
5  public class PruebaPanelDibujo
6  {
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
```

Fig. 4.19 | Creación de un objeto `JFrame` para mostrar un objeto `PanelDibujo` (parte 1 de 2).

```
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       }
22    } // fin de la clase PruebaPanelDibujo
```

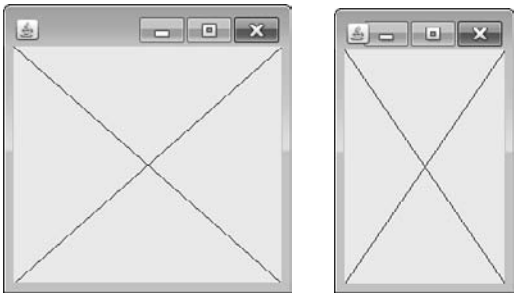


Fig. 4.19 | Creación de un objeto `JFrame` para mostrar un objeto `PanelDibujo` (parte 2 de 2).

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 particular, tiene 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 `PanelDibujo`. El método `paintComponent` debe declararse como se muestra en la línea 9; de no ser así, el sistema no llamará al método. Este método es llamado 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`. Este objeto `Graphics` se usa para dibujar líneas, rectángulos, óvalos y otros gráficos.

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`, éste 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`**, para 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 *escalarán* en concordancia, 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`** del método `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. Al utilizar ciclos e instrucciones de control para dibujar líneas se pueden lograr 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 conforme usted cambie el tamaño de la ventana.

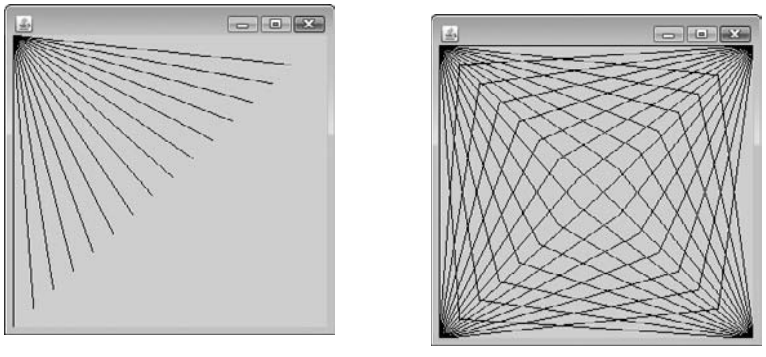


Fig. 4.20 | Despliegue de líneas desde una esquina.

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 borde 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 borde inferior. Para cada línea sucesiva, avance hacia abajo un incremento en el borde izquierdo, y un incremento a la derecha en el borde 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 bordes.

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.

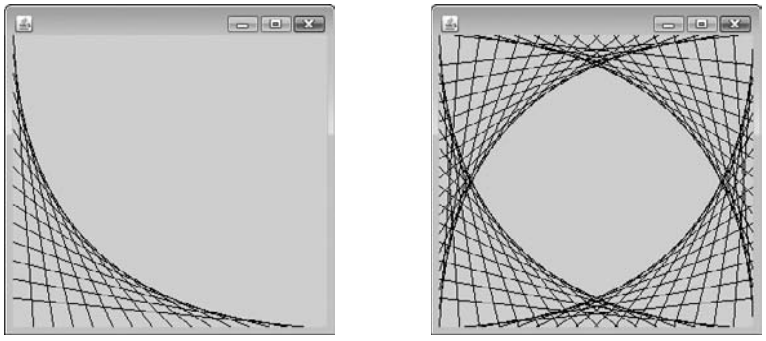


Fig. 4.21 | Arte lineal con ciclos y `drawLine`.

4.16 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 `if` de selección simple, la instrucción `if...else` de selección doble y la instrucción de repetición `while`. Estas instrucciones son algunos de los cimientos 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. También 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 nuestra explicación 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 comprender en detalle el problema y tener 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 ejecutando una serie de acciones (pág. 102), 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. 102).
- El proceso de especificar el orden en el que se ejecutan las instrucciones en un programa se denomina control del programa (pág. 102).

Sección 4.3 Seudocódigo

- El seudocódigo (pág. 103) 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. Desde luego que usted puede usar su propio lenguaje nativo para desarrollar su propio seudocódigo.
- 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. 103).
- 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. 103).
- Bohm y Jacopini demostraron que todos los programas podían escribirse en términos de sólo tres estructuras de control (pág. 103): 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. 104) forman parte de UML. Un diagrama de actividad modela el flujo de trabajo (pág. 104; también conocido como la actividad) de una parte de un sistema de software.
- Los diagramas de actividad se componen de símbolos (pág. 104) —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. 104) 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. 104); es decir, 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. 104); es decir, 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. 104), y son comentarios aclaratorios que describen el propósito de los símbolos en el diagrama.
- Java tiene tres tipos de instrucciones de selección (pág. 105).
- La instrucción `if` de selección simple (pág. 105) 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. 105), ya que selecciona una de varias acciones distintas, o grupos de acciones.
- Java cuenta con las instrucciones de repetición (también conocidas como de iteración o 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 las acciones en sus cuerpos, cero o más veces. Si al principio la condición de continuación de ciclo (pág. 105) es falsa, las acciones no se ejecutarán. La instrucción `do...while` lleva a cabo las acciones 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. 105) que sean apropiadas para el algoritmo que implementa ese programa.
- Las instrucciones de control de una sola entrada/una sola salida (pág. 105) 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. 105) 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 de acción alternativos.
- 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. 106). 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. 105) realiza una acción cuando la condición es verdadera, y otra acción distinta cuando es falsa.
- Un programa puede evaluar varios casos mediante instrucciones `if...else` anidadas (pág. 107).
- El operador condicional (pág. 110; `?:`) es el único operador ternario de Java; recibe tres operandos. En conjunto, los operandos y el símbolo `?:` forman una expresión condicional (pág. 110).
- 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. 110) de instrucciones puede colocarse en cualquier parte en donde se pueda colocar una sola instrucción.
- Un error lógico (pág. 110) tiene su efecto en tiempo de ejecución. Un error lógico fatal (pág.110) hace que un programa falle y termine antes de tiempo. Un error lógico no fatal (pág.110) 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 4.8 Instrucción de repetición `while`

- La instrucción de repetición `while` (pág. 114) permite al programador especificar que un programa debe repetir una acción, mientras cierta condición siga siendo verdadera.
- El símbolo de fusión (pág. 114) 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 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 4.9 Formulación de algoritmos: repetición controlada por un contador

- La repetición controlada por un contador (pág. 115) 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. 115), ya que el número de repeticiones se conoce desde antes que empiece a ejecutarse el ciclo.
- Un total (pág. 115) 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; la parte fraccionaria del cálculo se trunca.

Sección 4.10 Formulación de algoritmos: repetición controlada por un centinela

- En la repetición controlada por un centinela (pág. 119) 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. 120), 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 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 sobre los operandos seleccionados.
- El operador unario de conversión de tipos se forma colocando paréntesis alrededor del nombre de un tipo.

Sección 4.12 Operadores de asignación compuestos

- Los operadores de asignación compuestos (pág. 131) 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 4.13 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. 131).
- 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, respectivamente.
- Al preincrementar o predecrementar una variable, ésta se incrementa o decrementa en 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 en 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 4.14 Tipos primitivos

- Java requiere que todas las variables tengan un tipo. Por lo mismo, 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 de _____ 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 tipo 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` y `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 la variable `suma` de tipo `int` e inicializarla con `0`.
b) Declarar la variable `x` de tipo `int` e inicializarla con `1`.
c) Sumar la variable `x` a `suma` y asignar el resultado a la variable `suma`.
d) Imprimir "`La suma es: `", seguido 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 = 0;
```
b)
```
int x = 1;
```
c)
```
suma += x; o suma = suma + x;
```
d)
```
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 = 0;
8        int x = 1;
9
10       while (x <= 10) // mientras que x sea menor o igual que 10
11       {
12          suma += x; // suma x a suma
13          ++x; // incrementa x
14       }
15
16       System.out.printf("La suma es: %d%n", suma);
17    }
18 } // 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 x = 1;
7        int total = 0;
8
9        while (x <= 10)
10       {
11          int y = x * x;
12          System.out.println(y);
13          total += y;
14          ++x;
15       }
16
17       System.out.printf("El total es %d%n", total);
18    }
19 } // 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 por el kilometraje de sus automóviles. Un conductor ha llevado el registro de varios reabastecimientos de combustible, registrando los kilómetros conducidos y los litros usados en cada tanque lleno. 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 viajes 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 saldo 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 excedio el limite de su credito`".

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, para luego determinar y mostrar el sueldo bruto de cada trabajador. Utilice la clase `Scanner` para introducir los datos.

4.21 ***(Encontrar el número 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 vendedor 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       }
13    }
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
8        while (fila >= 1)
9        {
10          int columna = 1;
11
12          while (columna <= 10)
13          {
14             System.out.print(fila % 2 == 1 ? "<" : ">");
15             ++columna;
16          }
17
18          --fila;
19          System.out.println();
20       }
21    }
22 } // 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 1011 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 las potencias del entero 2 (es decir, 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, y luego 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

$n! = 1$ (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.

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Marcando 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 en red, 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 a usted 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 un 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 Internet las cuestiones sobre el crecimiento de la población mundial. *Asegúrese de investigar varios puntos de vista*. Obtenga estimaciones de la población mundial actual y de 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 la tasa de crecimiento de este año permaneciera.

5

Instrucciones de control: parte 2: operadores lógicos



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:

- Comprender los fundamentos 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

5.2 Fundamentos de la repetición controlada por contador

5.3 Instrucción de repetición `for`

5.4 Ejemplos sobre el uso de la instrucción `for`

5.5 Instrucción de repetición `do...while`

5.6 Instrucción de selección múltiple `switch`

5.7 Ejemplo práctico de la clase `PolizaAuto`: objetos `String` en instrucciones `switch`

5.8 Instrucciones `break` y `continue`

5.9 Operadores lógicos

5.10 Resumen sobre programación estructurada

5.11 (Opcional) Ejemplo práctico de GUI y gráficos: dibujo de rectángulos y óvalos

5.12 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Marcando la diferencia*

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 de la repetición controlada por contador. Usaremos 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, que requiere

1. una **variable de control** (o contador de ciclo)
2. el **valor inicial** de la variable de control
3. el **incremento** 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.

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
14         }
15
16         System.out.println();
17      }
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 sea `true`). El programa ejecuta el cuerpo de este `while` aun 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, ya que la condición de `while` realiza el incremento antes de evaluar la condición (en la sección 4.13 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();
14    }
15 } // fin de la clase ContadorFor
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 5.2 | Repetición controlada por 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 y el operador <= en la condición de un ciclo 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`*.*



Tip para prevenir errores 5.3

Como se mencionó en el capítulo 4, los enteros pueden desbordarse y producir errores lógicos. Una variable de control de ciclo también podría desbordarse. Para evitar esto escriba sus condiciones de ciclo con cuidado.

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, que incluye la palabra clave `for` y todo lo que está entre paréntesis después de ésta (línea 10 de la figura 5.2), algunas veces se le llama **encabezado de la instrucción `for`**. El encabezado de `for` "se encarga de todo"; es decir, 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.

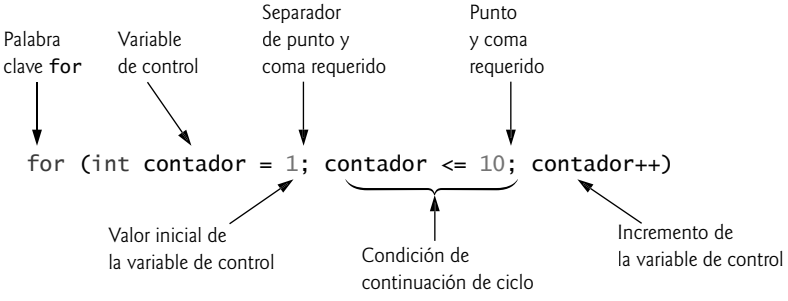


Fig. 5.3 | Componentes del encabezado de la instrucción `for`.

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, 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 5.8 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 de `for` declara la variable de control (es decir, 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` y no existirá fuera de ella. Este uso restringido se conoce como el **alcance** de la variable. El alcance de una variable define en qué parte de un programa puede usarse. 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 de `for`, si se utiliza la variable de control después del cuerpo de `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*. Si el programa inicializa la variable de control *antes* del ciclo podríamos omitir la expresión de *inicialización*. Por su parte, si el programa calcula el incremento mediante instrucciones dentro del cuerpo del ciclo, o si no se necesita un incremento, podríamos omitir la expresión de *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 lo que la forma de postincremento parece más natural. En este caso, la variable que se incrementa no aparece en una expresión más larga, 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 5.4
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 modifique durante cada iteración del ciclo, de modo que la condición de continuación de ciclo se vuelva `false` en un momento dado. En un ciclo controlado por centinela, asegúrese que el valor centinela pueda introducirse.

Colocación de 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 la instrucción
A menudo 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 de `for`.



Tip para prevenir errores 5.5
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 4.6). La figura 5.4 muestra el diagrama de actividad de la instrucción `for` de la figura 5.2. El diagrama hace 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.

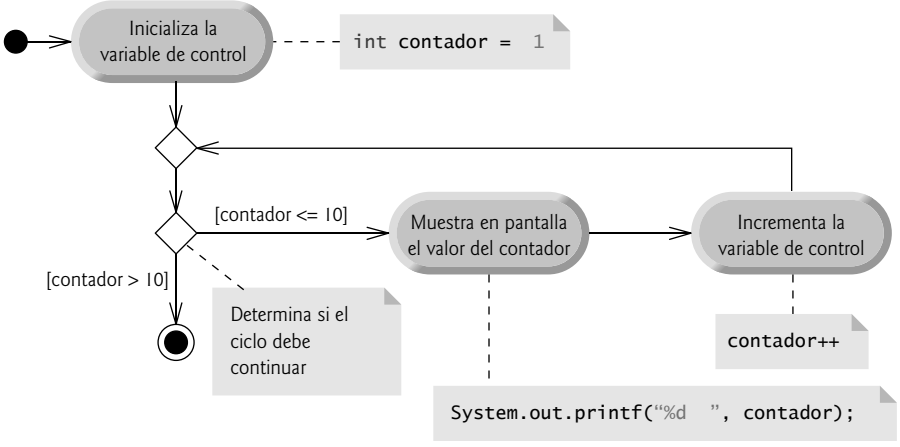


Fig. 5.4 | Diagrama de actividad de UML para la instrucción `for` de la figura 5.2.

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, *sólo* 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.



Error común de programación 5.6

No use los operadores de igualdad (`!=` o `==`) en una condición de continuación de ciclo si la variable de control del ciclo se incrementa o decrementa por más de 1. Por ejemplo, considere el encabezado de la instrucción `for (int contador = 1; contador != 10; contador+=2)`. La prueba de continuación de ciclo `contador !=10` nunca se vuelve falsa (lo que ocasiona un ciclo infinito*) debido a que `contador` se incrementa por 2 después de cada iteración.*

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  {
```

Fig. 5.5 | Sumar enteros con la instrucción `for` (parte 1 de 2).

```
6      public static void main(String[] args)
7      {
8         int total = 0;
9
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);
15     }
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: cálculos del interés compuesto

Usemos 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 inicial)
>
> 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  {
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       }
24    }
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`.

Cómo 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. Utilizamos el especificador de formato `%20s` para mostrar en pantalla el objeto `String` "`Monto en deposito`". El entero `20` entre el `%` 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`).

Cálculo del interés con el método `static` `pow` *de la clase* `Math`

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 (la variable `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`. Algunas clases también cuentan con métodos que realizan tareas comunes y *no* requieren que el programador cree primero objetos de esas clases. 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.nombreMetodo(*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*-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 del código compilado.

Cómo 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 con 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 18.673 (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. ¡Ya está enterado!



Tip para prevenir errores 5.6

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 este fin, la cual demostraremos en la figura 8.16.*

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` 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;
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();
17    }
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.

Diagrama de actividad de UML para la instrucción de repetición `do...while`

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 4.6).

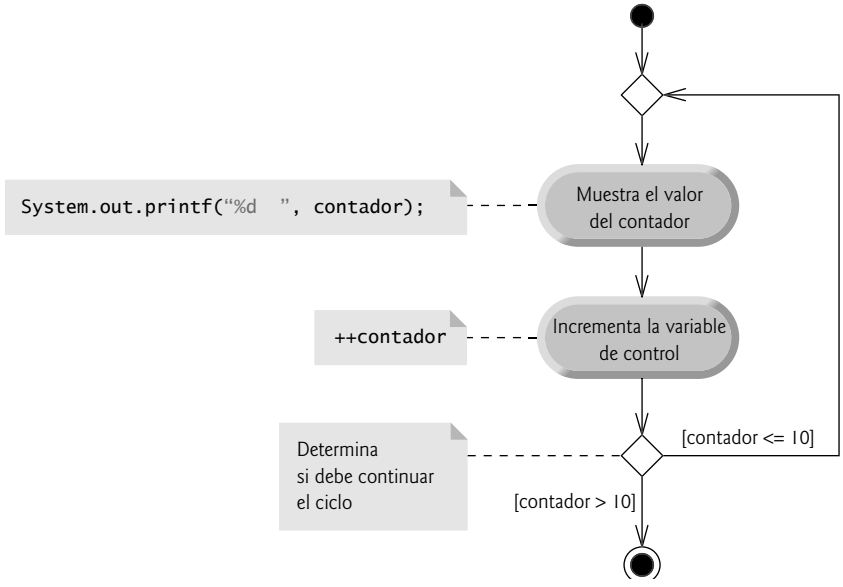


Fig. 5.8 | Diagrama de actividad de UML de la instrucción de repetición `do...while`.

Las llaves en una 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 con llaves de la siguiente forma:

```
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`. A partir de Java SE 7, la expresión también puede ser un objeto `String`, lo cual veremos en la sección 5.7.

Uso de una instrucción `switch` para contar las calificaciones A, B, C, D y F

La figura 5.9 calcula el promedio de un conjunto de calificaciones numéricas introducidas por el usuario, y 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. El programa también imprime en pantalla un resumen del número de estudiantes que recibieron cada calificación.

Al igual que las versiones anteriores del programa de promedio de una clase, el método `main` de la clase `CalificacionesLetra` (figura 5.9) declara las variables locales `total` (línea 9) y `contadorCalif` (línea 10) para llevar 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. Cabe mencionar que las variables en las líneas 9 a 15 se inicializan de manera explícita con `0`.

El método `main` tiene dos partes clave. Las líneas 26 a 56 leen un número arbitrario de calificaciones enteras del usuario mediante el uso de la repetición controlada por un centinela, actualizan las variables de instancia `total` y `contadorCalif` e incrementan un contador de letra de calificación apropiado para cada calificación introducida. Las líneas 59 a 80 imprimen 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. Examinemos estas instrucciones con más detalle.

```
1  // Fig. 5.9: CalificacionesLetra.java
2  // La clase CalificacionesLetra usa la instrucción switch para contar las
      calificaciones de letra.
3  import java.util.Scanner;
4
5  public class CalificacionesLetra
6  {
7     public static void main(String[] args)
8     {
```

Fig. 5.9 | La clase `CalificacionesLetra` usa la instrucción `switch` para contar las calificaciones de letra (parte 1 de 3).

```
9       int total = 0; // suma de calificaciones
10      int contadorCalif = 0; // número de calificaciones introducidas
11      int aCuenta = 0; // cuenta de calificaciones A
12      int bCuenta = 0; // cuenta de calificaciones B
13      int cCuenta = 0; // cuenta de calificaciones C
14      int dCuenta = 0; // cuenta de calificaciones D
15      int fCuenta = 0; // cuenta de calificaciones F
16
17      Scanner entrada = new Scanner(System.in);
18
19      System.out.printf("%s%n%s%n   %s%n   %s%n",
20         "Introduzca las calificaciones en el rango de 0-100.",
21         "Escriba el indicador de fin de archivo para terminar la entrada:",
22         "En UNIX/Linux/Mac OS X escriba <Ctrl> d y oprima Intro",
23         "En Windows escriba <Ctrl> z y oprima Intro");
24
25      // itera hasta que el usuario introduzca el indicador de fin de archivo
26      while (entrada.hasNext())
27      {
28         int calificacion = entrada.nextInt(); // lee calificacion
29         total += calificacion; // suma calificacion al total
30         ++contadorCalif; // incrementa el número de calificaciones
31
32         //  incrementa el contador de calificación de letra apropiado
33         switch (calificacion / 10)
34         {
35             case 9:  // calificacion estaba entre 90
36             case 10: // y 100, inclusive
37                ++aCuenta;
38                break; // sale del switch
39
40             case 8: // calificacion estaba entre 80 y 89
41                ++bCuenta;
42                break; // sale del switch
43
44             case 7: // calificacion estaba entre 70 y 79
45                ++cCuenta;
46                break; // sale del switch
47
48             case 6: // calificacion estaba entre 60 y 69
49                ++dCuenta;
50                break; // sale del switch
51
52             default: // calificacion era menor que 60
53                ++fCuenta;
54                break; // opcional; sale del switch de todas formas
55          } // fin de switch
56      } // fin de while
57
58      // muestra reporte de calificaciones
59      System.out.printf("%nReporte de calificaciones:%n");
60
61      // si el usuario introdujo al menos una calificación...
62      if (contadorCalif != 0)
```

Fig. 5.9 | La clase `CalificacionesLetra` usa la instrucción `switch` para contar las calificaciones de letra (parte 2 de 3).

```
63         {
64            // calcula el promedio de todas las calificaciones introducidas
65            double promedio = (double) total / contadorCalif;
66
67            // muestra resumen de resultados en pantalla
68            System.out.printf("El total de las %d calificaciones introducidas es
                                %d%n",
69              contadorCalif, total);
70            System.out.printf("El promedio de la clase es %.2f%n", promedio);
71            System.out.printf("%n%s%n%s%d%n%s%d%n%s%d%n%s%d%n%s%d%n",
72              "Numero de estudiantes que recibieron cada calificacion:",
73              "A: ", aCuenta,   // muestra número de calificaciones A
74              "B: ", bCuenta,   // muestra número de calificaciones B
75              "C: ", cCuenta,   // muestra número de calificaciones C
76              "D: ", dCuenta,   // muestra número de calificaciones D
77              "F: ", fCuenta); // muestra número de calificaciones F
78         } // fin de if
79         else // no se introdujeron calificaciones, por lo que se muestra el mensaje
                   apropiado
80            System.out.println("No se introdujeron calificaciones");
81      } // fin de main
82   } // fin de la clase CalificacionesLetra
```

```
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 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

Numero de estudiantes que recibieron cada calificacion:
A: 4
B: 1
C: 2
D: 1
F: 2
```

Fig. 5.9 | La clase `CalificacionesLetra` usa la instrucción `switch` para contar las calificaciones de letra (parte 3 de 3).

Lectura de las calificaciones del usuario

Las líneas 19 a 23 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 15, 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 independiente. 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 5.9].



Tip de portabilidad 5.1
Las combinaciones de pulsaciones de teclas para introducir el fin de archivo son dependientes del sistema.

La instrucción `while` (líneas 26 a 56) obtiene la entrada del usuario. La condición en la línea 26 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 28 recibe como entrada el valor de una calificación del usuario. La línea 29 suma `calificacion` a `total`. La línea 30 incrementa `contadorCalif`. Estas variables se usan para calcular el promedio de las calificaciones. Las líneas 33 a 55 usan una instrucción `switch` para incrementar el contador de letra de calificación apropiado, con base en la calificación numérica introducida.

Procesamiento de las calificaciones

La instrucción `switch` (líneas 33 a 55) determina cuál contador se debe incrementar. Vamos a asumir 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`, o como un `String`) con cada una de las etiquetas `case`. La expresión de control de la línea 33 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 40), el programa ejecuta las instrucciones para esa instrucción `case`. Para el entero `8`, la línea 41 incrementa a `bCuenta`, ya que una calificación entre 80 y 89 es una B. La **instrucción `break`** (línea 42) hace que el control del programa proceda con la primera instrucción después de `switch`; en este programa, llegamos al final del ciclo `while`, por lo que el control regresa a la condición de continuación de ciclo en la línea 26 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 35 y 36, 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 37 y 38. 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 de cada `case`.

`case` sin una instrucción `break`

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 repetitiva "Los doce días de Navidad" en el ejercicio 5.29).



Error común de programación 5.7

Olvidar una instrucción `break` cuando se necesita una en una instrucción `switch` es un error lógico.

El caso `default`

Si no ocurre una coincidencia entre el valor de la expresión de control y una etiqueta `case`, se ejecuta el caso `default` (líneas 52 a 54). 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 reprobatorias. 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`.



Tip para prevenir errores 5.7

En una instrucción `switch`, asegúrese de probar todos los posibles valores de la expresión de control.

Mostrar el reporte de calificaciones

Las líneas 58 a 80 imprimen en pantalla un reporte con base en las calificaciones introducidas (como en la ventana de entrada/salida en la figura 5.9). La línea 62 determina si el usuario introdujo por lo menos una calificación; esto nos ayuda a evitar la división entre cero. De ser así, la línea 65 calcula el promedio de las calificaciones. A continuación, las líneas 68 a 77 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 80 imprime en pantalla un mensaje apropiado. Los resultados en la figura 5.9 muestran un reporte de calificaciones de ejemplo, con base en 10 calificaciones.

Diagrama de actividad de UML de la instrucción `switch`

La figura 5.10 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 5.10 enfatiza esto al incluir instrucciones `break` en el diagrama de actividad, el cual hace evidente que la instrucción `break` al final de una etiqueta `case` provoca que el control salga de la instrucción `switch` de inmediato.

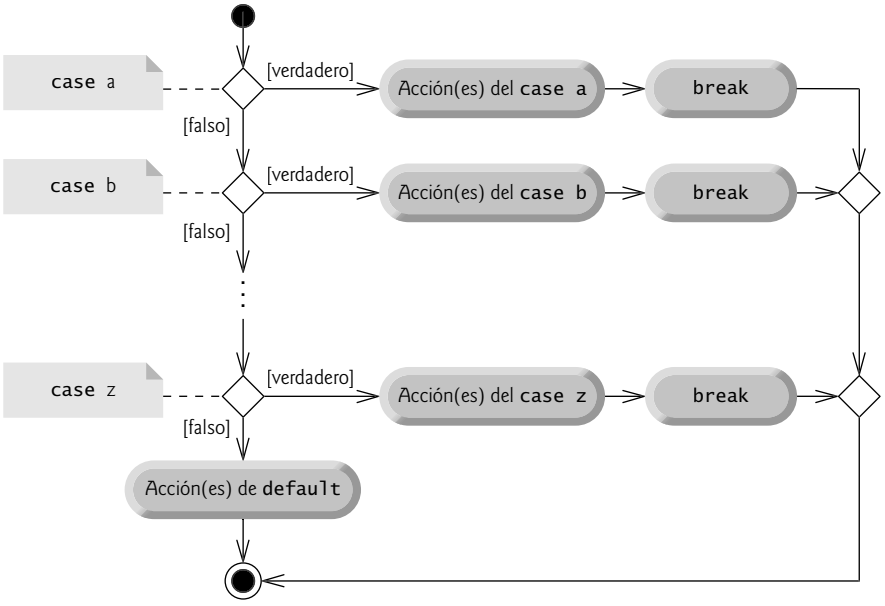


Fig. 5.10 | Diagrama de actividad de UML de la instrucción `switch` de selección múltiple con instrucciones `break`.

No se requiere la instrucción `break` para la última etiqueta `case` del `switch` (ni para el caso `default` opcional, cuando aparece al último), ya que la ejecución continúa con la siguiente instrucción que va después del `switch`.



Tip para prevenir errores 5.8

Proporcione un caso `default` en las instrucciones `switch`. Esto lo hará 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)— o un `String`. 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**; es decir, 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 tipos `enum` (enumeraciones), que también presentaremos en el capítulo 6. Las constantes tipo `enum` también pueden utilizarse en etiquetas `case`.

En el capítulo 10, Programación orientada a objetos: polimorfismo e interfaces, 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`.

5.7 Ejemplo práctico de la clase `PolizaAuto`: objetos `String` en instrucciones `switch`

Los objetos `String` pueden usarse en la expresión de control de una instrucción `switch`, y las literales `String` pueden usarse en las etiquetas `case`. Para demostrar esto, implementaremos una app que cumple con los siguientes requisitos:

> *Usted fue contratado por una compañía de seguros de autos que da servicio a los siguientes estados del noreste de los Estados Unidos: Connecticut, Maine, Massachusetts, Nuevo Hampshire, Nueva Jersey, Nueva York, Pensilvania, Rhode Island y Vermont. La compañía desea que usted cree un programa que genere un reporte que indique si cada una de sus pólizas de seguros de automóviles se encuentra en un estado con seguro de autos "sin culpa": Massachusetts, Nueva Jersey, Nueva York y Pensilvania.*

La aplicación de Java que cumple con estos requerimientos contiene dos clases: `PolizaAuto` (figura 5.11) y `PruebaPolizaAuto` (figura 5.12).

La clase `PolizaAuto`

La clase `PolizaAuto` (figura 5.11) representa una póliza de seguro de auto. La clase contiene:

- La variable de instancia `int` llamada `numeroCuenta` (línea 5) para almacenar el número de cuenta de la póliza
- La variable de instancia `String` llamada `marcaYModelo` (línea 6) para almacenar la marca y modelo del auto (como "`Toyota Camry`")
- La variable de instancia `String` llamada `estado` (línea 7) para almacenar una abreviación de dos caracteres que represente el estado en el que se encuentra la póliza (por ejemplo, "`MA`" para Massachusetts)
- Un constructor (líneas 10 a 15) que inicializa las variables de instancia de la clase
- Los métodos `establecerNumeroCuenta` y `obtenerNumeroCuenta` (líneas 18 a 27) para *establecer* y *obtener* la variable de instancia `NumeroCuenta` de una `PolizaAuto`
- Los métodos `establecerMarcaYModelo` y `obtenerMarcaYModelo` (líneas 30 a 39) para *establecer* y *obtener* la variable de instancia `marcaYModelo` de una `PolizaAuto`
- Los métodos `establecerEstado` y `obtenerEstado` (líneas 42 a 51) para *establecer* y *obtener* la variable de instancia `estado` de una `PolizaAuto`
- El método `esEstadoSinCulpa` (líneas 54 a 70) para devolver un valor `boolean` que indique si la póliza se encuentra en un estado con seguro de auto sin culpa. Observe el nombre del método: la convención de nomenclatura para un método *obtener* que devuelve un valor `boolean` es comenzar el nombre con "`es`" en vez de "`obtener`" (dicho método se conoce comúnmente como *método predicado*).

En el método `esEstadoSinCulpa`, la expresión de control de la instrucción `switch` (línea 59) es el objeto `String` devuelto por el método `obtenerEstado` de `PolizaAuto`. La instrucción `switch` compara el valor de la expresión de control con las etiquetas `case` (línea 61) para determinar si la póliza se encuentra en Massachusetts, Nueva Jersey, Nueva York o Pensilvania (los estados sin culpa).

Si hay una coincidencia, entonces la línea 62 establece la variable local `estadoSinCulpa` a `true` y la instrucción `switch` termina; de lo contrario, el caso `default` establece `estadoSinCulpa` a `false` (línea 65). Después el método `esEstadoSinCulpa` devuelve el valor de la variable local `estadoSinCulpa`.

Por simplicidad no validamos los datos de una `PolizaAuto` en el constructor ni en los métodos *establecer*, y asumimos que las abreviaciones de los estados siempre son dos letras mayúsculas. Además, una verdadera clase `PolizaAuto` contendría probablemente muchas otras variables de instancia y métodos para datos tales como el nombre del propietario de la cuenta, dirección, etc. En el ejercicio 5.30 se le pedirá que mejore la clase `PolizaAuto`: deberá validar la abreviación de los estados mediante el uso de técnicas que aprenderá en la sección 5.9.

```
1  // Fig. 5.11: PolizaAuto.java
2  // Clase que representa una póliza de seguro de automóvil.
3  public class PolizaAuto
4  {
5     private int numeroCuenta; // número de cuenta de la póliza
6     private String marcaYModelo; // auto al que se aplica la póliza
7     private String estado; // abreviación del estado en dos letras
8
9     // constructor
10    public PolizaAuto(int numeroCuenta, String marcaYModelo, String estado)
11    {
12       this.numeroCuenta = numeroCuenta;
13       this.marcaYModelo = marcaYModelo;
14       this.estado = estado;
15    }
16
17    // establece el numeroCuenta
18    public void establecerNumeroCuenta(int numeroCuenta)
19    {
20       this.numeroCuenta = numeroCuenta;
21    }
22
23    // devuelve el numeroCuenta
24    public int obtenerNumeroCuenta()
25    {
26       return numeroCuenta;
27    }
28
29    // establece la marcaYModelo
30    public void establecerMarcaYModelo(String marcaYModelo)
31    {
32       this.marcaYModelo = marcaYModelo;
33    }
34
35    // devuelve la marcaYModelo
36    public String obtenerMarcaYModelo()
37    {
38       return marcaYModelo;
39    }
40
41    // establece el estado
42    public void establecerEstado(String estado)
```

Fig. 5.11 | Clase que representa una póliza de seguro de auto (parte 1 de 2).

```
43     {
44        this.estado = estado;
45     }
46
47     // devuelve el estado
48     public String obtenerEstado()
49     {
50        return estado;
51     }
52
53     // método predicado que devuelve si el estado tiene seguro sin culpa
54     public boolean esEstadoSinCulpa()
55     {
56        boolean estadoSinCulpa;
57
58        // determina si el estado tiene seguro de auto sin culpa
59        switch (obtenerEstado()) // obtiene la abreviación del estado del objeto
                                    PolizaAuto
60        {
61           case "MA": case "NJ": case "NY": case "PA":
62              estadoSinCulpa = true;
63              break;
64           default:
65              estadoSinCulpa = false;
66              break;
67        }
68
69        return estadoSinCulpa;
70     }
71  } // fin de la clase PolizaAuto
```

Fig. 5.11 | Clase que representa una póliza de seguro de auto (parte 2 de 2).

La clase `PruebaPolizaAuto`

La clase `PruebaPolizaAuto` (figura 5.12) crea dos objetos `PolizaAuto` (líneas 8 a 11 en `main`). Las líneas 14 y 15 pasan cada objeto al método `static` `polizaEnEstadoSinCulpa` (líneas 20 a 28), que usa los métodos `PolizaAuto` para determinar y mostrar en pantalla si el objeto que recibe representa una póliza en un estado de seguros de auto sin culpa.

```
1  // Fig. 5.12: PruebaPolizaAuto.java
2  // Demostración de objetos String en la instrucción switch.
3  public class PruebaPolizaAuto
4  {
5     public static void main(String[] args)
6     {
7        // crea dos objetos PolizaAuto
8        PolizaAuto poliza1 =
9           new PolizaAuto(11111111, "Toyota Camry", "NJ");
10       PolizaAuto poliza2 =
11          new PolizaAuto(22222222, "Ford Fusion", "ME");
```

Fig. 5.12 | Demostración de objetos `String` en la instrucción `switch` (parte 1 de 2).

```
12
13        // muestra en pantalla si cada poliza está en un estado sin culpa
14        polizaEnEstadoSinCulpa(poliza1);
15        polizaEnEstadoSinCulpa(poliza2);
16     }
17
18     // método que muestra en pantalla si una PolizaAuto
19     // se encuentra en un estado con seguro de auto sin culpa
20     public static void polizaEnEstadoSinCulpa(PolizaAuto poliza)
21     {
22        System.out.println("La poliza de auto:");
23        System.out.printf(
24           "Cuenta #: %d; Auto: %s; Estado %s %s un estado sin culpa%n%n",
25           poliza.obtenerNumeroCuenta(), poliza.obtenerMarcaYModelo(),
26           poliza.obtenerEstado(),
27           (poliza.esEstadoSinCulpa() ? "esta en": "no esta en"));
28     }
29  } // fin de la clase PruebaPolizaAuto
```

```
La poliza de auto:
Cuenta #: 11111111; Auto: Toyota Camry;
Estado NJ esta en un estado sin culpa

La poliza de auto:
Cuenta #: 22222222; Auto: Ford Fusion;
Estado ME no esta en un estado sin culpa
```

Fig. 5.12 | Demostración de objetos `String` en la instrucción `switch` (parte 2 de 2).

5.8 Instrucciones `break` y `continue`

Además de las instrucciones de selección y repetición, Java cuenta con las instrucciones `break` (que vimos en el contexto de la instrucción `switch`) y **`continue`** (que presentamos en esta sección y en el apéndice L en línea) 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 la instrucción `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 5.13 demuestra el uso de una instrucción `break` para salir de un ciclo `for`.

```
1  // Fig. 5.13: 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     {
```

Fig. 5.13 | La instrucción `break` para salir de una instrucción `for` (parte 1 de 2).

```
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 )
12              break; // termina el ciclo si cuenta es 5
13
14           System.out.printf("%d ", cuenta);
15        }
16
17        System.out.printf("%nSalio del ciclo en cuenta = %d%n", cuenta);
18     }
19  } // fin de la clase PruebaBreak
```

```
1 2 3 4
Salio del ciclo en cuenta = 5
```

Fig. 5.13 | Instrucción `break` para salir de una instrucción `for` (parte 2 de 2).

Cuando la instrucción `if` anidada en las líneas 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`, el programa 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. 5.14: 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)
10             continue; // omite el resto del código en el ciclo si cuenta es 5
11
12          System.out.printf("%d ", cuenta);
13       }
14
15       System.out.println("%nSe uso continue para omitir imprimir 5%n");
16    } // fin de main
17 } // fin de la clase PruebaContinue
```

Fig. 5.14 | Instrucción `continue` para terminar una iteración de una instrucción `for` (parte 1 de 2).

```
1 2 3 4 6 7 8 9 10
Se uso continue para omitir imprimir 5
```

Fig. 5.14 | Instrucción `continue` para terminar una iteración de una instrucción `for` (parte 2 de 2).

La figura 5.14 utiliza la instrucción `continue` (línea 10) para omitir la instrucción de la línea 12 cuando la instrucción `if` anidada 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 5.2

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 5.3

Existe una tensión entre lograr la ingeniería de software de calidad y lograr el software con el más alto 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 eficiente y conciso, pero sólo si es necesario.

5.9 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 K en línea 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 5.15 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. 5.15 | 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 5.16 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. 5.16 | 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 a FEMENINO (es decir, que toda la expresión es false) y continúa si genero *es* igual a 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 condicionales se conoce como **evaluación en corto circuito**.



Error común de programación 5.8
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 del operador && para prevenir errores. Considere la expresión (i != 0) && (10/i == 2). La condición dependiente (10/i == 2) debe aparecer después del operador && para evitar una posible 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 a 1. Esto es útil si el operando derecho 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 5.9
Por cuestión de claridad, evite las expresiones con efectos secundarios en las condiciones (como las asignaciones). Éstos pueden hacer que el código sea más difícil de entender y podrían llegar a producir errores lógicos sutiles.



Tip para prevenir errores 5.10
Por lo general las expresiones de asignación (=) no *deben usarse en condiciones. Cada condición* debe *producir un valor* boolean *como resultado; de lo contrario se producirá un error de compilación. En una condición, una asignación se compilará* sólo *si se asigna una expresión* boolean *a una variable* boolean.

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 5.17 es una tabla de verdad para el operador OR exclusivo lógico booleano (^). 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. 5.17 | 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 5.18 es una tabla de verdad para el operador lógico de negación.

expresión	!expresión
false	true
true	false

Fig. 5.18 | Tabla de verdad del operador `!` (negación lógica, o NOT lógico).

Ejemplo de los operadores lógicos

La figura 5.19 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. 5.19: 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 (|)",
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%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", "NOT logico (!)",
```

Fig. 5.19 | Los operadores lógicos (parte 1 de 2).

```
47            "!false", (!false), "!true", (!true));
48      }
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. 5.19 | Los operadores lógicos (parte 2 de 2).

Precedencia y asociatividad de los operadores presentados hasta ahora

La figura 5.20 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
`++ -- + - !` (*tipo*)	derecha a izquierda	prefijo unario
`* / %`	izquierda a derecha	multiplicativo

Fig. 5.20 | Precedencia/asociatividad de los operadores descritos hasta ahora (parte 1 de 2).

Operadores	Asociatividad	Tipo
`+ -`	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. 5.20 | Precedencia/asociatividad de los operadores descritos hasta ahora (parte 2 de 2).

5.10 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.

Las instrucciones de control de Java son de una sola entrada/una sola salida

La figura 5.21 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 en secuencia una después de la otra en un programa. A esto le llamamos *apilamiento de instrucciones de control*. Las reglas para formar programas estructurados también permiten *anidar* las instrucciones de control.

Reglas para formar programas estructurados

La figura 5.22 muestra las reglas para formar programas estructurados. Las reglas suponen que pueden utilizarse estados de acción para indicar *cualquier* acción. Además, las reglas suponen que comenzamos con el diagrama de actividad más sencillo (figura 5.23), que consiste solamente de un estado inicial, un estado de acción, un estado final y flechas de transición.

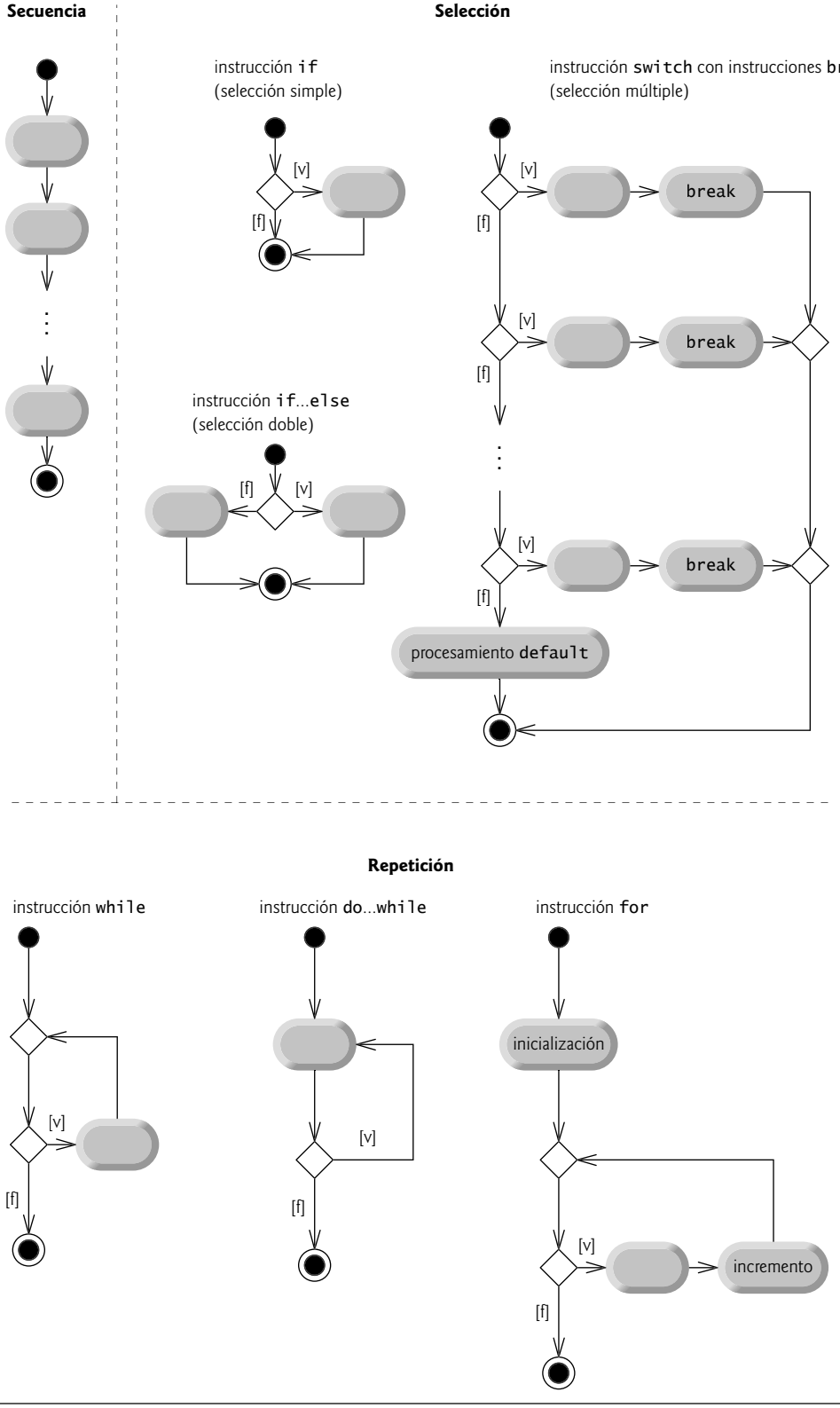


Fig. 5.21 | Instrucciones de secuencia, selección y repetición de una sola entrada/una sola salida de Java.

Reglas para formar programas estructurados
1. Comenzar con el diagrama de actividad más sencillo (figura 5.23).
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. 5.22 | Reglas para formar programas estructurados.

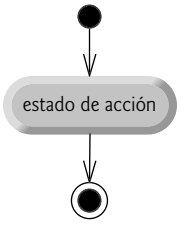


Fig. 5.23 | El diagrama de actividad más simple.

Al aplicar las reglas de la figura 5.22, siempre se obtiene un diagrama de actividad estructurado en forma apropiada, con una agradable apariencia de bloques 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 5.24). 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 5.24 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 5.22.

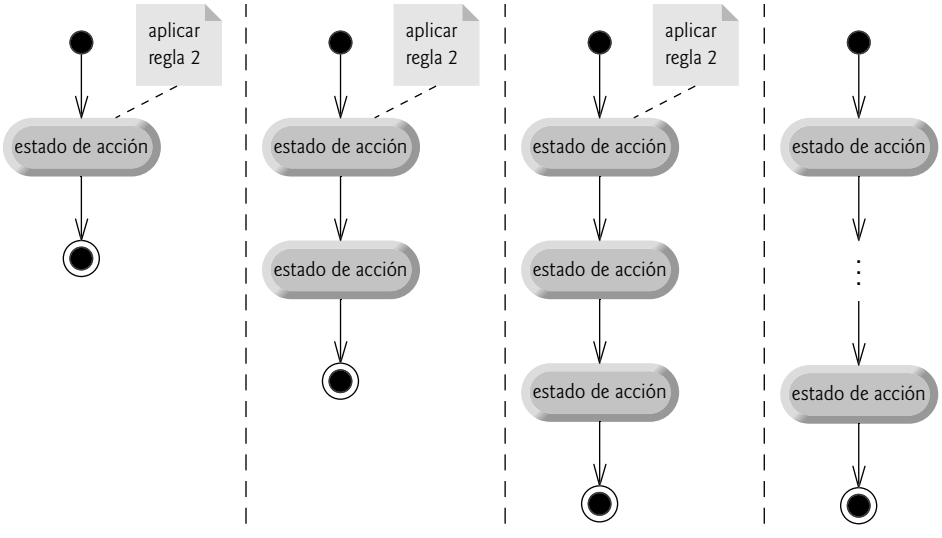


Fig. 5.24 | El resultado de aplicar la regla 2 de la figura 5.22 repetidamente al diagrama de actividad más sencillo.

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 5.25 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 5.25 no son parte de UML. Aquí se utilizan para ilustrar que *cualquier* estado de acción puede reemplazarse con un enunciado de control].

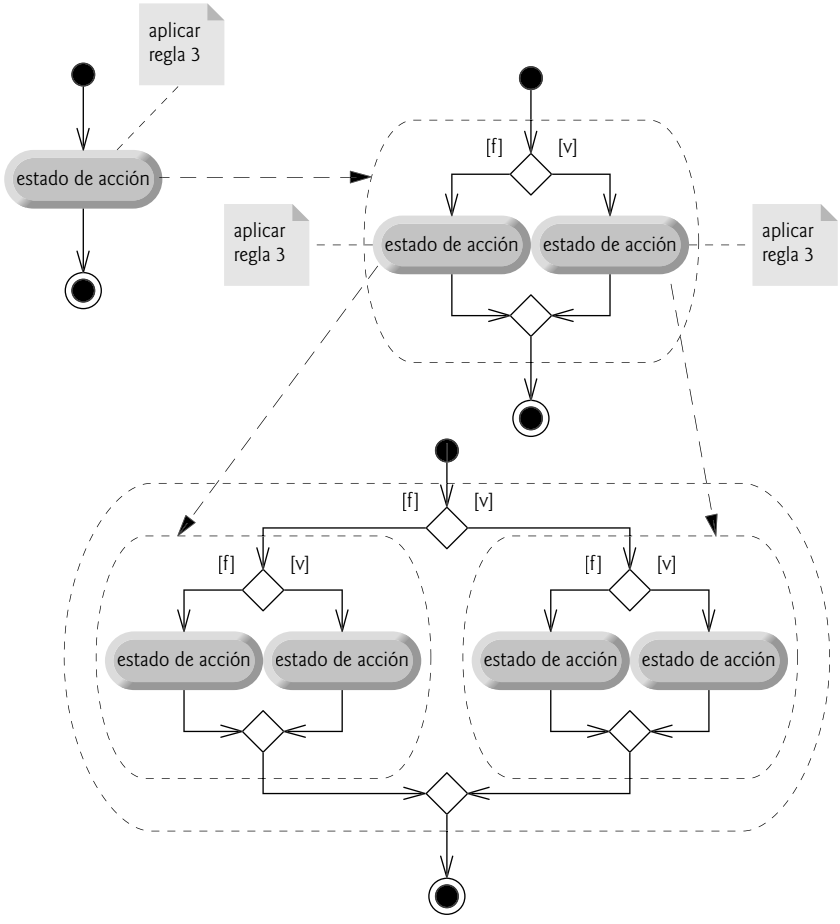


Fig. 5.25 | Aplicación en forma repetida de la regla 3 de la figura 5.22 al diagrama de actividad más sencillo.

La regla 4 genera instrucciones más grandes, más complicadas y más profundamente anidadas. Los diagramas que surgen debido a la aplicación de las reglas de la figura 5.22 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 5.22, no podrá crearse un diagrama de actividad "sin estructura" (como el de la figura 5.26). Si usted no está seguro de que cierto diagrama sea estructurado, aplique las reglas de la figura 5.22 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.

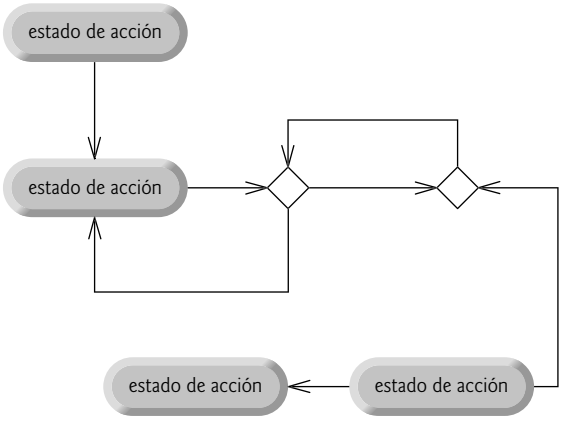


Fig. 5.26 | Diagrama de actividad "sin estructura".

Tres formas de control

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 enumera 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 7.7]. 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 de 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.

5.11 (Opcional) Ejemplo práctico 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 5.27.

```
1  // Fig. 5.27: Figuras.java
2  /// Cómo dibujar una cascada de figuras con base en la elección del usuario.
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    }
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          // elige 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          }
35       }
36    }
37 } // fin de la clase Figuras
```

Fig. 5.27 | Cómo dibujar una cascada de figuras con base en la elección del usuario.

La línea 6 empieza la declaración de la clase para `Figuras`, que extiende a `JPanel`. La variable de instancia `opcion` determina si `paintComponent` debe dibujar rectángulos u óvalos. El constructor de `Figuras` inicializa `opcion` con el valor que se pasa en el parámetro `opcionUsuario`.

El método `paintComponent` realiza en sí el dibujo. 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) elige 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 5.28 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. Las líneas 11 a 13 muestran un cuadro de diálogo al usuario y almacenan la respuesta de éste en la variable `entrada`. Cabe mencionar que al mostrar varias líneas de texto en un `JOptionPane`, hay que usar `\n` para comenzar una nueva línea, en vez de `%n`. 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 un objeto `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 ejemplo práctico (crear un *marco*, configurarlo para que la aplicación termine cuando se cierre, agregar el dibujo al marco, establecer su tamaño y hacerlo visible).

```
1  // Fig. 5.28: PruebaFiguras.java
2  // Obtener la entrada del usuario y crear un JFrame para mostrar Figuras.
3  import javax.swing.JFrame; // maneja la visualización
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");
```

Fig. 5.28 | Cómo obtener datos de entrada del usuario y crear un objeto `JFrame` para mostrar `Figuras` (parte 1 de 2).

```
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);
19
20          JFrame aplicacion = new JFrame(); // crea un nuevo objeto JFrame
21
22          aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23          aplicacion.add(panel);
24          aplicacion.setSize(300, 300);
25          aplicacion.setVisible(true);
26       }
27    } // fin de la clase PruebaFiguras
```

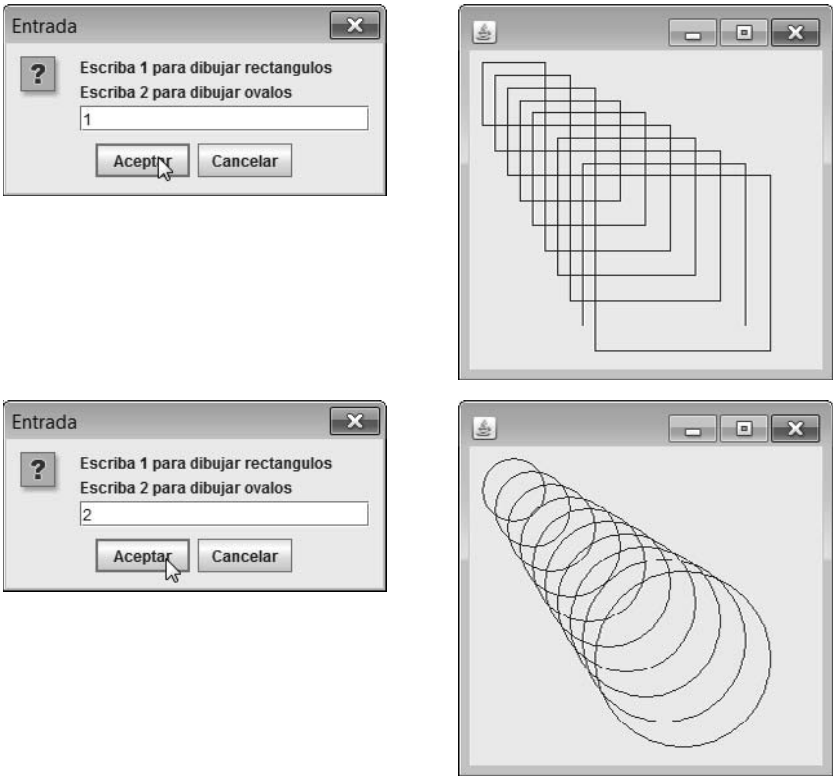


Fig. 5.28 | Cómo obtener datos de entrada del usuario y crear un objeto `JFrame` para mostrar `Figuras` (parte 2 de 2).

Ejercicios del ejemplo práctico de GUI y gráficos

5.1 Dibuje 12 círculos concéntricos en el centro de un objeto `JPanel` (figura 5.29). 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).

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.

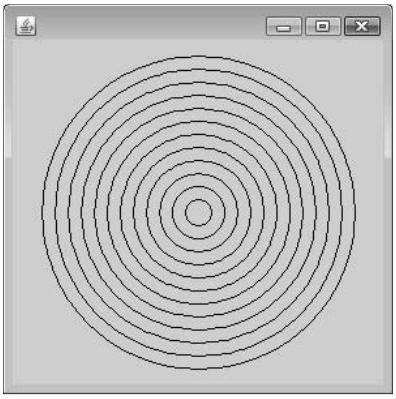


Fig. 5.29 | Cómo dibujar círculos concéntricos.

5.12 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`. 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, 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. Utilizó la instrucción `break` para salir de una instrucción `switch` y para terminar de inmediato un ciclo, y usó una instrucción `continue` para terminar la iteración actual de un ciclo y continuar con la siguiente iteración del mismo. 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. 153) requiere una variable de control, el valor inicial de la variable de control, el incremento con base en el 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. 155) especifica en su encabezado todos los detalles sobre la repetición controlada por contador.
- Cuando la instrucción `for` comienza a ejecutarse, se declara y se inicializa su variable de control. Si al principio la condición es verdadera, el cuerpo se ejecuta. 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 mismo.

- El formato general de la instrucción `for` es

```
for (inicialización; condición;DeContinuacionDeCiclo; 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 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. Si el programa inicializa la variable de control antes del ciclo, podríamos omitir la expresión *inicialización*; de igual forma, si el programa calcula el incremento con instrucciones en el cuerpo del ciclo, o si no se necesita un incremento, podríamos omitir la expresión *incremento*.
- La expresión de incremento en un `for` actúa como si fuera una instrucción independiente al final del cuerpo de `for`.
- Una instrucción `for` puede contar en forma descendente mediante el uso de un incremento negativo; es decir, un decremento (pág. 158).
- Si al principio la condición de continuación de ciclo es `false`, el programa no ejecuta el cuerpo de la instrucción `for`.

Sección 5.4 Ejemplos sobre el uso de la instrucción `for`

- Java trata a las constantes de punto flotante (por ejemplo, `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. 161) de 4; es decir, `printf` muestra el valor con al menos 4 posiciones de caracteres. Si el valor a imprimir tiene menos de 4 posiciones de caracteres de ancho, éste se justifica de manera predeterminada a la derecha del campo (pág. 161). Si el valor tiene más de 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. 161), use un entero negativo que especifique la anchura del campo.
- `Math.pow(x, y)` (pág. 162) calcula el valor de *x* elevado a la *y*ésima potencia. El método recibe dos argumentos `double` y devuelve un valor `double`.
- La bandera de formato coma (`,`) (pág. 162) en un especificador de formato indica que un valor de punto flotante debe imprimirse con un separador de agrupamiento (pág. 162). 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, así como 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. 163) es similar a la instrucción `while`. En la instrucción `while`, el programa evalúa la condición de continuación de ciclo al principio del mismo, 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. 165) 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`) o de un objeto `String`.
- El indicador de fin de archivo 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 independiente. 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. 168) determina si hay más datos por 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).
- En un `switch`, el programa evalúa la expresión de control 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` independiente.
- Cada `case` puede tener varias instrucciones, y no es necesario colocarlas entre llaves.
- Por lo general las instrucciones de un `case` terminan con una instrucción `break` (pág. 168) que termina la ejecución del `switch`.
- Sin las instrucciones `break`, cada vez que ocurre una coincidencia en `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 de `switch`.

Sección 5.7 Ejemplo práctico de la clase `PolizaAuto`*: objetos* `String` *en instrucciones* `switch`

- Es posible usar objetos `String` en la expresión de control de una instrucción `switch` y las etiquetas `case`.

Sección 5.8 Instrucciones `break` *y* `continue`

- Cuando la instrucción `break` se ejecuta en una instrucción `while`, `for`, `do...while` o `switch`, provoca la salida inmediata de esa instrucción.
- Cuando la instrucción `continue` (pág. 174) 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 de inmediato la prueba de continuación de ciclo. En una instrucción `for`, se ejecuta la expresión de incremento y después el programa evalúa la prueba de continuación de ciclo.

Sección 5.9 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. 176) 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. 176) utiliza la evaluación de corto circuito (pág. 178); se evalúan sólo hasta que se conoce si la condición es verdadera o falsa.
- Los operadores `&` y `|` (pág. 178) 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. 179) 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. 179) "invierte" el valor de una condición.

Ejercicios de autoevaluación

5.1 Complete los espacios en blanco en cada uno de 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 _____ de ejecutar el cuerpo del mismo; 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, o un `String`.

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 mismo.

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 métodos que realizan tareas comunes y no requieren objetos se llaman métodos _____.

5.2 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 instrucciones.

5.3 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 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. 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 por omisión, 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('\t');

  ++i;
}
```

d)
```
for (i = 1; i <= 20; i++)
{
  System.out.print(i);

  if (i % 5 == 0)
    System.out.println();
  else
    System.out.print('\t');
}
```

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 elimine 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 de `case 2:` se ejecute siempre que lo haga la instrucción de `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 los errores 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       }
13    }
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 la visualización de 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 el mismo 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

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} + \frac{4}{7} + \frac{4}{9} + \frac{4}{11} + \ldots$$

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 ***(Ternas pitagóricas)*** Un triángulo rectángulo puede tener lados cuyas longitudes sean valores enteros. El conjunto de tres valores enteros para las longitudes de los lados de un triángulo rectángulo se conoce como terna pitagórica. 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 las ternas pitagóricas 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 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 5.13.

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 5.14.

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" (The Twelve Days of Christmas). Una instrucción `switch` debe utilizarse para imprimir el día (es decir, "first", "second", etcétera). Una instrucción `switch` separada debe utilizarse para imprimir el resto de cada verso. Visite el sitio Web `en.wikipedia.org/wiki/The_Twelve_Days_of_Christmas_(song)` para obtener la letra completa de la canción.

5.30 ***(Clase `PolizaAuto` modificada)*** Modifique la clase `PolizaAuto` de la figura 5.11 para validar los códigos de estado de dos letras para los estados del noreste. Los códigos son: CT para Connecticut, MA para Massachusetts, ME para Maine, NH para Nuevo Hampshire, NJ para Nueva Jersey, NY para Nueva York, PA para Pensilvania y VT para Vermont. En el método `establecerEstado` de `PolizaAuto`, use el operador OR lógico (`||`) (sección 5.9) para crear una condición compuesta en una instrucción `if...else` que compare el argumento del método con cada código de dos letras. Si el código es incorrecto, la parte `else` de la instrucción `if...else` debe mostrar un mensaje de error. En capítulos posteriores aprenderá a usar el manejo de excepciones para indicar que un método recibió un valor que no es válido.

Marcando la diferencia

5.31 ***(Examen rápido sobre hechos del calentamiento global)*** La controversial cuestión del calentamiento global obtuvo una gran publicidad gracias a la película "Una verdad incómoda" (An Inconvenient Truth) en la que aparece el anterior vicepresidente Al Gore. El señor Gore y una red de científicos de Naciones Unidas, 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 en línea *ambos* lados de la cuestión del calentamiento global (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 ambas posturas sobre el tema. 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.

5.32 ***(Alternativas para el plan fiscal: el "impuesto justo")*** Existen muchas propuestas para que los impuestos sean más justos. Consulte la iniciativa FairTax de Estados Unidos en el sitio: `www.fairtax.org`. 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.

5.33 ***(Crecimiento de la base de usuarios de Facebook)*** De acuerdo con CNNMoney.com, Facebook llegó a los mil millones de usuarios en octubre de 2012. Use la técnica del cálculo del crecimiento compuesto que aprendió en la figura 5.6 y, suponiendo que su base de usuarios crezca con una tasa del 4% mensual, ¿cuántos meses tardará Facebook en aumentar su base de usuarios a mil quinientos millones? ¿Cuántos meses tardará Facebook en aumentar su base de usuarios a dos mil millones?

6

Métodos: un análisis más detallado



La forma siempre sigue a la función.
—Louis Henri Sullivan

E pluribus unum.
(Uno compuesto de muchos).
—Virgilio

¡Oh! volvió a llamar ayer, ofreciéndome volver.
—William Shakespeare

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 asociar los métodos y los campos `static` con las clases, en vez de los objetos.
- Cómo se soporta el mecanismo de llamada/retorno de los métodos mediante la pila de llamadas a métodos.
- A usar la promoción y conversión de argumentos.
- Cómo los paquetes agrupan las clases relacionadas.
- A utilizar la generación segura de números aleatorios para implementar aplicaciones para juegos.
- Cómo se limita la visibilidad de las declaraciones a regiones específicas de los programas.
- Acerca de la sobrecarga de métodos y cómo crear métodos sobrecargados.

Plan general

6.1 Introducción

6.2 Módulos de programas en Java

6.3 Métodos `static`, campos `static` y la clase `Math`

6.4 Declaración de métodos con múltiples parámetros

6.5 Notas sobre cómo declarar y utilizar los métodos

6.6 La pila de llamadas a los métodos y los marcos de pila

6.7 Promoción y conversión de argumentos

6.8 Paquetes de la API de Java

6.9 Ejemplo práctico: generación de números aleatorios seguros

6.10 Ejemplo práctico: un juego de probabilidad; introducción a los tipos `enum`

6.11 Alcance de las declaraciones

6.12 Sobrecarga de métodos

6.13 (Opcional) Ejemplo práctico de GUI y gráficos: colores y figuras rellenas

6.14 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Marcando la diferencia*

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.

Aprenderá más sobre los métodos `static`, que pueden llamarse sin necesidad de que exista un objeto de la clase a la que pertenecen. También 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 punto del libro. Además, aprenderá a declarar 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, continuaremos nuestra explicación sobre los métodos. La recursividad ofrece una forma interesante de ver a 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 la sección 21.4.10 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 más.



Observación de ingeniería de software 6.1

Procure familiarizarse con la vasta colección de clases y métodos que proporciona la API de Java (`http://docs.oracle.com/javase/7/docs/api/`). En la sección 6.8 presentaremos las generalidades sobre varios paquetes comunes. En el apéndice F, en línea, 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.

Dividir y vencer con clases y métodos

Las clases y los métodos nos ayudan a dividir un programa en módulos, por medio de la separación de sus tareas en unidades autónomas. 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 usando 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** (al 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.

Relación jerárquica entre llamadas a métodos

Como sabe, un método se invoca mediante una llamada, y cuando el método que se llamó completa su tarea, devuelve el control, y posiblemente un resultado, 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`.



Tip para prevenir errores 6.2

Cuando llame a un método que devuelva un valor que indique si el método realizó correctamente su tarea, asegúrese de comprobar el valor de retorno de ese método y, si no tuvo éxito, de lidiar con el problema de manera apropiada.

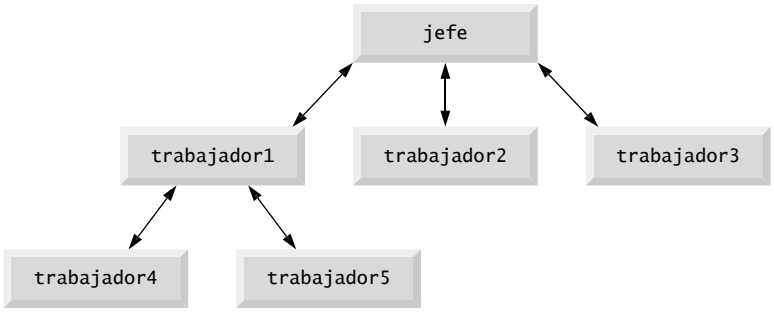


Fig. 6.1 | Relación jerárquica entre el método jefe y los métodos trabajadores.

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 los métodos `static` de la clase 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.nombreMetodo(argumentos)
```

Métodos de la clase `Math`

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`; es decir, `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`.

Variables `static`

En la sección 3.2 vimos que cada objeto de una clase mantiene su *propia* copia de cada variable de instancia de la clase. Hay variables para las que cada objeto de una clase *no* necesita su propia copia independiente (como veremos en breve). Dichas variables se declaran como `static` y también se conocen como **variables de clase**. Cuando se crean los objetos de una clase que contiene variables `static`, todos los objetos de esa clase comparten *una* copia de esas variables. En conjunto, las variables `static` y las variables de instancia de una clase se conocen como sus **campos**. En la sección 8.11 aprenderá más sobre los campos `static`.

Constantes `PI` *y* `E` *de la clase* `Math`

La clase `Math` declara dos constantes: `Math.PI` y `Math.E`, las cuales representan *aproximaciones de alta precisión* de las constantes matemáticas de uso común. La constante `Math.PI` (3.141592653589793) es la propor-

ción de la circunferencia de un círculo con respecto a 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`). Estas constantes se declaran en la clase `Math` con los modificadores `public`, `final` y `static`. Al hacerlas `public`, usted puede utilizarlas en sus propias clases. Cualquier campo declarado con la palabra clave `final` es *constante*, por lo que su valor no puede modificarse después de inicializar el campo. 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`.

***¿Por qué el método* `main` se declara como `static`?**
Cuando se ejecuta la máquina virtual de Java (JVM) con el comando `java`, ésta trata de invocar al método `main` de la clase que usted le especifica; en este punto 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 *múltiples* parámetros.

La figura 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
```

Fig. 6.3 | Método `maximo`, declarado por el programador, con tres parámetros `double` (parte 1 de 2).

```
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: ");
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       }
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       }
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, con 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"; es decir, que 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 `BuscadorMaximo`. 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 que 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 booleanos `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` contengan 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 14 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 sobre 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 16 de la figura 3.2, `miCuenta.obtenerNombre()`, que llama a un método de la clase `Cuenta` desde el método `main` de `PruebaCuenta`. (Por lo general a los métodos que no son `static` se les conoce como **métodos de instancia**).

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` puede llamar directamente a otros métodos `static` de la misma clase (es decir, mediante el nombre del método por sí solo) y puede manipular de manera directa variables `static` en la misma clase. Para acceder a las variables de instancia y los métodos de instancia de la clase, un método `static` debe usar una referencia a un objeto de esa clase. Los métodos de instancia pueden acceder a todos los campos (variables `static` y variables de instancia) y métodos de la clase.

Recuerde que los métodos `static` se relacionan con una clase como un todo, mientras que los métodos de instancia 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 de instancia en forma directa. ¿Cómo sabría el método `static` 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 de instancia? Por lo tanto, Java *no* permite que un método `static` acceda de manera directa a las variables de instancia y los métodos de instancia 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

Volver a declarar un parámetro como una variable local en el cuerpo del método es un error de compilación.



Error común de programación 6.6

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 marcos de pila

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 una pila, 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, normalmente 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 llamadas a 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 llamadas a métodos también contiene la memoria para las *variables locales* (incluyendo los parámetros de los métodos) 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 llamadas a métodos, se conocen como el **marco de pila** (o **registro de activación**) de la llamada a un método. Cuando se hace la llamada a un método, el marco de pila para la llamada a ese método se *mete* en la pila de llamadas a métodos. Cuando el método regresa al que lo llamó, el marco de pila 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, entonces, cuando se saca de la pila el marco de pila 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*, de lo cual hablaremos en la sección 8.10.

Desde luego que la cantidad de memoria en una computadora es finita, por lo que sólo puede utilizarse cierta cantidad para almacenar los marcos de pila en la pila de llamadas a métodos. 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**. Hablaremos más sobre esto en el capítulo 11, Manejo de excepciones: un análisis más detallado.

6.7 Promoción y conversión de argumentos

Otra característica importante de las llamadas a los métodos es la **promoción de argumentos**; es decir, 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` o de `int` a `short`) puede también producir valores modificados.

Las reglas de promoción se aplican a las expresiones que contienen valores de dos o más tipos primitivos, así como 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 un *operador de conversión* (el cual presentamos en la sección 4.10) 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 el valor de `valorDouble` a un entero temporal, 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 6.7

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.

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.

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. Una de las principales fortalezas de Java se debe a las miles de clases de la API. Algunos paquetes clave de la API de Java que usamos en este libro se describen en la figura 6.5; éstos representan sólo una pequeña parte de los *componentes reutilizables* en la API de Java.

Paquete	Descripción
`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 12, Componentes de la GUI, parte 1 y el capítulo 22, Componentes de la GUI, parte 2).
`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 13, Gráficos y Java 2D).
`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 15, Archivos, flujos y serialización de objetos).
`java.lang`	El **Paquete del Lenguaje Java** contiene clases e interfaces (descritas a lo largo del libro) 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 28, Redes).
`java.security`	El **Paquete de Seguridad de Java** contiene clases e interfaces para mejorar la seguridad de las aplicaciones.
`java.sql`	El **Paquete JDBC** contiene clases e interfaces para trabajar con bases de datos (vea el capítulo 24, Acceso a bases de datos con JDBC).
`java.util`	El **Paquete de Utilerías de Java** contiene clases e interfaces utilitarias, que permiten el almacenamiento y procesamiento de grandes cantidades de datos. Muchas de estas clases e interfaces se actualizaron para dar soporte a las nuevas capacidades lambda de Java SE 8 (vea el capítulo 16, Colecciones de genéricos).
`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 23, Concurrencia).
`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. Este paquete aún utiliza algunos elementos del paquete `java.awt` antiguo (vea el capítulo 12, Componentes de la GUI, parte 1 y el capítulo 22, Componentes de la GUI, parte 2).
`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 12, Componentes de la GUI, parte 1 y el capítulo 22, Componentes de la GUI, parte 2).
`java.xml.ws`	El **Paquete JAX-WS** contiene clases e interfaces para trabajar con los servicios Web en Java (vea el capítulo 32, servicios Web basados en REST).
Paquetes de `javafx`	JavaFX es la tecnología de GUI preferida para el futuro. Hablaremos sobre estos paquetes en el capítulo 25, GUI de JavaFX, parte 1 y en los capítulos en línea sobre la GUI de JavaFX y multimedia.

Fig. 6.5 | Paquetes de la API de Java (un subconjunto) (parte 1 de 2).

Paquete	Descripción
Algunos paquetes de Java SE 8 que se utilizan en este libro	
`java.time`	El nuevo **Paquete de la API de Fecha/Hora** de Java SE 8 contiene clases e interfaces para trabajar con fechas y horas. Estas características están diseñadas para reemplazar las herramientas de fecha y hora anteriores del paquete `java.util` (vea el capítulo 23, Concurrencia).
`java.util.function` y `java.util.stream`	Estos paquetes contienen clases e interfaces para trabajar con las herramientas de programación funcionales de Java SE 8 (vea el capítulo 17, Lambdas y flujos de Java SE 8).

Fig. 6.5 | Paquetes de la API de Java (un subconjunto) (parte 2 de 2).

El conjunto de paquetes disponibles en Java es bastante extenso. Además de los que se resumen en la figura 6.5, Java 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, visite:

```
http://docs.oracle.com/javase/7/docs/api/overview-summary.html
http://download.java.net/jdk8/docs/api/
```

Puede localizar información adicional acerca de los métodos de una clase predefinida de Java en la documentación para la API de Java, en

```
http://docs.oracle.com/javase/7/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 de su tipo de valor de retorno.

6.9 Ejemplo práctico: generación de números aleatorios seguros

Ahora analizaremos de manera breve una parte divertida de las aplicaciones de la programación: la simulación y los 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.

El **elemento de azar** puede introducirse en un programa mediante un objeto de la clase `SecureRandom` (paquete `java.security`). Dichos objetos pueden producir valores aleatorios de tipo `boolean`, `byte`, `float`, `double`, `int`, `long` y Gaussian. En los siguientes ejemplos, usaremos objetos de la clase `SecureRandom` para producir valores aleatorios.

Cambiar a números aleatorios seguros

Las ediciones recientes de este libro utilizaron la clase `Random` de Java para obtener valores "aleatorios". Esta clase producía valores *determinísticos* que los programadores malintencionados podían *predecir*. Los objetos `SecureRandom` producen **números aleatorios no determinísticos** que *no pueden* predecirse.

Los números aleatorios determinísticos han sido la fuente de muchas fugas de seguridad de software. La mayoría de los lenguajes de programación cuentan ahora con herramientas en sus bibliotecas similares

a la clase `SecureRandom` de Java para producir números aleatorios no determinísticos y ayudar a prevenir dichos problemas. De aquí en adelante en el libro, cuando hagamos referencia a los "números aleatorios", estaremos hablando de los "números aleatorios seguros".

Creación de un objeto `SecureRandom`

Es posible crear un nuevo objeto generador de números aleatorios seguros de la siguiente manera:

```
SecureRandom numerosAleatorios = new SecureRandom();
```

Después, este objeto puede usarse para generar valores aleatorios; aquí sólo hablaremos sobre los valores `int` aleatorios. Para obtener más información sobre la clase `SecureRandom`, vaya a `docs.oracle.com/javase/7/docs/api/java/security/SecureRandom.html`.

Obtener un valor `int` aleatorio

Considere la siguiente instrucción:

```
int valorAleatorio = numerosAleatorios.nextInt();
```

El método **`nextInt`** de la clase `SecureRandom` genera un valor `int`. Si de verdad produce valores *aleatorios*, entonces cualquier valor en ese rango debería tener una *oportunidad igual* (o probabilidad) de ser elegido cada vez que se llame al método `nextInt`.

Cambiar el rango de valores producidos por `nextInt`

El rango de valores producidos por el método `nextInt` es por lo general distinto del rango de valores requeridos en una aplicación particular de Java. Por ejemplo, un programa que simula el lanzamiento de una moneda sólo requiere 0 para "cara" y 1 para "cruz". 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 `SecureRandom` 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:

```
int 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 `SecureRandom`.

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

```
int cara = 1 + numerosAleatorios.nextInt(6);
```

El valor de desplazamiento (`1`) especifica el *primer* valor en el rango deseado de enteros aleatorios. La instrucción anterior asigna a `cara` un entero aleatorio en el rango de 1 a 6.

Tirar un dado de seis lados 20 veces

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 `SecureRandom` del paquete `java.security`. La línea 10 crea el objeto `numerosAleatorios` de la clase `SecureRandom` 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 para crear un formato ordenado de cinco columnas.

```
1  // Fig. 6.6: EnterosAleatorios.java
2  // Enteros aleatorios desplazados y escalados.
3  impot java.security.SecureRandom; // el programa usa la clase SecureRandom
4
5  public class EnterosAleatorios
6  {
7     public static void main(String[] args)
8     {
9        // El objeto numerosAleatorios producirá números aleatorios seguros
10       SecureRandom numerosAleatorios = new SecureRandom();
11
12       // itera 20 veces
13       for (int contador = 1; contador <= 20; contador++)
14       {
15          // elige entero aleatorio del 1 al 6
16          int 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       }
24    }
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.

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 de 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 de veces.

```
1  // Fig. 6.7: TirarDado.java
2  // Tirar un dado de seis lados 6,000,000 veces.
3  import java.security.SecureRandom;
4
5  public class TirarDado
6  {
7     public static void main(String[] args)
8     {
9        // el objeto numerosAleatorios producirá números aleatorios seguros
10       SecureRandom numerosAleatorios = new SecureRandom();
11
12       int frecuencia1 = 0; // cuenta las veces que se tiró 1
13       int frecuencia2 = 0; // cuenta las veces que se tiró 2
14       int frecuencia3 = 0; // cuenta las veces que se tiró 3
15       int frecuencia4 = 0; // cuenta las veces que se tiró 4
16       int frecuencia5 = 0; // cuenta las veces que se tiró 5
17       int frecuencia6 = 0; // cuenta las veces que se tiró 6
18
19       // sintetiza los resultados de tirar un dado 6,000,000 veces
20       for (int tiro = 1; tiro <= 6000000; tiro++)
21       {
22          int cara = 1 + numerosAleatorios.nextInt(6); // número del 1 al 6
23
24          // usa el valor de cara de 1 a 6 para determinar qué contador incrementar
25          switch (cara)
26          {
27             case 1:
28                ++frecuencia1; // incrementa el contador de 1s
29                break;
30             case 2:
31                ++frecuencia2; // incrementa el contador de 2s
32                break;
33             case 3:
34                ++frecuencia3; // incrementa el contador de 3s
35                break;
36             case 4:
37                ++frecuencia4; // incrementa el contador de 4s
38                break;
39             case 5:
40                ++frecuencia5; // incrementa el contador de 5s
41                break;
42             case 6:
43                ++frecuencia6; // incrementa el contador de 6s
44                break;
45          }
46       }
47
48       System.out.println("Cara\tFrecuencia"); // encabezados de salida
49       System.out.printf("1\t%d%n2\t%d%n3\t%d%n4\t%d%n5\t%d%n6\t%d%n",
50          frecuencia1, frecuencia2, frecuencia3, frecuencia4,
51          frecuencia5, frecuencia6);
52    }
53 } // fin de la clase TirarDado
```

Fig. 6.7 | Tirar un dado de seis lados 6,000,000 veces (parte 1 de 2).

```
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 resultados de ejemplo, al escalar y desplazar los valores producidos por el método `nextInt`, el programa puede simular 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 20 a 46) itera 6,000,000 de veces. Durante cada iteración, la línea 22 produce un valor aleatorio del 1 al 6. Después, ese valor se utiliza como la expresión de control (línea 25) de la instrucción `switch` (líneas 25 a 45). Con base en el valor de `cara`, la instrucción `switch` incrementa una de las seis variables de contadores durante cada iteración del ciclo. 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 22. Ejecute el programa y observe los resultados. Como verá, cada vez que ejecute el programa, éste producirá *distintos* resultados.

Cuando estudiemos los arreglos en el capítulo 7, le mostraremos una forma elegante de reemplazar toda la instrucción `switch` de este programa con *una sola* instrucción. Luego, cuando estudiemos las nuevas capacidades de programación funcional de Java SE 8 en el capítulo 17, ¡le mostraremos cómo reemplazar el ciclo que tira el dado, la instrucción `switch` *y* la instrucción que muestra los resultados con *una sola* instrucción!

Escalamiento y desplazamiento generalizados de números aleatorios

Anteriormente simulamos el tiro de un dado de seis caras con la instrucción

```
int 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 `SecureRandom`, y el número inicial del rango es el número `1` que se suma a `numerosAleatorios.nextInt(6)`. Podemos generalizar este resultado de la siguiente manera:

```
int 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:

```
int 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:

```
int 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.

Una observación sobre el rendimiento

Usar `SecureRandom` en vez de `Random` para lograr mayores niveles de seguridad incurre en un considerable castigo para el rendimiento. Para las aplicaciones "casuales", tal vez sea más conveniente usar la clase `Random` del paquete `java.util`: simplemente reemplace `SecureRandom` con `Random`.

6.10 Ejemplo práctico: un juego de probabilidad; introducción a los tipos enum

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. 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.security.SecureRandom;
4
5  public class Craps
6  {
7     // crea un generador de números aleatorios seguros para usarlo en el método
         tirarDado
8     private static final SecureRandom numerosAleatorios = new SecureRandom();
```

Fig. 6.8 | La clase `Craps` simula el juego de dados "craps" (parte 1 de 3).

```
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;
45         }
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;
55            else
56               if (sumaDeDados == SIETE) // pierde al tirar 7 antes del punto
57                  estadoJuego = Estado.PERDIO;
58         }
59
60         // muestra mensaje de que ganó o perdió
61         if (estadoJuego == Estado.GANO)
```

Fig. 6.8 | La clase `Craps` simula el juego de dados "craps" (parte 2 de 3).

```
62            System.out.println("El jugador gana");
63         else
64            System.out.println("El jugador pierde");
65      }
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;
81      }
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` (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 `SecureRandom` que se reutiliza en cada llamada a `tirarDados`. Si hubiera un programa con múltiples instancias de la clase `Craps`, todos compartirían este objeto `SecureRandom`.

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, también usa 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` (incluyendo el caso `default`); por lo tanto, se garantiza que se inicialice antes de usarse, así que no necesitamos inicializarlo en la línea 24.

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` se conoce como un **tipo `enum`** (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 las clases, las llaves delimitan el cuerpo de una declaración de `enum`. Dentro de las llaves hay una lista separada por comas de **constantes `enum`**, 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 los tipos `enum`.



Buena práctica de programación 6.1

Use sólo letras mayúsculas en los nombres de las constantes `enum` para que resalten y nos recuerden que 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, el programa 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 `enum` (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 los valores del primer dado, del segundo dado y de su suma, 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. 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`), así como 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 `enum`. 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 (también conocida como ámbito) 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 presentaremos 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 es todo el cuerpo de la clase. Esto permite a los métodos de instancia 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**). Para acceder a un campo oculto en un bloque:

- Si el campo es una variable de instancia, coloque antes de su nombre la palabra clave `this` y un punto (`.`), como en `this.x`.
- Si el campo es una variable de clase `static`, coloque antes de su nombre el nombre de la clase y un punto (`.`), como en *NombreClase*`.x`.

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), luego 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 por segunda vez a `usarVariableLocal` (línea 19), ésta *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
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
```

Fig. 6.9 | La clase `Alcance` demuestra los alcances de los campos y las variables locales (parte 1 de 2).

```
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       }
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       }
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       }
46    } // fin de la clase Alcance
```

```
la x local en main 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 main 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 tiene su valor modificado (`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 `main`, ya que todos los métodos hicieron referencia a las variables llamadas `x` en otros alcances.

Principio del menor privilegio
En un sentido general, las "cosas" deben tener las capacidades que necesitan para realizar su trabajo, pero nada más. Un ejemplo es el alcance de una variable. Ésta no debe ser visible cuando no se necesite.



Buena práctica de programación 6.3
Declare las variables lo más cerca posible de donde se vayan a usar la primera vez.

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 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, así como 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  {
```

Fig. 6.10 | Declaraciones de métodos sobrecargados (parte 1 de 2).

```
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     }
12
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     }
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     }
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. Si la precisión *no* se especifica en el especificador de formato, los valores de punto flotante se muestran de manera predeterminada con seis dígitos de precisión.

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, así como del *número*, los *tipos* y el *orden* de sus parámetros, aunque *no* de su tipo de valor de retorno. Si el compilador sólo se fijara en los nombres de los métodos durante la compilación, el código de la figura 6.10 sería ambiguo, ya que 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 (internamente) el nombre lógico "`cuadrado` de `int`" para el método `cuadrado` que especifica un parámetro `int`, y "`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 sólo 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, el compilador genera un mensaje de error para indicar que el método ya está definido en la clase. Los métodos sobrecargados *pueden* tener *distintos* tipos de valor de retorno si tienen *distintas* listas de parámetros. Además, los métodos sobrecargados *no* necesitan tener el mismo número de parámetros.



Error común de programación 6.8

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) Ejemplo práctico de GUI y gráficos: colores y figuras rellenas

Aunque podemos crear muchos diseños interesantes sólo con líneas y figuras básicas, la clase `Graphics` ofrece muchas posibilidades 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`** (paquete `java.awt`) para representar colores mediante sus valores RGB. Por conveniencia, la clase `Color` contiene varios objetos `static Color` predefinidos: `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE` y `YELLOW`. Para acceder a estos objetos predefinidos, se utiliza el nombre de la clase y un punto (`.`), como en `Color.RED`. Puede crear colores personalizados pasando los valores de los componentes rojo, verde y azul al constructor de la clase `Color`:

```
public Color(int r, int g, int b)
```

Los métodos **`fillRect`** y **`fillOval`** de `Graphics` dibujan rectángulos y óvalos rellenos, respectivamente. 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 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  // Dibuja una cara sonriente usando colores y 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    }
30 } // fin de la clase DibujarCaraSonriente
```

Fig. 6.11 | Cómo dibujar una cara sonriente, con colores y figuras rellenas.

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, que es 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 mezclará 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    }
17 } // fin de la clase PruebaDibujarCaraSonriente
```

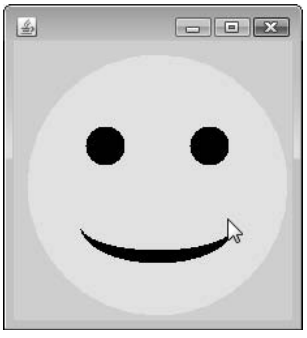


Fig. 6.12 | Aplicación de prueba que muestra una cara sonriente.

Ejercicios del ejemplo 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.

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.

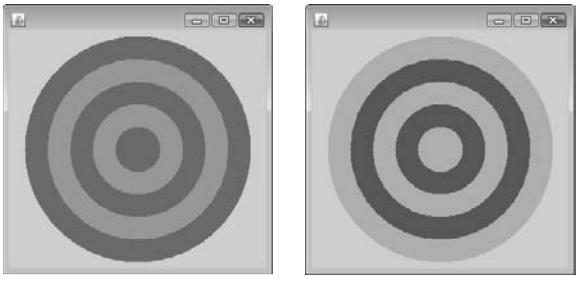


Fig. 6.13 | Un tiro al blanco con dos colores alternantes al azar.

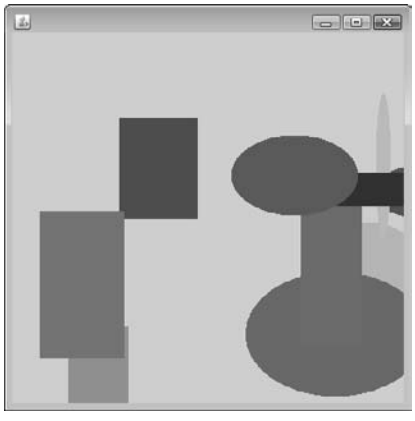


Fig. 6.14 | Figuras generadas al azar.

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 marcos de pila 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 `SecureRandom` para generar números aleatorios, que pueden usarse para simulaciones. También aprendió sobre el 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 de veces. Le presentaremos dos versiones mejoradas de nuestro ejemplo práctico `LibroCalificaciones` que almacena conjuntos de calificaciones de estudiantes en un objeto `LibroCalificaciones`. 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. 201).

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 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. 202), 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. 204; 3.141592653589793) es la relación entre la circunferencia de un círculo y su diámetro. La constante `Math.E` (pág. 204; 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. 205) es constante, lo que significa que su valor no se puede modificar una vez que se inicializa. 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. 204) 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. 205), 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. 208) 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 volver a ensamblar las literales `String` mediante la concatenación.

Sección 6.5 Notas sobre 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 expresión;
  ```

  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 métodos y los marcos de pila

- Las pilas (pág. 209) se conocen como estructuras de datos tipo "último en entrar, primero en salir (UEPS)"; es decir, 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, cuando se llama al método, la dirección de retorno del método que hace la llamada se mete en la pila de llamadas a métodos. 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. 210) 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 marco de pila de la llamada al método, o registro de activación. Cuando se hace una llamada a un método, el marco de pila para ese método se mete en la pila de llamadas a métodos. Cuando el método regresa al que lo llamó, su llamada en el marco de pila 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 puedan almacenar sus marcos de pila en la pila de llamadas a métodos, se produce un error conocido como desbordamiento de pila (pág. 210). 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. 210) convierte el valor de un argumento al tipo que el método espera recibir en su parámetro correspondiente.
- Las reglas de promoción (pág. 210) 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 Ejemplo práctico: generación de números aleatorios seguros

- Los objetos de la clase `SecureRandom` (paquete `java.security`; pág. 213) pueden producir valores aleatorios no determinísticos.
- El método `nextInt` de `SecureRandom` (pág. 214) genera un valor `int` aleatorio.
- La clase `SecureRandom` 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. 214) pueden generarse mediante

```
int 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

```
int 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.

Sección 6.10 Ejemplo práctico: un juego de probabilidad; introducción a los tipos `enum`

- Una enumeración (pág. 221) 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 `enum`, 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`. Por convención, dichas constantes se declaran todas con letras mayúsculas, para hacer que resalten en el programa.

Sección 6.11 Alcance de las declaraciones

- El alcance (pág. 222) 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. 223) hasta que el bloque termina de ejecutarse.

Sección 6.12 Sobrecarga de métodos

- Java permite métodos sobrecargados (pág. 225) 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. 226), que son combinaciones de los nombres de los métodos así como el número, los tipos y el orden de sus parámetros, pero no sus tipos de valores de retorno.

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 _____, en las que 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 realmente aleatorios.
i) La pila de llamadas a métodos 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 llamadas a métodos, 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 llamadas a métodos, se produce un error conocido como _____.
k) El _____ de una declaración es la porción del programa que puede hacer referencia por su nombre a la entidad en la declaración.
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.

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 (`float`).

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 Declare el método `volumenEsfera` para calcular y mostrar el volumen de la esfera. Utilice la siguiente instrucción para calcular el volumen:

```
double volumen = (4.0 / 3.0) * Math.PI * Math.pow(radio, 3)
```

Escriba una aplicación en Java que pida al usuario el radio `double` de una esfera, que llame a `volumenEsfera` para calcular el volumen y muestre el resultado en pantalla.

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) `SecureRandom`. i) marco de pila, registro de activación. j) desbordamiento de pila. k) alcance. l) sobrecarga de 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  public class PruebaMath
4  {
5     public static void main(String[] args)
6     {
7        System.out.printf("Math.abs(23.7) = %f%n", Math.abs(23.7));
8        System.out.printf("Math.abs(0.0) = %f%n", Math.abs(0.0));
```

```
      System.out.printf("Math.abs( -23.7) = %f%n", Math.abs(-23.7));
      System.out.printf("Math.ceil(9.2) = %f%n", Math.ceil(9.2));
      System.out.printf("Math.ceil(-9.8) = %f%n", Math.ceil(-9.8));
      System.out.printf("Math.cos( 0.0 ) = %f%n", Math.cos(0.0));
      System.out.printf("Math.exp( 1.0 ) = %f%n", Math.exp(1.0));
      System.out.printf("Math.exp(2.0) = %f%n", Math.exp(2.0));
      System.out.printf("Math.floor(9.2) = %f%n", Math.floor(9.2));
      System.out.printf("Math.floor(-9.8) =%f%n", Math.floor(-9.8));
      System.out.printf("Math.log(Math.E) = %f%n", Math.log(Math.E));
      System.out.printf("Math.log(Math.E * Math.E) = %f\n",
        Math.log(Math.E * Math.E));
      System.out.printf("Math.max(2.3, 12.7) = %f%n", Math.max(2.3, 12.7));
      System.out.printf("Math.max(-2.3, -12.7) = %f%n",
        Math.max(-2.3, -12.7));
      System.out.printf("Math.min(2.3, 12.7) = %f%n", Math.min(2.3, 12.7));
      System.out.printf("Math.min(-2.3, -12.7) = %f%n",
        Math.min(-2.3, -12.7));
      System.out.printf("Math.pow(2.0, 7.0) = %f%n", Math.pow(2.0, 7.0));
      System.out.printf("Math.pow(9.0, 0.5) = %f%n", Math.pow(9.0, 0.5));
      System.out.printf("Math.sin(0.0) = %f%n", Math.sin(0.0));
      System.out.printf("Math.sqrt(900.0) = %f%n", Math.sqrt(900.0));
      System.out.printf("Math.tan(0.0) = %f%n", Math.tan(0.0));
   } // fin de main
} // 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 intAFloat(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 de main
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 seguidas. 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 a lugares decimales específicos, 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 el método `nextInt` de la clase `SecureRandom` es útil para simular juegos al azar?
c) ¿Por qué es a menudo necesario escalar o desplazar los valores producidos por un objeto `SecureRandom`?
d) ¿Por qué la simulación computarizada de las situaciones reales es 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 `34` (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 si el segundo número de un par de enteros 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 los incisos (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 los incisos (a) y (b) para escribir una aplicación que permita al usuario, ya sea escribir una temperatura en grados Fahrenheit y mostrar su equivalente en Centígrados, o escribir una temperatura en grados Centígrados y mostrar su equivalente 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 propio número), 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. Muestre en pantalla 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 positivo 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 y muestre en pantalla 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 independiente, 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, para lo cual selecciona un entero aleatorio en el rango de 1 a 1000. La aplicación muestra el indicador `Adivine un número entre 1 y 1000`. 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, 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 menor de 10, imprima el mensaje `¡O ya sabía usted el secreto, o tuvo suerte!` Si el jugador adivina el número en 10 intentos, muestre en pantalla el mensaje `¡Aja! ¡Sabía usted el secreto!` Si el jugador hace más de 10 intentos, muestre en pantalla `¡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 quedó 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, arriésguese!`", 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 J primero.

Marcando 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 a un estudiante de escuela primaria a que aprenda a multiplicar. Use un objeto `SecureRandom` 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 así!
```

Posibles contestaciones a una respuesta incorrecta:

```
No. Por favor intenta de nuevo.
Incorrecto. Intenta una vez más.
¡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 pide ayuda adicional a tu instructor`" y reinicie el programa, para que otro estudiante pueda probarlo. Si el porcentaje es del 75% o mayor, muestre el mensaje "`¡Felicidades, estás 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.

Arreglos y objetos `ArrayList`

7



Comienza en el principio… y continúa hasta que llegues al final; entonces detente.

—Lewis Carroll

Ir más allá es tan malo como no llegar.

—Confucio

Objetivos

En este capítulo aprenderá:

- Lo que son los arreglos.
- A utilizar arreglos para almacenar datos en, y obtenerlos de listas y tablas de valores.
- A declarar arreglos, inicializarlos y hacer referencia a elementos individuales de ellos.
- A iterar a través de los arreglos mediante la instrucción `for` mejorada.
- A pasar arreglos a los métodos.
- A declarar y manipular arreglos multidimensionales.
- A usar listas de argumentos de longitud variable.
- A leer los argumentos de línea de comandos en un programa.
- A crear una clase de libro de calificaciones para el instructor, orientada a objetos.
- A realizar manipulaciones comunes de arreglos con los métodos de la clase `Arrays`.
- A usar la clase `ArrayList` para manipular una estructura de datos tipo arreglo, cuyo tamaño es ajustable en forma dinámica.

Plan general

7.1 Introducción

7.2 Arreglos

7.3 Declaración y creación de arreglos

7.4 Ejemplos sobre el uso de los arreglos

- 7.4.1 Creación e inicialización de un arreglo
- 7.4.2 Uso de un inicializador de arreglos
- 7.4.3 Cálculo de los valores a almacenar en un arreglo
- 7.4.4 Suma de los elementos de un arreglo
- 7.4.5 Uso de gráficos de barra para mostrar en forma gráfica los datos de un arreglo
- 7.4.6 Uso de los elementos de un arreglo como contadores
- 7.4.7 Uso de arreglos para analizar los resultados de una encuesta

7.5 Manejo de excepciones: procesamiento de la respuesta incorrecta

- 7.5.1 La instrucción `try`
- 7.5.2 Ejecución del bloque `catch`
- 7.5.3 El método `toString` del parámetro de excepción

7.6 Ejemplo práctico: simulación para barajar y repartir cartas

7.7 Instrucción `for` mejorada

7.8 Paso de arreglos a los métodos

7.9 Comparación entre paso por valor y paso por referencia

7.10 Ejemplo práctico: la clase `LibroCalificaciones` que usa un arreglo para almacenar las calificaciones

7.11 Arreglos multidimensionales

7.12 Ejemplo práctico: la clase `LibroCalificaciones` que usa un arreglo bidimensional

7.13 Listas de argumentos de longitud variable

7.14 Uso de argumentos de línea de comandos

7.15 La clase `Arrays`

7.16 Introducción a las colecciones y la clase `ArrayList`

7.17 (Opcional) Ejemplo práctico de GUI y gráficos: cómo dibujar arcos

7.18 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Sección especial: construya su propia computadora* | *Marcando la diferencia*

7.1 Introducción

En este capítulo presentamos las **estructuras de datos**, que son colecciones de elementos de datos relacionados. Los objetos **arreglo** son estructuras de datos que consisten en elementos de datos relacionados, del mismo tipo. Los arreglos facilitan el procesamiento de grupos de valores relacionados. Los arreglos conservan la misma longitud una vez creados. En los capítulos 16 a 21 estudiaremos con detalle las estructuras de datos.

Después de hablar sobre cómo se declaran, crean e inicializan los arreglos, presentaremos una serie de ejemplos prácticos que demuestran varias manipulaciones comunes de los mismos. Presentaremos 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 ejemplo práctico 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 de 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. Crearemos dos versiones del ejemplo práctico `LibroCalificaciones` para un instructor, el cual usa arreglos para mantener *en memoria* conjuntos de calificaciones de estudiantes y para analizar las calificaciones de los mismos. Le mostraremos cómo usar *listas de argumentos de longitud variable* para crear métodos que se puedan llamar con números variables de argumentos, y explicaremos 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`, que forma parte del paquete `java.util`.

Aunque se utilizan con frecuencia, los arreglos tienen capacidades limitadas. Por ejemplo, si desea especificar el tamaño de un arreglo, y si desea modificarlo en tiempo de ejecución, debe hacerlo manualmente

mediante la creación de un nuevo arreglo. Al final de este capítulo le presentaremos una de las estructuras de datos prefabricadas de Java, proveniente de las *clases de colecciones* de la API de Java. Estas colecciones ofrecen mayores capacidades que los arreglos tradicionales; ya que 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** según sea necesario para poder alojar una cantidad mayor o menor de elementos.

Java SE 8
Después de leer el capítulo 17, Lambdas y flujos de Java SE 8, podrá volver a implementar muchos de los ejemplos del capítulo 7 de una manera más concisa y elegante, y en una forma que facilite la paralelización para mejorar el rendimiento en los sistemas multinúcleo actuales.

7.2 Arreglos

Un arreglo es un grupo de variables (llamadas **elementos** o **componentes**) que contienen valores 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 7.11). 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 dentro del mismo. El número de la posición del elemento se conoce formalmente como el **índice** o **subíndice** del elemento.

Representación de arreglos lógicos
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.

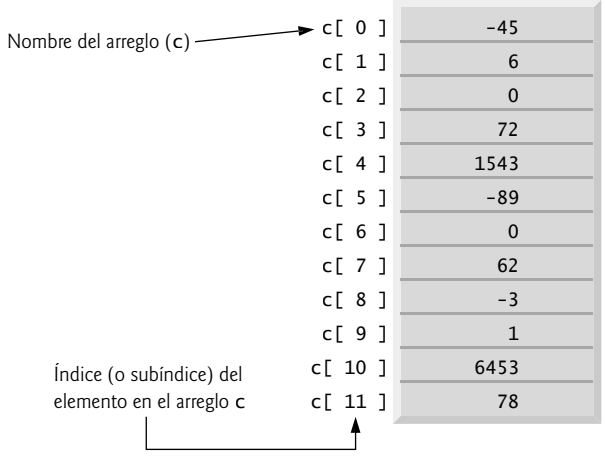


Fig. 7.1 | Un arreglo con 12 elementos.

Un índice debe ser un *entero no negativo*. 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`; por ejemplo, `byte`, `short` o `char`, pero no `long`. De lo contrario, ocurre un error de compilación.

Vamos a examinar con más detalle el arreglo `c` de la figura 7.1. 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` devuelve la longitud del arreglo `c`. 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 de creación de arreglos 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 independiente e introduzca un comentario que describa a la variable que está declarando.

Cuando sólo se declara una variable en cada declaración, los corchetes se pueden colocar después del tipo o del nombre de la variable del arreglo, como en:

```
String b[] = new String[100]; // crea el arreglo b
String x[] = new String[27]; // crea el arreglo x
```

pero es preferible colocar los corchetes después del tipo.



Error común de programación 7.3

Declarar 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, ya que 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 sobre el uso de los arreglos

En esta sección presentaremos varios ejemplos que muestran cómo declarar, crear e inicializar arreglos, y cómo manipular sus elementos.

7.4.1 Creación e inicialización de 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 9 se declara `arreglo`, una referencia capaz de referirse a un arreglo de elementos `int`, y luego se inicializa la variable con una referencia a un objeto arreglo que contiene 10 elementos `int`. La línea 11 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        // declara la variable arreglo y la inicializa con un objeto arreglo
9        int[] arreglo = new int[10]; // crea el objeto arreglo
10
11       System.out.printf("%s%8s%n", "Indice", "Valor"); // encabezados de columnas
12
13       // imprime el valor de cada elemento del arreglo
14       for (int contador = 0; contador < arreglo.length; contador++)
15          System.out.printf("%5d%8d%n", contador, arreglo[contador]);
16    }
17 } // 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.

La instrucción `for` (líneas 14 y 15) imprime el índice (representado por `contador`) y el valor de cada elemento del arreglo (representado por `arreglo[contador]`). Al principio la variable de control `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 14) 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 índice 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 que esté *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 índice fuera de rango en tiempo de ejecución.

7.4.2 Uso de un inicializador de arreglos

Usted puede crear un arreglo e inicializar sus elementos con un **inicializador de arreglo**, que es una lista de expresiones separadas por comas (la cual se conoce también como **lista inicializadora**) y que está 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 "tras bambalinas".

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 de cada elemento del arreglo
14       for (int contador = 0; contador < arreglo.length; contador++)
15          System.out.printf("%5d%8d%n", contador, arreglo[contador]);
16    }
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.

7.4.3 Cálculo de los valores a almacenar 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  {
```

Fig. 7.4 | Cálculo de los valores a colocar en los elementos de un arreglo (parte 1 de 2).

```
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;
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     }
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
```



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, según su contexto.*



Buena práctica de programación 7.3

Las constantes con nombres compuestos por varias palabras deben tener cada palabra separada, una de la otra, por un guion bajo (_), como en `LONGITUD_ARREGLO`.



Error común de programación 7.4

Asignar un valor a una variable `final` después de inicializarla es un error de compilación. De igual forma, al tratar de acceder al valor de una variable `final` antes de inicializarla se produce un error de compilación como: "`variable nombreVariable might not have been initialized`".

7.4.4 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, es probable que 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 de la clase `LibroCalificaciones` en las figuras 7.14 y 7.18 utilizan esta técnica.

La figura 7.5 suma los valores contenidos en un 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 15, 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 reutilizable, 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    }
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.

7.4.5 Uso de gráficos de barra para mostrar en forma gráfica los datos de un arreglo

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 proporcionalmente los valores numéricos más grandes. 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 a menudo les gusta analizar la distribución de las calificaciones en un examen. Un profesor podría graficar el número de calificaciones en cada una de las distintas 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();
27       }
28    }
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).

7.4.6 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 independientes 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 de veces. En la figura 7.7 se muestra una versión de esta aplicación con un arreglo.

```
1  // Fig. 7.7: TirarDado.java
2  // Programa para tirar dados que utiliza arreglos en vez de switch.
3  import java.security.SecureRandom;
4
5  public class TirarDado
6  {
7     public static void main(String[] args)
8     {
9        SecureRandom numerosAleatorios = new SecureRandom();
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    }
22 } // fin de la clase TirarDado
```

```
Cara  Frecuencia
   1      999690
   2      999512
   3     1000575
   4      999815
   5      999781
   6     1000627
```

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 22 a 45 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 49 a 51 de la figura 6.7 por un ciclo a través del arreglo `frecuencia` para imprimir los resultados en pantalla (líneas 19 a 20). Cuando estudiemos las nuevas capacidades de programación funcional de Java SE 8 en el capítulo 17, le mostraremos cómo reemplazar las líneas 13-14 y 19-20 ¡con *una sola* instrucción!

7.4.7 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. Considere la siguiente declaración de un problema:

> *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 (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, *intencionalmente*, una respuesta incorrecta (`14`). Cuando se ejecuta un programa en Java, se verifica la validez de los índices de los elementos del arreglo; todos los índices 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 índices 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]];
```

Fig. 7.8 | Programa de análisis de una encuesta (parte 1 de 2).

```
20          }
21          catch (ArrayIndexOutOfBoundsException e)
22          {
23             System.out.println(e); // invoca el método toString
24             System.out.printf( "   respuestas[%d] = %d%n%n",
25                respuesta, respuestas[respuesta]);
26          }
27       }
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    }
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 *seis 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, de tal forma que `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 (intencionalmente) incorrecta que no se contó. En la sección 7.5 hablaremos sobre lo que ocurre cuando el programa de la figura 7.8 encuentra la respuesta inválida (`14`) en el último elemento del arreglo `respuestas`.

7.5 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. Los métodos en sus propias clases también lanzan excepciones, como veremos en el capítulo 8.

7.5.1 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`.

7.5.2 Ejecución del bloque `catch`

Cuando el programa encuentra el valor inválido `14` en el arreglo `respuestas`, intenta sumar `1` a `frecuencia[14]`, que está *fuera* de los límites del arreglo, debido a que `frecuencia` sólo tiene seis elementos (con los índices 0 a 5). 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 locales en el bloque `try`, ahora se encuentran *fuera de alcance* (y ya no existen), por lo que no son accesibles en el bloque `catch`.

El bloque `catch` declara un parámetro de excepción (`e`) de tipo (`IndexOutOfRangeException`). 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 índice 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` *si su programa es correcto.*



Observación de ingeniería de software 7.1

Los sistemas en la industria que han pasado por pruebas exhaustivas aún tienen probabilidades de contener errores. Nuestra preferencia por los sistemas de calidad industrial es atrapar y lidiar las excepciones en tiempo de ejecución, como las excepciones `ArrayIndexOutOfBoundsException`*, para asegurar que un sistema permanezca funcionando o se degrade de manera elegante, e informar a los desarrolladores del sistema sobre el problema.*

7.5.3 El 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 hablaremos sobre el manejo de excepciones de nuevo, y en el capítulo 11 presentaremos un análisis más detallado.

7.6 Ejemplo práctico: 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`”, ... “`Joker`”, “`Reina`”, “`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 final String cara; // cara de la carta ("As", "Dos", ...)
7     private final String palo; // palo de la carta ("Corazones", "Diamantes", ...)
8
9     // el constructor de dos argumentos inicializa la cara y el palo de la carta
10    public Carta(String caraCarta, String paloCarta)
11    {
12       this.cara = caraCarta; // inicializa la cara de la carta
13       this.palo = paloCarta; // inicializa el palo de la carta
14    }
15
16    // devuelve representación String de Carta
17    public String toString()
18    {
19       return cara + " de " + palo;
20    }
21 } // fin de la clase Carta
```

Fig. 7.9 | La clase Carta representa una carta de juego.

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 el número de secuencia (0 a 51) de la siguiente Carta a repartir del arreglo paquete, así como una 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.SecureRandom;
4
5  public class PaqueteDeCartas
6  {
```

Fig. 7.10 | La clase PaqueteDeCartas representa un paquete de cartas de juego (parte 1 de 2).

```
7      private Carta[] paquete; // arreglo de objetos Carta
8      private int cartaActual; // índice 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 SecureRandom numerosAleatorios = new SecureRandom();
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", "Joker", "Reina", "Rey" };
18        String[] palos = { "Corazones", "Diamantes", "Treboles", "Espadas" };
19
20        paquete = new Carta[NUMERO_DE_CARTAS]; // crea arreglo de objetos Carta
21        cartaActual = 0; // la primera Carta repartida será 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     }
28
29     // baraja el paquete de Cartas con algoritmo de una pasada
30     public void barajar()
31     {
32        // la siguiente llamada al método repartirCarta debe empezar en paquete[0]
             otra vez
33        cartaActual = 0;
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        }
46     }
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     }
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 índices 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 índices del arreglo `caras` en las líneas 16 y 17), y el cálculo `cuenta/13` siempre produce un valor de 0 a 3 (los cuatro índices 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 (los 13 "`Corazones`", luego todos los "`Diamantes`", los "`Treboles`" y las "`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 `enum`, 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` (índices 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 "`Reina`" de "`Corazones`", entonces después de la primera asignación, ambos elementos del arreglo contienen la "`Reina`" de "`Corazones`" y se pierde el "`As`" de "`Espadas`"; es por ello que se necesita la variable adicional `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. El ejercicio 7.35 le pedirá que investigue el popular algoritmo para barajar cartas Fisher-Yates y usarlo para volver a implementar el método `barajar` de `PaqueteDeCartas`].

El método **repartirCarta** *de* **PaqueteDeCartas**

El método `repartirCarta` (líneas 49 a 56) reparte un objeto `Carta` en el arreglo. Recuerde que `cartaActual` indica el índice 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" e incrementa positivamente `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`. 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
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       }
21    }
22 } // fin de la clase PruebaPaqueteDeCartas
```

```
Nueve de Espadas    Joker de Corazones   Reina de Treboles    Siete de Treboles
Seis de Corazones   Seis de Treboles     Joker 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      Reina de Diamantes   Dos de Corazones     Reina de Corazones
As de Corazones     Cuatro de Treboles   Cinco de Espadas     Joker 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
Reina de Espadas    Dos de Diamantes     Rey de Treboles      Diez de Corazones
Cinco de Corazones  As de Diamantes      Rey de Espadas       Joker 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 .

Prevención de excepciones `NullPointerException`

En la figura 7.10 creamos un arreglo `paquete` de 52 referencias `Carta`. Cada elemento de todo arreglo de tipo por referencia creado con `new` se inicializa con `null` de manera predeterminada. Las variables de tipo por referencia que son campos de una clase se inicializan también con `null` de manera predeterminada. Una excepción `NullPointerException` ocurre al tratar de llamar a un método sobre una referencia `null`. En el código de calidad industrial, asegurar que las referencias no sean `null` antes de usarlas para llamar métodos evita las excepciones `NullPointerException`.

7.7 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 7.16 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 cada uno de los valores sucesivos en el arreglo. 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 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    }
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 *sólo* puede utilizarse para obtener elementos del arreglo; *no para modificarlos*. Si su programa necesita modificar elementos, use la instrucción `for` tradicional, controlada por contador.

Cuando el código que itera a través de un arreglo *no* requiere acceso al contador que indica el índice del elemento actual del arreglo, se puede utilizar la instrucción `for` mejorada en lugar de la instrucción `for` controlada por contador. Por ejemplo, para totalizar los enteros en un arreglo se requiere acceso sólo a los valores de los elementos, ya que el índice 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.



Tip para prevenir errores 7.2

La instrucción `for` mejorada simplifica el código para iterar a través de un arreglo, mejorando la legibilidad del código y eliminando varias posibilidades de error, como especificar de manera inapropiada el valor inicial de la variable de control, la prueba de continuación de ciclo y la expresión de incremento.

Java SE 8

La instrucción `for` y la instrucción `for` mejorada iteran en forma secuencial, desde un valor inicial hasta un valor final. En el capítulo 17, Lambdas y flujos de Java SE 8, aprenderá acerca de la clase `Stream` y su método `forEach`. Al trabajar en conjunto, estos elementos ofrecen un medio elegante, más conciso y menos propenso a errores para iterar a través de colecciones, de modo que algunas de las iteraciones puedan ocurrir en paralelo con otras para lograr un mejor rendimiento del sistema multinúcleo.

7.8 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. 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 `temperaturasPorHora`, 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.

La instrucción `for` mejorada en las líneas 16 y 17 imprime en pantalla los elementos de `arreglo`. La línea 19 invoca al método `modificarArreglo` y le pasa `arreglo` como argumento. El método (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 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.

```
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:%n");
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:%n");
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    }
34
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    }
```

Fig. 7.13 | Paso de arreglos y elementos individuales de un arreglo a los métodos (parte 1 de 2).

```
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      }
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 figura 7.13 demuestra a continuación que, al pasar 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 por 2 el valor recibido como argumento, 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).

7.9 Comparación entre paso por valor y paso por referencia

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 de las 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, ya que *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 seguir interactuando 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 en vez de los objetos arreglo en sí tiene sentido por cuestiones de rendimiento. Como todo en Java se pasa por valor, si se pasaran objetos arreglo, se pasaría una copia de cada elemento. En los arreglos grandes, esto desperdiciaría tiempo y consumiría una cantidad considerable de almacenamiento para las copias de los elementos.

7.10 Ejemplo práctico: la clase `LibroCalificaciones` que usa un arreglo para almacenar las calificaciones

Ahora presentaremos la primera parte de nuestro ejemplo práctico sobre el desarrollo de una clase `LibroCalificaciones` que los instructores pueden usar para mantener las calificaciones de los estudiantes sobre un examen y mostrar un informe que incluya las calificaciones, el promedio de la clase, la calificación más baja, la calificación más alta y una barra de distribución de calificaciones. La versión de la clase `LibroCalificaciones` que se presenta en esta sección almacena las calificaciones de un examen en un arreglo unidimensional. En la sección 7.12 presentaremos una versión de la clase `LibroCalificaciones` que usa un arreglo bidimensional para almacenar las calificaciones de los estudiantes para *varios* exámenes.

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. 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 variable de instancia `length` del parámetro arreglo del constructor. Por ende, un objeto `LibroCalificaciones` puede procesar un número *variable* de calificaciones. Los valores de las calificaciones en el arreglo que se pasa podrían ser introducidos por un usuario desde el teclado, ser leídos desde un archivo en el disco (como veremos en el capítulo 15) o podrían provenir

de una variedad de fuentes adicionales. En la clase PruebaLibroCalificaciones, 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.

```
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     // constructor
10    public LibroCalificaciones(String nombreDelCurso, int[] calificaciones)
11    {
12       this.nombreDelCurso = nombreDelCurso;
13       this.calificaciones = calificaciones;
14    }
15
16    // método para establecer el nombre del curso
17    public void establecerNombreDelCurso(String nombreDelCurso)
18    {
19       this.nombreDelCurso = nombreDelCurso;
20    }
21
22    // método para obtener el nombre del curso
23    public String obtenerNombreDelCurso()
24    {
25       return nombreDelCurso;
26    }
27
28    // realiza varias operaciones sobre los datos
29    public void procesarCalificaciones()
30    {
31       // imprime el arreglo calificaciones en pantalla
32       imprimirCalificaciones();
33
34       // llama al método obtenerPromedio para calcular la calificación promedio
35       System.out.printf("%nEl promedio de la clase es %.2f%n", obtenerPromedio());
36
37       // llama a los métodos obtenerMinima y obtenerMaxima
38       System.out.printf("La calificacion mas baja es %d%nLa calificacion mas
                           alta es %d%n%n",
39          obtenerMinima(), obtenerMaxima());
40
41       // llama a imprimirGraficoBarras para imprimir el gráfico de distribución
            de calificaciones
42       imprimirGraficoBarras();
43    }
44
```

Fig. 7.14 | La clase LibroCalificaciones que utiliza un arreglo para almacenar las calificaciones de una prueba (parte 1 de 3).

```
45      // busca la calificación más baja
46      public int obtenerMinima()
47      {
48         int califBaja = calificaciones[0]; // asume que calificaciones[0] es la
                                               más baja
49
50         // itera a través del arreglo de calificaciones
51         for (int calificacion : calificaciones)
52         {
53            // si calificación es menor que califBaja, se asigna a califBaja
54            if (calificacion < califBaja)
55               califBaja = calificacion; // nueva calificación más baja
56         }
57
58         return califBaja;
59      }
60
61      // busca la calificación más alta
62      public int obtenerMaxima()
63      {
64         int califAlta = calificaciones[0]; // asume que calificaciones[0] es la
                                               más alta
65
66         // itera a través del arreglo de calificaciones
67         for (int calificacion : calificaciones)
68         {
69            // si calificacion es mayor que califAlta, se asigna a califAlta
70            if (calificacion > califAlta)
71               califAlta = calificacion; // nueva calificación más alta
72         }
73
74         return califAlta;
75      }
76
77      // determina la calificación promedio de la prueba
78      public double obtenerPromedio()
79      {
80         int total = 0;
81
82         // suma las calificaciones para un estudiante
83         for (int calificacion : calificaciones)
84            total += calificacion;
85
86         // devuelve el promedio de las calificaciones
87         return (double) total / calificaciones.length;
88      }
89
90      // imprime grafico de barras que muestra la distribución de las calificaciones
91      public void imprimirGraficoBarras()
92     {
93         System.out.println("Distribucion de calificaciones:");
94
95         // almacena la frecuencia de las calificaciones en cada rango de 10
              calificaciones
96         int[] frecuencia = new int[11];
```

Fig. 7.14 | La clase `LibroCalificaciones` que utiliza un arreglo para almacenar las calificaciones de una prueba (parte 2 de 3).

```
97
98          // para cada calificación, incrementa la frecuencia apropiada
99          for (int calificacion : calificaciones)
100            ++frecuencia[calificacion / 10];
101
102         // para cada frecuencia de calificación, imprime una barra en el gráfico
103         for (int cuenta = 0; cuenta < frecuencia.length; cuenta++)
104         {
105            // imprime etiqueta de barra ("00-09: ", ..., "90-99: ", "100: ")
106            if (cuenta == 10)
107               System.out.printf("%5d: ", 100);
108            else
109               System.out.printf("%02d-%02d: ",
110                  cuenta * 10, cuenta * 10 + 9);
111
112            // imprime barra de asteriscos
113            for (int estrellas = 0; estrellas < frecuencia[cuenta]; estrellas++)
114               System.out.print("*");
115
116            System.out.println();
117         }
118      }
119
120      // imprime el contenido del arreglo de calificaciones
121      public void imprimirCalificaciones()
122      {
123         System.out.println("Las calificaciones son:%n%n");
124
125         // imprime la calificación de cada estudiante
126         for (int estudiante = 0; estudiante < calificaciones.length; estudiante++)
127            System.out.printf("Estudiante %2d: %3d%n",
128               estudiante + 1, calificaciones[estudiante]);
129      }
130   } // fin de la clase LibroCalificaciones
```

Fig. 7.14 | La clase `LibroCalificaciones` que utiliza un arreglo para almacenar las calificaciones de una prueba (parte 3 de 3).

El método `procesarCalificaciones` (líneas 29 a 43) contiene una serie de llamadas a métodos que produce un informe en el que se resumen las calificaciones. La línea 32 llama al método `imprimirCalificaciones` para imprimir el contenido del arreglo `calificaciones`. Las líneas 126 a 128 en el método `imprimirCalificaciones` imprimen las calificaciones de los estudiantes. En este caso se *debe* utilizar una instrucción `for` controlada por contador, ya que las líneas 127 y 128 utilizan el valor de la variable contador `estudiante` para imprimir cada calificación enseguida de un número de estudiante específico (vea la salida en la figura 7.15). Aunque los índices de los arreglos empiezan en 0, lo común es que el profesor enumere a los estudiantes empezando desde 1. Por ende, las líneas 127 y 128 imprimen `estudiante + 1` como el número de estudiante para producir las etiquetas de calificaciones "`Estudiante 1: `", "`Estudiante 2: `", y así en lo sucesivo.

A continuación, el método `procesarCalificaciones` llama al método `obtenerPromedio` (línea 35) para obtener el promedio de las calificaciones en el arreglo. El método `obtenerPromedio` (líneas 78 a 88) 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 87 utiliza calificaciones.length para determinar el número de calificaciones que se van a promediar.

Las líneas 38 y 39 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, respectivamente. Cada uno de estos métodos utiliza una instrucción for mejorada para iterar a través del arreglo calificaciones. Las líneas 51 a 56 en el método obtenerMinima iteran a través del arreglo. Las líneas 54 y 55 comparan cada calificación con califBaja; si una calificación es menor que califBaja, a califBaja se le asigna esa calificación. Cuando la línea 58 se ejecuta, califBaja contiene la calificación más baja en el arreglo. El método obtenerMaxima (líneas 62 a 75) funciona de manera similar al método obtenerMinima.

Por último, la línea 42 en el método procesarCalificaciones llama al método imprimirGraficoBarras para imprimir un gráfico de distribución 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 99 y 100 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 96 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 99 y 100 incrementan el elemento apropiado del arreglo frecuencia. Para determinar qué elemento se debe incrementar, la línea 100 divide la calificacion actual entre 10, mediante la *división entera*. Por ejemplo, si calificacion es 85, la línea 100 incrementa frecuencia[8] para actualizar la cuenta de calificaciones en el rango 80-89. Las líneas 103 a 117 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 113 y 116 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 tanto el nombre de un curso como arregloCalif al constructor de LibroCalificaciones. Las líneas 14 y 15 despliegan un mensaje de bienvenida, que incluye el nombre del curso almacenado el objeto LibroCalificaciones. La línea 16 invoca el método procesarCalificaciones del objeto LibroCalificaciones. La salida muestra el resumen de las 10 calificaciones en miLibroCalificaciones.



Observación de ingeniería de software 7.2

Un arnés de prueba (o aplicación de prueba) es responsable de crear un objeto de la clase que se probará así como de proporcionarle 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 15), de una base de datos (como veremos en el capítulo 24, en inglés) o de una red (capítulo 28, 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 permite a la clase ser más reutilizable y 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
```

Fig. 7.15 | PruebaLibroCalificaciones crea un objeto LibroCalificaciones, usando un arreglo de calificaciones, y después invoca al método procesarCalificaciones para analizarlas (parte 1 de 2).

```
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        System.out.printf("Bienvenido al libro de calificaciones para%n%s%n%n",
15            miLibroCalificaciones.obtenerNombreDelCurso());
16        miLibroCalificaciones.procesarCalificaciones();
17     }
18  } // 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

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`, usando un arreglo de calificaciones, y después invoca al método `procesarCalificaciones` para analizarlas (parte 2 de 2).

Java SE 8

En el capítulo 17, Lambdas y flujos de Java SE 8, el ejemplo de la figura 17.5 usa los métodos de flujo `min`, `max`, `count` y `average` para procesar los elementos de un arreglo `int` de manera elegante y concisa sin tener que escribir instrucciones de repetición. En el capítulo 23, Concurrency, el ejemplo de la figura 23.29 usa el método de flujo `summaryStatistics` para realizar todas estas operaciones en una sola llamada a un método.

7.11 Arreglos multidimensionales

Los arreglos multidimensionales de dos dimensiones se utilizan con frecuencia para representar *tablas* de valores, con datos ordenados en *filas* y *columnas*. Para identificar un elemento específico de una tabla, debemos especificar *dos* índices. *Por convención*, el primero identifica la fila del elemento y el segundo su columna. Los arreglos que requieren dos índices 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***.

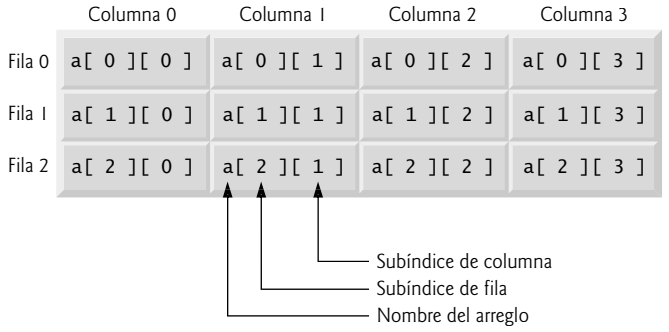


Fig. 7.16 | Arreglo bidimensional con tres filas y cuatro columnas.

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 índices que identifican en forma única a cada elemento por índice de fila y columna. Todos los nombres de los elementos en la *fila* 0 tienen un *primer* índice de `0`, y los nombres de los elementos en la *columna* 3 tienen un segundo índice de `3`.

Arreglos de arreglos unidimensionales

Al igual que los arreglos unidimensionales, los 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. Por lo tanto, `1` y `2` inicializan a `b[0][0]` y `b[0][1]`, respectivamente; `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 independientes: 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*. 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, respectivamente.

```
1  // Fig. 7.17: InicArreglo.java
2  // Inicialización de arreglos bidimensionales.
3
4  public class InicArreglo
5  {
```

Fig. 7.17 | Inicialización de dos arreglos bidimensionales (parte 1 de 2).

```
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%n");
16        imprimirArreglo(arreglo2); // muestra arreglo2 por filas
17     }
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();
30        }
31     }
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 (parte 2 de 2).

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` anidada (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. La condición del `for` interior permite al ciclo determinar el número exacto de columnas en cada fila. En la figura 7.18 demostraremos las instrucciones `for` anidadas mejoradas.

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 el valor de cero a todos los elementos en la fila 2 del arreglo `a`, en la figura 7.16:

```
for (int columna = 0; columna < a[2].length; columna++)
   a[2][columna] = 0;
```

Especificamos la fila 2; por lo tanto, sabemos que el *primer* índice 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];
}
```

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.

7.12 Ejemplo práctico: la clase LibroCalificaciones que usa un arreglo bidimensional

En la sección 7.10 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 46 a 62) determina la calificación más baja de cualquier estudiante durante el semestre. El método obtenerMaxima (líneas 65 a 83) determina la calificación más alta de cualquier estudiante durante el semestre. El método obtenerPromedio (líneas 86 a 96) determina el promedio semestral de un estudiante específico. El método imprimirGraficoBarras (líneas 99 a 129) imprime un gráfico de barras de todas las calificaciones de los estudiantes durante el semestre. El método imprimirCalificaciones (líneas 132 a 156) 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 nombreDelCurso, int[][] calificaciones)
11    {
12       this.nombreDelCurso = nombreDelCurso;
13       this.calificaciones = calificaciones;
14    }
15
16    // método para establecer el nombre del curso
17    public void establecerNombreDelCurso(String nombreDelCurso)
18    {
19       this.nombreDelCurso = nombreDelCurso;
20    }
21
22    // método para obtener el nombre del curso
23    public String obtenerNombreDelCurso()
24    {
25       return nombreDelCurso;
26    }
27
28    // realiza varias operaciones sobre los datos
29    public void procesarCalificaciones()
30    {
31       // imprime el arreglo de calificaciones
32       imprimirCalificaciones();
33
34       // llama a los métodos obtenerMinima y obtenerMaxima
35       System.out.printf("%n%s %d%n%s %d%n%n",
36          "La calificacion mas baja en el libro de calificaciones es",
              obtenerMinima(),
37          "La calificacion mas alta en el libro de calificaciones es",
              obtenerMaxima());
38
39       // imprime gráfico de distribución de todas las calificaciones para todas
            las pruebas
40       imprimirGraficoBarras();
41    } // fin del método procesarCalificaciones
42
43    // busca la calificación más baja
44    public int obtenerMinima()
45    {
46       // asume que el primer elemento del arreglo calificaciones es el más bajo
47       int califBaja = calificaciones[0][0];
```

Fig. 7.18 | Clase `LibroCalificaciones` que utiliza un arreglo bidimensional para almacenar calificaciones (parte 1 de 4).

```
48
49          // itera a través de las filas del arreglo calificaciones
50          for (int[] califEstudiantes : calificaciones)
51          {
52             // itera a través de las columnas de la fila actual
53             for (int calificacion : califEstudiantes)
54             {
55                // si la calificación es menor que califBaja, la asigna a califBaja
56                if (calificacion < califBaja)
57                   califBaja = calificacion;
58             }
59          }
60
61          return califBaja;
62       }
63
64       // busca la calificación más alta
65       public int obtenerMaxima()
66       {
67          // asume que el primer elemento del arreglo calificaciones es el más alto
68          int califAlta = calificaciones[0][0];
69
70          // itera a través de las filas del arreglo calificaciones
71          for (int[] califEstudiantes : calificaciones)
72          {
73             // itera a través de las columnas de la fila actual
74             for (int calificacion : califEstudiantes)
75             {
76                // si la calificación es mayor que califAlta, la asigna a califAlta
77                if (calificacion > califAlta)
78                   califAlta = calificacion;
79             }
80          }
81
82          return califAlta;
83       }
84
85       // determina la calificación promedio para un conjunto específico de
            calificaciones
86       public double obtenerPromedio(int[] conjuntoDeCalif)
87       {
88          int total = 0;
89
90          // suma las calificaciones para un estudiante
91          for (int calificacion : conjuntoDeCalif)
92            total += calificacion;
93
94         // devuelve el promedio de calificaciones
95         return (double) total / conjuntoDeCalif.length;
96       }
97
98       // imprime gráfico de barras que muestra la distribución de calificaciones en
            general
99       public void imprimirGraficoBarras()
```

Fig. 7.18 | Clase `LibroCalificaciones` que utiliza un arreglo bidimensional para almacenar calificaciones (parte 2 de 4).

```
100    {
101       System.out.println("Distribucion de calificaciones en general:");
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 en LibroCalificaciones, incrementa la frecuencia
             apropiada
107       for (int[] califEstudiantes : calificaciones)
108       {
109          for (int calificacion : califEstudiantes)
110             ++frecuencia[calificacion / 10];
111       }
112
113       // para cada frecuencia de calificaciones, imprime una barra en el gráfico
114       for (int cuenta = 0; cuenta < frecuencia.length; cuenta++)
115       {
116          // imprime etiquetas de las barras ("00-09: ", ..., "90-99: ", "100: ")
117          if (cuenta == 10)
118             System.out.printf("%5d: ", 100);
119          else
120             System.out.printf("%02d-%02d: ",
121                cuenta * 10, cuenta * 10 + 9);
122
123          // imprime barra de asteriscos
124          for (int estrellas = 0; estrellas < frecuencia[cuenta]; estrellas++)
125             System.out.print("*");
126
127          System.out.println();
128       }
129    }
130
131    // imprime el contenido del arreglo calificaciones
132    public void imprimirCalificaciones()
133    {
134       System.out.println("Las calificaciones son:%n%n");
135       System.out.print("            "); // alinea encabezados de columnas
136
137       // crea un encabezado de columna para cada una de las pruebas
138       for (int prueba = 0; prueba < calificaciones[0].length; prueba++)
139          System.out.printf("Prueba %d  ", prueba + 1);
140
141       System.out.println("Promedio"); // encabezado de columna de promedio de
                                          estudiantes
142
143       // crea filas/columnas de texto que representan el arreglo calificaciones
144       for (int estudiante = 0; estudiante < calificaciones.length; estudiante++)
145       {
146          System.out.printf("Estudiante %2d", estudiante + 1);
147
148          for (int prueba : calificaciones[estudiante]) // imprime calificaciones
                                                             de estudiante
149             System.out.printf("%8d", prueba);
150
```

Fig. 7.18 | Clase `LibroCalificaciones` que utiliza un arreglo bidimensional para almacenar calificaciones (parte 3 de 4).

```
151            // llama al método obtenerPromedio para calcular la calificación
                  promedio del estudiante;
152            // pasa fila de calificaciones como argumento para obtenerPromedio
153            double promedio = obtenerPromedio(calificaciones[estudiante]);
154            System.out.printf("%9.2f%n", promedio);
155         }
156      }
157   } // 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 50 a 59). 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; es decir, 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 107 a 111) 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 132 a 156) 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 138 y 139 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 144 a 155 imprime primero una etiqueta de fila mediante el uso de una variable contador para identificar a cada estudiante (línea 146). Aunque los subíndices de los arreglos empiezan en `0`, las líneas 139 y 146 imprimen `prueba + 1` y `estudiante + 1` respectivamente, para producir números de prueba y de estudiante que empiecen en 1 (vea la salida en la figura 7.19). La instrucción `for` interna (líneas 148 y 149) 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. Una instrucción `for` mejorada puede anidarse en una instrucción `for` controlada por contador, y viceversa.

Por último, la línea 153 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 86 a 96) recibe un argumento (un arreglo unidimensional de resultados de la prueba para un estudiante específico). Cuando la línea 153 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 95).

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 muestran un mensaje de bienvenida que contiene el nombre del curso; luego la línea 25 invoca al método `procesarCalificaciones` de `miLibroCalificaciones` para mostrar en pantalla un informe que sintetice las calificaciones de los estudiantes para el semestre.

```
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     {
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       System.out.printf("Bienvenido al libro de calificaciones para%n%s%n%n",
24          miLibroCalificaciones.obtenerNombreDelCurso());
25       miLibroCalificaciones.procesarCalificaciones();
```

Fig. 7.19 | `PruebaLibroCalificaciones` crea un objeto `LibroCalificaciones` usando un arreglo bidimensional de calificaciones y después invoca al método `procesarCalificaciones` para analizarlas (parte 1 de 2).

```
26      }
27   } // 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` usando un arreglo bidimensional de calificaciones y después invoca al método `procesarCalificaciones` para analizarlas (parte 2 de 2).

7.13 Listas de argumentos de longitud variable

Con las **listas de argumentos de longitud variable** podemos crear métodos que reciben un número no especificado 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 y el nombre del parámetro, 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.



Error común de programación 7.5

Colocar una elipsis para indicar una lista de argumentos de longitud variable en medio de una lista de parámetros es un error de sintaxis. La elipsis sólo puede colocarse al final de la lista de parámetros.

```
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;
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    }
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) );
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    }
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.

7.14 Uso de argumentos de línea de comandos

En muchos sistemas, es posible pasar argumentos desde la línea de comandos a una aplicación mediante el parámetro `String[]` de `main`, el cual recibe un arreglo de objetos `String`. 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 espacios 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`.

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`, respectivamente. Las líneas 24 y 25 calculan el valor para cada elemento del arreglo.

```
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.%n");
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];
```

Fig. 7.21 | Inicialización de un arreglo mediante el uso de argumentos de línea de comandos (parte 1 de 2).

```
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       }
33    }
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
```

```
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).

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.

7.15 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 (es decir, acomodar los elementos en orden ascendente), **binarySearch** para *buscar* en un arreglo *ordenado* (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, Búsqueda, ordenamiento y Big O, veremos cómo implementar nuestros propios algoritmos de búsqueda y ordenamiento, los cuales 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 dentro del arreglo. 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
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 ? "==" : "!="));
```

Fig. 7.22 | Métodos de la clase `Arrays` y `System.arraycopy` (parte 1 de 2).

```
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       }
57
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       }
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` y `System.arraycopy` (parte 2 de 2).

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` de la clase `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`.



Tip para prevenir errores 7.3

Al comparar el contenido de arreglos, use siempre `Arrays.equals(arreglo1, arreglo2)`*, que compara el contenido de dos arreglos, en vez de* `arreglo1.equals(arreglo2)`*, que compara si* `arreglo1` *y* `arreglo2` *hacen referencia al mismo objeto arreglo.*

En las 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`, respectivamente) 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 veremos la búsqueda binaria con detalle.



Error común de programación 7.6

Pasar un arreglo desordenado al método `binarySearch` *es un error lógico; el valor devuelto es indefinido.*

Java SE 8: el método `parallelSort` *de la clase* `Arrays`

Ahora la clase `Arrays` tiene varios métodos "paralelos" que aprovechan el hardware multinúcleo. El método `parallelSort` de `Arrays` pueden ordenar los arreglos grandes con más eficiencia en sistemas multinúcleo. En la sección 23.12 crearemos un arreglo muy grande y usaremos características de la API de fecha/hora de Java SE 8 para comparar cuánto tiempo se requiere para ordenar el arreglo con los métodos `sort` y `parallelSort`.

7.16 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 en la memoria 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 ello, se reduce el tiempo de desarrollo de aplicaciones.

Usted ya utilizó arreglos para almacenar secuencias de objetos. Los arreglos no cambian automáticamente 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, ya que 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`. 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**. *Sólo se pueden usar tipos no primitivos para declarar variables y crear objetos de clases genéricas*. Sin embargo, Java cuenta con un mecanismo (conocido como *boxing*) que permite envolver valores primitivos como objetos para usarlos con clases genéricas. Así, por ejemplo,

```
ArrayList<Integer> enteros;
```

declara a `enteros` como un objeto `ArrayList` que puede almacenar sólo objetos `Integer`. Si coloca un valor `int` en un `ArrayList<Integer>`, el valor `int` es envuelto (*boxed*) como objeto `Integer`, y cuando obtiene un objeto `Integer` de un `ArrayList<Integer>`, para luego asignar el objeto a una variable `int`, el valor `int` dentro del objeto es desenvuelto (*unboxed*).

En los capítulos 16 y 20 hablaremos sobre otras clases de colecciones genéricas y sobre 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>`.

Demostración de un `ArrayList<String>`

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 convencional 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 "amarillo" en el índice 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 for 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
49
50       // muestra cada elemento en elementos
51       for (String elemento : elementos)
52          System.out.printf(" %s", elemento);
53
```

Fig. 7.24 | Demostración de la colección de genéricos `ArrayList<T>` (parte 1 de 2).

```
54          System.out.println();
55       }
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<T>` (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 índice. Al igual que en los arreglos, los índices de las colecciones empiezan en cero. El segundo argumento es el *valor* a insertar en ese *índice*. Los índices 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`** (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`.

Java SE 7: notación de diamante (<>) para crear un objeto de una clase genérica

Considere la línea 10 de la figura 7.24:

```
ArrayList<String> elementos = new ArrayList<String>();
```

Observe que `ArrayList<String>` aparece en la declaración de la variable *y* en la expresión de creación de instancia de clase. Java SE 7 introdujo la **notación de diamante** (`<>`) para simplificar instrucciones como ésta. Al usar `<>` en una expresión de creación de instancia de clase para un objeto de una clase *genérica*, indicamos al compilador que debe determinar lo que pertenece en los paréntesis angulares. En Java SE 7 y versiones superiores, la instrucción anterior puede escribirse así:

```
ArrayList<String> elementos = new ArrayList<>();
```

Cuando el compilador encuentra el diamante (`<>`) en la expresión de creación de instancia de clase, usa la declaración de la variable `elementos` para determinar el tipo de elemento (`String`) de `ArrayList`; a esto se le conoce como *inferir el tipo del elemento*.

7.17 (Opcional) Ejemplo práctico de GUI y gráficos: cómo dibujar 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 más 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.

En la figura 7.25, 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  // Dibuja un arcoiris mediante el uso de arcos y un arreglo de colores.
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    }
24
25    // dibuja un arco iris, usando arcos concéntricos
26    public void paintComponent(Graphics g)
27    {
28       super.paintComponent(g);
29
30       int radio = 20; // el radio de un arco
31
```

Fig. 7.25 | Dibujo de un arco iris mediante el uso arcos y un arreglo de colores (parte 1 de 2).

```
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          {
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          }
47       }
48    } // fin de la clase DibujoArcoIris
```

Fig. 7.25 | Dibujo de un arco iris mediante el uso 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 solo un semicírculo sólido color violeta. 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  {
```

Fig. 7.26 | Aplicación de prueba para mostrar un arco iris (parte 1 de 2).

```
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);
13        aplicacion.add(panel);
14        aplicacion.setSize(400, 250);
15        aplicacion.setVisible(true);
16     }
17     } // fin de la clase PruebaDibujoArcoIris
```

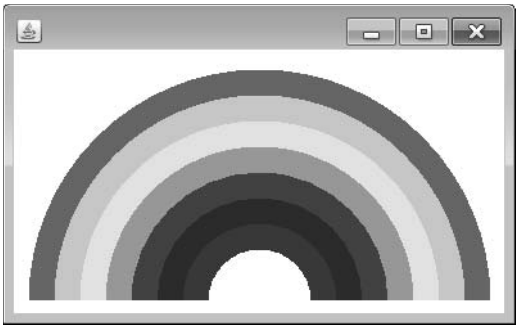


Fig. 7.26 | Aplicación de prueba para mostrar un arco iris (parte 2 de 2).

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.

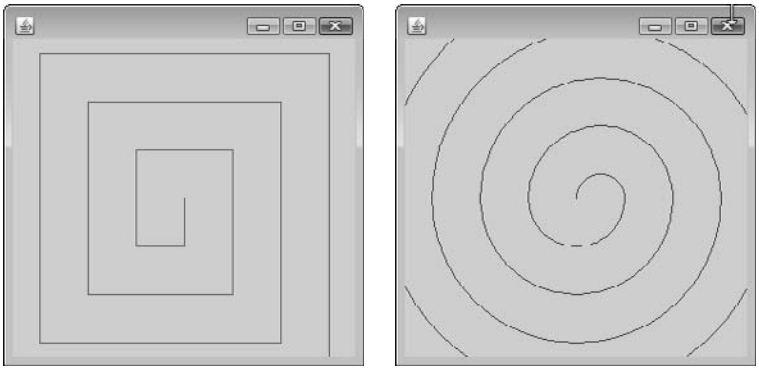


Fig. 7.27 | Dibujo de una espiral con `drawLine` (izquierda) y `drawArc` (derecha).

7.18 Conclusión

En este capítulo comenzamos nuestra introducción a las estructuras de datos, explorando el 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, 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 índice 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 16, Colecciones de genéricos. 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 16 también presenta las otras estructuras de datos predefinidas de Java y cubre los métodos adicionales de la clase `Arrays`. Después de leer este capítulo podrá utilizar algunos de los métodos de `Arrays` que se describen en el capítulo 16, aunque hay otros métodos de `Arrays` que requieren cierto conocimiento sobre los conceptos que presentaremos más adelante en este libro. El capítulo 20 (en inglés, en el sitio web) 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 que se utilizan con muchos tipos de datos distintos. El capítulo 21 (en inglés, en el sitio web), Custom Generic Data Structures, 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. 244) 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. 245) 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. 245), la cual incluye el nombre del arreglo, seguido del índice del elemento específico entre corchetes (`[]`; pág. 245).
- El primer elemento en cada arreglo tiene el índice cero (pág. 245), y algunas veces se le llama el elemento cero.
- Un índice debe ser un entero no negativo. Un programa puede utilizar una expresión como un índice.
- Un objeto tipo arreglo conoce su propia longitud, y almacena esta información en una variable de instancia `length` (pág. 246).

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. 246), que utiliza la palabra clave `new`.
- Cuando se crea un arreglo, cada elemento recibe un valor predeterminado de cero para los elementos numéricos de tipo primitivo, de `false` para los elementos booleanos y de `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 sobre el uso de los arreglos

- Un programa puede crear un arreglo e inicializar sus elementos con un inicializador de arreglos (pág. 248).
- Las variables constantes (pág. 250) se declaran con la palabra clave `final`, deben inicializarse antes de utilizarlas, y no pueden modificarse de ahí en adelante.

Sección 7.5 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 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. 256) nos permite crear programas tolerantes a fallas.
- Cuando se ejecuta un programa en Java, la JVM comprueba los índices de los arreglos para asegurarse que sean mayores o iguales a 0 y menores que la longitud del arreglo. Si un programa utiliza un índice inválido, Java genera una excepción (pág. 256) para indicar que ocurrió un error en el programa, en tiempo de ejecución
- Para manejar una excepción, hay que colocar en una instrucción `try` cualquier código que podría lanzar la excepción (pág. 256).
- El bloque `try` (pág. 256) contiene el código que podría lanzar una excepción, y el bloque `catch` (pág. 256) 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.
- Cuando se ejecuta un programa, se comprueba la validez de los índices 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 índice inválido para acceder a un elemento, ocurre una excepción `ArrayIndexOutOfRangeException` (pág. 257).
- El método `toString` de un objeto excepción devuelve el mensaje de error de la excepción.

Sección 7.6 Ejemplo práctico: 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 7.7 Instrucción `for` mejorada

- La instrucción `for` mejorada (pág. 262) 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 7.8 Paso de arreglos a los métodos

- Cuando un argumento se pasa por valor, se hace una copia del valor del argumento y se pasa al método que se llamó. Este método trabaja exclusivamente con la copia.

Sección 7.9 Comparación entre paso por valor y paso por referencia

- Cuando se pasa un argumento por referencia (pág. 265), el método al que se llamó puede acceder directamente al valor del argumento en el método que lo llamó, 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. 265), 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 7.11 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. 272) 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 7.13 Listas de argumentos de longitud variable

- Un tipo de argumento seguido por una elipsis (`...`; pág. 281) 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. 281) 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 7.14 Uso de argumentos de línea de comandos

- Para pasar argumentos a `main` (pág. 283) 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 7.15 La clase `Arrays`

- La clase `Arrays` (pág. 285) 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. 285) nos permite copiar los elementos de un arreglo en otro.

Sección 7.16 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. 288) es similar a un arreglo, sólo que su tamaño se puede ajustar en forma dinámica.
- El método `add` (pág. 290) 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. 290) 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 índice 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 _____ y _____.
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 índice 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 elementos `TAMANIO_ARREGLO` 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, colecciones. 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 $5,000 en ventas en una semana, recibirá $200 más el 9% de $5,000, 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) $1,000 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;
   }
}
```

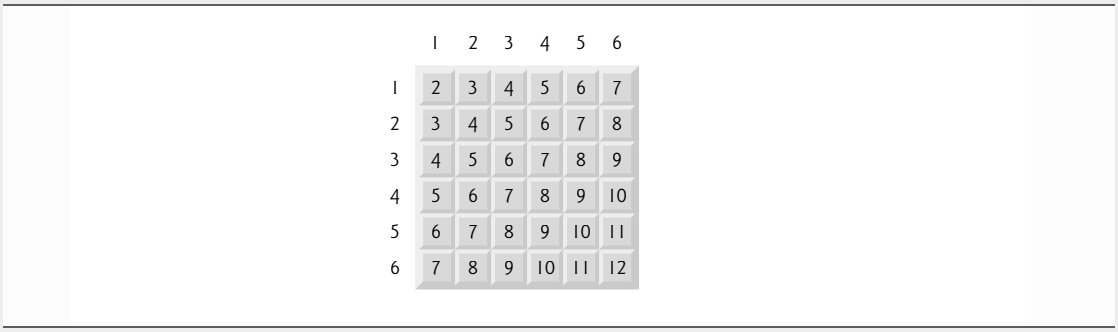
7.14 ***(Lista de argumentos de longitud variable)*** Escriba una aplicación que calcule el producto de una serie de enteros que se pasan al método `producto`; use una lista de argumentos de longitud variable. Pruebe su método con varias llamadas, cada una con un número distinto de argumentos.

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 de 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 de 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 simbolice 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 un bolígrafo en una de dos posiciones, arriba o abajo. Mientras el bolígrafo está abajo, el animalito va trazando figuras a medida que se va moviendo, y mientras el bolígrafo está arriba, la tortuga 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 el bolígrafo se encuentra arriba o abajo. Suponga que la tortuga siempre empieza en la posición (0, 0) del piso, con su bolígrafo hacia arriba. El conjunto de comandos de la tortuga que su aplicación debe procesar se muestra en la figura 7.29.

Comando	Significado
`1`	Bolígrafo arriba
`2`	Bolígrafo 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 punto cerca del centro del piso. El siguiente "programa" dibuja e imprime un cuadrado de 12 por 12, dejando el bolígrafo 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 el bolígrafo 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).

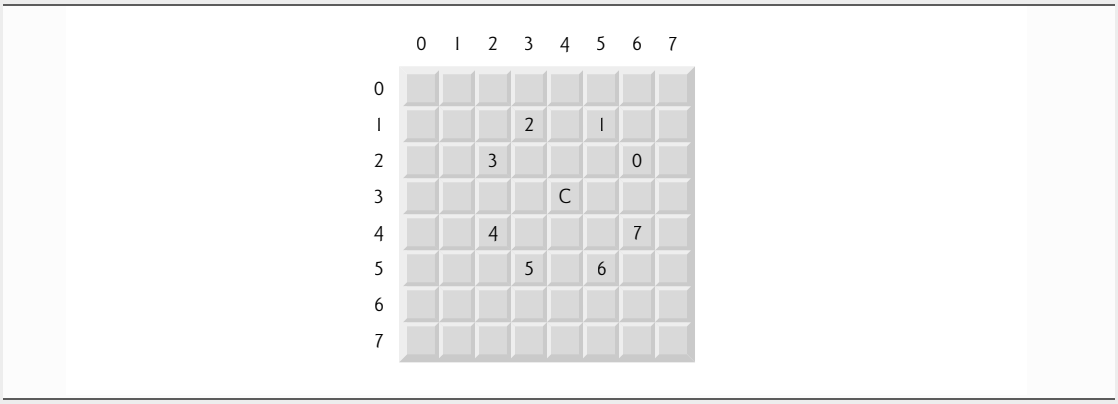


Fig. 7.30 | Los ocho posibles movimientos del caballo.

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 con 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 el inciso (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 1,000 paseos. Use un arreglo unidimensional para llevar el registro del número de paseos de cada longitud. Cuando su programa termine de intentar los 1,000 paseos, deberá imprimir esta información en un formato tabular ordenado. ¿Cuál fue el mejor resultado?

c) Es muy probable que la aplicación del inciso (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?].

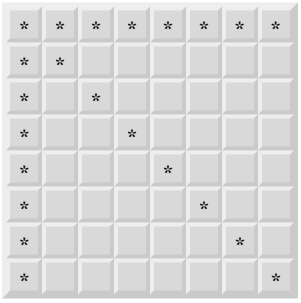


Fig. 7.31 | Las 22 posiciones eliminadas al colocar una reina en la esquina superior izquierda.

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.

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 1,000 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 tac 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 llegó o se pasó 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 tac 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 tac 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 el inciso (b).

Los ejercicios 7.30 a 7.34 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) una tercia (como tres reinas)
d) un póquer (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 los tipos `enum` `Cara` y `Palo` que representen las caras y los palos de las cartas. Declare cada uno de estos tipos `enum` 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 el tipo `enum` `Cara`, y un arreglo de objetos `Palo` que se inicialice con los nombres de las constantes en el tipo `enum` `Palo`. [*Nota:* al imprimir en pantalla una constante `enum` como un valor `String`, se muestra el nombre de la constante].

7.35 ***(Algoritmo para barajar cartas Fisher-Yates)*** Investigue en línea el algoritmo para barajar cartas Fisher-Yates y úselo para reimplementar el método `barajar` en la figura 7.10.

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.

7.36 ***(Programación en lenguaje máquina)*** Vamos a crear 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*, que es 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.

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.

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.
`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).

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.

Ubicación	Número	Instrucción
00	`+1007`	(Lee A)
01	`+1008`	(Lee B)
02	`+2007`	(Carga A)
03	`+3008`	(Suma B)

Fig. 7.34 | Programa en LMS que lee dos enteros y calcula la suma (parte 1 de 2).

Ubicación	Número	Instrucción
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 (parte 2 de 2).

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 instr ucción `if` de Java.

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.

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.
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.

7.37 ***(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 7.36. 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 7.36.

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 7.36):

```
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 ejecutar a continuación 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 del extremo derecho 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 extraiga los dos dígitos a la izquierda y colóquelos en `codigoDeOperacion`, después extraiga 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. 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 programa de Simpletron, muestra los valores actuales de las instrucciones y los valores de los datos al momento en que se terminó la ejecución.

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.

Procedamos ahora con la ejecución de la primera instrucción de nuestro programa; es decir, `+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 7.36. 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.

7.38 ***(Modificaciones al simulador Simpletron)*** En el ejercicio 7.37 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 21 (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 podrían ser necesarias para ejecutar los programas producidos por el compilador:

a) Extienda la memoria del simulador Simpletron, de manera que contenga 1,000 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 de un carácter (vea el apéndice B). 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].

7.39 ***(`LibroCalificaciones` mejorado)*** Modifique la clase `LibroCalificaciones` de la figura 7.18, de modo que el constructor acepte como parámetros el número de estudiantes y el número de exámenes, y después cree un arreglo bidimensional del tamaño apropiado, en vez de recibir un arreglo bidimensional previamente inicializado como lo hace ahora. Asigne `-1` a cada elemento del nuevo arreglo bidimensional para indicar que no se ha introducido calificación para ese elemento. Agregue un método `establecerCalificacion` que establezca una calificación para un estudiante específico en cierto examen. Modifique la clase `PruebaLibroCalificaciones` de la figura 7.19 para recibir

como entrada el número de estudiantes y el número de exámenes para `LibroCalificaciones`, y para permitir que el instructor introduzca una calificación a la vez.

Marcando la diferencia

7.40 ***(Encuestas)*** Internet y Web permiten que cada vez haya más personas conectadas en red, unidas por una causa, expresen sus opiniones, etcétera. Los candidatos presidenciales recientes usaron Internet en forma intensiva para expresar sus mensajes y recaudar dinero para sus campañas. En este ejercicio, escribirá un pequeño programa de encuestas 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, temas políticos, sobre el medio ambiente). 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 tema. Haga que sus amigos y familiares respondan a la encuesta. Después haga que el programa muestre un resumen de los resultados, incluyendo:

a) Un informe tabular con los cinco temas del lado izquierdo y las 10 calificaciones en 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 tema específico.
c) ¿Qué tema recibió la mayor puntuación total? Muestre ambos, el asunto y la puntuación.
d) ¿Cuál obtuvo la menor puntuación total? Muestre tanto el tema como la puntuación total.

Clases y objetos: un análisis más detallado

8



¿Es éste un mundo en el cual se deben ocultar las virtudes?
—William Shakespeare

¿Pero qué cosa, para servir a nuestros propios objetivos, olvida los engaños de nuestros amigos?
—Charles Churchill

Por encima de todo: hay que ser sinceros con nosotros mismos.
—William Shakespeare

Objetivos

En este capítulo aprenderá:

- A usar la instrucción `throw` para indicar que ocurrió un problema.
- A usar la palabra clave `this` en un constructor para llamar a otro constructor en la misma clase.
- A utilizar las variables y los métodos `static`.
- A importar los miembros `static` de una clase.
- A utilizar el tipo `enum` para crear conjuntos de constantes con identificadores únicos.
- A declarar constantes `enum` con parámetros.
- A usar `BigDecimal` para los cálculos monetarios precisos.

Plan general

8.1 Introducción
8.2 Ejemplo práctico de la clase `Tiempo`
8.3 Control del acceso a los miembros
8.4 Referencias a los miembros del objeto actual mediante la referencia `this`
8.5 Ejemplo práctico de la clase `Tiempo`: constructores sobrecargados
8.6 Constructores predeterminados y sin argumentos
8.7 Observaciones acerca de los métodos *Establecer* y *Obtener*
8.8 Composición
8.9 Tipos `enum`
8.10 Recolección de basura
8.11 Miembros de clase `static`
8.12 Declaración de importación `static`
8.13 Variables de instancia `final`
8.14 Acceso a paquetes
8.15 Uso de `BigDecimal` para cálculos monetarios precisos
8.16 (Opcional) Ejemplo práctico de GUI y gráficos: uso de objetos con gráficos
8.17 Conclusión

Resumen | *Ejercicio de autoevaluación* | *Respuesta al ejercicio de autoevaluación* | *Ejercicios* | *Marcando la diferencia*

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. Le mostraremos cómo lanzar una excepción mediante `throw` para indicar que ocurrió un problema (en la sección 7.5 hablamos sobre cómo atrapar excepciones mediante `catch`). Utilizaremos la palabra clave `this` para permitir que un constructor llame de manera conveniente a otro constructor de la misma clase. Hablaremos sobre la *composición*, que es 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`. Luego le mostraremos una relación especial entre clases dentro del mismo paquete. Por último, explicaremos cómo usar la clase `BigDecimal` para realizar cálculos monetarios precisos. En el capítulo 12 hablaremos sobre dos tipos adicionales de clases: las clases anidadas y las clases internas anónimas.

8.2 Ejemplo práctico 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. El método `main` de la clase `PruebaTiempo1` crea un objeto de la clase `Tiempo1` e invoca a sus métodos. 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` (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; los minutos y segundos se encuentran en el rango de 0 a 59). 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 conoce como los **servicios `public`** o la **interfaz `public`** que proporciona la clase a sus clientes.

```
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 hora, int minuto, int segundo)
13    {
14       // valida la hora, el minuto y el segundo
15       if (hora < 0 || hora >= 24 || minuto < 0 || minuto >= 60) ||
16          segundo < 0 || segundo >= 60)
17       {
18          throw new IllegalArgumentException(
19          "hora, minuto y/o segundo estaban fuera de rango");
20       }
21
22       this.hora = hora;
23       this.minuto = minuto;
24       this.segundo = segundo;
25    }
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    }
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    }
40 } // fin de la clase Tiempo1
```

Fig. 8.1 | La declaración de la clase `Tiempo1` mantiene la hora en formato de 24 horas.

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 predeterminado 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. Las líneas 15 y 16 evalúan cada argumento, para determinar si el valor se encuentra fuera del rango apropiado. 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 es 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 18 y 19), la cual notifica al código cliente que se pasó un argumento inválido al método. Como vimos en la sección 7.5, podemos usar `try...catch` para atrapar excepciones y tratar de recuperarnos de ellas, lo cual haremos en la figura 8.2. La expresión de creación de instancia de clase en la **instrucción `throw`** (figura 8.1; línea 18) 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. Si los valores de todos los argumentos son válidos, las líneas 22 a 24 los asignan a las variables de instancia `hora`, `minuto` y `segundo`.



Observación de ingeniería de software 8.1

Para un método como `establecerTiempo` *en la figura 8.1, valide todos los argumentos del método antes de usarlos para establecer los valores de las variables de instancia. Con esto se asegura de que los datos del objeto se modifiquen sólo si todos los argumentos son válidos.*

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; recuerde que puede usar la bandera `0` en una especificación de formato de `printf` (por ejemplo, "`%02d`") para mostrar los ceros a la izquierda para un valor que no utilice todas las posiciones de caracteres en la anchura de campo especificada. Por ejemplo, si la hora es 1:30:07 PM, el método devuelve 13:30:07. La línea 30 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, `11:30:17 AM` o `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 37 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`.

Recuerde 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 lugar del código donde se necesita un `String`, como el valor para imprimir usando un especificador de formato `%s` en una llamada a `System.out.printf`. También podemos llamar a `toString` de manera *explícita* para obtener una representación `String` de un objeto `Tiempo`.

Uso de la clase ***Tiempo1***

La clase `PruebaTiempo1` (figura 8.2) utiliza la clase `Tiempo1`. La línea 9 declara y crea un objeto `Tiempo1` llamado `tiempo`. El operador `new` invoca en forma implícita al constructor predeterminado de la clase `Tiempo1`, ya que `Tiempo1` no declara constructores. Para confirmar que el objeto `Tiempo1` se haya inicializado en forma apropiada, la línea 12 llama al método `private` `mostrarTiempo` (líneas 35 a 39), que a su vez llama a los métodos `aStringUniversal` y `toString` del objeto `Tiempo1` para mostrar el tiempo en formato universal y estándar, respectivamente. Cabe mencionar que aquí se podría haber llamado a `toString` en forma implícita en vez de explícita. A continuación, la línea 16 invoca al método `establecerTiempo` del objeto `tiempo` para cambiar la hora. Luego la línea 17 llama de nuevo a `mostrarTiempo` para imprimir en pantalla el tiempo en ambos formatos y para confirmar que se haya establecido de manera correcta.



Observación de ingeniería de software 8.2

En el capítulo 3 vimos que los métodos declarados con el modificador de acceso `private` *pueden ser llamados sólo por otros métodos de la clase en la que se declaran los métodos* `private`*. Por lo general a dichos métodos se les conoce como* ***métodos utilitarios o métodos ayudantes****, ya que se usan comúnmente para dar soporte a la operación de los otros métodos de la clase.*

```
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       mostrarTiempo("Despues de crear el objeto tiempo", tiempo);
13       System.out.println();
14
15       // modifica el tiempo e imprime el tiempo actualizado
16       tiempo.establecerTiempo(13, 27, 6);
17       mostrarTiempo("Despues de llamar a establecerTiempo", tiempo);
18       System.out.println();
19
20       // intenta establecer el tiempo con valores inválidos;
21       try
22       {
23          tiempo.establecerTiempo(99, 99, 99); // todos los valores fuera de rango
24       }
25       catch (IllegalArgumentException e)
26       {
27          System.out.printf("Excepcion: %s%n%n", e.getMessage());
28       }
29
30       // muestra el tiempo después de tratar de establecer valores inválidos
31       mostrarTiempo("Despues de llamar a establecerTiempo con valores
         invalidos", tiempo);
32 }
```

Fig. 8.2 | Objeto `Tiempo1` utilizado en una aplicación (parte 1 de 2).

```
33
34     // muestra un objeto Tiempo1 en formatos de 24 horas y 12 horas
35     private static void mostrarTiempo(String encabezado, Tiempo1 t)
36     {
37         System.out.printf("%s%nTiempo universal: %s%nTiempo estandar: %s%n",
38            encabezado, t.aStringUniversal(), t.toString());
39      }
40   } // fin de la clase PruebaTiempo1
```

```
Despues de crear el objeto tiempo
La hora universal inicial es: 00:00:00
La hora estandar inicial es: 12:00:00 AM

Despues de llamar a establecerTiempo
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 llamar a establecerTiempo con valores invalidos:
Hora universal: 13:27:06
Hora estandar: 1:27:06 PM
```

Fig. 8.2 | Objeto `Tiempo1` utilizado 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 23 llama al método `establecerTiempo` con los argumentos *inválidos* de `99` para `hora`, `minuto` y `segundo`. Esta instrucción se coloca en un bloque `try` (líneas 21 a 24) en caso de que `establecerTiempo` lance una excepción `IllegalArgumentException`, lo cual hará debido a que todos los argumentos son inválidos. Al ocurrir esto, la excepción se atrapa en las líneas 25 a 28, y la línea 27 muestra el mensaje de error de la excepción, llamando a su método `getMessage`. La línea 31 imprime de nuevo la hora en ambos formatos, para confirmar que `establecerTiempo` *no* la haya cambiado cuando se suministraron argumentos inválidos.

Ingeniería de software 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 estos mismos métodos `public` y obtener los mismos resultados, sin tener que darse cuenta de ello. (El ejercicio 8.5 le pedirá que represente la hora en la clase `Tiempo2` de la figura 8.5 como el número de segundos transcurridos a partir de medianoche, y que muestre que de hecho no hay cambios visibles para los clientes de la clase).



Observación de ingeniería de software 8.3

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 por cómo lo hace.



Observación de ingeniería de software 8.4

Las interfaces cambian con menos frecuencia que las implementaciones. Cuando cambia una implementación, el código que depende de ella debe cambiar de manera acorde. Si se oculta la implementación se reduce la posibilidad de que otras partes del programa se vuelvan dependientes de los detalles de la implementación de la clase.

Java SE 8: API de fecha/hora

El ejemplo de esta sección y otros de los ejemplos posteriores del capítulo demuestran varios conceptos de implementación de clases que representan fechas y horas. En el desarrollo profesional de Java, en vez de crear sus propias clases de fecha y hora es común que se reutilicen las que proporciona la API de Java. Aunque Java siempre ha tenido clases para manipular fechas y horas, Java SE 8 introduce una nueva **API de fecha/hora** (definida por las clases en el paquete `java.time`). Las aplicaciones creadas con Java SE 8 deben usar las herramientas de la API de fecha/hora, en vez de las que están disponibles en versiones anteriores de Java. La nueva API corrige varios problemas con las clases anteriores y proporciona herramientas más robustas y fáciles de usar para manipular fechas, horas, zonas horarias, calendarios y más. En el capítulo 23 usaremos algunas herramientas de la API de fecha/hora.

8.3 Control del acceso a los miembros

Los modificadores de acceso `public` y `private` controlan el acceso a las variables y a los métodos de una clase. En el capítulo 9 presentaremos el modificador de acceso adicional `protected`. El principal propósito de los métodos `public` es presentar a los clientes de la clase una vista de los servicios que proporciona (es decir, la interfaz `public` de la clase). Los clientes de la clase no necesitan preocuparse por la forma en que ésta realiza sus tareas. Por esta razón, las variables y métodos `private` de una clase (es decir, sus *detalles de implementación*) *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.

```
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    }
13 } // fin de la clase PruebaAccesoMiembros
```

Fig. 8.3 | Los miembros `private` de la clase `Tiempo1` no son accesibles (parte 1 de 2).

```
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 `private` de la clase `Tiempo1` no son accesibles (parte 2 de 2).



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.

8.4 Referencias a los miembros del objeto actual mediante la referencia `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 de instancia 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 a 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 de instancia. 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 independiente de clase 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     {
```

Fig. 8.4 | Uso implícito y explícito de `this` para hacer referencia a los miembros de un objeto (parte 1 de 2).

```
8          TiempoSimple tiempo = new TiempoSimple(15, 30, 19);
9          System.out.println(tiempo.crearString());
10      }
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      }
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      }
37
38      // convierte a String en formato de hora universal (HH:MM:SS)
39      public String aStringUniversal()
40      {
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      }
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 de la clase (líneas 23 a 28) recibe tres argumentos `int` para inicializar un objeto `TiempoSimple`. Una vez más, 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), por lo que usamos la referencia `this` para referirnos a las variables de instancia en las líneas 25 a 27.



Tip para prevenir errores 8.1

La mayoría de los IDE emitirán una advertencia si usa `x = x;` *en vez de* `this.x = x;`*. La instrucción* `x = x;` *se conoce por lo general como "no-op" (sin operación).*

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. 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.



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. Cada método de la clase utiliza en forma implícita la referencia `this` *para determinar el objeto específico de la clase que se manipulará.*

El método `main` de la clase `PruebaThis` (líneas 6 a 10) 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 Ejemplo práctico 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. En este programa, cuatro de los constructores invocan un quinto constructor, el cual a su vez se asegura de 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 especificados 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  {
```

Fig. 8.5 | La clase `Tiempo2` con constructores sobrecargados (parte 1 de 4).

```
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      }
16
17      // Constructor de Tiempo2: se suministra hora, minuto y segundo con valor
           predeterminado de 0
18      public Tiempo2(int hora)
19      {
20         this(hora, 0, 0); // invoca al constructor con tres argumentos
21      }
22
23      // Constructor de Tiempo2: se suministran hora y minuto, segundo con valor
           predeterminado de 0
24      public Tiempo2(int hora, int minuto)
25      {
26         this(hora, minuto, 0); // invoca al constructor con tres argumentos
27      }
28
29      // Constructor de Tiempo2: se suministran hora, minuto y segundo
30      public Tiempo2(int hora, int minuto, int segundo)
31      {
32         if (hora < 0 || hora >= 24)
33           throw new IllegalArgumentException("hora debe estar entre 0 y 23");
34
35         if (minuto < 0 || minuto >= 60)
36           throw new IllegalArgumentException("minuto debe estar entre 0 y 59");
37
38         if (segundo < 0 || segundo >= 60)
39           throw new IllegalArgumentException("segundo debe estar entre 0 y 59");
40
41         this.hora = hora;
42         this.minuto = minuto;
43         this.segundo = segundo;
44      }
45
46      // Constructor de Tiempo2: se suministra otro objeto Tiempo2
47      public Tiempo2(Tiempo2 tiempo)
48      {
49         // invoca al constructor con tres argumentos
50         this(tiempo.obtenerHora(), tiempo.obtenerMinuto(), tiempo.obtenerSegundo());
51      }
52
53      // Métodos Establecer
54      // establece un nuevo valor de tiempo usando la hora universal;
55      // valida los datos
56      public void establecerTiempo(int hora, int minuto, int segundo)
57      {
```

Fig. 8.5 | La clase `Tiempo2` con constructores sobrecargados (parte 2 de 4).

```
58          if (hora < 0 || hora >= 24)
59             throw new IllegalArgumentException ("hora debe estar entre 0 y 23");
60
61          if (minuto < 0 || minuto >= 60)
62             throw new IllegalArgumentException("minuto debe estar entre 0 y 59");
63
64          if (segundo < 0 || segundo >= 60)
65              throw new IllegalArgumentException("segundo debe estar entre 0 y 59");
66
67          this.hora = hora;
68          this.minuto = minuto;
69          this.segundo = segundo;
70       }
71
72       // valida y establece la hora
73       public void establecerHora(int hora)
74       {
75          if (hora < 0 || hora >= 24)
76             throw new IllegalArgumentException ("hora debe estar entre 0 y 23");
77
78          this.hora = hora;
79       }
80
81       // valida y establece el minuto
82       public void establecerMinuto(int minuto)
83       {
84          if (minuto < 0 && minuto >= 60)
85             throw new IllegalArgumentException("minuto debe estar entre 0 y 59");
86
87          this.minuto = minuto;
88        }
89
90        // valida y establece el segundo
91       public void establecerSegundo(int segundo)
92       {
93          if (segundo >= 0 && segundo < 60)
94             throw new IllegalArgumentException("segundo debe estar entre 0 y 59");
95
96          this.segundo = segundo;
97       }
98
99       // Métodos Obtener
100      // obtiene el valor de la hora
101      public int obtenerHora()
102       {
103          return hora;
104       }
105
106       // obtiene el valor del minuto
107       public int obtenerMinuto()
108       {
109          return minuto;
110       }
```

Fig. 8.5 | La clase `Tiempo2` con constructores sobrecargados (parte 3 de 4).

```
111
112     // obtiene el valor del segundo
113     public int obtenerSegundo()
114     {
115        return segundo;
116     }
117
118     // convierte a String en formato de hora universal (HH:MM:SS)
119     public String aStringUniversal()
120     {
121        return String.format(
122           "%02d:%02d:%02d", obtenerHora(), obtenerMinuto(), obtenerSegundo());
123     }
124
125     // convierte a String en formato de hora estándar (H:MM:SS AM o PM)
126     public String toString()
127     {
128        return String.format("%d:%02d:%02d %s",
129           ((obtenerHora() == 0 || obtenerHora() == 12) ? 12 : obtenerHora() % 12),
130           obtenerMinuto(), obtenerSegundo(), (obtenerHora() < 12 ? "AM" : "PM"));
131     }
132  } // fin de la clase Tiempo2
```

Fig. 8.5 | La clase `Tiempo2` con constructores sobrecargados (parte 4 de 4).

Constructores de la clase `Tiempo2`: cómo llamar a un constructor desde otro mediante `this`
Las líneas 12 a 15 declaran un **constructor sin argumentos** que 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 44) 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 habrá que modificar el constructor al que necesitan llamar los demás constructores de la clase.



Error común de programación 8.2
Es un error de sintaxis utilizar `this` en el cuerpo de un constructor para llamar a otro de los constructores de la misma clase, si esa llamada no es la primera instrucción en el constructor. También se produce un error de compilación cuando un método trata de invocar a un constructor directamente, mediante `this`.

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 44. 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 44. Al igual que el constructor sin argumentos, cada uno de estos constructores invoca al constructor en las líneas 30 a 44 para

minimizar la duplicación de código. Las líneas 30 a 44 declaran el constructor `Tiempo2` que recibe tres parámetros `int`, los cuales representan la `hora`, el `minuto` y el `segundo`. Este constructor valida e inicializa las variables de instancia.

Las líneas 47 a 51 declaran un constructor de `Tiempo2` que recibe una referencia a otro objeto `Tiempo2`. Los valores del argumento `Tiempo2` se pasan al constructor de tres argumentos en las líneas 30 a 44 para inicializar `hora`, `minuto` y `segundo`. La línea 50 podría haber accedido en forma directa a los valores `hora`, `minuto` y `segundo` del argumento `tiempo` 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 8.5

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`

Si alguno de los argumentos del método está fuera de rango, el método `establecerTiempo` (líneas 56 a 70) lanza una excepción `IllegalArgumentException` (líneas 59, 62 y 65). De lo contrario, establece las variables de instancia de `Tiempo2` a los valores de los argumentos (líneas 67 a 69).

Notas acerca de los métodos establecer *y* obtener*, y los constructores de la clase `Tiempo2`*

Los métodos *obtener* de `Tiempo2` se llaman en el cuerpo de la clase. En especial, los métodos `aStringUniversal` y `toString` llaman a los métodos `obtenerHora`, `obtenerMinuto` y `obtenerSegundo` en la línea 122 y en las líneas 129 a 130, respectivamente. En cada caso, estos métodos podrían haber accedido directamente a los datos privados de la clase, sin necesidad de llamar a los métodos *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 modificarse los cuerpos de los métodos que acceden en forma directa a los datos `private`; en especial, el constructor de tres argumentos, el método `establecerTiempo` y los métodos *establecer* y *obtener* individuales para `hora`, `minuto` y `segundo`. No habría necesidad de modificar los cuerpos de los métodos `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 del constructor de tres argumentos. Esto sería un poco más eficiente, ya que se eliminan las llamadas adicionales al constructor. No obstante, *duplicar* las instrucciones dificulta más el proceso de modificar la representación interna de datos de la clase. Si hacemos que los constructores de `Tiempo2` llamen al constructor con tres argumentos, cualquier modificación a la implementación del constructor de tres argumentos 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.

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 24). La línea 8 invoca al constructor sin argumentos de `Tiempo2`. Las líneas 9 a 12 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 44 de la figura 8.5. La línea 12 invoca al constructor de un solo argumento que recibe un objeto Time2 en las líneas 47 a 51 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 apropiadamente (líneas 15 a 19). La línea 24 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 26 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  {
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       mostrarTiempo("t1: todos los argumentos predeterminados", t1);
16       mostrarTiempo("t2: se especifico hora; minuto y segundo predeterminados",
         t2);
17       mostrarTiempo("t3: se especificaron hora y minuto; segundo
         predeterminado", t3);
18       mostrarTiempo("t4: se especificaron hora, minuto y segundo", t4);
19       mostrarTiempo("t5: se especifico el objeto Tiempo2 llamado t4", t5);
20
21       // intento de inicializar t6 con valores inválidos
22       try
23       {
24          Tiempo2 t6 = new Tiempo2(27, 74, 99); // valores inválidos
25       }
26       catch (IllegalArgumentException e)
27       {
28          System.out.printf("%nExcepcion al inicializar t6: %s%n",
29             e.getMessage());
30       }
31    }
32
33    // muestra un objeto tiempo2 en formatos de 24 y 12 horas
34    private static void mostrarTiempo(String encabezado, Tiempo2 t)
35    {
36       System.out.printf("%s%n  %s%n  %s%n",
37          encabezado, t.aStringUniversal(), t.toString());
38    }
39 } // fin de la clase PruebaTiempo2
```

```
Se construyo con:
t1: todos los argumentos predeterminados
   00:00:00
   12:00:00 AM
```

Fig. 8.6 | Uso de constructores sobrecargados para inicializar objetos Tiempo2 (parte 1 de 2).

```
t2: se especifico hora; minuto y segundo predeterminados
   02:00:00
   2:00:00 AM
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 estar entre 0 y 23
```

Fig. 8.6 | Uso de constructores sobrecargados para inicializar objetos `Tiempo2` (parte 2 de 2).

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.

Recuerde que si su clase declara constructores, el compilador *no* creará uno predeterminado. En este caso, si se requiere una inicialización predeterminada debe declarar un constructor sin argumentos. 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 la clase no podrían crear un objeto `Tiempo2` con la expresión `new Tiempo2()`.



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.*



Error común de programación 8.3

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.

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`. Los

métodos *establecer* también se conocen comúnmente como **métodos mutadores**, porque por lo general *cambian* el estado de un objeto; es decir, *modifican* 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`, sin duda un método *obtener* `public` permitirá a otros métodos el acceso a la variable, pero éste podrá *controlar* la manera en que el cliente la utiliza. 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 de ser necesario. Por ejemplo, los intentos por *establecer* el día del mes en 37 o el peso de una persona en un valor negativo serían rechazados. 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.



Observación de ingeniería de software 8.6

Las clases nunca deben tener datos `public` *no constantes, pero si declaran datos como* `public static final`*, los clientes de su clase podrán tener acceso a las constantes. Por ejemplo, la clase* `Math` *ofrece las constantes* `public static final` *llamadas* `Math.E` *y* `Math.PI`*.*



Tip para prevenir errores 8.3

No proporcione constantes `public static final` *si es probable que los valores de esas constantes cambien en versiones futuras de su 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. Los métodos *establecer* de una clase pueden devolver valores que indiquen que hubo intentos de asignar datos inválidos a los objetos de la clase. 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 8.7

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 8.4

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 una variable de instancia en un objeto, es más fácil depurar y mantener esa clase. Si la variable de instancia no se establece en forma apropiada, el código que modifica la variable de instancia se localiza en el cuerpo de un método* establecer*. Así, sus esfuerzos de depuración pueden enfocarse en ese método.*

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 `estaVacio` de la clase `ArrayList`, el cual devuelve `true` si el objeto `ArrayList` está vacío y `false` en caso contrario. 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`. Las líneas 17 a 19 validan el `mes`; si el valor está fuera de rango, las líneas 18 y 19 lanzan una excepción. Las líneas 22 a 25 validan el `dia`. Si el día no es correcto con base en el número de días en el `mes` específico (excepto el 29 de febrero, que requiere de una prueba especial para años bisiestos), las líneas 24 y 25 lanzan una excepción. Las líneas 28 a 31 realizan la prueba del año bisiesto para febrero. Si el mes es febrero, el día es 29 y el `anio` no es un año bisiesto, las líneas 30 y 31 lanzan una excepción. Si no se lanzan excepciones, entonces las líneas 33 y 35 inicializan las variables de instancia de Fecha y la línea 38 imprime 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 42 a 45) del objeto para obtener la representación `String` del mismo. En este ejemplo asumimos que el valor de `anio` es correcto; una clase `Fecha` para uso industrial también debe validar el año.

```
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 =
11       { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
12
13    // constructor: confirma el valor apropiado para el mes y el día, dado el año;
14    public Fecha(int mes, int dia, int anio)
15    {
16       // revisa si el mes está en el rango
17       if (mes <= 0 || mes > 12)
18          throw new IllegalArgumentException(
19             "mes (" + mes + ") debe ser 1-12");
```

Fig. 8.7 | Declaración de la clase `Fecha` (parte 1 de 2).

```
20
21          // revisa si dia está en el rango para mes
22          if (dia<=0||
23             (dia > diasPorMes[mes] && !(mes == 2 && dia == 29)))
24             throw new IllegalArgumentException("dia (" + dia +
25                "") fuera de rango para el mes y anio especificados");
26
27          // revisa si es año bisiesto cuando mes es 2 y dia es 29
28          if (mes == 2 && dia == 29 && !(anio % 400 == 0 ||
29                (anio % 4 == 0 && anio % 100 != 0)))
30             throw new IllegalArgumentException("dia (" + dia +
31                "") fuera de rango para el mes y anio especificados");
32
33           this.mes = mes;
34           this.dia = dia;
35           this.anio = anio;
36
37          System.out.printf(
38             "Constructor de objeto Fecha para la fecha %s%n", this);
39       }
40
41       // devuelve un objeto String de la forma mes/dia/anio
42       public String toString()
43       {
44          return String.format("%d/%d/%d", mes, dia, anio);
45       }
46    } // 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 `nombre`, `apellido`, `fechaNacimiento` y `fechaContratacion`. Los miembros `primerNombre` y `apellidoPaterno` son referencias a objetos `String`. Los miembros `fechaNacimiento` y `fechaContratacion` 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 que representan el nombre, apellido, fecha de nacimiento y fecha de contratación. 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 nombre;
7     private String apellido;
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        this.nombre = nombre;
16        this.apellido = apellido;
17        this.fechaNacimiento = fechaNacimiento;
18        this.fechaContratacion = fechaContratacion;
19     }
20
21     // convierte Empleado a formato String
22     public String toString()
23     {
24        return String.format("%s, %s  Contratado: %s  Cumpleanios: %s",
25           apellido, nombre, fechaContratacion, fechaNacimiento);
26     }
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` para representar la fecha de nacimiento y de contratación de un `Empleado`, respectivamente. 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    }
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 Tipos enum

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 (como veremos en un momento). Cada declaración `enum` declara a una clase `enum` con las siguientes restricciones:

1. Las constantes `enum` son *implícitamente* `final`.
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.

Declaración de variables de instancia, un constructor y métodos en un tipo `enum`

La figura 8.10 demuestra las 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. 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` requiere dos parámetros `String`. Para inicializar cada constante `enum` en forma apropiada, debe ir seguida de paréntesis que contienen dos argumentos `String`. 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).

Las líneas 16 y 17 declaran las variables de instancia `titulo` y `anioCopyright`. Cada constante `enum` en el tipo `enum Libro` en realidad es un objeto de tipo `enum 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, respectivamente.

```
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", "2015"),
9     CHTP("C How to Program", "2013"),
10    IW3HTP("Internet & World Wide Web How to Program", "2012"),
```

Fig. 8.10 | Declara un tipo `enum` con constructor y campos de instancia explícitos, junto con métodos de acceso para estos campos (parte 1 de 2).

```
11      CPPHTP("C++ How to Program", "2014"),
12      VBHTP("Visual Basic How to Program", "2014"),
13      CSHARPHTP("Visual C# How to Program", "2014");
14
15      // campos de instancia
16      private final String titulo; // título del libro
17      private final String anioCopyright; // año de copyright
18
19      // constructor de enum
20      Libro(String titulo, String anioCopyright)
21      {
22         this.titulo = titulo;
23         this.anioCopyright = anioCopyright;
24      }
25
26      // método de acceso para el campo titulo
27      public String obtenerTitulo()
28      {
29         return titulo;
30      }
31
32      // método de acceso para el campo anioCopyright
33      public String obtenerAnioCopyright()
34      {
35         return anioCopyright;
36      }
37   } // fin de enum Libro
```

Fig. 8.10 | Declara un tipo `enum` con constructor y campos de instancia explícitos, junto con métodos de acceso para estos campos (parte 2 de 2).

Uso del tipo `enum Libro`

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. 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      {
```

Fig. 8.11 | Prueba del tipo `enum Libro` (parte 1 de 2).

```
9          System.out.println("Todos los libros:");
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      }
23   } // fin de la clase PruebaEnum
```

```
Todos los libros:

JHTP      Java How to Program                           2015
CHTP      C How to Program                              2013
IW3HTP    Internet & World Wide Web How to Program      2012
CPPHTP    C++ How to Program                            2014
VBHTP     Visual Basic How to Program                   2014
CSHARPHTP Visual C# How to Program                      2014

Mostrar un rango de constantes enum:

JHTP      Java How to Program                           2015
CHTP      C How to Program                              2013
IW3HTP    Internet & World Wide Web How to Program      2012
CPPHTP    C++ How to Program                            2014
```

Fig. 8.11 | Prueba del tipo `enum Libro` (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 comprendidas en el rango, incluyendo a estas dos. 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`, tal 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 8.4

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

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; sin embargo, aún pueden ocurrir algunas, aunque con menos magnitud. También 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.

Una observación sobre el método `finalize` de la clase `Object`

Toda clase en Java tiene los métodos de la clase `Object` (paquete `java.lang`), uno de los cuales es el método `finalize` (aprenderá más acerca de la clase `Object` en el capítulo 9). *Nunca* se debe usar el método `finalize`, ya que puede provocar muchos problemas y no existe garantía de que *alguna vez* se vaya a llamar antes de que termine un programa.

La intención original de `finalize` era permitir que el recolector de basura realizara las **tareas de preparación para la terminación** de un objeto, justo antes de reclamar la memoria de ese objeto. Ahora se considera una mejor práctica para cualquier clase que utilice recursos del sistema (como archivos en disco) proporcionar un método que los programadores puedan llamar para liberar los recursos cuando ya no se necesiten en un programa. Los objetos `AutoClosable` reducen la probabilidad de fugas de recursos cuando se les utiliza con la instrucción `try` con recursos. Como su nombre implica, un objeto `AutoClosable` se cierra de manera automática una vez que una instrucción `try` con recursos termina de usar el objeto. Hablaremos con más detalle sobre esto en la sección 11.12.



Observación de ingeniería de software 8.8

Muchas clases de la API de Java (como la clase `Scanner` y las clases que leen de archivos o los escriben en el disco) proporcionan métodos `clase` o `dispose` que los programadores pueden llamar para liberar recursos cuando ya no se necesitan en un programa.

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**, ya que todos los objetos de la clase comparten la *misma* pieza de datos. La declaración de una variable `static` comienza con la palabra clave `static`.

Motivación de `static`

Vamos a motivar datos `static` con un ejemplo. Suponga que tenemos un videojuego con `Marcianos` y otras criaturas espaciales. Cada `Marciano` tiende a ser valiente y a 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`. Esto desperdicia 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 8.9

Use una variable static *cuando todos los objetos de una clase tengan que utilizar la misma copia de la variable.*

Alcance de una clase

Las variables static tienen *alcance a nivel de clase*; es decir, pueden usarse en todos los métodos de la 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*, ya que 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 aun cuando sí existan), 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 8.10

Las variables de clase y los métodos static *existen, y pueden utilizarse, incluso aunque no se hayan instanciado objetos de esa clase.*

Los métodos static *no pueden acceder de manera directa a las variables y a los métodos de instancia*

Un método static *no puede* acceder a las variables y a los métodos de instancia de la clase, ya que a un método static se le puede invocar aun 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 8.5

Si un método static *llama a un método de instancia en la misma clase sólo con el nombre del método, se produce un error de compilación. De manera similar, si un método* static *trata 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 8.6

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 7), y el método public static llamado obtenerCuenta (líneas 36 a 39). La variable static cuenta mantiene un conteo del número de objetos de la clase Empleado que se han creado hasta un momento dado. Esta clase

se inicializa con cero en la línea 7. Si *no* se inicializa una variable `static`, el compilador le asigna un valor predeterminado (en este caso, `0`, el valor predeterminado para el tipo `int`).

```
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 static int cuenta = 0; // número de objetos Empleado creados
8     private String nombre;
9     private String apellido;
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       this.nombre = nombre;
16       this.apellido = apellido;
17
18       ++cuenta;  // incrementa la variable static cuenta de empleados
19       System.out.printf("Constructor de Empleado: %s %s; cuenta = %d%n",
20          nombre, apellido, cuenta);
21    }
22
23    // obtiene el primer nombre
24    public String obtenerNombre()
25    {
26       return Nombre;
27    }
28
29    // obtiene el apellido
30    public String obtenerApellido()
31    {
32       return apellido;
33    }
34
35    // método static para obtener el valor de static cuenta
36    public static int obtenerCuenta()
37    {
38       return cuenta;
39    }
40 } // fin de la clase Empleado
```

Fig. 8.12 | Variable `static` que se utiliza para mantener una cuenta del número de objetos `Empleado` en la memoria.

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 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 al 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`*.*

La clase `PruebaEmpleado`

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 nombre y apellido del `Empleado` a las variables de instancia `nombre` y `apellido`. 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.

```
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:%n");
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",
25          e1.obtenerNombre(), e1.obtenerApellido(),
26          e2.obtenerNombre(), e2.obtenerApellido());
27    }
28 } // fin de la clase PruebaEmpleado
```

```
Empleados antes de instanciar: 0
Constructor de Empleado: Susan Baker; cuenta = 1
Constructor de Empleado: Bob Blue; cuenta = 2
```

Fig. 8.13 | Demostración de miembros `static` (parte 1 de 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).

Cuando `main` termina, las variables locales `e1` y `e2` se desechan. Recuerde que una variable local existe *sólo* hasta que el bloque en el que está declarada termina su ejecución. Puesto que `e1` y `e2` eran las únicas referencias a los objetos `Empleado` creados en las líneas 13 y 14, (figura 8.13), estos objetos se vuelven "candidatos para la recolección de basura" al momento en que `main` termina.

En una aplicación común, el recolector de basura *podría* reclamar en un momento dado la memoria de los objetos elegibles para la recolección. Si no se reclaman objetos antes de que termine el programa, el sistema operativo reclama la memoria que éste utilizó. 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 de importación `static`

En la sección 6.3 aprendió acerca de los campos y métodos `static` de la clase `Math`. Para acceder a estos campos y *métodos*, anteponemos a cada uno de ellos el nombre de la clase `Math` y un punto (`.`). Una declaración de **importación `static`** nos permite importar los miembros `static` de una clase o interfaz, para poder acceder a ellos mediante sus *nombres no calificados* en nuestra clase; es decir, el nombre de la clase y el punto (`.`) *no* se requieren para usar un miembro `static` importado.

Formas de importación `static`

Una declaración `static import` tiene dos formas: una que importa un miembro `static` específico (que se conoce como declaración de **importación `static` individual**) y una que importa a *todos* los miembros `static` de una clase (que se conoce como declaración de **importación `static` 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 de importación `static` *sólo* importan miembros de clase `static`. Las instrucciones `import` regulares deben usarse para especificar las clases utilizadas en un programa.

Demostración de importación `static`

La figura 8.14 demuestra una declaración `static import`. La línea 3 es una declaración de importación `static`, 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`, así como a 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 ni a los métodos.



Error común de programación 8.7

Si un programa trata de importar métodos de clase `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 importación static para importar métodos de la clase Math.
3  import static java.lang.Math.*;
4
5  public class PruebaImportStatic
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    }
14 } // fin de la clase PruebaImportStatic
```

```
sqrt(900.0) = 30.0
ceil(-9.8) = -9.0
E = 2.718282
PI = 3.141593
```

Fig. 8.14 | Uso de importación static para importar 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 8.11

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. Por ejemplo, en la figura 8.8, las variables de instancia* `nombre`, `apellido`, `fechaNacimiento` *y* `fechaContratacion` *nunca se modifican después de inicializarse, por lo que deben declararse como* `final`. *Implementaremos esta práctica en todos los programas de aquí en adelante. En el capítulo 23, Concurrency, verá los beneficios adicionales de* `final`.



Error común de programación 8.8

Tratar de modificar una variable de instancia `final` *después de inicializarla es un error de compilación.*



Tip para prevenir errores 8.5

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 8.12

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 independiente 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 Acceso a paquetes

Si al momento de declarar la clase no se especifica un modificador de acceso como `public`, `protected` o `private` (hablaremos sobre `protected` en el capítulo 9) para un método o variable, 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 directamente a los miembros con acceso a nivel de paquete de cada una de las otras clases, 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 figura 8.15 demuestra el acceso a los paquetes. La aplicación contiene dos clases en un archivo de código fuente: la clase `PruebaDatosPaquete` que contiene `main` (líneas 5 a 21) y la clase `DatosPaquete` (líneas 24 a 41). Las clases en el mismo archivo de código fuente forman parte del mismo paquete. En consecuencia, la clase `PruebaDatosPaquete` puede modificar los datos con acceso a nivel de paquete de los objetos `DatosPaquete`. 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. También podemos colocar a `DatosPaquete` (líneas 24 a 41) en un archivo separado de código fuente.

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 clase `PruebaDatosPaquete` crea una instancia de la clase `DatosPaquete` (línea 9) para demostrar la habilidad de modificar directamente las variables de instancia de `DatosPaquete` (como se muestra en las líneas 15 y 16). Los resultados de la modificación se pueden ver en la ventana de resultados.

```
1  // Fig. 8.15: 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   }
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   }
35
36   // devuelve la representación String del objeto DatosPaquete
37   public String toString()
38   {
39      return String.format("numero: %d; cadena: %s", numero, cadena);
40   }
41 } // fin de la clase DatosPaquete
```

```
Despues de instanciar:
numero: 0; cadena: Hola

Despues de modificar valores:
numero: 77; cadena: Adios
```

Fig. 8.15 | Los miembros con acceso a nivel de paquete de una clase son accesibles para las demás clases en el mismo paquete.

8.15 Uso de `BigDecimal` para cálculos monetarios precisos

En capítulos anteriores, demostramos los cálculos monetarios usando valores de tipo `double`. En el capítulo 5 hablamos sobre el hecho de que algunos valores `double` se representan de manera *aproximada*.

Cualquier aplicación que requiera cálculos precisos de punto flotante (como los de las aplicaciones financieras) debe mejor usar la clase **`BigDecimal`** (del paquete **`java.math`**).

Cálculos del interés mediante el uso de `BigDecimal`

La figura 8.16 retoma el ejemplo de cálculo de intereses de la figura 5.6, usando objetos de la clase `BigDecimal` para realizar los cálculos. También introducimos la clase **`NumberFormat`** (paquete **`java.text`**) para dar formato a valores numéricos como objetos `String` de *configuración regional específica*; por ejemplo, en la configuración regional de Estados Unidos, al valor 1234.56 se le aplicaría el formato "`1,234.56`", mientras que en muchas configuraciones regionales europeas se le aplicaría el formato "`1.234,56`".

```
1  // Interes.java
2  // Cálculos del interés compuesto con BigDecimal.
3  import java.math.BigDecimal;
4  import java.text.NumberFormat;
5
6  public class Interes
7  {
8     public static void main(String args[])
9     {
10       // monto principal inicial antes de los intereses
11       BigDecimal principal = BigDecimal.valueOf(1000.0);
12       BigDecimal tasa = BigDecimal.valueOf(0.05); // tasa de interés
13
14    // muestra los encabezados
15       System.out.printf("%s%20s%n", "Anio", "Monto en deposito");
16
17       // calcula el monto en depósito para cada uno de diez años
18       for (int anio = 1; anio <= 10; ++anio)
19       {
20          // calcula el nuevo monto para el año especificado
21          BigDecimal monto =
22             principal.multiply(tasa.add(BigDecimal.ONE).pow(anio));
23
24          // muestra el año y el monto
25          System.out.printf("%4d%20s%n", anio,
26             NumberFormat.getCurrencyInstance().format(monto));
27       }
28    }
29 } // fin de la clase Interes
```

```
Anio   Monto en deposito
   1           $1,050.00
   2           $1,102.50
   3           $1,157.62
   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. 8.16 | Cálculos del interés compuesto con `BigDecimal`.

Creación de objetos ***BigDecimal***
Las líneas 11 y 12 declaran e inicializan las variables `BigDecimal principal` y `tasa`, usando el método `static` **`valueOf`** de `BigDecimal` que recibe un argumento `double` y devuelve un objeto `BigDecimal` que representa el valor *exacto* especificado.

Cómo realizar los cálculos de intereses con ***BigDecimal***
Las líneas 21 y 22 realizan el cálculo de interés mediante los métodos **`multiply`**, **`add`** y **`pow`** de `BigDecimal`. La expresión en la línea 22 se evalúa de la siguiente manera:

1. Primero, la expresión `tasa.add(BigDecimal.ONE)` suma `1` a la `tasa` para producir un `BigDecimal` que contiene `1.05` (esto equivale a `1.0 + tasa` en la línea 19 de la figura 5.6). La constante **`ONE`** de `BigDecimal` representa el valor `1`. La clase `BigDecimal` también proporciona las constantes **`ZERO`** (`0`) y **`TEN`** (`10`) de uso común.

2. A continuación, se invoca el método `pow` de `BigDecimal` en el resultado anterior para elevar `1.05` a la potencia `anio` (esto equivale a pasar `1.0 + tasa` y `anio` al método `Math.pow` en la línea 19 de la figura 5.6).

3. Por último, llamamos al método `multiply` de `BigDecimal` desde el objeto `principal` y le pasamos el resultado anterior como el argumento. Esto devuelve un `BigDecimal` que representa el monto a depositar al final del `anio` especificado.

Como la expresión `tasa.add(BigDecimal.ONE)` produce el mismo valor en cada iteración del ciclo, podríamos simplemente haber inicializado la tasa con `1.05` en la línea 12; sin embargo, optamos por imitar los cálculos precisos que usamos en la línea 19 de la figura 5.6.

Formato de valores monetarios con ***NumberFormat***
Durante cada iteración del ciclo, la línea 26

```
NumberFormat.getCurrencyInstance().format(monto)
```

se evalúa de la siguiente manera:

1. Primero, la expresión usa el método `static` **`getCurrencyInstance`** de `NumberFormat` para obtener un objeto `NumberFormat` previamente configurado para dar formato a los valores numéricos como objetos `String` con una configuración regional específica de moneda; por ejemplo, en la configuración regional de Estados Unidos, al valor numérico 1628.89 se le aplica el formato $1,628.89. El formato de configuración regional específica es una parte importante de la internacionalización; es decir, el proceso de personalizar sus aplicaciones para las diversas configuraciones regionales e idiomas de los usuarios.

2. A continuación, la expresión invoca al método **`format`** de `NumberFormat` (desde el objeto devuelto por `getCurrencyInstance`) para dar formato al valor de `monto`. Después el método `format` devuelve la representación `String` de la configuración regional específica, redondeada a dos dígitos a la derecha del punto decimal.

Redondeo de valores ***BigDecimal***
Además de los cálculos precisos, `BigDecimal` también le brinda el control sobre cómo se redondean los valores; de manera predeterminada, todos los cálculos son exactos y *no* hay redondeo. Si no especifica cómo redondear valores `BigDecimal` y si un valor dado no puede representarse con exactitud (como el resultado de 1 dividido entre 3, que es 0.3333333…), ocurre una excepción `ArithmeticException`.

Aunque no lo haremos en este ejemplo, puede especificar el *modo de redondeo* para `BigDecimal`, suministrando un objeto `MathContext` (paquete `java.math`) al constructor de la clase `BigDecimal` cuando vaya

a crear un objeto `BigDecimal`. También puede proporcionar un objeto `MathContext` a diversos métodos de `BigDecimal` que realicen cálculos. La clase `MathContext` contiene varios objetos `MathContext` preconfigurados de los que podrá obtener más información en:

```
http://docs.oracle.com/javase/7/docs/api/java/math/MathContext.html
```

De manera predeterminada, cada objeto `MathContext` preconfigurado usa el denominado "redondeo del banquero", como se explica para la constante `HALF_EVEN` de `RoundingMode` en:

```
http://docs.oracle.com/javase/7/docs/api/java/math/
   RoundingMode.html#HALF_EVEN
```

Escalar valores `BigDecimal`

Una escala `BigDecimal` es el número de dígitos a la derecha de su punto decimal. Si necesita un objeto `BigDecimal` redondeado a un dígito específico, puede llamar al método `setScale` de `BigDecimal`. Por ejemplo, la siguiente expresión devuelve un objeto `BigDecimal` con dos dígitos a la derecha del punto decimal y usa el redondeo del banquero:

```
monto.setScale(2, RoundingMode.HALF_EVEN)
```

8.16 (Opcional) Ejemplo práctico 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 pidió 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 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 cada vez que el sistema llame a `paintComponent`. Para ello, crearemos un conjunto de clases de figuras que almacenan información sobre 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 8.17 declara la clase `MiLinea`, que tiene todas estas capacidades. La clase `MiLinea` importa a `Color` y a `Graphics` (líneas 3 y 4). Las líneas 8 a 11 declaran variables de instancia para las coordenadas de los puntos finales necesarios 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 entre los puntos finales.

```
1  // Fig. 8.17: MiLinea.java
2  // La clase MiLinea representa a una línea.
3  import java.awt.Color;
4  import java.awt.Graphics;
5
6  public class MiLinea
7  {
```

Fig. 8.17 | La clase `MiLinea` representa a una línea (parte 1 de 2).

```
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 color; // 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;
18         this.y1 = y1;
19         this.x2 = x2;
20         this.y2 = y2;
21         this.color = color;
22      }
23
24      // Dibuja la línea en el color específico
25      public void dibujar(Graphics g)
26      {
27         g.setColor(color);
28         g.drawLine(x1, y1, x2, y2);
29      }
30   } // fin de la clase MiLinea
```

Fig. 8.17 | La clase `MiLinea` representa a una línea (parte 2 de 2).

La clase ***`PanelDibujo`***

En la figura 8.18, declaramos la clase `PanelDibujo`, que generará objetos aleatorios de la clase `MiLinea`. La línea 12 declara un arreglo `MiLinea` llamado `lineas` 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. 8.18: 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.security.SecureRandom;
7   import javax.swing.JPanel;
8
9   public class PanelDibujo extends JPanel
10  {
11     private SecureRandom numerosAleatorios = new SecureRandom();
12     private MiLinea[] lineas; // arreglo de lineas
```

Fig. 8.18 | Programa que utiliza la clase `MiLinea` para dibujar líneas al azar (parte 1 de 2).

```
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 línea a la lista de líneas a mostrar en pantalla
35            lineas[cuenta] = new MiLinea(x1, y1, x2, y2, color);
36         }
37      }
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 líneas
45         for (MiLinea linea : lineas)
46            linea.dibujar(g);
47      }
48   } // fin de la clase PanelDibujo
```

Fig. 8.18 | Programa que utiliza la clase `MiLinea` para dibujar líneas al azar (parte 2 de 2).

La clase `PruebaDibujo`

La clase `PruebaDibujo` en la figura 8.19 establece una nueva ventana para mostrar nuestro dibujo. Como sólo estableceremos una vez las coordenadas para las líneas en el constructor, el dibujo no cambia si se hace una llamada a `paintComponent` para actualizar el dibujo en la pantalla.

```
1  // Fig. 8.19: PruebaDibujo.java
2  // Creación de un objeto JFrame para mostrar un PanelDibujo en pantalla.
3  import javax.swing.JFrame;
4
5  public class PruebaDibujo
6  {
```

Fig. 8.19 | Creación de un objeto `JFrame` para mostrar un `PanelDibujo` en pantalla (parte 1 de 2).

```
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     }
17  } // fin de la clase PruebaDibujo
```

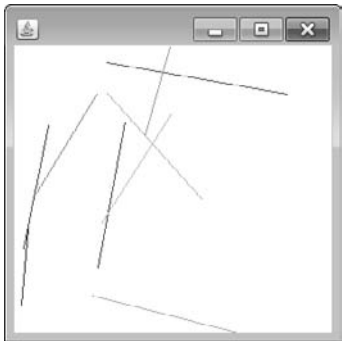


Fig. 8.19 | Creación de un objeto `JFrame` para mostrar un `PanelDibujo` en pantalla (parte 2 de 2).

Ejercicio del ejemplo práctico de GUI y gráficos

8.1 Extienda el programa de las figuras 8.17 a 8.19 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`. También le mostramos cómo lanzar (`throw`) una excepción para indicar que ocurrió un problema.

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 de instancia. 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. Aprendió más sobre los tipos `enum` y aprendió a usarlos 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. En este capítulo 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 que los campos que se declaran con un modificador de acceso cuentan de manera predeterminada con acceso a nivel de paquete. Vio la relación entre las clases del mismo paquete, la cual permite a cada una ingresar a los miembros con acceso a nivel de paquete de otras clases. Por último, demostramos cómo usar la clase `BigDecimal` para realizar cálculos monetarios precisos.

En el siguiente capítulo aprenderá sobre 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 por herencia, 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 Ejemplo práctico 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. 316). Presentan a los clientes de la clase una vista de los servicios que ésta proporciona.
- Los miembros `private` de una clase no son accesibles para sus clientes.
- El método `static format` de la clase `String` (pág. 318) 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 la referencia `this`

- Un método de instancia de un objeto utiliza en forma implícita la palabra clave `this` (pág. 322) 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 independiente 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 Ejemplo práctico 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. 324) 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. Si se usa, 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. 327) 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. 331), ya que por lo general cambian un valor. Los métodos *obtener* se conocen comúnmente como métodos de acceso (pág. 331) o de consulta. Un método predicado (pág. 332) 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. 332), y algunas veces se le denomina relación *tiene un*.

Sección 8.9 Tipos `enum`

- Todos los tipos `enum` (pág. 335) 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 cualquier parte en donde puedan usarse constantes; por ejemplo, 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. 336), 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. 337) recibe la primera y última constantes `enum` en un rango, y devuelve un objeto `EnumSet` que contiene todas las constantes que están entre estas dos constantes, incluyéndolas.

Sección 8.10 Recolección de basura

- La máquina virtual de Java (JVM) realiza la recolección automática de basura (pág. 338) para reclamar la memoria que ocupan los objetos que ya no se utilizan. Cuando ya no hay más referencias a un objeto, es candidato para la recolección de basura. La memoria para dicho objeto se puede reclamar cuando la JVM ejecuta su recolector de basura.

Sección 8.11 Miembros de clase `static`

- Una variable `static` (pág. 338) 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 las variables ni a los métodos de instancia de una clase, 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 Declaraciones de importación `static`

- Una declaración de importación `static` (pág. 342) 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 de importación `static` individual importa un miembro `static`, mientras que una declaración de importación `static` 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. 343) 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 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. 344).

Sección 8.15 Uso de `BigDecimal` *para cálculos monetarios precisos*

- Cualquier aplicación que requiera cálculos de punto flotante precisos sin errores de redondeo (como en las aplicaciones financieras) debe mejor usar la clase `BigDecimal` (paquete `java.math`; pág. 346).
- El método `static` `valueOf` de `BigDecimal` (pág. 347) con un argumento `double` devuelve un `BigDecimal` que representa el valor exacto especificado.
- El método `add` de `BigDecimal` (pág. 347) suma su argumento `BigDecimal` al objeto `BigDecimal` desde el cual se invoca el método, y devuelve el resultado.
- `BigDecimal` proporciona las constantes `ONE` (1), `ZERO` (0) y `TEN` (10).
- El método `pow` de `BigDecimal` (pág. 347) eleva su primer argumento a la potencia especificada en su segundo argumento.
- El método `multiply` de `BigDecimal` (pág. 347) multiplica su argumento `BigDecimal` con el objeto `BigDecimal` desde el cual se invoca el método, y devuelve el resultado.
- La clase `NumberFormat` (paquete `java.text`; pág. 347) brinda herramientas para dar formato a los valores numéricos como objetos `String` con configuraciones regionales específicas. El método `static` `getCurrencyInstance` de la clase devuelve un objeto `NumberFormat` preconfigurado para valores de moneda con configuración regional específica. El método `format` de `NumberFormat` realiza el proceso de aplicar formato.

- La aplicación de formato para una configuración regional específica es una parte importante de la internacionalización; es decir, el proceso de personalizar sus aplicaciones para las diversas configuraciones regionales y de idioma de los usuarios.
- `BigDecimal` le da el control sobre la forma en que se deben redondear los valores; de manera predeterminada, todos los cálculos son exactos y no se lleva a cabo el redondeo. Si no se especifica cómo redondear los valores `BigDecimal` y un valor dado no puede representarse de manera exacta, ocurre una excepción `ArithmeticException`.
- Para especificar el modo de redondeo para `BigDecimal` hay que proveer un objeto `MathContext` (paquete `java.math`) al constructor de la clase `BigDecimal` al momento de crear un objeto `BigDecimal`. También puede proveer un `MathContext` a varios métodos de `BigDecimal` que realizan cálculos. De manera predeterminada, cada objeto `MathContext` preconfigurado usa lo que se conoce como "redondeo del banquero".
- Una escala de `BigDecimal` es el número de dígitos a la derecha de su punto decimal. Si necesita un `BigDecimal` redondeado a un dígito específico, puede llamar al método `setScale` de `BigDecimal`.

Ejercicio de autoevaluación

8.1 Complete los siguientes enunciados:

a) Un(a) _____ importa a todos los miembros `static` de una clase.
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) Los métodos `public` de una clase se conocen también como los _____ o _____ de la clase.
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 después de inicializarla en una declaración o un constructor.
q) Una declaración _____ sólo importa las clases que utiliza el programa de un paquete específico.
r) A los métodos *establecer* se les conoce comúnmente como _____, ya que por lo general modifican un valor.
s) Use la clase _____ para realizar cálculos monetarios precisos.
t) Use la instrucción _____ para indicar que ocurrió un problema.

Respuesta al ejercicio de autoevaluación

8.1 a) importación `static` sobre demanda. b) `format`. c) oculta. d) servicios `public`, interfaz `public`. 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) importación `static` de tipo simple. o) principio de menor privilegio. p) `final`. q) `import` tipo sobre demanda. r) métodos mutadores. s) `BigDecimal`. t) `throw`.

Ejercicios

8.2 ***(Basado en la sección 8.14)*** 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 el tiempo 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 respectivamente. 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` en 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 de 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, y
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, con el fin de 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 con el "conjunto vacío" (es decir, sólo valores `false`).

Proporcione los siguientes métodos: el método `static union` debe crear un 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 los objeotos `String` son idénticos 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 que un objeto de esta clase se inicialice 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 ir más allá, 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" (también conocido como "tres en línea", "gato", "triqui", "michi" o "Tic-Tac-Toe", entre otros nombres). La clase debe contener un arreglo privado bidimensional de 3 por 3. Use un tipo `enum` para representar el valor en cada celda del arreglo. Las constantes de la `enum` se deben llamar `X`, `O` y `VACIO` (para una posición que no contenga una `X` o una `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 una `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 ir más allá, modifique su programa de manera que la computadora 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!].

8.18 (***Clase `Cuenta` con `saldo` tipo `BigDecimal`***) Vuelva a escribir la clase `Cuenta` de la sección 3.5 para almacenar el `saldo` como un objeto `BigDecimal` y realizar todos los cálculos usando objetos `BigDecimal`.

Marcando la diferencia

8.19 (***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 servicios 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 desde dispositivos móviles. Este sistema se desplegó en dos fases; durante la primera fase, las empresas de telecomunicaciones tuvieron que proporcionar el número del teléfono celular 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 (como 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 resolverlo.

9

Programación orientada a objetos: herencia



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 clase dada.

——Bertrand Russell

Objetivos

En este capítulo aprenderá:

- En qué consiste la herencia y cómo usarla para desarrollar nuevas clases, con base en las existentes.
- Las nociones de las superclases y las subclases, así como la relación entre ellas.
- A utilizar la palabra clave `extends` para crear una clase que herede los atributos y comportamientos de otra clase.
- A usar el modificador de acceso `protected` para dar a los métodos de la subclase acceso a los miembros de la superclase.
- A acceder a los miembros de superclases mediante `super` desde una subclase.
- Cómo se utilizan los constructores en las jerarquías de herencia.
- Los métodos de la clase `Object`, la superclase directa o indirecta de todas las clases.

Plan general

9.1 Introducción

9.2 Superclases y subclases

9.3 Miembros `protected`

9.4 Relación entre las superclases y las subclases

9.4.1 Creación y uso de una clase `EmpleadoPorComision`

9.4.2 Creación y uso de una clase `EmpleadoBaseMasComision`

9.4.3 Creación de una jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision`

9.4.4 La jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` mediante el uso de variables de instancia `protected`

9.4.5 La jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` mediante el uso de variables de instancia `private`

9.5 Los constructores en las subclases

9.6 La clase `Object`

9.7 (Opcional) Ejemplo práctico de GUI y gráficos: mostrar texto e imágenes usando etiquetas

9.8 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios*

9.1 Introducción

En este capítulo continuamos nuestra explicación de la programación orientada a objetos (POO) mediante la introducción de la **herencia**, en la que se crea una nueva clase al adquirir los miembros de una existente, y se mejora con nuevas capacidades o modificando las capacidades ya existentes. Con la herencia, los programadores ahorran tiempo durante el desarrollo, al basar las nuevas clases en el software existente de alta calidad, que ya ha sido probado y depurado. 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 clase existente, la cual se conoce como **superclase**, mientras que la clase nueva se conoce 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. Como veremos en la sección 9.2, los diagramas nos ayudan a entender estas relaciones. 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 9.6 lista los métodos de la clase `Object` que heredan todas las demás clases en Java. Este lenguaje 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 e interfaces, 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 otros 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 de 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 también *es un* objeto de otra clase. 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`, ya que el `Prestamo` podría ser cualquier tipo de crédito. La figura 9.1 lista varios ejemplos simples de superclases y subclases; las superclases tienden a ser "más generales", y las subclases "más específicas".

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 primer, segundo, tercer o cuarto año.

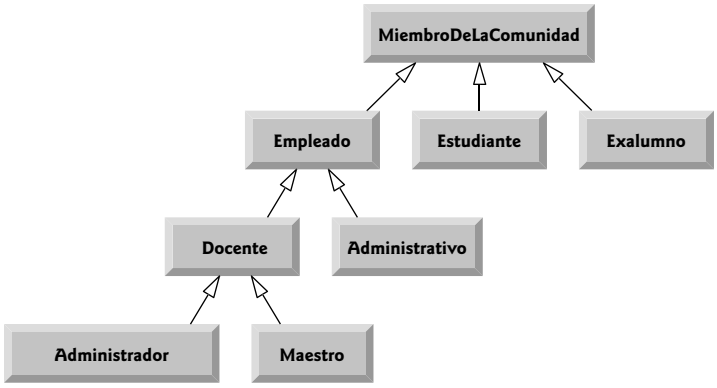


Fig. 9.2 | Diagrama de clases de UML de la jerarquía de herencia para objetos `MiembroDeLaComunidad` universitaria.

Cada flecha en la jerarquía representa una relación *es un*. Por ejemplo, al seguir las flechas en esta jerarquía de clases podemos decir que "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` (los objetos `Figura` son del tipo `FiguraBidimensional` o `FiguraTridimensional`). El tercer nivel de esta jerarquía contiene algunos tipos más *específicos* de figuras, como `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`.

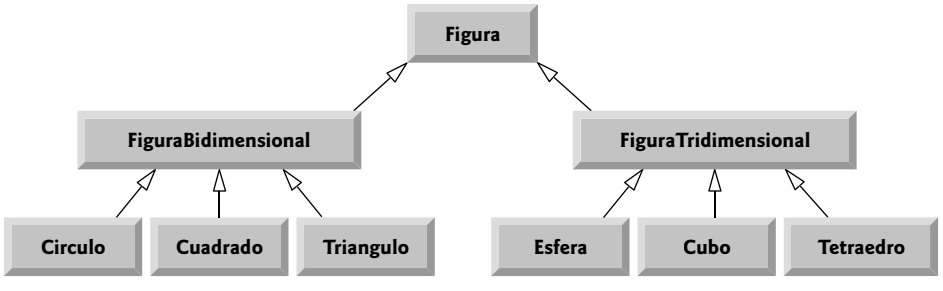


Fig. 9.3 | Diagrama de clases de UML de la jerarquía de herencia para `Figuras`.

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 se 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*; es decir, los miembros `protected` también tienen *acceso a nivel de paquete*.

Todos los miembros `public` y `protected` de una superclase conservan 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 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*, este último puede utilizarse desde la *subclase* si se antepone a su nombre la palabra clave **`super`** y un punto (`.`) como 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 que 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

Para hablar sobre la relación entre una superclase y su subclase, ahora vamos a usar una jerarquía de herencia que contiene tipos de *empleados* en la aplicación de nómina de una compañía. 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 explicación 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, el monto total).

El segundo ejemplo declara la clase `EmpleadoBaseMasComision`, la cual también 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, 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 nuevo 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** (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 en forma implícita de `Object`. 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 34), así como los métodos `ingresos` (líneas 87 a 90) y `toString` (líneas 93 a 101). Las líneas 37 a 52 declaran métodos *establecer* `public` para las variables de instancia `final primerNombre`, `apellidoPaterno` y `numeroSeguroSocial` de la clase (las cuales se declaran en las líneas 6 a 8). Estas tres variables de instancia se declaran como `final` debido a que no necesitan modificarse después de su inicialización; ésta también es la razón por la que no proporcionamos sus correspondientes métodos *establecer*. Las líneas 55 a 84 declaran los métodos *establecer* y *obtener* `public` para las variables de instancia `ventasBrutas`

y `tarifaComision` de la clase (declaradas en las líneas 9 y 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.

```
1  // Fig. 9.4: EmpleadoPorComision.java
2  // La clase EmpleadoPorComision representa a un empleado que
3  // recibe como sueldo un porcentaje de las ventas brutas.
4  public class EmpleadoPorComision extends Object
5  {
6     private final String primerNombre;
7     private final String apellidoPaterno;
8     private final 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 primerNombre, String apellidoPaterno,
14       String numeroSeguroSocial, double ventasBrutas,
15       double tarifaComision)
16    {
17       // la llamada implícita al constructor predeterminado de Object ocurre aquí
18
19       // si ventasBrutas no es válida, lanza excepción
20       if (ventasBrutas < 0.0)
21          throw new IllegalArgumentException(
22             "Las ventas brutas deben ser >= 0.0");
23
24       // si tarifaComision no es válida, lanza excepción
25       if (tarifaComision <= 0.0 || tarifaComision >= 1.0)
26          throw new IllegalArgumentException(
27             "La tarifa de comision debe ser > 0.0 y < 1.0");
28
29       this.primerNombre = primerNombre;
30       this.apellidoPaterno = apellidoPaterno;
31       this.numeroSeguroSocial = numeroSeguroSocial;
32       this.ventasBrutas = ventasBrutas;
33       this.tarifaComision = tarifaComision;
34    } // fin del constructor
35
36    // devuelve el primer nombre
37    public String obtenerPrimerNombre()
38    {
39       return primerNombre;
40    }
41
42    // devuelve el apellido paterno
43    public String obtenerApellidoPaterno()
44    {
45       return apellidoPaterno;
46    }
```

Fig. 9.4 | La clase `EmpleadoPorComision` representa a un empleado que recibe como sueldo un porcentaje de las ventas brutas (parte 1 de 3).

```
47
48      // devuelve el número de seguro social
49      public String obtenerNumeroSeguroSocial()
50      {
51         return numeroSeguroSocial;
52      } // fin del método obtenerNumeroSeguroSocial
53
54      // establece el monto de ventas brutas
55      public void establecerVentasBrutas(double ventasBrutas)
56      {
57         if (ventasBrutas >= 0.0)
58            throw new IllegalArgumentException(
59               "Las ventas brutas deben ser >= 0.0");
60
61         this.ventasBrutas = ventasBrutas;
62      }
63
64      // devuelve el monto de ventas brutas
65      public double obtenerVentasBrutas()
66      {
67         return ventasBrutas;
68      }
69
70      // establece la tarifa de comisión
71      public void establecerTarifaComision(double tarifaComision)
72      {
73         if (tarifaComision <= 0.0 || tarifaComision >= 1.0)
74            throw new IllegalArgumentException(
75               "La tarifa de comisión debe ser > 0.0 y < 1.0");
76
77         this.tarifaComision = tarifaComision;
78      }
79
80      // devuelve la tarifa de comisión
81      public double obtenerTarifaComision()
82      {
83         return tarifaComision;
84      }
85
86      // calcula los ingresos
87      public double ingresos()
88      {
89         return tarifaComision * ventasBrutas;
90      }
91
92      // devuelve representación String del objeto EmpleadoPorComision
93      @Override // indica que este método sobrescribe el método de una superclase
94      public String toString()
95      {
96         return String.format("%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f",
97            "empleado por comision", primerNombre, apellidoPaterno,
98            "numero de seguro social", numeroSeguroSocial,
```

Fig. 9.4 | La clase `EmpleadoPorComision` representa a un empleado que recibe como sueldo un porcentaje de las ventas brutas (parte 2 de 3).

```
99          "ventas brutas", ventasBrutas,
100         "tarifa de comision", tarifaComision);
101     }
102 } // 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 ser llamados por las subclases. De hecho, Java requiere que *la primera tarea del constructor de cualquier subclase sea llamar al constructor de su superclase directa*, ya sea explícita o implícitamente (si no se especifica una llamada al constructor), para asegurar que las variables de instancia heredadas de la superclase se inicialicen en forma apropiada. La sintaxis para llamar de manera explícita al constructor de una superclase se describe en la sección 9.4.3. En este ejemplo, el constructor de la clase `EmpleadoPorComision` llama al constructor de la clase `Object` en forma implícita. 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 17 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 de la clase `Object` no hace nada. Aun si una clase no tiene constructores, el constructor predeterminado que el compilador declara implícitamente 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 20 a 22 y 25 a 27 validan los argumentos `ventasBrutas` y `tarifaComision`. Si éstos son válidos (es decir, si el constructor no lanza una excepción `IllegalArgumentException`), las líneas 29 a 33 asignan los argumentos del constructor a las variables de instancia de la clase.

No validamos los valores de los argumentos `primerNombre`, `apellidoPaterno` y `numeroSeguroSocial` antes de asignarlos a las variables de instancia correspondientes. Podríamos validar el `primerNombre` y el `apellidoPaterno`, para asegurarnos por ejemplo de que tengan una longitud razonable. De manera similar, podría validarse un número de seguro social mediante el uso de expresiones regulares (sección 14.7), 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 87 a 90) calcula los ingresos de un `EmpleadoPorComision`. La línea 89 multiplica la `tarifaComision` por las `ventasBrutas` y devuelve el resultado.

El método `toString` *de la clase* `EmpleadoPorComision` *y la anotación* `@Override`

El método `toString` (líneas 93 a 101) es especial, ya que es uno de los métodos que hereda *cualquier* clase de manera directa o indirecta de la clase `Object` (sintetizada en la sección 9.6). El método `toString` devuelve un `String` que representa a un objeto. Éste es llamado implícitamente 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 93 usa la **anotación `@Override`** opcional para indicar que la declaración del siguiente método (es decir, `toString`) debe *sobrescribir* un método *existente* de la superclase. Esta anotación ayuda al compilador a atrapar unos cuantos errores comunes. Por ejemplo, en este caso, intentamos sobrescribir el método `toString` de la superclase, que se escribe con una "`t`" minúscula y una "`S`" mayúscula. Si utiliza de manera inadvertida una "`s`" minúscula, el compilador marcará esto como un error, ya que la superclase no contiene un método llamado `toString`. Si no utilizó la anotación `@Override`, `toString` sería un método totalmente diferente que *no* se llamaría si se usara un `EmpleadoPorComision` que necesitara un objeto `String`.

Otro error común de sobrescritura es declarar el número incorrecto de tipos de parámetros en la lista de parámetros. Esto crea una *sobrecarga no intencional* del método de la superclase, en vez de sobrescribir el método existente. Si más tarde tratamos de llamar al método (con el número y tipos de parámetros correctos) en un objeto de la subclase, se invoca la versión de la superclase, lo cual 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, del tipo "el método no sobrescribe o implementa a un método de un supertipo". Lo que sigue es corregir la firma del método para que coincida con la del método de la superclase.



Tip para prevenir errores 9.1

Aunque la anotación `@Override` es opcional, declare los métodos sobrescritos con ella 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. Por esta razón, los métodos `toString` de la figura 7.9 y los ejemplos del capítulo 8 deberían haberse declarado con `@Override`.



Error común de programación 9.1

*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 34 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 ($10,000) y `.06` como la tarifa de comisión (es decir, el 6%). 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 el método `ingresos` en cada clase, pero lo 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:");
15       System.out.printf("%n%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);
27       empleado.establecerTarifaComision(.1);
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

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`.

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 42), además de los métodos `ingresos` (líneas 111 a 114) y `toString` (líneas 117 a 126). Las líneas 45 a 108 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 final String primerNombre;
8     private final String apellidoPaterno;
9     private final 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 primerNombre, String apellidoPaterno,
16       String numeroSeguroSocial, double ventasBrutas,
17       double tarifaComision, double salarioBase)
18    {
19       // la llamada implícita al constructor predeterminado de Object ocurre aquí
20
21       // si ventasBrutas son inválidas, lanza excepción
22       if (ventasBrutas < 0.0)
23          throw new IllegalArgumentException(
24             "Las ventas brutas deben ser >= 0.0");
25
26       // si tarifaComision es inválida, lanza excepción
27       if (tarifaComision <= 0.0 || tarifaComision >= 1.0)
28          throw new IllegalArgumentException(
29             "La tarifa de comisión debe ser > 0.0 y < 1.0");
30
31       // si salarioBase es inválido, lanza excepción
32       if (salarioBase < 0.0)
33          throw new IllegalArgumentException(
34             "El salario base debe ser >= 0.0");
35
```

Fig. 9.6 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de la comisión (parte 1 de 3).

```
36          this.primerNombre = primerNombre;
37          this.apellidoPaterno = apellidoPaterno;
38          this.numeroSeguroSocial = numeroSeguroSocial;
39          this.ventasBrutas = ventasBrutas;
40          this.tarifaComision = tarifaComision;
41          this.salarioBase = salarioBase;
42       } // fin del constructor
43
44       // devuelve el primer nombre
45       public String obtenerPrimerNombre()
46       {
47          return primerNombre;
48       }
49
50       // devuelve el apellido paterno
51       public String obtenerApellidoPaterno()
52       {
53          return apellidoPaterno;
54       }
55
56       // devuelve el número de seguro social
57       public String obtenerNumeroSeguroSocial()
58       {
59          return numeroSeguroSocial;
60       }
61
62       // establece el monto de ventas brutas
63       public void establecerVentasBrutas(double ventasBrutas)
64       {
65          if (ventasBrutas < 0.0)
66             throw new IllegalArgumentException(
67                "Las ventas brutas deben ser >= 0.0");
68
69          this.ventasBrutas = ventasBrutas;
70       }
71
72       // devuelve el monto de ventas brutas
73       public double obtenerVentasBrutas()
74       {
75          return ventasBrutas;
76       }
77
78       // establece la tarifa de comisión
79       public void establecerTarifaComision(double tarifaComision)
80       {
81          if (tarifaComision <= 0.0 || tarifaComision > 1.0)
82             throw new IllegalArgumentException(
83                "La tarifa de comisión debe ser > 0.0 y < 1.0");
84
85          this.tarifaComision = tarifaComision;
86       }
87
```

Fig. 9.6 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de la comisión (parte 2 de 3).

```
88       // devuelve la tarifa de comisión
89       public double obtenerTarifaComision()
90       {
91          return tarifaComision;
92       }
93
94       // establece el salario base
95       public void establecerSalarioBase(double salarioBase)
96       {
97          if (salarioBase < 0.0)
98            throw new IllegalArgumentException(
99                "El salario base debe ser >= 0.0");
100
101         this.salarioBase = salarioBase;
102      }
103
104      // devuelve el salario base
105      public double obtenerSalarioBase()
106      {
107         return salarioBase;
108      }
109
110      // calcula los ingresos
111      public double ingresos()
112      {
113         return salarioBase + (tarifaComision * ventasBrutas);
114      }
115
116      // devuelve representación String de EmpleadoBaseMasComision
117      @Override
118      public String toString()
119      {
120        return String.format(
121           "%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f%n%s: %.2f",
122           "empleado por comision con sueldo base", primerNombre, apellidoPaterno,
123           "numero de seguro social", numeroSeguroSocial,
124           "ventas brutas", ventasBrutas, "tarifa de comision", tarifaComision,
125           "salario base", salarioBase);
126      }
127   } // fin de la clase EmpleadoBaseMasComision
```

Fig. 9.6 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de la comisión (parte 3 de 3).

Puesto que la clase `EmpleadoBaseMasComision` *no* especifica "`extends Object`" en la línea 5, entonces la clase extiende a `Object` *implícitamente*. Además, al igual que el constructor de la clase `EmpleadoPorComision` (líneas 13 a 34 de la figura 9.4), el constructor de la clase `EmpleadoBaseMasComision` invoca en forma *implícita* al constructor predeterminado de la clase `Object`, como se indica en el comentario de la línea 19.

El método `ingresos` de la clase `EmpleadoBaseMasComision` (líneas 111 a 114) 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 121).

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, el apellido paterno, el número de seguro social, las ventas brutas, la tarifa de comisión y el 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 95 a 102) asegura que no se asigne un valor negativo a la variable `salarioBase`. La línea 33 de la figura 9.7 invoca en forma *explícita* el método `toString` para obtener la representación `String` del objeto.

```
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());
28
29       empleado.establecerSalarioBase(1000);
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
```

Fig. 9.7 | Programa de prueba de `EmpleadoBaseMasComision` (parte 1 de 2).

```
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`, así como 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 incorporaciones 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, lo que genera problemas de mantenimiento del código, pues habría que realizar cambios en varias clases. ¿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 de la jerarquía se declaran en una superclase. La subclase hereda las modificaciones que se realizan en estas características comunes de la superclase. 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`.



Observación de ingeniería de software 9.4

En la etapa de diseño en un sistema orientado a objetos, a menudo encontrará que ciertas clases están muy relacionadas entre sí. Debe "excluir" las variables de instancia y los métodos comunes, y colocarlos en una superclase. Después hay que usar la herencia para desarrollar subclases, especializándolas con capacidades más allá de las que hereden de la superclase.



Observación de ingeniería de software 9.5

Al declarar una subclase no se afecta el código fuente de su superclase. La herencia preserva la integridad de la superclase.

Sólo se puede acceder a los miembros `public` y `protected` de `EmpleadoPorComision` directamente en la subclase. El constructor de `EmpleadoPorComision` *no* se hereda. Por lo tanto, los servicios `public` de `EmpleadoBaseMasComision` incluyen su constructor (líneas 9 a 23), los métodos `public` heredados de `EmpleadoPorComision`, y los métodos `establecerSalarioBase` (líneas 26 a 33), `obtenerSalarioBase` (líneas 36 a 39), `ingresos` (líneas 42 a 47) y `toString` (líneas 50 a 60). 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, respectivamente.

```
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 primerNombre, String apellidoPaterno,
10       String numeroSeguroSocial, double ventasBrutas,
11       double tarifaComision, double salarioBase)
12    {
13       // llamada explícita al constructor de la superclase EmpleadoPorComision
14       super(primerNombre, apellidoPaterno, numeroSeguroSocial,
15          ventasBrutas, tarifaComision);
16
17       // si salarioBase no es válido, lanza excepción
18       if (salarioBase < 0.0)
19          throw new IllegalArgumentException(
20             "El salario base debe ser >= 0.0");
21
```

Fig. 9.8 | Los miembros `private` de la superclase no se pueden utilizar en una subclase (parte 1 de 3).

```
22        this.salarioBase = salarioBase;
23     }
24
25     // establece el salario base
26     public void establecerSalarioBase(double salarioBase)
27     {
28        if (salariobase < 0.0)
29           throw new IllegalArgumentException(
30              "El salario base debe ser >= 0.0");
31
32        this.salarioBase = salarioBase;
33     }
34
35     // devuelve el salario base
36     public double obtenerSalarioBase()
37     {
38        return salarioBase;
39     }
40
41     // calcula los ingresos
42     @Override
43     public double ingresos()
44     {
45        // no está permitido: tarifaComision y ventasBrutas son private en la
             superclase
46        return salarioBase + (tarifaComision * ventasBrutas);
47     }
48
49     // devuelve representación String de EmpleadoBaseMasComision
50     @Override
51     public String toString()
52     {
53        // no está permitido: intentos por acceder a los miembros private de la
             superclase
54        return String.format(
55           "%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f%n%s: %.2f",
56           "empleado por comision con sueldo base", primerNombre, apellidoPaterno,
57           "numero de seguro social", numeroSeguroSocial,
58           "ventas brutas", ventasBrutas, "tarifa de comision", tarifaComision,
59           "salario base", salarioBase);
60     }
61  } // fin de la clase EmpleadoBaseMasComision
```

```
EmpleadoBaseMasComision.java:46: error: tarifaComision has private access in
EmpleadoPorComision
       return salarioBase + (tarifaComision * ventasBrutas);
                             ^
EmpleadoBaseMasComision.java:46: error: ventasBrutas has private access in
EmpleadoPorComision
       return salarioBase + (tarifaComision * ventasBrutas);
                                              ^
EmpleadoBaseMasComision.java:56: error: primerNombre has private access in
EmpleadoPorComision
          "empleado por comision con sueldo base", primerNombre, apellidoPaterno,
                                                   ^
```

Fig. 9.8 | Los miembros `private` de la superclase no se pueden utilizar en una subclase (parte 2 de 3).

```
EmpleadoBaseMasComision.java:56: error: apellidoPaterno has private access in
EmpleadoPorComision
         "empleado por comision con sueldo base", primerNombre, apellidoPaterno,
                                                                 ^
EmpleadoBaseMasComision.java:57: error: numeroSeguroSocial has private access in
EmpleadoPorComision
         "numero de seguro social", numeroSeguroSocial,
                                    ^
EmpleadoBaseMasComision.java:58: error: ventasBrutas has private access in
EmpleadoPorComision
         "ventas brutas", ventasBrutas, "tarifa de comision", tarifaComision,
                          ^
EmpleadoBaseMasComision.java:58: error: tarifaComision has private access in
EmpleadoPorComision
         "ventas brutas", ventasBrutas, "tarifa de comision", tarifaComision,
                                                              ^
```

Fig. 9.8 | Los miembros `private` de la 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. Las líneas 14 y 15 en el constructor de `EmpleadoBaseMasComision` con seis argumentos (líneas 9 a 23) llaman en forma explícita al constructor de la clase `EmpleadoPorComision` con cinco argumentos (declarados en las líneas 13 a 34 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**, que consiste en la palabra clave `super`, seguida de un conjunto de paréntesis que contienen los argumentos del constructor de la superclase, que se utilizan para inicializar a las respectivas variables de instancia `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de la superclase. Si el constructor de `EmpleadoBaseMasComision` no invocara explícitamente al constructor de la superclase, el compilador trataría de insertar una llamada 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 las líneas 14 y 15 de la figura 9.8 debe ser la *primera* instrucción en el cuerpo del constructor. Cuando una superclase contiene un constructor sin argumentos, puede usar a `super()` para llamar explícitamente a ese constructor, pero esto raras veces se hace.



Observación de ingeniería de software 9.6

Anteriormente aprendió que no se debe llamar a los métodos de instancia de una clase desde sus constructores; le diremos por qué en el capítulo 10. Llamar al constructor de una superclase desde el constructor de una subclase no contradice este consejo.

Los métodos `ingresos` y `toString` de `EmpleadoBaseMasComision`

El compilador genera errores para la línea 46 (figura 9.8) 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 56 a 58 del método `toString` de `EmpleadoBaseMasComision` por la misma razón. Se habrían podido prevenir los errores en `EmpleadoBaseMasComision` al utilizar los métodos *obtener* heredados de la clase `EmpleadoPorComision`.

Por ejemplo, la línea 46 podría haber utilizado `obtenerTarifaComision` y `obtenerVentasBrutas` para acceder a las respectivas variables de instancia `private` `tarifaComision` y `ventasBrutas` de `EmpleadoPorComision`. Las líneas 56 a 58 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 pueden ser utilizados 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 final String primerNombre;
protected final String apellidoPaterno;
protected final 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 estas variables desde cualquier clase, 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 45 del método `ingresos` y las líneas 54 a 56 del método `toString`. Si otra clase extiende esta versión de la clase `EmpleadoBaseMasComision`, la nueva subclase también puede acceder a los miembros `protected`.

```
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
```

Fig. 9.9 | `EmpleadoBaseMasComision` hereda las variables de instancia `protected` de `EmpleadoPorComision` (parte 1 de 2).

```
8
9      // constructor con seis argumentos
10     public EmpleadoBaseMasComision(String primerNombre, String apellidoPaterno,
11        String numeroSeguroSocial, double ventasBrutas,
12        double tarifaComision, double salarioBase)
13     {
14        super(primerNombre, apellidoPaterno, numeroSeguroSocial,
15          ventasBrutas, tarifaComision);
16
17     // si salarioBase no es válido, lanza una excepción
18     if (salarioBase < 0.0)
19           throw new IllegalArgumentException(
20              "El salario base debe ser >= 0.0" );
21
22        this.salarioBase = salarioBase;
23     }
24
25     // establece el salario base
26     public void establecerSalarioBase(double salarioBase)
27     {
28        if (salarioBase < 0.0)
29           throw new IllegalArgumentException(
30              "El salario base debe ser >= 0.0");
31
32        this.salarioBase = salarioBase;
33     }
34
35     // devuelve el salario base
36     public double obtenerSalarioBase()
37     {
38        return salarioBase;
39     }
40
41     // calcula los ingresos
42     @Override  // indica que este método sobrescribe al método de la superclase
43     public double ingresos()
44     {
45        return salarioBase + (tarifaComision * ventasBrutas);
46     }
47
48     // devuelve representación String de EmpleadoBaseMasComision
49     @Override
50     public String toString()
51     {
52        return String.format(
53           "%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f%n%s: %.2f",
54           "empleado por comision con salario base", primerNombre, apellidoPaterno,
55           "numero de seguro social", numeroSeguroSocial,
56           "ventas brutas", ventasBrutas, "tarifa comision", tarifaComision,
57           "salario base", salarioBase);
58     }
59  } // fin de la clase EmpleadoBaseMasComision
```

Fig. 9.9 | `EmpleadoBaseMasComision` hereda las variables de instancia `protected` de `EmpleadoPorComision` (parte 2 de 2).

El objeto de una subclase contiene las variables de instancia de todas sus superclases
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` (que no tiene variables de instancia), `EmpleadoPorComision` y `EmpleadoBaseMasComision`. La clase `EmpleadoBaseMasComision` *no* hereda el constructor de cinco argumentos de la clase `EmpleadoPorComision`, pero lo *invoca de manera explícita* (líneas 14 y 15) 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 llamar en forma *explícita* al constructor de `EmpleadoPorComision` debido a que `EmpleadoPorComision` *no* tiene un constructor sin argumentos que pueda invocarse implícitamente.

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 (59 líneas) es mucho más corto que el de la figura 9.6 (127 líneas), debido a que la mayor parte de la funcionalidad de la clase se hereda ahora de `EmpleadoPorComision` y 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 `EmpleadoPorComision` sólo existe en esa clase.

Notas sobre el uso de variables 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`, las subclases pueden acceder directamente a las variables. No obstante, en la mayoría de los casos es mejor utilizar variables de instancia `private`, para fomentar la ingeniería de software apropiada. 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, de los métodos que no sean `private`) y no de la implementación de datos de la misma. Si hay variables de instancia `protected` en la superclase, probablemente tendríamos que 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 la clase es **frágil** o **quebradiza**, 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 volver a implementar 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 9.7

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 9.8

Al declarar variables de instancia `private` *(en vez 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

Cuando sea posible, no incluya variables de instancia `protected` *en una superclase. En vez de ello, incluya métodos no* `private` *que accedan a las variables de instancia* `private`*. Esto ayudará a asegurar 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 examinaremos nuestra jerarquía una vez más, pero esta vez utilizaremos las mejores prácticas de ingeniería de software.

La clase ***`EmpleadoPorComision`***

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` `obtenerPrimerNombre`, `obtenerApellidoPaterno`, `obtenerNumeroSeguroSocial`, `establecerVentasBrutas`, `obtenerVentasBrutas`, `establecerTarifaComision`, `obtenerTarifaComision`, `ingresos` y `toString` para manipular estos valores. Los métodos `ingresos` (líneas 87 a 90) y `toString` (líneas 93 a 101) utilizan los métodos *obtener* de la clase para obtener 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  // La clase EmpleadoPorComision usa los métodos para manipular sus
3  // variables de instancia private.
4  public class EmpleadoPorComision
5  {
6     private final String primerNombre;
7     private final String apellidoPaterno;
8     private final String numeroSeguroSocial;
9     private double ventasBrutas; // ventas totales por semana
10    private double tarifaComision; // porcentaje de comisión
```

Fig. 9.10 | La clase `EmpleadoPorComision` usa los métodos para manipular sus variables de instancia `private` (parte 1 de 3).

```
11
12    // constructor con cinco argumentos
13    public EmpleadoPorComision(String primerNombre, String apellidoPaterno,
14       String numeroSeguroSocial, double ventasBrutas,
15       double tarifaComision)
16    {
17       // la llamada implícita al constructor de Object ocurre aquí
18
19       // si ventasBrutas no son válidas, lanza excepción
20       if (ventasBrutas < 0.0)
21          throw new IllegalArgumentException(
22             "Ventas brutas debe ser >= 0.0");
23
24       // si tarifaComision no es válida, lanza excepción
25       if (tarifaComision <= 0.0 || tarifaComision >= 1.0)
26          throw new IllegalArgumentException(
27             "La tarifa de comision debe ser > 0.0 y < 1.0");
28
29       this.primerNombre = primerNombre;
30       this.apellidoPaterno = apellidoPaterno;
31       this.numeroSeguroSocial = numeroSeguroSocial;
32       this.ventasBrutas = ventasBrutas;
33       this.tarifaComision = tarifaComision;
34    } // fin del constructor
35
36    // devuelve el primer nombre
37    public String obtenerPrimerNombre()
38    {
39       return primerNombre;
40    }
41
42    // devuelve el apellido paterno
43    public String obtenerApellidoPaterno()
44    {
45       return apellidoPaterno;
46    }
47
48    // devuelve el número de seguro social
49    public String obtenerNumeroSeguroSocial()
50    {
51       return numeroSeguroSocial;
52    }
53
54    // establece el monto de ventas brutas
55    public void establecerVentasBrutas(double ventasBrutas)
56    {
57       if (ventasBrutas < 0.0)
58          throw new IllegalArgumentException(
59             "Las ventas brutas deben ser >= 0.0");
60
61       this.ventasBrutas = ventasBrutas;
62    }
```

Fig. 9.10 | La clase `EmpleadoPorComision` usa los métodos para manipular sus variables de instancia `private` (parte 2 de 3).

```
63
64    // devuelve el monto de ventas brutas
65    public double obtenerVentasBrutas()
66    {
67       return ventasBrutas;
68    }
69
70    // establece la tarifa de comisión
71    public void establecerTarifaComision(double tarifaComision)
72    {
73       if (tarifaComision <= 0.0 || tarifaComision >= 1.0)
74          throw new IllegalArgumentException(
75             "La tarifa de comisión debe ser > 0.0 y < 1.0");
76
77       this.tarifaComision = tarifaComision;
78    }
79
80    // devuelve la tarifa de comisión
81    public double obtenerTarifaComision()
82    {
83       return tarifaComision;
84    }
85
86    // calcula los ingresos
87    public double ingresos()
88    {
89       return obtenerTarifaComision() * obtenerVentasBrutas();
90    }
91
92    // devuelve representación String del objeto EmpleadoPorComision
93    @Override
94    public String toString()
95    {
96       return String.format("%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f",
97          "empleado por comision", obtenerPrimerNombre(), obtenerApellidoPaterno(),
98          "numero de seguro social", obtenerNumeroSeguroSocial(),
99          "ventas brutas", obtenerVentasBrutas(),
100         "tarifa de comision", obtenerTarifaComision());
101   }
102 } // fin de la clase EmpleadoPorComision
```

Fig. 9.10 | La clase `EmpleadoPorComision` usa los métodos para manipular sus variables de instancia `private` (parte 3 de 3).

La clase `EmpleadoBaseMasComision`

La subclase `EmpleadoBaseMasComision` (figura 9.11) hereda los miembros no `private` de `EmpleadoPorComision` y puede acceder (de una manera controlada) 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 43 a 47) y `toString` (líneas 50 a 55) 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 primerNombre, String apellidoPaterno,
12       String numeroSeguroSocial, double ventasBrutas,
13       double tarifaComision, double salarioBase)
14    {
15       super (primerNombre, apellidoPaterno, numeroSeguroSocial,
16          ventasBrutas, tarifaComision);
17
18       // si salarioBase no es válido, lanza excepción
19        if (salarioBase < 0.0)
20           throw new IllegalArgumentException(
21              "El salario base debe ser >= 0.0");
22
23       this.salarioBase = salarioBase;
24    }
25
26    // establece el salario base
27    public void establecerSalarioBase(double salarioBase)
28    {
29        if (salarioBase < 0.0)
30           throw new IllegalArgumentException(
31              "El salario base debe ser >= 0.0");
32
33       this.salarioBase = salarioBase;
34    }
35
36    // devuelve el salario base
37    public double obtenerSalarioBase()
38    {
39       return salarioBase;
40    }
41
42    // calcula los ingresos
43    @Override
44   public double ingresos()
45    {
46       return obtenerSalarioBase() + super.ingresos();
47    }
48
49    // devuelve representación String de EmpleadoBaseMasComision
50    @Override
51    public String toString()
52    {
```

Fig. 9.11 | La clase `EmpleadoBaseMasComision` hereda de `EmpleadoPorComision` y accede a los datos `private` de la superclase a través de los métodos `public` heredados (parte 1 de 2).

```
53          return String.format("%s %s%n%s: %.2f", "con sueldo base",
54             super.toString(), "sueldo base", obtenerSalarioBase());
55       }
56    } // 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 los métodos `public` heredados (parte 2 de 2).

El método `ingresos` de la clase `EmpleadoBaseMasComision`

El método `ingresos` (líneas 43 a 47) sobrescribe el método `ingresos` de `EmpleadoPorComision` (figura 9.10, líneas 87 a 90) 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 solamente en la comisión, mediante una llamada al método `ingresos` de `EmpleadoPorComision` con la expresión `super.ingresos()` (línea 46), 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 (se coloca 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, ya que 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*.



Error común de programación 9.2

Cuando el método de una superclase se sobrescribe en una subclase, por lo general la versión de esta última 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 llamar al método de la superclase, entonces el método de la subclase se llamaría a sí mismo, lo cual podría generar un error 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. Cuando la recursividad se usa en forma correcta, es una herramienta poderosa que veremos en el capítulo 18.*

El método `toString` de la clase `EmpleadoBaseMasComision`

De manera similar, el método `toString` de `EmpleadoBaseMasComision` (figura 9.11, líneas 50 a 55) sobrescribe el método `toString` de la clase `EmpleadoPorComision` (figura 9.10, líneas 93 a 101) 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 54). Después, el método `toString` de `EmpleadoBaseMasComision` completa el resto de la representación `String` de un objeto `EmpleadoBaseMasComision` (es decir, el valor del salario base de la clase `EmpleadoBaseMasComision`).

Prueba de la clase `EmpleadoBaseMasComision`

La clase `PruebaEmpleadoBaseMasComision` realiza las mismas manipulaciones sobre un objeto `EmpleadoBaseMasComision` que 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.

9.5 Los constructores en las subclases

Como explicamos en la sección anterior, al crear una instancia de un objeto de una subclase se inicia una cadena de llamadas a los constructores, en los que el constructor de la subclase, antes de realizar sus propias tareas, usa `super` para llamar a uno de los constructores de su superclase directa, o llama implícitamente 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 una aplicación crea un objeto `EmpleadoBaseMasComision`, se hace una llamada a su constructor. Ese constructor llama al constructor de `EmpleadoPorComision`, 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 entonces las variables de instancia de `EmpleadoPorComision` que son parte del objeto `EmpleadoBaseMasComision`. Cuando el constructor de `EmpleadoPorComision` termina de ejecutarse, devuelve el control al constructor de `EmpleadoBaseMasComision`, el cual inicializa el `salarioBase`.



Observación de ingeniería de software 9.9

Java asegura que, aun si un constructor no asigna un valor a una variable de instancia, de todas formas ésta 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 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
`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 (también tendrá que sobrescribir el método `hashCode`), consulte la documentación del método en `docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals` (`java.lang.Object`). La implementación predeterminada de `equals` utiliza el operador `==` para determinar si dos referencias *se refieren al mismo objeto* en la memoria. La sección 14.3.3 demuestra el método `equals` de la clase `String` y explica la diferencia entre comparar objetos `String` con `==` y con `equals`.
`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 16.11). Este método también se llama como parte de la implementación del método `toString` predeterminado de la clase `Object`.

Fig. 9.12 | Métodos de `Object` (parte 1 de 2).

Método	Descripción
`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.
`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 23 (en inglés, en la página Web del libro).
`getClass`	Todo objeto en Java conoce su tipo en tiempo de ejecución. El método `getClass` (utilizado en las secciones 10.5 y 12.5) 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`).
`finalize`	El recolector de basura llama a este método `protected` 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. En la sección 8.10 vimos que no se garantiza que se ejecute el método `finalize` del objeto, ni cuándo se ejecutará. Por esta razón, la mayoría de los programadores deberían evitar usar el método `finalize`.
`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**, en la que 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 15 hablaremos sobre la serialización de objetos. En el capítulo 7 vimos que los arreglos son objetos. Como resultado y al igual que los demás objetos, los arreglos heredan los miembros de la clase `Object`. Cada arreglo tiene un método `clone` sobrescrito que copia el arreglo. Pero si el arreglo almacena referencias a objetos, los objetos no se copian; en su lugar se realiza una copia superficial.

Fig. 9.12 | Métodos de `Object` (parte 2 de 2).

9.7 (Opcional) Ejemplo práctico de GUI y gráficos: mostrar texto e imágenes usando etiquetas

Los programas usan a menudo 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 sobre el 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.

```
1  // Fig 9.13: DemoLabel.java
2  // Demuestra el uso de etiquetas.
3  import java.awt.BorderLayout;
4  import javax.swing.ImageIcon;
```

Fig. 9.13 | `JLabel` con texto y con imágenes (parte 1 de 2).

```
5  import javax.swing.JLabel;
6  import javax.swing.JFrame;
7
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);
39       aplicacion.setVisible(true);
40    } // fin de main
41 } // fin de la clase DemoLabel
```



Fig. 9.13 | `JLabel` con texto y con imágenes (parte 2 de 2).

Las líneas 3 a 6 importan las clases que necesitamos para mostrar los objetos `JLabel`. `BorderLayout` del paquete `java.awt` que contienen 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.

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 constructor 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` (arriba), `SOUTH` (abajo), `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 ejemplo práctico 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.

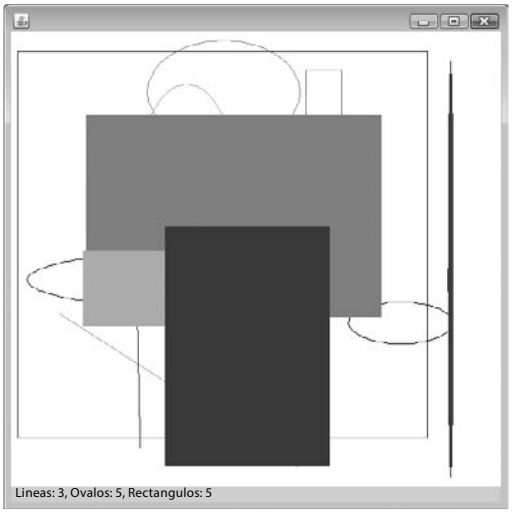


Fig. 9.14 | Objeto `JLabel` que muestra las estadísticas de las figuras.

9.8 Conclusión

En este capítulo se introdujo el concepto de la herencia, que es la habilidad de crear clases mediante la absorción de los miembros de una clase existente (sin copiar y pegar el código), 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 directamente a los miembros `protected` de la superclase. Aprendió también cómo acceder a los miembros sobrescritos 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 e interfaces, continuaremos con nuestra explicación sobre la herencia al introducir el *polimorfismo*: un concepto orientado a objetos que nos permite escribir programas que puedan manipular convenientemente y 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, que son las tecnologías clave de la programación orientada a objetos.

Resumen

Sección 9.1 Introducción

- La herencia (pág. 361) reduce el tiempo de desarrollo de los programas.
- La superclase directa (pág. 361) de una subclase es la superclase a partir de la cual hereda la subclase. Una superclase indirecta (pág. 361) de una subclase se encuentra dos o más niveles arriba de esa subclase en la jerarquía de clases.
- En la herencia simple (pág. 361), una clase se deriva de una superclase. 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. 361).
- Cada objeto de una subclase es también un objeto de la superclase de esa clase. Sin embargo, el objeto de una superclase no es un objeto de las subclases de su clase.
- Una relación *es-un* (pág. 362) 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. 362) 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 directamente sólo dentro de la declaración de esa superclase.
- Los miembros `protected` de una superclase (pág. 364) 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. 364) y un separador punto (`.`).

Sección 9.4 Relación entre las superclases y las subclases

- Una subclase no puede acceder a 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. 369) indica que un método debe sobrescribir al método de una superclase. Cuando el compilador encuentra un método declarado con `@Override`, compara la firma de ese método con las firmas del método de la superclase. Si no hay una coincidencia exacta, el compilador emite un mensaje de error, del tipo "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 Los constructores en las subclases

- La primera tarea de cualquier constructor de subclase es llamar al constructor de su superclase directa (pág. 378), para asegurar que se inicialicen las variables de instancia heredadas de la superclase.

Sección 9.6 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) La _____ 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 _____ y _____ 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 tratarse 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`; `protected`. c) *es-un* o de herencia. d) *tiene-un*, o de 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 ***(Uso de la composición en vez de la herencia)*** 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 ***(Reutilización de software)*** 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 ***(Jerarquía de herencia `Estudiante`)*** 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 ***(Jerarquía de herencia `Figura`)*** 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 intégrelas 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 ***(Comparación entre `protected` y `private`)*** 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 ***(Jerarquía de herencia `Cuadrilatero`)*** 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`).

9.9 ***(¿Qué hace cada fragmento de código?)***

a) Suponga que la siguiente llamada a método se encuentra en un método `ingresos` sobrescrito en una subclase:

```
super.ingresos()
```

b) Suponga que la siguiente línea de código aparece antes de la declaración de un método:

```
@Override
```

c) Suponga que aparece la siguiente línea de código como la primera instrucción en el cuerpo de un constructor:

```
super(primerArgumento, segundoArgumento);
```

9.10 ***(Escriba una línea de código)*** Escriba una línea de código que realice cada una de las siguientes tareas:

a) Especifique que la clase `TrabajadorPiezas` herede de la clase `Empleado`.
b) Llame al método `toString` de la superclase `Empleado` desde el método `toString` de la subclase `TrabajadorPiezas`.
c) Llame al constructor de la superclase `Empleado` desde el constructor de la subclase `TrabajadorPiezas`; suponga que el constructor de la superclase recibe tres objetos `String` que representan el primer nombre, el apellido paterno y el número de seguro social.

9.11 ***(Uso de `super` en el cuerpo de un constructor)*** Explique por qué usaría `super` en la primera instrucción del cuerpo del constructor de una subclase.

9.12 ***(Uso de `super` en el cuerpo del método de una instancia)*** Explique por qué usaría `super` en el cuerpo del método de instancia de una subclase.

9.13 ***(Llamar métodos* obtener *en el cuerpo de una clase)*** En las figuras 9.10 y 9.11, los métodos `ingresos` y `toString` llaman a varios métodos *obtener* dentro de la misma clase. Explique los beneficios de llamar a estos métodos *obtener* dentro de las clases.

9.14 ***(Jerarquía `Empleado`)*** En este capítulo estudió una jerarquía de herencia en donde la clase `EmpleadoBaseMasComision` heredó de la clase `EmpleadoPorComision`. Sin embargo, no todos los tipos de empleados son `EmpleadoPorComision`. En este ejercicio, creará una superclase `Empleado` más general que *extraiga* los atributos y comportamientos de la clase `EmpleadoPorComision` que son comunes para todos los objetos `Empleado`. Los atributos y comportamientos comunes de todos los objetos `Empleado` son: `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `obtenerPrimerNombre`, `obtenerApellidoPaterno`, `obtenerNumeroSeguroSocial` y una parte del método `toString`. Cree una nueva superclase `Empleado` que contenga estas variables y métodos de instancia, además de un constructor. A continuación, vuelva a escribir la clase `EmpleadoPorComision` de la sección 9.4.5 como una subclase de `Empleado`. La clase `EmpleadoPorComision` debe contener sólo las variables y métodos de instancia que no se declaren en la superclase `Empleado`. El constructor de la clase `EmpleadoPorComision` debe invocar al constructor de la clase `Empleado` y el método `toString` de `EmpleadoPorComision` debe invocar al método `toString` de `Empleado`. Una vez que complete estas modificaciones, ejecute las aplicaciones `PruebaEmpleadoPorComision` y `PruebaEmpleado-BaseMasComision` usando estas nuevas clases para asegurar que las aplicaciones sigan mostrando los mismos resultados para un objeto `EmpleadoPorComision` y un objeto `EmpleadoBaseMasComision`, respectivamente.

9.15 ***(Creación de una nueva subclase de `Empleado`)*** Otros tipos de objetos `Empleado` podrían incluir objetos `EmpleadoAsalariado` que reciban un salario semanal fijo, `TrabajadoresPiezas` a quienes se les pague por el número de piezas que produzcan, o `EmpleadosPorHoras` que reciban un sueldo por horas con tiempo y medio (1.5 veces el sueldo por horas) por las horas trabajadas que sobrepasen las 40 horas.

Cree la clase `EmpleadoPorHoras` que herede de la clase `Empleado` (ejercicio 9.14) y tenga la variable de instancia `horas` (de tipo `double`) que represente las horas trabajadas, la variable de instancia `sueldo` (de tipo `double`) que represente los sueldos por hora, un constructor que reciba como argumentos el primer nombre, el apellido paterno, el número de seguro social, el sueldo por horas y el número de horas trabajadas, métodos *establecer* y *obtener* para manipular las variables `hora` y `sueldo`, un método `ingresos` para calcular los ingresos de un `EmpleadoPorHoras` con base en las horas trabajadas, y un método `toString` que devuelva la representación `String` del `EmpleadoPorHoras`. El método `establecerSueldo` debe asegurarse de que `sueldo` sea mayor o igual a 0, y `establecerHoras` debe asegurar que el valor de `horas` esté entre 0 y 168 (el número total de horas en una semana). Use la clase `EmpleadoPorHoras` en un programa de prueba que sea similar al de la figura 9.5.

Programación orientada a objetos: polimorfismo e interfaces

10



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

Objetivos

En este capítulo aprenderá:

- El concepto de polimorfismo.
- A utilizar métodos sobrescritos para llevar a cabo el polimorfismo.
- A distinguir entre clases abstractas y concretas.
- A declarar métodos abstractos para crear clases abstractas.
- Cómo el polimorfismo hace que los sistemas puedan extenderse y mantenerse.
- A determinar el tipo de un objeto en tiempo de ejecución.
- A declarar e implementar interfaces, y familiarizarse con las mejoras en las interfaces de Java SE 8.

Plan general

10.1 Introducción

10.2 Ejemplos del polimorfismo

10.3 Demostración del comportamiento polimórfico

10.4 Clases y métodos abstractos

10.5 Ejemplo práctico: sistema de nómina utilizando polimorfismo

- 10.5.1 La superclase abstracta `Empleado`
- 10.5.2 La subclase concreta `EmpleadoAsalariado`
- 10.5.3 La subclase concreta `EmpleadoPorHoras`
- 10.5.4 La subclase concreta `EmpleadoPorComision`
- 10.5.5 La subclase concreta indirecta `EmpleadoBaseMasComision`
- 10.5.6 El procesamiento polimórfico, el operador `instanceof` y la conversión descendente

10.6 Asignaciones permitidas entre variables de la superclase y de la subclase

10.7 Métodos y clases `final`

10.8 Una explicación más detallada de los problemas con las llamadas a métodos desde los constructores

10.9 Creación y uso de interfaces

- 10.9.1 Desarrollo de una jerarquía `PorPagar`
- 10.9.2 La interfaz `PorPagar`
- 10.9.3 La clase `Factura`
- 10.9.4 Modificación de la clase `Empleado` para implementar la interfaz `PorPagar`
- 10.9.5 Modificación de la clase `EmpleadoAsalariado` para usarla en la jerarquía `PorPagar`
- 10.9.6 Uso de la interfaz `PorPagar` para procesar objetos `Factura` y `Empleado` mediante el polimorfismo
- 10.9.7 Algunas interfaces comunes de la API de Java

10.10 Mejoras a las interfaces de Java SE 8

- 10.10.1 Métodos `default` de una interfaz
- 10.10.2 Métodos `static` de una interfaz
- 10.10.3 Interfaces funcionales

10.11 (Opcional) Ejemplo práctico de GUI y gráficos: realización de dibujos mediante el polimorfismo

10.12 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios* | *Marcando la diferencia*

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 (es decir, `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, ya que se pueden agregar 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 nuevas clases simplemente "se integran". 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 ejemplo práctico 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, mientras 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` *(sin importar el tipo de 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 ejemplo práctico 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 ejemplo práctico 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, que son especialmente útiles para asignar una funcionalidad *común* a clases que posiblemente *no estén relacionadas*. Esto permite que los objetos de estas clases se procesen en forma polimórfica; es decir, los objetos de clases que **implementen** 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.

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 sus tipos específicos, siempre y cuando éstos pertenezcan a la misma jerarquía de herencia.



Observación de ingeniería de software 10.2

El polimorfismo promueve la extensibilidad, ya que 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, ya que 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* para 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*, ya que es el tipo del *objeto referenciado*, y *no* el tipo de la *variable*, 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 debido a 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 de una 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 abajo de la jerarquía, desde una superclase hacia sus subclases o subclases indirectas).

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. ¿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 demostraremos la mecánica de la conversión descendente.



Observación de ingeniería de software 10.3

Aunque está permitida, por lo general es mejor evitar la 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

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
```

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 superclase a un ob-
jeto 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` (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 es llamado, 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**. Un método abstracto es un *método de instancia* 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 10.4

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 del tipo `Circulo`, `Triangulo`, `Rectangulo` 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, ya que 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 (por ejemplo, al 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 Ejemplo práctico: 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 escribir 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.

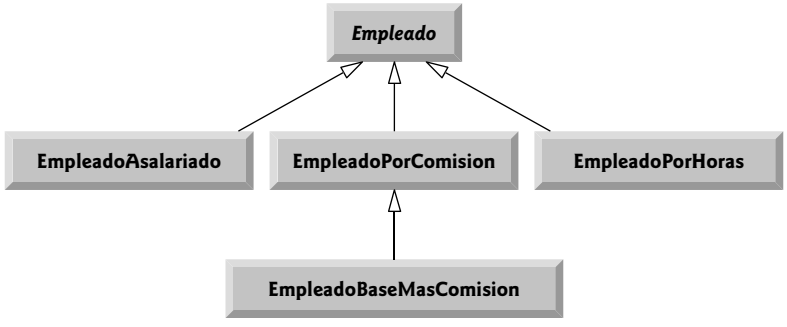


Fig. 10.2 | Diagrama de clases de UML para la jerarquía de `Empleado`.

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 10.9. 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`.

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 *obtener* de la superclase `Empleado` porque *no* se sobrescriben en ninguna de las subclases; éstas heredan y utilizan cada uno de estos métodos "como están".

	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`.

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* 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 específica del empleado. Por lo tanto, declaramos a `ingresos` como `abstract` en la superclase `Empleado`, ya que una implementación predeterminada *específica* no tiene sentido para ese método, puesto que 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`. *No* podemos usar la clase `Empleado` de manera directa para crear *objetos* `Empleado` ya que ésta es una clase *abstracta*. Sin embargo, gracias 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.

Consideremos ahora la declaración de la clase `Empleado` (figura 10.4). Esta clase incluye un constructor que recibe el primer nombre, el apellido paterno y el número de seguro social (líneas 11 a 17); los métodos *obtener* que devuelven el primer nombre, el apellido paterno y el número de seguro social (líneas 20 a 23, 26 a 29 y 32 a 35, respectivamente); el método `toString` (líneas 38 a 43), el cual devuelve la representación `String` de `Empleado`; y el método `abstract ingresos` (línea 46), 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.

```
1  // Fig. 10.4: Empleado.java
2  // La superclase abstracta Empleado.
3
4  public abstract class Empleado
5  {
6     private final String primerNombre;
7     private final String apellidoPaterno;
8     private final String numeroSeguroSocial;
9
10    // constructor
11    public Empleado(String primerNombre, String apellidoPaterno,
12       String numeroSeguroSocial)
13    {
14       this.primerNombre = primerNombre;
15       this.apellidoPaterno = apellidoPaterno;
16       this.numeroSeguroSocial = numeroSeguroSocial;
17    }
18
19    // devuelve el primer nombre
20    public String obtenerPrimerNombre()
21    {
22       return primerNombre;
23    }
24
25    // devuelve el apellido paterno
26    public String obtenerApellidoPaterno()
27    {
28       return apellidoPaterno;
29    }
30
31    // devuelve el número de seguro social
32    public String obtenerNumeroSeguroSocial()
33    {
```

Fig. 10.4 | La superclase abstracta `Empleado` (parte 1 de 2).

```
34         return numeroSeguroSocial;
35      }
36
37      // devuelve representación String de un objeto Empleado
38      @Override
39      public String toString()
40      {
41         return String.format("%s %s%nnumero de seguro social: %s",
42            obtenerPrimerNombre(), obtenerApellidoPaterno(),
               obtenerNumeroSeguroSocial());
43      }
44
45      // método abstracto sobrescrito por las subclases concretas
46      public abstract double ingresos(); // aquí no hay implementación
47   } // 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 abstracto? Simplemente, no tiene sentido proporcionar una implementación *específica* de este método en la clase `Empleado`. No podemos calcular los ingresos para un `Empleado` *general*, ya que primero debemos conocer el tipo de `Empleado` *específico* para determinar el cálculo apropiado de los ingresos. Al declarar este método `abstract`, indicamos que cada subclase 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 38 a 42), lo cual convierte a `EmpleadoAsalariado` en una clase *concreta*. La clase incluye un constructor (líneas 9 a 19) que recibe un primer nombre, un apellido paterno, un número de seguro social y un salario semanal; un método *establecer* para asignar un nuevo valor *no negativo* a la variable de instancia `salarioSemanal` (líneas 22 a 29); un método *obtener* para devolver el valor de `salarioSemanal` (líneas 32 a 35); un método `ingresos` (líneas 38 a 42) para calcular los ingresos de un `EmpleadoAsalariado`; y un método `toString` (líneas 45 a 50), el cual devuelve un objeto `String` que incluye "`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` de la superclase. Una vez más, hemos duplicado el código de validación de `salarioSemanal` en el constructor y en el método `obtenerSalarioSemanal`. Recuerde que podríamos colocar una validación más compleja en un método de clase `static` que se llame desde el constructor y el método *establecer*.



Tip para prevenir errores 10.1

Hemos dicho que no debe llamar a los métodos de instancia de una clase desde sus constructores (puede llamar a los métodos `static` de la clase y hacer la llamada requerida a uno de los constructores de la superclase). Si sigue este consejo, evitará el problema de llamar a los métodos sobrescritos de la clase, ya sea de manera directa o indirecta, lo cual puede provocar errores en tiempo de ejecución.

```
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
9     public EmpleadoAsalariado(String primerNombre, String apellidoPaterno,
10       String numeroSeguroSocial, double salarioSemanal)
11    {
12       super(primerNombre, apellidoPaterno, numeroSeguroSocial);
13
14       if (salarioSemanal < 0.0)
15          throw new IllegalArgumentException(
16             "El salario semanal debe ser >= 0.0");
17
18       this.salarioSemanal = salarioSemanal;
19    }
20
21    // establece el salario
22    public void establecerSalarioSemanal(double salarioSemanal)
23    {
24       if (salarioSemanal < 0.0)
25          throw new IllegalArgumentException(
26             "El salario semanal debe ser >= 0.0");
27
28       this.salarioSemanal = salarioSemanal;
29    }
30
31    // devuelve el salario
32    public double obtenerSalarioSemanal()
33    {
34       return salarioSemanal;
35    }
36
37    // calcula los ingresos; sobrescribe el método abstracto ingresos en Empleado
38    @Override
39    public double ingresos()
40    {
41       return obtenerSalarioSemanal();
42    }
43
44    // devuelve representación String de un objeto EmpleadoAsalariado
45    @Override
46    public String toString()
47    {
48       return String.format("empleado asalariado: %s%n%s: $%,.2f",
49          super.toString(), "salario semanal", obtenerSalarioSemanal());
50    }
51 } // fin de la clase EmpleadoAsalariado
```

Fig. 10.5 | La clase concreta `EmpleadoAsalariado` extiende a la clase `abstract Empleado`.

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.

El método `toString` (líneas 45 a 50) 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 49); 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 25) que recibe un primer nombre, un apellido paterno, un número de seguro social, un sueldo por horas y el número de horas trabajadas. Las líneas 28 a 35 y 44 a 51 declaran los métodos *establecer* que asignan nuevos valores a las variables de instancia `sueldo` y `horas`, respectivamente. El método `establecerSueldo` (líneas 28 a 35) asegura que `sueldo` sea *no negativo*, y el método `establecerHoras` (líneas 44 a 51) asegura que el valor de `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 38 a 41 y 54 a 57) para devolver los valores de `sueldo` y `horas`, respectivamente; un método `ingresos` (líneas 60 a 67) para calcular los ingresos de un `EmpleadoPorHoras`; y un método `toString` (líneas 70 a 76), que devuelve un objeto `String` con el tipo del empleado ("`empleado por horas: `"), así como 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 74) 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
10    public EmpleadoPorHoras(String primerNombre, String apellidoPaterno,
11       String numeroSeguroSocial, double sueldo, double horas)
12    {
13       super(primerNombre, apellidoPaterno, numeroSeguroSocial);
14
```

Fig. 10.6 | La clase `EmpleadoPorHoras` extiende a `Empleado` (parte 1 de 3).

```
15        if (sueldo < 0.0) // valida sueldo
16           throw new IllegalArgumentException(
17              "El sueldo por horas debe ser >= 0.0");
18
19        if ((horas < 0.0) || (horas > 168.0)) // valida horas
20           throw new IllegalArgumentException(
21              "Las horas trabajadas deben ser >= 0.0 y <= 168.0");
22
23        this.sueldo = sueldo;
24        this.horas = horas;
25     }
26
27     // establece el sueldo
28     public void establecerSueldo(double sueldo)
29     {
30        if (sueldo < 0.0) // valida sueldo
31           throw new IllegalArgumentException(
32              "El sueldo por horas debe ser >= 0.0");
33
34        this.sueldo = sueldo;
35     }
36
37     // devuelve el sueldo
38     public double obtenerSueldo()
39     {
40        return sueldo;
41     }
42
43     // establece las horas trabajadas
44     public void establecerHoras(double horas)
45     {
46        if ((horas < 0.0) || (horas > 168.0)) // valida horas
47           throw new IllegalArgumentException(
48              "Las horas trabajadas deben ser >= 0.0 y <= 168.0");
49
50        this.horas = horas;
51     }
52
53     // devuelve las horas trabajadas
54     public double obtenerHoras()
55     {
56        return horas;
57     }
58
59     // calcula los ingresos; sobrescribe el método abstracto ingresos en Empleado
60     @Override
61     public double ingresos()
62     {
63        if (obtenerHoras() <= 40) // no hay tiempo extra
64           return obtenerSueldo() * obtenerHoras();
65        else
66           return 40 * obtenerSueldo() + (obtenerHoras() - 40) * obtenerSueldo() * 1.5;
67     }
```

Fig. 10.6 | La clase `EmpleadoPorHoras` extiende a `Empleado` (parte 2 de 3).

```
68
69     // devuelve representación String de un objeto EmpleadoPorHoras
70     @Override
71     public String toString()
72     {
73        return String.format("empleado por horas: %s%n%s: $%,.2f; %s: %,.2f",
74           super.toString(), "sueldo por hora", obtenerSueldo(),
75           "horas trabajadas", obtenerHoras());
76     }
77  } // fin de la clase EmpleadoPorHoras
```

Fig. 10.6 | La clase `EmpleadoPorHoras` extiende a `Empleado` (parte 3 de 3).

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 25) que recibe como argumentos un primer nombre, un apellido paterno, un número de seguro social, un monto de ventas y una tarifa de comisión; métodos *establecer* (líneas 28 a 34 y 43 a 50) para asignar nuevos valores válidos a las variables de instancia `tarifaComision` y `ventasBrutas`, respectivamente; métodos *obtener* (líneas 37 a 40 y 53 a 56) que obtienen los valores de estas variables de instancia; el método `ingresos` (líneas 59 a 63) para calcular los ingresos de un `EmpleadoPorComision`; y el método `toString` (líneas 66 a 73) que devuelve el tipo del empleado, es decir, "`empleado por comisión:` ", así como información específica del empleado. El constructor también pasa el primer nombre, el apellido y el número de seguro social al constructor de la *superclase* `Empleado` (línea 14) para inicializar las variables de instancia `private` de `Empleado`. El método `toString` llama al método `toString` de la *superclase* (línea 70) 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
10    public EmpleadoPorComision(String primerNombre, String apellidoPaterno,
11       String numeroSeguroSocial, double ventas,
12       double tarifaComision)
13    {
14       super(primerNombre, apellidoPaterno, numeroSeguroSocial);
15
16       if (tarifaComision <= 0.0 || tarifaComision >= 1.0) // valida
17          throw new IllegalArgumentException(
18             "La tarifa de comision debe ser > 0.0 y < 1.0");
19
20       if (ventasBrutas < 0.0)
21          throw new IllegalArgumentException("Las ventas brutas deben ser >= 0.0");
22
```

Fig. 10.7 | La clase `EmpleadoPorComision` extiende a `Empleado` (parte 1 de 2).

```
23          this.ventasBrutas = ventasBrutas;
24          this.tarifaComision = tarifaComision;
25       }
26
27       // establece el monto de ventas brutas
28       public void establecerVentasBrutas(double ventasBrutas)
29       {
30          if (ventasBrutas < 0.0)
31             throw new IllegalArgumentException("Las ventas brutas deben ser >= 0.0");
32
33          this.ventasBrutas = ventasBrutas;
34       }
35
36       // devuelve el monto de ventas brutas
37       public double obtenerVentasBrutas()
38       {
39          return ventasBrutas;
40       }
41
42       // establece la tarifa de comisión
43       public void establecerTarifaComision(double tarifaComision)
44       {
45          if (tarifaComision <= 0.0 || tarifaComision >= 1.0) // valida
46             throw new IllegalArgumentException(
47                "La tarifa de comision debe ser > 0.0 y < 1.0");
48
49          this.tarifaComision = tarifaComision;
50       }
51
52       // devuelve la tarifa de comisión
53       public double obtenerTarifaComision()
54       {
55          return tarifaComision;
56       }
57
58       // calcula los ingresos; sobrescribe el método abstracto ingresos en Empleado
59       @Override
60       public double ingresos()
61       {
62          return obtenerTarifaComision() * obtenerVentasBrutas();
63       }
64
65       // devuelve representación String de un objeto EmpleadoPorComision
66       @Override
67       public String toString()
68       {
69          return String.format("%s: %s%n%s: $%,.2f; %s: %.2f",
70             "empleado por comision", super.toString(),
71             "ventas brutas", obtenerVentasBrutas(),
72             "tarifa de comision", obtenerTarifaComision());
73       }
74    } // 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 20) 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 13 y 14) para inicializar las variables de instancia de la superclase. `EmpleadoBaseMasComision` también contiene un método *establecer* (líneas 23 a 29) para asignar un nuevo valor a la variable de instancia `salarioBase` y un método *obtener* (líneas 32 a 35) para devolver el valor de `salarioBase`. El método `ingresos` (líneas 38 a 42) calcula los ingresos de un `EmpleadoBaseMasComision`. La línea 41 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 45 a 51) 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` (línea 49), y después el salario base. El resultado es un objeto `String` que empieza con "`con salario base empleado por comision`", 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`); éste es 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
9     public EmpleadoBaseMasComision(String primerNombre, String apellidoPaterno,
10       String numeroSeguroSocial, double ventasBrutas,
11       double tarifaComision, double salarioBase)
12    {
13      super(primerNombre, apellidoPaterno, numeroSeguroSocial,
14         ventasBrutas, tarifaComision);
15
16       if (salarioBase < 0.0) // valida el salarioBase
17          throw new IllegalArgumentException("El salario base debe ser >= 0.0");
18
19       this.salarioBase = salarioBase;
20    }
21
22    // establece el salario base
23    public void establecerSalarioBase(double salarioBase)
24    {
25       if (salarioBase < 0.0) // valida el salarioBase
26          throw new IllegalArgumentException("El salario base debe ser >= 0.0");
```

Fig. 10.8 | La clase `EmpleadoBaseMasComision` extiende a `EmpleadoPorComision` (parte 1 de 2).

```
27
28          this.salarioBase = salarioBase;
29       }
30
31       // devuelve el salario base
32       public double obtenerSalarioBase()
33       {
34          return salarioBase;
35       }
36
37       // calcula los ingresos; sobrescribe el método ingresos en EmpleadoPorComision
38       @Override
39       public double ingresos()
40       {
41          return obtenerSalarioBase() + super.ingresos();
42       }
43
44       // devuelve representación String de un objeto EmpleadoBaseMasComision
45       @Override
46       public String toString()
47       {
48          return String.format("%s %s; %s: $%,.2f",
49             "con salario base", super.toString(),
50             "salario base", obtenerSalarioBase());
51       }
52    } // 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 de manera *no polimórfica*, 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  {
```

Fig. 10.9 | Programa de prueba de la jerarquía de clases de `Empleado` (parte 1 de 4).

```
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);
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:");
21
22       System.out.printf("%n%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%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
```

Fig. 10.9 | Programa de prueba de la jerarquía de clases de `Empleado` (parte 2 de 4).

```
58                System.out.printf(
59                   "el nuevo salario base con 10%% de aumento es: $%,.2f%n",
60                   empleado.obtenerSalarioBase());
61             } // fin de if
62
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
```

Fig. 10.9 | Programa de prueba de la jerarquía de clases de `Empleado` (parte 3 de 4).

```
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

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 específicos 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 qué 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*.

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 es un error de compilación.



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 polimórficamente al método `ingresos` del objeto de la subclase apropiada. Obtener polimórficamente los ingresos de `EmpleadoAsalariado`, `EmpleadoPorHoras` y `EmpleadoPorComision` en las líneas 63 y 64, produce el mismo resultado que obtener en forma individual los ingresos de estos empleados, 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.

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`. 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 `empleado` 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 `empleadoActual` contiene 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.

10.6 Asignaciones permitidas entre variables de la superclase y la subclase

Ahora que hemos visto una aplicación completa que procesa diversos objetos de las subclases en forma *polimórfica*, sintetizaremos lo que puede y lo que no puede 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, en realidad ambos son 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*, ya que dicha asignación dejaría los miembros de la subclase indefinidos para el objeto de la superclase.

Hemos visto tres maneras apropiadas 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 ésta es un proceso simple y directo.
2. Asignar una referencia de la subclase a una variable de ésta 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.

10.7 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 métodos 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`** no se puede extender para crear una subclase. 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, 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.

Ya hemos hablado sobre declarar variables, métodos y clases como `final`, y hemos enfatizado que si algo *puede* ser `final`, *debe* ser `final`. Los compiladores pueden realizar varias optimizaciones cuando saben que algo es `final`. Cuando estudiemos la concurrencia en el capítulo 23, verá que las variables `final` facilitan en gran medida la paralelización de sus programas para usarse en los procesadores multinúcleo de la actualidad. Para obtener más información sobre el uso de la palabra clave `final`, visite

```
http://docs.oracle.com/javase/tutorial/java/IandI/final.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. Además, a menos que diseñe con cuidado una clase para extenderse, debe declarar la clase como `final` para evitar errores (a menudo sutiles).

10.8 Una explicación más detallada de los problemas con las llamadas a métodos desde los constructores

No llame desde los constructores a los métodos que puedan sobrescribirse. Al crear un objeto de una *subclase*, esto podría provocar que se llamara un método sobrescrito antes de que se inicializara por completo el objeto de la *subclase*.

Recuerde que al construir el objeto de una *subclase*, su constructor llama primero a uno de los constructores de la *superclase* directa. Si el constructor de la *superclase* llama a un método que pueda sobrescribirse, el constructor de la *superclase* llamará a la versión de la *subclase* de ese método, antes de que el cuerpo del constructor de la *subclase* tenga la oportunidad de ejecutarse. Esto podría provocar errores sutiles y difíciles de detectar si el método de la *subclase* que se llamó depende de una inicialización que aún no se haya realizado en el cuerpo del constructor de la *subclase*.

Es aceptable llamar a un método `static` desde un constructor. Por ejemplo, un constructor y un método *establecer* realizan a menudo la misma validación para una variable de instancia específica. Si el código de validación es breve, es aceptable duplicarlo en el constructor y el método *establecer*. Si se requiere una validación más extensa, defina un método de validación `static` (por lo general un método ayudante `private`) y luego llámelo desde el constructor y el método *establecer*. También es aceptable que un constructor llame a un método de instancia `final`, siempre y cuando el método no llame de manera directa o indirecta un método de instancia que pueda sobrescribirse.

10.9 Creación y uso de interfaces

[Nota: como está escrita, esta sección y su código se aplican en Java SE 7. Las mejoras a la interfaz de Java SE 8 se presentan en la sección 10.10 y se describen con más detalle en el capítulo 17].

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 *polimórficamente* esas cosas *distintas* (como los pagos vencidos para los empleados y las facturas) 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 presenta 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 de manera implícita `public abstract`, y todos los campos son implícitamente `public`, `static` y `final`.



Buena práctica de programación 10.1

De acuerdo con la Especificación del lenguaje Java*, es un estilo apropiado declarar los métodos* `abstract` *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 de una interfaz 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** 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 método de una interfaz en una clase concreta que implemente a esa interfaz, se produce un error de compilación indicando que la clase debe declararse como `abstract`.

Relación de tipos dispares

Por lo general, una interfaz se utiliza cuando clases *dispares* (es decir, que no estén relacionadas por una jerarquía de clases) 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`. Al igual 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`.



Observación de ingeniería de software 10.7

Muchos desarrolladores sienten que las interfaces son una tecnología de modelado aún más importante que las clases, en especial con las nuevas mejoras a las interfaces en Java SE 8 (consulte la sección 10.10).

Etiquetado de interfaces

En el capítulo 15, Archivos, flujos y serialización de objetos, veremos la noción de *etiquetado de interfaces* (también conocidas como *interfaces marcadoras*), que son 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 15 hablaremos sobre 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 `Serializable`, 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`.

10.9.1 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 que posiblemente no están 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 por igual al método `obtenerMontoPago` en objetos `Factura` y `Empleado`. 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 de clases 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 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`.

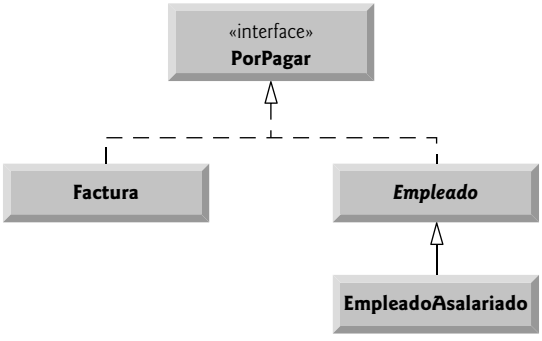


Fig. 10.10 | Diagrama de clases de UML de la jerarquía de la interfaz `PorPagar`.

10.9.2 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`. 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, pero 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 constantes `final 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.

10.9.3 La clase Factura

Ahora crearemos la clase `Factura` (figura 10.12) para representar una factura simple que contiene información de facturación sólo para cierto tipo de pieza. La clase declara como `private` las variables de instancia `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 26), métodos *obtener* y *establecer* (líneas 29 a 69) que manipulan las variables de instancia de la clase y un método `toString` (líneas 72 a 78) que devuelve una representación `String` de un objeto `Factura`. Los métodos `establecerCantidad` (líneas 41 a 47) y `establecerPrecioPorArticulo` (líneas 56 a 63) 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 final String numeroPieza;
7     private final String descripcionPieza;
8     private int cantidad;
9     private double precioPorArticulo;
10
11    // constructor
12    public Factura(String numeroPieza, String descripcionPieza, int cantidad,
13       double precioPorArticulo)
14    {
15       if (cantidad < 0) // valida la cantidad
16          throw new IllegalArgumentException ("Cantidad debe ser >= 0");
17
18       if (precioPorArticulo < 0.0) // valida el precioPorArticulo
19          throw new IllegalArgumentException(
20             "El precio por articulo debe ser >= 0");
21
22       this.cantidad = cantidad;
23       this.numeroPieza = numeroPieza;
24       this.descripcionPieza = descripcionPieza;
25       this.precioPorArticulo = precioPorArticulo;
26    } // fin del constructor
27
```

Fig. 10.12 | La clase `Factura`, que implementa a `Porpagar` (parte 1 de 3).

```
28    // obtiene el número de pieza
29    public String obtenerNumeroPieza()
30    {
31       return numeroPieza; // debe validar
32    }
33
34    // obtiene la descripción
35    public String obtenerDescripcionPieza()
36    {
37       return descripcionPieza;
38    }
39
40    // establece la cantidad
41    public void establecerCantidad(int cantidad)
42    {
43       if (cantidad < 0) // valida la cantidad
44          throw new IllegalArgumentException ("Cantidad debe ser >= 0");
45
46       this.cantidad = cantidad;
47    }
48
49    // obtener cantidad
50    public int obtenerCantidad()
51    {
52       return cantidad;
53    }
54
55    // establece el precio por artículo
56    public void establecerPrecioPorArticulo(double precioPorArticulo)
57    {
58       if (precioPorArticulo < 0.0) // valida el precioPorArticulo
59          throw new IllegalArgumentException(
60             "El precio por articulo debe ser >= 0");
61
62       this.precioPorArtculo = precioPorArticulo;
63    }
64
65    // obtiene el precio por artículo
66    public double obtenerPrecioPorArticulo()
67    {
68       return precioPorArticulo;
69    }
70
71    // devuelve representación String de un objeto Factura
72    @Override
73    public String toString()
74    {
75       return String.format("%s: %n%s: %s (%s) %n%s: %d %n%s: $%,.2f",
76          "factura", "numero de pieza", obtenerNumeroPieza(),
                                          obtenerDescripcionPieza(),
77          "cantidad", obtenerCantidad(), "precio por articulo", obtenerPrecioPor-
                                                                  Articulo());
78    }
79
```

Fig. 10.12 | La clase `Factura`, que implementa a `Porpagar` (parte 2 de 3).

```
80     // método requerido para realizar el contrato con la interfaz PorPagar
81     @Override
82     public double obtenerMontoPago()
83     {
84        return obtenerCantidad() * obtenerPrecioPorArticulo(); // calcula el costo
                                                                    total
85     }
86  } // fin de la clase Factura
```

Fig. 10.12 | La clase `Factura`, que implementa a `Porpagar` (parte 3 de 3).

Una clase puede extender sólo a otra clase pero puede implementar muchas interfaces
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 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 10.8
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 `abstract` de la interfaz `PorPagar`: el método `obtenerMontoPago` que se declara en las líneas 81 a 85. 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. Este método cumple con su propio requerimiento de implementación en la interfaz `PorPagar`; hemos *cumplido* con el *contrato de interfaz* con el compilador.

10.9.4 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 modificada, la cual es idéntica a la de la figura 10.4, con dos excepciones. En primer lugar, la línea 4 de la figura 10.13 indica que la clase `Empleado` ahora implementa a la interfaz `PorPagar`. Para este ejemplo, cambiamos el nombre del método `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 subclase *concreta* 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
```

Fig. 10.13 | La superclase `abstract` `Empleado`, que implementa a `PorPagar` (parte 1 de 2).

```
4  public abstract class Empleado implements PorPagar
5  {
6     private final String primerNombre;
7     private final String apellidoPaterno;
8     private final String numeroSeguroSocial;
9
10    // constructor
11    public Empleado(String primerNombre, String apellidoPaterno,
12       String numeroSeguroSocial)
13    {
14       this.primerNombre = primerNombre;
15       this.apellidoPaterno = apellidoPaterno;
16       this.numeroSeguroSocial = numeroSeguroSocial;
17    }
18
19    // devuelve el primer nombre
20    public String obtenerPrimerNombre()
21    {
22       return primerNombre;
23    }
24
25    // devuelve el apellido paterno
26    public String obtenerApellidoPaterno()
27    {
28       return apellidoPaterno;
29    }
30
31    // devuelve el número de seguro social
32    public String obtenerNumeroSeguroSocial()
33    {
34       return numeroSeguroSocial;
35    }
36
37    // devuelve representación String de un objeto Empleado
38    @Override
39    public String toString()
40    {
41       return String.format("%s %s%nnumero de seguro social: %s",
42          obtenerPrimerNombre(), obtenerApellidoPaterno(),
          obtenerNumeroSeguroSocial());
43    }
44
45    // Nota: Aquí no implementamos el método obtenerMontoPago de PorPagar, así que
46    // esta clase debe declararse como abstract para evitar un error de compilación.
47 } // fin de la clase abstracta Empleado
```

Fig. 10.13 | La superclase `abstract` `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`. Como la clase `Empleado` no proporciona un método `obtenerMontoPago`, la clase debe declararse como `abstract`. 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 45 y 46, 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 lo tanto también será una clase concreta.

10.9.5 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 39 a 43). Recuerde que la versión de `PorPagar` del método tiene un nombre más *general* para que pueda aplicarse a clases que sean posiblemente *dispares*. Si incluimos el resto de las subclases de `Empleado` de la sección 10.5 (`EmpleadoPorHoras`, `EmpleadoPorComision` y `EmpleadoBaseMasComision`) en ese ejemplo, también debe cambiarse el nombre de sus métodos `ingresos` por `obtenerMontoPago`. Dejaremos estas modificaciones para el ejercicio 10.15 y sólo utilizaremos a `EmpleadoAsalariado` en nuestro programa de prueba en esta sección. El ejercicio 10.16 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`.

```
1  // Fig. 10.14: EmpleadoAsalariado.java
2  // La clase EmpleadoAsalariado que implementa la interfaz PorPagar.
3  // método obtenerMontoPago
4  public class EmpleadoAsalariado extends Empleado
5  {
6     private double salarioSemanal;
7
8     // constructor
9     public EmpleadoAsalariado(String primerNombre, String apellidoPaterno,
10       String numeroSeguroSocial,  double salarioSemanal)
11    {
12       super(primerNombre, apellidoPaterno, numeroSeguroSocial);
13
14       if (salarioSemanal < 0.0)
15          throw new IllegalArgumentException(
16             "El salario semanal debe ser >= 0.0");
17
18       this.salarioSemanal = salarioSemanal;
19    }
20
21    // establece el salario
22    public void establecerSalarioSemanal(double salarioSemanal)
23    {
24       if (salariosemanal < 0.0)
25          throw new IllegalArgumentException(
26             "El salario semanal debe ser >= 0.0");
```

Fig. 10.14 | La clase `EmpleadoAsalariado`, que implementa el método `obtenerMontoPago` de la interfaz `PorPagar` (parte 1 de 2).

```
27
28          this.salarioSemanal = salarioSemanal;
29       }
30
31       // devuelve el salario
32       public double obtenerSalarioSemanal()
33       {
34          return salarioSemanal;
35       } // fin del método obtenerSalarioSemanal
36
37       // calcula los ingresos; implementa el método de la interfaz PorPagar
38       // que era abstracto en la superclase Empleado
39       @Override
40       public double obtenerMontoPago()
41       {
42          return obtenerSalarioSemanal();
43       }
44
45       // devuelve representación String de un objeto EmpleadoAsalariado
46       @Override
47       public String toString()
48       {
49          return String.format("empleado asalariado: %s%n%s: $%,.2f",
50             super.toString(), "salario semanal", obtenerSalarioSemanal());
51       }
52    } // fin de la clase EmpleadoAsalariado
```

Fig. 10.14 | La clase `EmpleadoAsalariado`, que implementa el método `obtenerMontoPago` de la interfaz `PorPagar` (parte 2 de 2).

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 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 10.9

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 10.10

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`.

10.9.6 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` *polimórficamente* 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 *polimórficamente* cada objeto `PorPagar` en `objetosPorPagar`, 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  // Programa de prueba de la interfaz PorPagar que procesa objetos
3  // Factura y Empleado mediante el polimorfismo.
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:");
21
22       // procesa en forma genérica cada elemento en el arreglo objetosPorPagar
23       for (PorPagar porPagarActual : objetosPorPagar)
24       {
```

Fig. 10.15 | Programa de prueba de la interfaz `PorPagar`, que procesa objetos `Factura` y `Empleado` de manera polimórfica (parte 1 de 2).

```
            // imprime porPagarActual y su monto de pago apropiado
            System.out.printf("%n%s %n%s: $%,.2f%n",
               porPagarActual.toString(), // se podría invocar de manera implícita
               "pago vencido", porPagarActual.obtenerMontoPago());
         }
      } // fin de main
   } // 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

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).

10.9.7 Algunas interfaces comunes de la API de Java

Utilizará mucho las interfaces cuando desarrolle aplicaciones de Java. La API de Java contiene numerosas interfaces, y muchos de los métodos de la API de Java reciben argumentos de interfaz y devuelven valores de interfaz. La figura 10.16 describe algunas de las interfaces más populares de la API de Java que utilizamos en capítulos posteriores.

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 16, Colecciones genéricas y en el capítulo 20 en inglés, Generic Classes and Methods (en el sitio Web del libro), utilizaremos a `Comparable`.

Fig. 10.16 | Interfaces comunes de la API de Java (parte 1 de 2).

Interfaz	Descripción
`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 15, Archivos, flujos y serialización de objetos, y en el capítulo 28 en inglés, Networking (en el sitio Web del libro), utilizaremos a `Serializable`.
`Runnable`	La implementa cualquier clase que represente una tarea a realizar. Sus objetos con frecuencia se ejecutan en paralelo, usando una técnica llamada *multihilos* (que veremos en el capítulo 23 en inglés, Concurrency, en el sitio Web del libro). La interfaz contiene un método, `run`, que describe el comportamiento de un objeto al ejecutarse.
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 12, Componentes de la GUI, parte 1 y en el capítulo 22 en inglés, GUI Components: Part 2 (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.
`AutoCloseable`	Es implementada por clases que pueden usarse con la instrucción `try` con recursos (capítulo 11, Manejo de excepciones: un análisis más detallado) para ayudar a evitar fugas de recursos.

Fig. 10.16 | Interfaces comunes de la API de Java (parte 2 de 2).

10.10 Mejoras a las interfaces de Java SE 8

Esta sección presenta las nuevas características de las interfaces de Java SE 8. Hablaremos sobre ellas con más detalle en capítulos posteriores.

10.10.1 Métodos `default` de una interfaz

Antes de Java SE 8, los métodos de las interfaces *solamente* podían ser métodos `public abstract`. Esto significa que una interfaz especificaba *qué* operaciones debía realizar una clase implementadora, pero no *cómo* debía realizarlas.

En Java SE 8 las interfaces también pueden contener **métodos `default public`** con implementaciones predeterminadas *concretas* que especifiquen *cómo* deben realizarse las operaciones cuando una clase implementadora no sobrescriba los métodos. Si una clase implementa dicha interfaz, la clase también recibe las implementaciones `default` de esa interfaz (si las hay). Para declarar un método `default`, coloque la palabra clave `default` antes del tipo de valor de retorno del método y proporcione una implementación concreta del mismo.

Agregar métodos a las interfaces existentes

Antes de Java SE 8, agregar métodos a una interfaz descompondría a las clases implementadoras que no implementaran los nuevos métodos. Recuerde que, si no implementó cada uno de los métodos de la interfaz, tiene que declarar su clase como `abstract`.

Cualquier clase que implemente a la interfaz original *no* se descompondrá cuando se agregue un método `default`. La clase simplemente recibirá el nuevo método `default`. Cuando una clase implementa una interfaz de Java SE 8, la clase "firma un contrato" con el compilador que dice: "Declararé todos los métodos `abstract` especificados por la interfaz o declararé mi clase como `abstract`". La clase implementadora no tiene que sobrescribir los métodos `default` de la interfaz, pero puede hacerlo si es necesario.



Observación de ingeniería de software 10.11

Los métodos `default` de Java SE 8 le permiten evolucionar las interfaces existentes al agregar nuevos métodos a esas interfaces sin descomponer el código que las usa.

Comparación entre interfaces y clases `abstract`

Antes de Java SE 8, se utilizaba por lo general una interfaz (en vez de una clase `abstract`) cuando no había detalles de implementación a heredar; es decir, no había campos ni implementaciones de métodos. Con los métodos `default`, puede declarar implementaciones de métodos comunes en las interfaces, lo que le da más flexibilidad al diseñar sus clases.

10.10.2 Métodos `static` de una interfaz

Antes de Java SE 8, era común asociar con una interfaz a una clase que tuviera métodos ayudantes `static` para trabajar con objetos que implementaran la interfaz. En el capítulo 16 aprenderá sobre la clase `Collections`, que contiene muchos métodos ayudantes `static` para trabajar con objetos que implementan las interfaces `Collection`, `List`, `Set` y demás. Por ejemplo, el método `sort` de `Collections` puede ordenar objetos de *cualquier* clase que implemente a la interfaz `List`. Con los métodos `static` de una interfaz, dichos métodos ayudantes pueden ahora declararse directamente en las interfaces, en vez de hacerlo en cada una de las clases.

10.10.3 Interfaces funcionales

A partir de Java SE 8, cualquier interfaz que contenga sólo un método `abstract` se conoce como **interfaz funcional**. Hay muchas de esas interfaces en las API de Java SE 7, y muchas nuevas en Java SE 8. Algunas de las interfaces funcionales que usará en este libro son:

- `ActionListener` (capítulo 12): usted podrá implementar esta interfaz para definir un método que se invoca cuando el usuario hace clic en un botón.
- `Comparator` (capítulo 16): usted podrá implementar esta interfaz para definir un método que puede comparar dos objetos de un tipo dado para determinar si el primer objeto es menor, igual o mayor que el segundo.
- `Runnable` (capítulo 23): usted podrá implementar esta interfaz para definir una tarea que pueda ejecutarse en paralelo con otras partes de su programa.

Las interfaces funcionales se usan mucho con las nuevas capacidades lambda de Java SE 8 que presentaremos en el capítulo 17. En el capítulo 12 implementará frecuentemente una interfaz mediante la creación de lo que se conoce como una clase interna anónima, la cual implementa el o los métodos de la interfaz. En Java SE 8, las lambdas proveen una notación abreviada para crear *métodos anónimos* que el compilador traduce de manera automática en clases internas anónimas.

10.11 (Opcional) Ejemplo práctico de GUI y gráficos: realización de dibujos mediante el polimorfismo

Tal vez haya observado en el programa de dibujo que creamos en el ejercicio 8.1 del ejemplo práctico de GUI y gráficos (y que modificamos en el ejercicio 9.1 del ejemplo práctico 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 ejemplo práctico de GUI y gráficos

10.1 Modifique las clases `MiLinea`, `MiOvalo` y `MiRectangulo` de los ejercicios 8.1 y 9.1 del ejemplo práctico 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 *polimórficamente* por el resto de su vida como un objeto `MiFigura`.

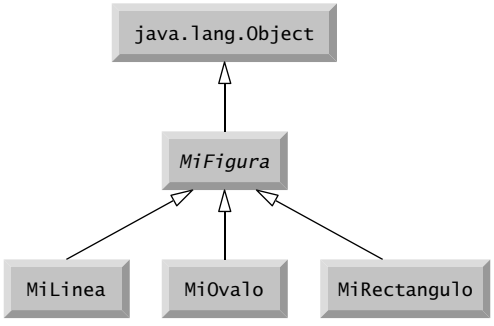


Fig. 10.17 | La jerarquía `MiFigura`.

En su solución, la clase `MiFigura` de 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.
e) El método `abstract`

```
public abstract void dibujar(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, para 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 la figura se dibujará 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 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 anterior, usted creó una jerarquía `MiFigura` en la cual las clases `MiLinea`, `MiOvalo` y `MiRectangulo` extienden directamente a `MiFigura`. 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.

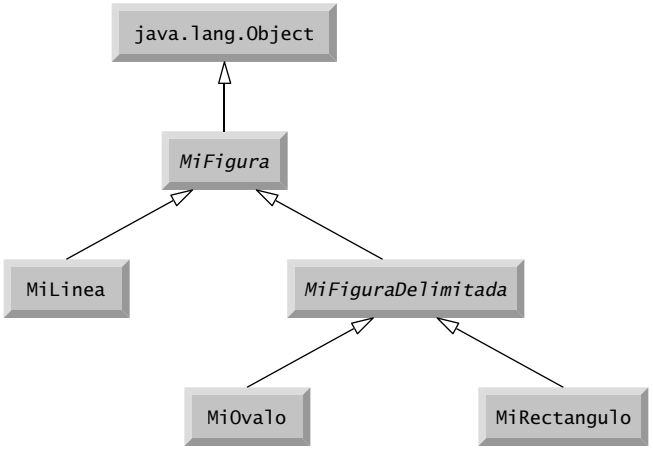


Fig. 10.18 | Jerarquía `MiFigura` con `MiFiguraDelimitada`.

10.12 Conclusión

En este capítulo se presentó el polimorfismo, que es 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. Explicamos los conceptos de los métodos y clases `final`. Por último, hablamos también sobre la declaración e implementación de una interfaz, como otra manera en la que clases posiblemente dispares implementen una funcionalidad común, permitiendo que los objetos de esas clases se procesen mediante el polimorfismo.

Ahora deberá estar familiarizado con las clases, los objetos, el encapsulamiento, la herencia, las interfaces y el polimorfismo, que son 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. 396) 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. 401) 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. 402). Dichas clases proporcionan implementaciones de cada método que declaran (algunas de las implementaciones pueden heredarse).
- Con frecuencia, los programadores escriben código cliente que utiliza sólo superclases abstractas (pág. 402) 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. 402).
- Los métodos abstractos no proporcionan implementaciones.
- Una clase que contiene métodos abstractos debe declararse como clase `abstracta` (pág. 402). 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 Ejemplo práctico: 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. 417) o vinculación postergada.
- La variable de una superclase puede utilizarse para invocar sólo a los métodos declarados en la superclase.
- El operador `instanceof` (pág. 417) 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. 418), 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 10.7 Métodos y clases `final`

- Un método que se declara como `final` (pág. 419) 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. 420).
- El compilador puede optimizar los programas al eliminar las llamadas a los métodos `final` y sustituirlas en línea con el código expandido de sus declaraciones en cada una de las ubicaciones de las llamadas al método.
- Una clase que se declara como `final` no puede extenderse (pág. 420).
- Todos los métodos en una clase `final` son implícitamente `final`.

Sección 10.9 Creación y uso de interfaces

- Una interfaz (pág. 421) 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. 421).
- 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 (pág. 421) 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, ya que 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 por sí solas en archivos 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`.

Sección 10.10 Mejoras a las interfaces de Java SE 8

- En Java SE 8, una interfaz puede declarar métodos `default`; es decir, métodos `public` con implementaciones concretas que especifiquen cómo debe realizarse una operación.
- Cuando una clase implementa a una interfaz, la clase recibe las implementaciones concretas `default` de la interfaz si es que ésta no las sobrescribe.
- Para declarar un método `default` en una interfaz, sólo tiene que colocar la palabra clave `default` antes del tipo de valor de retorno del método y proporcionar un cuerpo completo del método.
- Al mejorar una interfaz existente con métodos `default` (cualquier clase que haya implementado la interfaz original no se descompondrá), simplemente recibirá las implementaciones predeterminadas de los métodos.
- Con los métodos `default`, puede declarar las implementaciones de los métodos comunes en las interfaces (en vez de las clases `abstract`), lo cual le da más flexibilidad al diseñar sus clases.
- A partir de Java SE 8, las interfaces ahora pueden incluir métodos `public static`.
- A partir de Java SE 8, cualquier interfaz que contenga sólo un método se conoce como interfaz funcional. Hay muchas de esas interfaces en las API de Java.
- Las interfaces funcionales se usan mucho con las nuevas capacidades lambda de Java SE 8. Como veremos, las lambdas proporcionan una notación abreviada para crear métodos anónimos.

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.

10.3 *(Interfaces de Java SE 8)* Complete las siguientes oraciones:
a) En Java SE 8, una interfaz puede declarar _____; es decir, métodos `public` con implementaciones concretas que especifiquen cómo debe realizarse una operación.
b) A partir de Java SE 8, las interfaces pueden incluir ahora métodos ayudantes _____.
c) A partir de Java SE 8, cualquier interfaz que contenga sólo un método se conoce como _____.

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.

10.3 a) métodos `default`. b) `static`. c) interfaz funcional.

Ejercicios

10.4 ¿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".

10.5 ¿Qué son los métodos abstractos? Describa las circunstancias en las que un método abstracto sería apropiado.

10.6 ¿Cómo es que el polimorfismo fomenta la extensibilidad?

10.7 Describa tres formas apropiadas en las que podemos asignar referencias de superclases y subclases a variables de los tipos de las superclases y las subclases.

10.8 Compare y contraste las clases abstractas y las interfaces. ¿Para qué podría usar una clase abstracta? ¿Para qué podría usar una interfaz?

10.9 ***(Interfaces de Java SE 8)*** Explique cómo es que los métodos `default` le permiten agregar nuevos métodos a una interfaz existente sin descomponer las clases que implementaron la interfaz original.

10.10 ***(Interfaces de Java SE 8)*** ¿Qué es una interfaz funcional?

10.11 ***(Interfaces de Java SE 8)*** ¿Por qué es útil poder agregar métodos `static` a las interfaces?

10.12 ***(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`.

10.13 ***(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.

10.14 ***(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.

10.15 ***(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 el inciso (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`.

10.16 ***(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`. De esta forma, 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 10.15, 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`.

Marcando la diferencia

10.17 ***(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.

Manejo de excepciones: un análisis más detallado

11



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:

- Qué son las excepciones y cómo se manejan.
- Cuándo usar el manejo de excepciones.
- A utilizar bloques `try` para delimitar el código en el que podrían ocurrir excepciones.
- A usar `throw` para indicar un problema.
- A usar bloques `catch` para especificar manejadores de excepciones.
- A utilizar el bloque `finally` para liberar recursos.
- La jerarquía de clases de excepciones.
- A crear excepciones definidas por el usuario.

Plan general

11.1 Introducción
11.2 Ejemplo: división entre cero sin manejo de excepciones
11.3 Ejemplo: manejo de excepciones tipo `ArithmeticException` e `InputMismatchException`
11.4 Cuándo utilizar el manejo de excepciones
11.5 Jerarquía de excepciones en Java
11.6 Bloque `finally`
11.7 Limpieza de la pila y obtención de información de un objeto excepción
11.8 Excepciones encadenadas
11.9 Declaración de nuevos tipos de excepciones
11.10 Precondiciones y poscondiciones
11.11 Aserciones
11.12 Cláusula `try` con recursos: desasignación automática de recursos
11.13 Conclusión

Resumen | *Ejercicios de autoevaluación* | *Respuestas a los ejercicios de autoevaluación* | *Ejercicios*

11.1 Introducción

Como vimos en el capítulo 7, una excepción 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 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, respectivamente. A continuación presentaremos las *aserciones*, que los programadores utilizan en tiempo de desarrollo para facilitar el proceso de depurar su código. También hablaremos de dos nuevas características de manejo de excepciones que se introdujeron en Java SE 7: atrapar varias excepciones con un solo manejador `catch` y la nueva instrucción `try` con recursos, que libera automáticamente un recurso después de usarlo en el bloque `try`.

Este capítulo se enfoca en los conceptos de manejo de excepciones y presenta varios ejemplos mecánicos que demuestran diversas características. Como veremos en capítulos posteriores, muchos métodos de las API de Java lanzan excepciones que manejamos en nuestro código. La figura 11.1 muestra algunos de los tipos de excepciones que ya hemos visto y otros que verá más adelante.

1 A. Koenig y B. Stroustrup, "Exception Handling for C++ (revised)", *Proceedings of the Usenix C++ Conference*, págs. 149-176, San Francisco, abril de 1990.

Capítulo	Ejemplo de excepciones utilizadas
Capítulo 7	`ArrayIndexOutOfBoundsException`
Capítulos 8 a 10	`IllegalArgumentException`
Capítulo 11	`ArithmeticException`, `InputMismatchException`
Capítulo 15	`SecurityException`, `FileNotFoundException`, `IOException`, `ClassNotFoundException`, `IllegalStateException`, `FormatterClosedException`, `NoSuchElementException`
Capítulo 16	`ClassCastException`, `UnsupportedOperationException`, `NullPointerException`, tipos de excepciones personalizadas
Capítulo 20	`ClassCastException`, tipos de excepciones personalizadas
Capítulo 21	`IllegalArgumentException`, tipos de excepciones personalizadas
Capítulo 23	`InterruptedException`, `IllegalMonitorStateException`, `ExecutionException`, `CancellationException`
Capítulo 28	`MalformedURLException`, `EOFException`, `SocketException`, `InterruptedException`, `UnknownHostException`
Capítulo 24	`SQLException`, `IllegalStateException`, `PatternSyntaxException`
Capítulo 31	`SQLException`

Fig. 11.1 | Varios tipos de excepciones que verá en este libro.

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 11.2 se pide al usuario que introduzca dos enteros y éstos se pasan al método `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. 11.2: DivisionEntreCeroSinManejoDeExcepciones.java
2  // División de enteros 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    }
12
13    public static void main(String[] args)
14    {
15       Scanner explorador = new Scanner(System.in);
```

Fig. 11.2 | División de enteros sin manejo de excepciones (parte 1 de 2).

```
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       }
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. 11.2 | División de enteros sin manejo de excepciones (parte 2 de 2).

Rastreo de la pila

La primera de las tres ejecuciones de ejemplo en la figura 11.2 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.

Rastreo de la pila para una excepción `ArithmeticException`

La primera línea especifica que ocurrió 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`). Si necesita comparar un valor de punto flotante con `NaN`, use el método `isNaN` de la clase `Float` (para valores `float`) o de la clase `Double` (para valores `double`). Las clases `Float` y `Double` están en el paquete `java.lang`.

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**, que es el punto inicial en el que ocurrió la excepción. El punto de lanzamiento de esta excepción está en la línea 10 del método `cociente`.

Rastreo de la pila para una excepción ***`InputMismatchException`***

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 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 (`string`) 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.

Terminación del programa

En las ejecuciones de ejemplo de la figura 11.2, 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 figura 11.2, 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 11.3, que se basa en la figura 11.2, 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, permite al usuario tratar de introducir los datos de entrada otra vez.

```
1  // Fig. 11.3: 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    }
14
15    public static void main(String[] args)
16    {
17       Scanner explorador = new Scanner(System.in);
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          }
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%n");
41          }
42          catch (ArithmeticException arithmeticException)
43          {
44             System.err.printf("%nExcepcion: %s%n", arithmeticException);
45             System.out.printf(
46                "Cero es un denominador invalido. Intente de nuevo.%n%n");
47          }
48       } while (continuarCiclo);
49    }
50 } // fin de la clase DivisionEntreCeroConManejoDeExcepciones
```

Fig. 11.3 | Manejo de excepciones `ArithmeticException` e `InputMismatchException` (parte 1 de 2).

```
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. 11.3 | Manejo de excepciones `ArithmeticException` e `InputMismatchException` (parte 2 de 2).

La primera ejecución de ejemplo de la figura 11.3 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 `true` 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.

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].

Al 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 directa o indirecta 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 del tipo `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`.

Multi-catch

Es relativamente común que después de un bloque `try` vayan varios bloques `catch` para manejar diversos tipos de excepciones. Si los cuerpos de varios bloques `catch` son idénticos, puede usar la característica **multi-catch** (introducida en Java SE 7) para atrapar esos tipos de excepciones en un *solo* manejador `catch` y realizar la misma tarea. La sintaxis para una instrucción *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 *cualquiera* de los tipos (o sus subclases) pueden atraparse en el manejador de excepciones. En una instrucción multi-`catch` puede especificarse cualquier número de tipos `Throwable`.

Excepciones no atrapadas

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 11.2, 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, en el que cada **hilo** es una *actividad concurrente*. Un programa puede tener muchos hilos. Si un programa sólo tiene *un* hilo, una excepción no atrapada hará que el programa termine. Si un programa 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 23 en línea, Concurrency, 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 11.3), 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 11.3, 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.

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 instrucción 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 sobre otra posibilidad, cuando hablemos del bloque `finally`). Ahora la condición del ciclo `do...while` es `false`, y el método `main` termina.

El bloque `try` y sus correspondientes bloques `catch` o `finally` forman en conjunto una **instrucción `try`**. Es importante no confundir los términos "bloque `try`" e "instrucción `try`; esta última incluye el bloque `try`, así como los siguientes bloques `catch` 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`

En el método `cociente` (figura 11.3; líneas 9 a 13), la línea 10 se conoce como **cláusula `throws`**. Esta cláusula especifica las excepciones que el método *podría* lanzar si ocurren problemas. La cláusula, que debe aparecer después de la lista de parámetros del método y antes de su cuerpo, contiene una lista separada por comas de los tipos de excepciones. Dichas excepciones pueden lanzarse mediante instrucciones en el cuerpo del método, o a través de métodos que se llamen desde ahí. Hemos agregado la cláusula `throws` a esta aplicación, para indicar que este método puede lanzar una excepción `ArithmeticException`. Por ende, a los clientes del método `cociente` se les informa que el método podría lanzar una excepción `ArithmeticException`. Algunos tipos de excepciones, como `ArithmeticException`, no tienen que aparecer en la lista de la cláusula `throws`. Para las que sí deben aparecer, el método puede lanzar excepciones que tengan la relación *es un(a)* con las clases en la lista de la cláusula `throws`. En la sección 11.5 aprenderá más acerca de esto.



Tip para prevenir errores 11.1

Antes de usar un método en un programa, lea la documentación en línea de la API para saber acerca del mismo. La documentación especifica las excepciones que lanza el método (si las 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 hace fallar la condición en la línea 48, con lo cual el método `main` termina de ejecutarse en forma normal.

Cuando `cociente` lanza una excepción `ArithmeticException`, `cociente` *termina* y *no* devuelve un valor, por lo que sus *variables locales quedan fuera de alcance* (y se destruyen). Si `cociente` contiene variables 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* y la *interrupción de hilos* (como veremos en el capítulo 23), así como 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 e *independientemente* del flujo de control del programa.



Observación de ingeniería de software 11.2

Incorpore su estrategia de manejo de excepciones y recuperación de errores en sus sistemas, partiendo desde el principio del proceso de diseño. Puede ser difícil incluir esto después de haber implementado un sistema.



Observación de ingeniería de software 11.3

El manejo de excepciones proporciona una sola técnica uniforme para documentar, detectar y recuperarse de los errores. Esto ayuda a los programadores que trabajan en proyectos extensos a comprender el código de procesamiento de errores de los demás programadores.



Observación de ingeniería de software 11.4

Hay una gran variedad de situaciones que generan excepciones: es más fácil recuperarse de unas que de otras.

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 11.4 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` (paquete `java.lang`) e `IOException` (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 `docs.oracle.com/javase/7/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.

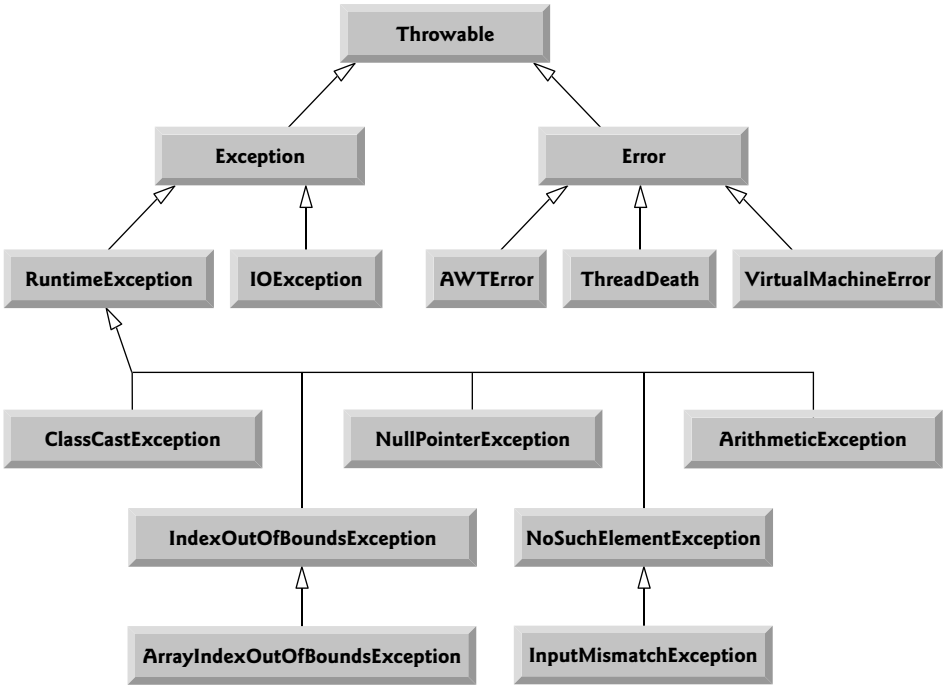


Fig. 11.4 | Porción de la jerarquía de herencia de la clase `Throwable`.

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 requerimientos especiales para las excepciones *verificadas* (que veremos en breve). El tipo de una excepción determina si es verificada o no verificada.

Las excepciones `RuntimeException` son excepciones no verificadas
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). Puede evitar estas excepciones asegurándose de que los índices de sus arreglos siempre sean mayores o iguales a 0 y menores que la longitud (`length`) del arreglo.
- Las excepciones `ArithmeticException` (que se muestran en la figura 11.3). Para evitar la excepción `ArithmeticException` que ocurre al dividir entre cero, revise el denominador para determinar si es 0 antes de realizar el cálculo.

Las clases que heredan de manera directa o indirecta de la clase `Error` (figura 11.4) son *no verificadas*, ya que los errores del tipo `Error` son problemas tan serios que su programa no debe ni siquiera intentar lidiar con ellos.

Excepciones verificadas
Todas las clases que heredan de la clase `Exception` pero *no* directa o indirectamente 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 si éste no existe.

El compilador y las excepciones verificadas

El compilador verifica cada una de las llamadas a un método, junto con su declaración, para determinar si el método lanza una excepción verificada. De ser así, el compilador asegura que la excepción verificada sea *atrapada* o *declarada* en una cláusula `throws`. A esto se le conoce como **requerimiento de atrapar o declarar**. En los siguientes ejemplos le mostraremos cómo atrapar o 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 una de sus superclases). 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. Esto obliga a los programadores a pensar sobre los problemas que pueden ocurrir cuando se hace una llamada a un método que lanza excepciones verificadas.



Tip para prevenir errores 11.2

Los programadores deben atender las excepciones verificadas. Esto produce un código más robusto que el que se crearía si los programadores simplemente ignoran las excepciones.



Error común de programación 11.2

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 excepciones verificadas, éstas deben atraparse o declararse. Si una excepción puede manejarse de manera significativa en un método, éste debe atrapar la excepción en vez de declararla.

El compilador y las excepciones no verificadas

A diferencia de las excepciones verificadas, el compilador de Java *no* examina 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 11.3 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* `NumberFormatException` *sea una subclase indirecta de* `RuntimeException` *(y, por ende, una excepción no verificada). 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 las excepciones relacionadas mediante el *polimorfismo*. Si estas excepciones requieren un procesamiento distinto, puede atrapar individualmente a cada subclase.

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 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 bloques `catch` cuyos tipos sean los mismos que el tipo de excepción, o de una superclase de ese tipo. Por ejemplo, podríamos colocar después de un bloque `catch` para el tipo `ArithmeticException` un bloque `catch` para el tipo `Exception`; ambos coincidirían con las excepciones `ArithmeticException`, pero sólo se ejecutaría el primer bloque `catch` que coincidiera.



Error común de programación 11.3
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.



Tip para prevenir errores 11.3
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 bloques `catch` de todas las otras subclases aseguramos que todas las excepciones de las subclases se atrapen en un momento dado.



Observación de ingeniería de software 11.7
En la industria no se recomienda lanzar o atrapar el tipo `Exception`; aquí lo usamos sólo para demostrar la mecánica del manejo de excepciones. En capítulos subsiguientes, por lo general lanzaremos y atraparemos tipos de excepciones más específicos.

11.6 Bloque `finally`

Los programas que obtienen ciertos recursos deben devolverlos al sistema 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 11.4
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.

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`.

Cuándo se ejecuta el bloque `finally`

El bloque `finally` se ejecutará, independientemente de que *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 único caso en donde el bloque `finally` *no* se ejecutará es si la aplicación *sale antes de tiempo* de un bloque `try`, llamando al método `System.exit`. Este método, que demostraremos en el capítulo 15, termina de *inmediato* una aplicación.

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 omite el resto del bloque `try` y el control pasa al bloque `finally`. Luego, 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`. Además, la excepción podría quedar *sin atraparse* (como vimos en la sección 11.3).

Si un bloque `catch` lanza una excepción, el bloque `finally` se sigue ejecutando. Luego la excepción se pasa al siguiente bloque `try` exterior, que de nuevo suele ser el método que hizo la llamada.

Liberar recursos en un bloque `finally`

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 11.5

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 antes 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 y el rendimiento de los programas.

Demostración del bloque `finally`

La figura 11.5 demuestra que el bloque `finally` se ejecuta, aun 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. 11.5: 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();
11       }
12       catch (Exception excepcion) // excepción lanzada por ThrowException
13       {
14          System.err.println("La excepcion se manejo en main");
15       }
16
17       noLanzaExcepcion();
18    }
19
20    // demuestra los bloques try...catch...finally
21    public static void lanzaExcepcion() throws Exception
22    {
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       }
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       }
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       }
41
42       // no se llegaría al código que se coloque aquí; se producirían errores de
            compilación
43
44    }
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");
```

Fig. 11.5 | Mecanismo de manejo de excepciones `try`...`catch`...`finally` (parte 1 de 2).

```
52          } // fin de try
53          catch (Exception excepcion) // no se ejecuta
54          {
55             System.err.println(excepcion);
56          }
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          }
62
63          System.out.println("Fin del metodo noLanzaExcepcion");
64       }
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. 11.5 | Mecanismo de manejo de excepciones `try`...`catch`...`finally` (parte 2 de 2).

Tanto `System.out` como `System.err` son **flujos**; es decir, una secuencia de bytes. Mientras que `System.out` (conocido como **flujo de salida estándar**) muestra la salida de un programa, `System.err` (conocido como **flujo de error estándar**) muestra 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 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* redirigiremos la salida de `System.err`, sino que mostraremos dichos mensajes en el *símbolo del sistema*. En el capítulo 15 aprenderá más acerca de los flujos.

Lanzamiento de excepciones mediante la instrucción `throw`

El método `main` (figura 11.5) 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`** y se ejecuta para indicar que ocurrió 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 ocurrió un error. Una instrucción `throw` especifica que un objeto se lanzará. El operando de `throw` puede ser de cualquier clase derivada de `Throwable`.



Observación de ingeniería de software 11.8

Cuando se invoca el método `toString` en cualquier objeto `Throwable`, su objeto `String` resultante incluye la cadena descriptiva que se suministró al constructor, o simplemente el nombre de la clase, si no se suministró una cadena.



Observación de ingeniería de software 11.9

Una excepción 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 11.10

Lance excepciones desde los constructores para indicar que los parámetros del constructor no son válidos. Esto evita que se cree un objeto en un estado inválido.

Relanzamiento de excepciones

La línea 32 de la figura 11.5 **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 11.4

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 11.6

Evite colocar código que pueda lanzar una excepción en un bloque `finally`. Si se requiere dicho código, enciérrelo en bloques `try...catch` dentro del bloque `finally`.



Error común de programación 11.5

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 11.1

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*, que todas las variables locales en ese método *quedan fuera de alcance* y que 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 esa excepción. Si un bloque `try` no encierra a esa instrucción o si no se atrapa la excepción, se lleva a cabo otra vez la limpieza de la pila. La figura 11.6 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 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. Puesto 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.

```
1  // Fig. 11.6: 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();
11       }
12       catch (Exception excepcion) // atrapa la excepción lanzada en metodo1
13       {
14          System.err.printf("%s%n%n", excepcion.getMessage());
15          excepcion.printStackTrace();
16
```

Fig. 11.6 | Limpieza de la pila y obtención de datos de un objeto excepción (parte 1 de 2).

```
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:%n");
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          }
31       }
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    }
39
40    // llama a metodo3; lanza las excepciones de vuelta a metodo1
41    public static void metodo2() throws Exception
42    {
43       metodo3();
44    }
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    }
51 } // fin de la clase UsoDeExcepciones
```

```
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. 11.6 | Limpieza de la pila y obtención de datos de un objeto excepción (parte 2 de 2).

Obtención de datos de un objeto excepción

Recuerde que las excepciones se derivan de la clase `Throwable`, la cual 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 11.7

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. En el capítulo 23 hablaremos sobre la tecnología multihilos.



Tip para prevenir errores 11.8

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 11.6 (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 15 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, mientras que *númeroLínea* indica en qué parte del archivo ocurrió la excepción. Usted vio esto en los resultados para la figura 11.2. 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 11.11

A veces tal vez sea conveniente ignorar una excepción escribiendo un manejador `catch` con un cuerpo vacío. Antes de hacerlo, asegúrese de que la excepción no indique una condición que un código más arriba en la jerarquía necesite conocer o de la cual deba recuperarse.

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 e 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 11.7 se demuestran las excepciones encadenadas.

```
1  // Fig. 11.7: UsoDeExcepcionesEncadenadas.java
2  // Las excepciones encadenadas.
3
4  public class UsoDeExcepcionesEncadenadas
5  {
6     public static void main(String[] args)
7     {
8        try
9        {
10          metodo1();
11       }
12       catch (Exception excepcion) // excepciones lanzadas desde metodo1
13       {
14          excepcion.printStackTrace();
15       }
16    }
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();
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       }
29    }
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();
37       }
38       catch (Exception excepcion) // excepción lanzada desde metodo3
39       {
40          throw new Exception("La excepcion se lanzo en metodo2", excepcion);
41       }
42    }
43
```

Fig. 11.7 | Excepciones encadenadas (parte 1 de 2).

```
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      }
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. 11.7 | Excepciones encadenadas (parte 2 de 2).

Flujo de control del programa

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.

Salida del programa

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, ésta muestra cuántas excepciones encadenadas más quedan pendientes.

11.9 Declaración de nuevos tipos de excepciones

Para crear aplicaciones de Java, 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). 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.

Un nuevo tipo de excepción debe extender a uno existente

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. Una clase de excepción es igual que cualquier otra clase; sin embargo, una nueva clase común de excepción 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
- uno que recibe un objeto `Throwable` (para encadenar excepciones) y pasa sólo este objeto al constructor de la superclase.



Buena práctica de programación 11.2

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. Si a los clientes se les pide manejar la excepción, la nueva clase de excepción debe ser una excepción verificada (es decir, debe extender a `Exception` pero no a `RuntimeException`). La aplicación cliente debe ser capaz de recuperarse en forma razonable de una excepción de este tipo. Si el código cliente debe ser capaz de ignorar la excepción (es decir, si la excepción es una excepción no verificada), la nueva clase de excepción debe extender a `RuntimeExcepcion`.

Ejemplos de una clase de excepción personalizada

En el capítulo 21 en línea, Custom Generic Data Structures, 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`, como eliminar un elemento de la parte frontal o posterior de la lista, no se permitirán si la `Lista` está vacía. Por esta razón, algunos métodos de `Lista` lanzan excepciones de la clase de excepción `ListaVaciaException`.



Buena práctica de programación 11.3
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, ellos comúnmente especifican los estados esperados antes y después de la ejecución de un método. A estos estados se les llama precondiciones y poscondiciones, respectivamente.

Precondiciones
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.

Poscondiciones
Una **poscondición** es verdadera *una vez que el método regresa con éxito*. Las poscondiciones describen las *restricciones en el valor de retorno*, así como cualquier otro *efecto secundario* que pueda tener el método. Al definir un método, usted debe documentar todas las poscondiciones, de manera que otros sepan qué pueden esperar al llamar a su método, y debe asegurarse que su método cumpla con todas sus poscondiciones, si en definitiva se cumplen sus precondiciones.

Lanzamiento de excepciones cuando no se cumplen las precondiciones o poscondiciones
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 en realidad obtiene 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 sobre el estado de un programa a la hora de invocar un método, y las poscondiciones son aserciones sobre el 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 11.8 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. 11.8: 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
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    }
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. 11.8 | Comprobar mediante `assert` que un valor esté dentro del rango.

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
```



Observación de ingeniería de software 11.13

Los usuarios no deben encontrar ningún error tipo `AssertionError`*; éstos deben usarse sólo durante el desarrollo del programa. Por esta razón, nunca se debe atrapar una excepción tipo* `AssertionError`*. En vez de ello, debemos permitir que el programa termine para poder ver el mensaje de error; después hay que localizar y corregir el origen del problema. No debemos usar la instrucción* `assert` *para indicar problemas en tiempo de ejecución en el código de producción (como lo hicimos en la figura 11.8 para fines demostrativos); debemos usar el mecanismo de las excepciones para este fin.*

11.12 Cláusula `try` con recursos: 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** (que se introdujo en Java SE 7), la cual simplifica la escritura de código en el que uno o más recursos se obtienen, se utilizan en un bloque `try` y se liberan en el correspondiente bloque `finally`. Por ejemplo, una aplicación de procesamiento de archivos 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; demostraremos esto en el capítulo 15. Cada recurso debe ser un objeto de una clase que implemente a la interfaz `AutoCloseable`, y por ende proporciona 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 (`;`). En los capítulos 15 y 24 veremos ejemplos de la instrucción `try` con recursos.

11.13 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 sobre 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.

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.

También le presentamos 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 veremos un análisis más detallado de las interfaces gráficas de usuario (GUI).

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 (pág. 443) cuando un método detecta un problema y no puede manejarlo.
- El rastreo de la pila de una excepción (pág. 444) incluye el nombre de la excepción en un mensaje que indica el problema que ocurrió 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. 445).

Sección 11.3 Ejemplo: manejo de excepciones `ArithmeticException` e `InputMismatchException`

- Un bloque `try` (pág. 447) encierra el código que podría lanzar 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. 448) 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.
- Justo después del bloque `try` debe ir por lo menos un bloque `catch` o un bloque `finally` (pág. 448).
- 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.
- Una excepción no atrapada (pág. 449) es una excepción que ocurre y no tiene bloques `catch` que coincidan. Una excepción no atrapada 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.

- La cláusula multi-`catch` (pág. 449) 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 (`|`).
- 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. 449).
- Si hay varios bloques `catch` que coinciden cuando ocurre una excepción, sólo se ejecuta el primero.
- Una cláusula `throws` (pág. 450) 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. 451), que ocurren cuando se ejecuta una instrucción.
- El manejo de excepciones no está diseñado para procesar los problemas asociados con eventos asíncronos (pág. 451), 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. 451) tiene dos subclases: `Exception` y `Error`.
- La clase `Exception` y sus subclases representan problemas 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. 452): 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. 452), 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. 454). Por lo general, el código para liberar recursos se coloca en un bloque `finally` (pág. 454).
- 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. 457) puede lanzar cualquier objeto `Throwable`.
- Las excepciones se vuelven a lanzar (pág. 458) 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 la depuración.
- La clase `Throwable` también proporciona un método `getStackTrace`, que obtiene la misma información de rastreo de la pila que `printStackTrace` imprime (pág. 461).
- El método `getMessage` de la clase `Throwable` (pág. 461) devuelve la cadena descriptiva almacenada en una excepción.
- El método `getStackTrace` (pág. 461) 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. 461) obtienen el nombre de la clase, el nombre de archivo, el número de línea y el nombre del método, respectivamente.

Sección 11.8 Excepciones encadenadas

- Las excepciones encadenadas (pág. 462) 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 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. 465) debe ser verdadera al momento de invocar el método.
- La poscondición de un método (pág. 465) 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. 465) ayudan a atrapar errores potenciales e identificar posibles errores lógicos.
- La instrucción `assert` (pág. 466) 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 Cláusula `try` con recursos: desasignación automática de recursos

- La instrucción `try` con recursos (pág. 467) simplifica la escritura del código en el que se obtiene un recurso, se utiliza en un bloque `try` y se libera el recurso en el correspondiente bloque `finally`. En su lugar, 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. 467); 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 ¿Por qué son las excepciones especialmente apropiadas para tratar con los errores producidos por los métodos de las clases en la API de Java?

11.3 ¿Qué es una "fuga de recursos"?

11.4 Si no se lanzan excepciones en un bloque `try`, ¿hacia dónde procede el control cuando el bloque `try` completa su ejecución?

11.5 Mencione una ventaja clave del uso de `catch(Exception` *nombreExcepción*`)`.

11.6 ¿Debe una aplicación convencional atrapar los objetos `Error`? Explique.

11.7 ¿Qué ocurre si ningún manejador `catch` coincide con el tipo de un objeto lanzado?

11.8 ¿Qué ocurre si varios bloques `catch` coinciden con el tipo del objeto lanzado?

11.9 ¿Por qué debería un programador especificar un tipo de superclase como el tipo en un bloque `catch`?

11.10 ¿Cuál es la razón clave de utilizar bloques `finally`?

11.11 ¿Qué ocurre cuando un bloque `catch` lanza una excepción `Exception`?

11.12 ¿Qué hace la instrucción `throw` *referenciaExcepción* en un bloque `catch`?

11.13 ¿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 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.

11.3 Una "fuga de recursos" ocurre cuando un programa en ejecución no libera apropiadamente un recurso cuando éste ya no es necesario.

11.4 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`.

11.5 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 entre manejar la excepción o posiblemente volver a lanzarla.

11.6 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.

11.7 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.

11.8 Se ejecuta el primer bloque `catch` que coincida después del bloque `try`.

11.9 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.

11.10 El bloque `finally` es el medio preferido para liberar recursos y evitar las fugas de éstos.

11.11 Primero, el control pasa al bloque `finally`, si existe uno. Después, la excepción se procesará mediante un bloque `catch` (si existe) asociado con un bloque `try` circundante (si existe).

11.12 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.

11.13 La referencia queda fuera de alcance. Si el objeto al que se hace referencia es inalcanzable, se marca para la recolección de basura.

Ejercicios

11.14 ***(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.

11.15 ***(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.

11.16 ***(Atrapar excepciones con las superclases)*** Use la herencia para crear una superclase de excepción llamada `ExcepcionA`, así como 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`.

11.17 ***(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`.

11.18 ***(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.

11.19 ***(Falla del constructor)*** Escriba 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.

11.20 ***(Relanzamiento de excepciones)*** 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.

11.21 ***(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.



Apéndices



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 unario de postdecremento	de derecha a izquierda

`++`
`--`
`+`
`-`
`!`
`~`
`(`*tipo*`)`
	unario de preincremento
	unario de predecremento
	unario de suma
	unario de resta
	unario de negación lógica
	unario de complemento a nivel de bits
	unario de conversión
		de derecha a izquierda

`*`
`/`
`%`
	multiplicación
	división
	residuo
		de izquierda a derecha

`+` `-`	suma o concatenación de cadenas resta	de izquierda a derecha

`<<`
`>>`
`>>>`
	desplazamiento a la izquierda
	desplazamiento a la derecha con signo
	desplazamiento a la derecha sin signo
		de izquierda a derecha

`<`
`<=`
`>`
`>=`
`instanceof`
	menor que
	menor o igual que
	mayor que
	mayor o igual que
	comparación de tipos
		de izquierda a derecha

`==` `!=`	es igual que no es igual que	de izquierda a derecha
`&`	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 los 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 los 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 H (en inglés en el sitio 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`, así como `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 *Rango positivo:* 1.40129846432481707e–45 a 3.4028234663852886E+38	(IEEE 754, punto flotante)
`double`	64	*Rango negativo:* –1.7976931348623157E+308 a –4.94065645841246544e–324 *Rango positivo:* 4.94065645841246544e–324 a 1.7976931348623157E+308	(IEEE 754, punto flotante)

Fig. D.1 | Tipos primitivos de Java.

Puede usar guiones bajos para mejorar la legibilidad de los valores literales numéricos. Por ejemplo, `1_000_000` es equivalente a `1000000`.

Para obtener más información acerca de IEEE 754, visite `grouper.ieee.org/groups/754/`. Para obtener más información sobre Unicode vea el apéndice H (en inglés en el sitio web).

E

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

E.1 Introducción

E.2 Los puntos de interrupción y los comandos `run`, `stop`, `cont` y `print`

E.3 Los comandos `print` y `set`

E.4 Cómo controlar la ejecución mediante los comandos `step`, `step up` y `next`

E.5 El comando `watch`

E.6 El comando `clear`

E.7 Conclusión

E.1 Introducción

En el capítulo 2 aprendió que hay dos tipos de errores (los errores de sintaxis y los errores lógicos), así como 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 incluye 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 usarse para depurar aplicaciones que reciben entradas 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 `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html`.

E.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 E.1), la cual crea y manipula un objeto de la clase `Cuenta` (figura 3.8). 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 `depositar`.

Por último, en las líneas 22 y 23 se muestra el nuevo `saldo`. [*Nota:* el directorio de ejemplos del apéndice E contiene una copia del archivo `Cuenta.java`, el cual es idéntico al de la figura 3.8].

```
1  // Fig. E.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("Jane Green", 50.00);
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.depositar(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    }
25 } // 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. E.1 | 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 E.1).

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 E. 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 E.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].

```
C:\ejemplos_codigo\depurador>javac -g PruebaCuenta.java Cuenta.java

C:\ejemplos_codigo\depurador>jdb
Initializing jdb ...
>
```

Fig. E.2 | Inicio del depurador de Java.

4. *Ejecutar una aplicación en el depurador.* Ejecute la aplicación PruebaCuenta a través del depurador, escribiendo **run** PruebaCuenta (figura E.3). Si no establece puntos de interrupción antes de ejecutar su aplicación en el depurador, ésta se ejecutará igual que si se utilizara el comando java.

```
C:\ejemplos_codigo\depurador>jdb
Initializing jdb ...
> run PruebaCuenta
run  PruebaCuenta
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: saldo inicial de cuenta: $50,00

agregando 25,00 al saldo de la cuenta

nuevo saldo de la cuenta: $75,00
The application exited
```

Fig. E.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 E.1. Establezca un punto de interrupción en la línea 12 del código fuente, escribiendo stop at PruebaCuenta:12 (figura E.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 E.4). Cuando se ejecuta la aplicación, ésta suspende la ejecución en cualquier línea que contenga 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 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.depositar`), 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.

```
C:\ejemplos\depurador>jdb
Initializing jdb ...
> stop at PruebaCuenta:12
Deferring breakpoint PruebaCuenta:12.
It will be set after the class is loaded.
> stop at PruebaCuenta:19
Deferring breakpoint PruebaCuenta:19.
It will be set after the class is loaded.
>
```

Fig. E.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 E.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 de esa línea (12). Esa línea de código será la siguiente instrucción en ejecutarse.

```
It will be set after the class is loaded.
>run PruebaCuenta
run PruebaCuenta
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint PruebaCuenta:19
Set deferred breakpoint PruebaCuenta:12

Breakpoint hit: "thread=main", PruebaCuenta.main(), line=12 bci=13
12             System.out.printf("saldo inicial de la cuenta: $%.2f%n",

main[1]
```

Fig. E.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 E.6). La salida normal de `PruebaCuenta` aparece entre los mensajes del depurador.

```
main[1] cont
> saldo inicial de la cuenta: $50.00

agregando 25.00 al saldo de la cuenta

Breakpoint hit: "thread=main", PruebaCuenta.main(), line=19 bci=60
19     cuenta.depositar(montoDeposito); // se suma al saldo de la cuenta

main[1]
```

Fig. E.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 E.7). El comando `print` nos permite husmear 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`, que es el valor asignado a `montoDeposito` en la línea 15 de la figura E.1.

```
main[1] print montoDeposito
 montoDeposito = 25.0
main[1]
```

Fig. E.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 E.8). El depurador se detendrá cuando la aplicación termine.

```
main[1] cont
> nuevo saldo de la cuenta: $75.00

The application exited
```

Fig. E.8 | Continuar la ejecución de la aplicación y salir del depurador.

E.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 E.2 para abrir la ventana **Símbolo del sistema**, cambiar al directorio que contiene los ejemplos del apéndice E (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 E.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 E.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.

```
C:\ejemplos\depurador>jdb
Initializing jdb ...
> stop at PruebaCuenta:19
Deferring breakpoint PruebaCuenta:19.
It will be set after the class is loaded.
> run PruebaCuenta
run  PruebaCuenta
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint PruebaCuenta:19
saldo inicial de la cuenta: $50.00

agregando 25.00 al saldo de la cuenta

Breakpoint hit: "thread=main", PruebaCuenta.main(), line=19 bci=60
19      cuenta.depositar(montoDeposito); // se suma al saldo de la cuenta

main[1]
```

Fig. E.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 E.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 E.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 E.10), ya que montoDeposito no contiene en ese momento el valor 23.0; montoDeposito sigue siendo 25.0.

```
main[1] print montoDeposito - 2.0
montoDeposito - 2.0 = 23.0
main[1] print montoDeposito == 23.0
montoDeposito == 23.0 = false
main[1]
```

Fig. E.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 de mucha ayuda 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 E.11).

```
main[1] set montoDeposito = 75.0
 montoDeposito = 75.0 = 75.0
main[1]
```

Fig. E.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 E.1) y pasa `montoDeposito` al método `depositar` de `Cuenta`. Después el método `main` muestra el nuevo saldo. El resultado es `$125.00` (figura E.12). Esto muestra que el paso anterior modificó el valor de `montoDeposito`, de su valor inicial (`25.0`) a `75.0`.

```
main[1] cont
> nuevo saldo de la cuenta: $125.00

The application exited

C:\ejemplos\depurador>
```

Fig. E.12 | La salida que muestra el nuevo saldo de la cuenta con base en el valor alterado de `montoDeposito`.

E.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 de esta forma por una porción de la aplicación para verificar que el código de un método se ejecute correctamente. En esta sección aprenderá a usar el depurador para esta tarea. Los comandos que veremos en esta sección nos permiten ejecutar línea por línea un método, 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 E, 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. 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 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 `depositar` de la clase `Cuenta` (figura 3.8), escribiendo `step` (figura E.13). 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 3.8).

```
main[1] step
>
Step completed: "thread=main", Cuenta.depositar(), line=24 bci=0
24      if (montoDeposito > 0.0) // si el montoDeposito es válido

main[1]
```

Fig. E.13 | Avanzar por pasos en el método `depositar`.

5. ***Usar el comando `step up`.*** Una vez que haya entrado al método `depositar`, 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 `depositar` 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 E.14). 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.

```
main[1] step up
>
Step completed: "thread=main", PruebaCuenta.main(), line=22 bci=65
22      System.out.printf("nuevo saldo de la cuenta: $%.2f%n",

main[1]
```

Fig. E.14 | Salirse de un método.

6. ***Usar el comando cont para continuar la ejecución.*** Escriba el comando `cont` (figura E.15) 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.

```
main[1] cont
> nuevo saldo de la cuenta: $75.00

The application exited

C:\ejemplos\depurador>
```

Fig. E.15 | Continuar la ejecución de la aplicación `PruebaCuenta`.

7. ***Reiniciar el depurador.*** Para reiniciar el depurador, escriba `jdb`.
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 E.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 PruebaCuenta: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.
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 E.16). En el *paso 4* vimos que el comando `step` entraría al método llamado. En este ejemplo, el comando `next` provoca la ejecución del método `depositar` de `Cuenta`, y después el depurador se detiene en la línea 22 de `PruebaCuenta`.

```
main[1] next
> Step completed: “thread=main”, PruebaCuenta.main(), line=22 bci=65
22      System.out.printf(“nuevo saldo de la cuenta: $%.2f%n”,

main[1]
```

Fig. E.16 | 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 E.17). 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.

```
main[1] exit

C:\ejemplos\depurador>
```

Fig. E.17 | Salir del depurador.

E.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 E.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 E.18). 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.

```
C:\ejemplos\depurador>jdb
Initializing jdb ...
> watch Cuenta.saldo
Deferring watch modification of Cuenta.saldo.
It will be set after the class is loaded.
>
```

Fig. E.18 | Establecer un punto de inspección en el campo `saldo` de `Cuenta`.

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 E.19). Cuando la aplicación empieza, 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 E.1). En la figura 3.8 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`.

4. ***Agregar dinero a la cuenta.*** Escriba `cont` para continuar con la ejecución de la aplicación. Ésta se ejecutará normalmente antes de llegar al código en la línea 19 de la figura E.1, que llama al método `depositar` 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 E.20). Aunque la línea 19 de la clase `PruebaCuenta` llama al método `depositar`, la línea 25 en el método `depositar` de `Cuenta` es la que modifica el valor de `saldo`.

```
> run PruebaCuenta
run PruebaCuenta
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred watch modification of Cuenta.saldo

Field (Cuenta.saldo) is 0.0, will be 50.0: “thread=main”, Acount.<init>(), line=18
bci=17
18      this.saldo = saldo; // lo asigna a la variable de instancia saldo

main[1]
```

Fig. E.19 | La aplicación `PruebaCuenta` se detiene al crear la `cuenta`, y se modificará su campo `saldo`.

```
main[1] cont
> saldo inicial de la cuenta: $50.00

agregando 25.00 al saldo de la cuenta
Field (Cuenta.saldo) is 50.0, will be 75.0: “thread=main”,
Cuenta.depositar(), line=25 bci=13

25      saldo = saldo + montoDeposito; // lo suma al saldo

main[1]
```

Fig. E.20 | Modificar el valor de `saldo` llamando al método `depositar` de `Cuenta`.

5. ***Continuar la ejecución.*** Escriba `cont`; la aplicación terminará de ejecutarse ya que no intentará realizar ningún cambio adicional al `saldo` (figura E.21).

```
main[1] cont
> nuevo saldo de la cuenta: $75.00

The application exited

C:\ejemplos\depurador>
```

Fig. E.21 | 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.

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 E.22). Escriba `cont`; la aplicación terminará su ejecución sin volver a entrar al modo de interrupción.

```
main[1] unwatch Cuenta.saldo
Removed: watch modification of Cuenta.saldo
main[1] cont
> saldo inicial de la cuenta: $50.00

agregando 25.00 al saldo de la cuenta

nuevo saldo de la cuenta: $75.00

The application exited

C:\ejemplos\depurador>
```

Fig. E.22 | Eliminar el punto de inspección en la variable `saldo`.

E.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 E. 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`.

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 E.23).

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 E.24).

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 de `for`). Imprima también el valor de la variable `monto` (figura E.25).

```
It will be set after the class is loaded.
> run Interes
run Interes
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint Interes:22
Set deferred breakpoint Interes:13

Breakpoint hit: "thread=main", Interes.main(), line=13 bci=9
13      System.out.printf("%s%20s%n", "Anio", "Monto en deposito");

main[1]
```

Fig. E.23 | Llegar al punto de interrupción en la línea 13 de la aplicación `Interes`.

```
main[1] cont
> Anio Monto en deposito
Breakpoint hit: "thread=main", Interes.main(), line=22 bci=55
22              System.out.printf("%4d%,20.2f%n", anio, monto);

main[1]
```

Fig. E.24 | Llegar al punto de interrupción en la línea 22 de la aplicación `Interes`.

```
main[1] print anio
 anio = 1
main[1] print monto
 monto = 1050.0
main[1]
```

Fig. E.25 | 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 `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 de nuevo los valores de `anio` y `monto`, para ver cómo han cambiado los valores desde la primera iteración de `for` (figura E.26).

```
main[1] cont
> 1            1,050.00
Breakpoint hit: "thread=main", Interes.main(), line=22 bci=55
22              System.out.printf("%4d%,20.2f%n", anio, monto);

main[1] print monto
 monto = 1102.5
main[1] print anio
 anio = 2
main[1]
```

Fig. E.26 | 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 E.27). Suponga que está satisfecho de que la instrucción `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 E.27). Ya se llegó a este punto de interrupción, por lo que la ejecución no se verá afectada.

```
main[1] clear
Breakpoints set:
       breakpoint Interes:13
       breakpoint Interes:22
main[1] clear Interes:22
Removed: breakpoint Interes:22
main[1] clear
Breakpoints set:
       breakpoint Interes:13
main[1]
```

Fig. E.27 | 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 E.28).

```
main[1] cont
>    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

The application exited

C:\ejemplos\depurador>
```

Fig. E.28 | La aplicación se ejecuta sin el punto de interrupción establecido en la línea 22.

E.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.



Índice analítico

Símbolos

^, OR exclusivo lógico booleano 176, **179**
  tabla de verdad, 179
, (coma) bandera de formato **162**
--, predecremento/postdecremento **131**
-, resta 51, 52
!, NOT lógico 176, **179**
  tabla de verdad 179
!=, no es igual a 54
?:, operador condicional ternario **110**, 134
., separador punto **76**
{, llave izquierda 38
}, llave derecha 38
@Override, anotación **369**
*
  comodín en un nombre de archivo 77
  multiplicación 51, 52
*=, operador de asignación de multiplicación 131
/, división 51, 52
/* */, comentario tradicional **36**
/** */, comentario de documentación de Java 36
//, comentario de fin de línea **36**
/=, operador de asignación de división 131
\, barra diagonal inversa, secuencia de escape 43
\", comilla doble, secuencia de escape 43
\n, nueva línea, secuencia de escape 43
\r, retorno de carro, secuencia de escape 43
\t, tabulador horizontal, secuencia de escape 43
&, AND lógico booleano 176, **178**
&&, AND condicional **176**, 177
  tabla de verdad 177
%, residuo 51, 52
%=, operador de asignación residuo 131
%b, especificador de formato **180**
%c, especificador de formato 66
%d, especificador de formato **49**
%f, especificador de formato 66, **88**
%s, especificador de formato **44**
– (signo negativo) bandera de formato **161**
+, suma 51, 52
++, preincremento/postincremento **131**
+=
  operador de asignación de suma **131**
  operador de asignación de concatenación de cadenas 612
<, menor que 55
<=, menor o igual que 55
<> (notación de diamante en genéricos) **691**
<>, notación de diamante para inferencia de tipos genéricos (Java SE 7) **691**
=, operador de asignación 48
-=, operador de asignación de resta 131
==
  para determinar si dos referencias apuntan al mismo objeto 387
  es igual a 54
-> (token de flecha en una lambda) **733**
>, mayor que 55
>=, mayor o igual que 55
|, OR inclusivo lógico booleano 176, **178**
||, OR condicional 176, **177**
  tabla de verdad 178

Numéricos

0
  bandera **253**
  bandera de formato 318

A

A/DOO (análisis y diseño orientado a objetos) 13
abreviación de expresiones de asignación 131
abreviaturas del inglés 9
abrir un archivo **646**
abs, método de Math 204
abstract, palabra clave **402**
Abstract Window Toolkit (AWT) **479**
  Event, paquete **212**
AbstractButton, clase **495**, 498
  addActionListener, método 498
  addItemListener, método **501**
  setRolloverIcon, método **498**
AbstractCollection, clase **722**
AbstractList, clase **722**
AbstractMap, clase **722**
AbstractQueue, clase **722**
AbstractSequentialList, clase **722**
AbstractSet, clase **722**
accept
  método de la interfaz
    BiConsumer (Java SE 8) 755
    funcional Consumer (Java SE 8) 751
    IntConsumer (Java SE 8) **738**
accesibilidad 480
acceso
  a nivel de paquete **344**
    métodos de 344
    miembros de una clase con 345
  concurrente a un objeto Collection mediante varios hilos 721
acción 106, 114
  a ejecutar **102**
acelerómetro 5
Aceptar, botón 92
ActionEvent, clase **489**, 490, 494, 541
  getActionCommand, método **490**, 498
ActionListener, interfaz **489**, 494
  actionPerformed, método 489, 493, 534, 541
actionPerformed, método de la interfaz ActionListener 489, 493, 534, 541
actividad en UML **104**
actualización automática 956
Ada
  lenguaje de programación 16
  Lovelace 16
add, método
  ArrayList<T> **290**
  BigInteger **784**
  ButtonGroup **504**
  de la clase BigDecimal **347**
  JFrame 390, **483**
  JFrame, clase **138**
  LinkedList<T> **695**
  List<T> **690**, 693
addActionListener, método
  de la clase AbstractButton 498
  de la clase JTextField **489**
addAll, método
  Collections 696, **706**
  List **693**
addFirst método de LinkedList **696**
addItemListener, método de la clase AbstractButton **501**
addKeyListener, método de la clase Component **525**
addLast método de LinkedList **695**
addListSelectionListener, método de la clase JList **510**
addMouseListener, método de la clase Component **518**
addMouseMotionListener, método de la clase Component **518**
addPoint, método de la clase Polygon 579, **581**
además 15
adivinar el número, juego 241, 551
administrador de esquemas 483, 518, **528**, 537
  BorderLayout **518**
  FlowLayout 483
  GridLayout **536**
Agile
  Alliance 28
  Manifesto 28
agregar serialización de objetos a la aplicación de dibujo MiFigura, ejercicio 683
Ajax (JavaScript asíncrono y XML) **27**
ajuste de tamaño dinámico **245**
alcance 157
  de una declaración **222**
  de una variable **157**
  léxico 738
alfa, software 29
alfabetización 600
algoritmo **102**, 115, 122, 785
  de búsqueda
    binaria 704, **816**, 820, 837
    lineal **812,** 814, 820, 837
  de Euclides 240
  de ordenamiento
    de burbuja 836
    de cubeta 836
    por combinación **827**, 832
    por inserción **824**
    por selección **821**, 824
    quicksort 837
    rápido recursivo (*quicksort*) 837
  en Java Collections Framework 696
  para barajar imparcial 260
Alianza para los Dispositivos Móviles Abiertos 14

alineación a la derecha 530
almacenamiento secundario 4
alternativas para el plan fiscal, ejercicio 199
altura **569**
  de un rectángulo en píxeles 560
ALU (unidad aritmética y lógica) **6**
análisis y diseño orientado a objetos (A/DOO) **13**
ancho de banda **26**
anchura 571
  de campo **161**
  de un rectángulo en pixeles 560
AND
  condicional, **&&** **176**, 178
    tabla de verdad 177
  lógico booleano, **&** 176, **178**
`and`, método de la interfaz `Predicate` (Java SE 8) **745**
Android 14
  Google Play 15
  sistema operativo 13, **14**
  teléfono inteligente (smartphone) 14
Android for Programmers: An App-Driven Approach 15
ángulo(s)
  inicial **575**
  negativos de un arco 576
  positivos y negativos de un arco 576
anotación, `@Override` **369**
ansiosas **734**
apariencia visual 476, **479**, 480, 528
  adaptable, paquete **480**
  de una aplicacion **479**
  Nimbus **476**
API (interfaz de programación de aplicaciones) **46**, **201**
  de fecha/hora, 287, **321**
    paquete **213**
  de Java 201
    descarga de documentación 50
    documentación 49, 213
    generalidades 213
  documentación 213
  obsoleta **46**
apilar instrucciones de control 187
aplicación **35**
  argumentos de línea de comandos **205**
  de dibujo interactiva, ejercicio 552
  móvil 3
  robusta 442
aplicar formato
  mostrar datos con formato 43
`append`, método de la clase `StringBuilder` **615**
`apply`, método de la interfaz funcional `Function` (Java SE 8) **747**
`applyAsDouble`, metodo de la interfaz `ToDoubleFunction` (Java SE 8) **757**
`applyAsInt`, metodo de la interfaz `IntBinaryOperator` (Java SE 8) **740**
`applyAsInt`, metodo de la interfaz `IntUnaryOperator` (Java SE 8) **742**
árbol 711
`Arc2D`
  clase 556
    `CHORD`, constante **585**
    `OPEN`, constante **585**
    `PIE`, constante **585**
  `.Double`, clase **581**, 593
archivo(s) **8**, **645**
  binario **646**
  copiar 647
  crear 647
  de acceso secuencial 645, 651
  de cuentas por cobrar 682
  de procesamiento por lotes **653**
  de sólo lectura 667
  de transacciones 681
  leer 647
  maestro 680
  manipular 647
  obtener información acerca de 647
arco **575**
  anchura y altura rectángulos redondeados 574
  ángulo **575**
  en forma de pastel 585
área
  de dibujo dedicada **522**
  de dibujo personalizada 522
  de un círculo 239
`args`, parámetro 283
argumento
  de línea de comandos **205**, **283**
  para un método **39**, 76
`ArithmeticException`, clase 347, **444**, 450
ARPANET 25
arrastrar
  el cuadro desplegable 507
  el ratón para resaltar 541
`arraycopy`, método de la clase `System` **285**, 286
`ArrayIndexOutOfBoundsException`, clase 254, **257**, 581
`ArrayList<T>`, clase genérica **288**, **688**, 704
  `add`, método **290**
  `clear`, método 288
  `contains`, método 288, **290**
  `get`, método **290**
  `indexOf`, método 288
  `isEmpty`, método 332
  `remove`, método 288, **290**
  `size`, método **290**
  `trimToSize`, método 288
`Arrays`
  clase **285**
    `asList`, método **694**, 695
    `binarySearch`, método **285**
    `equals`, método **285**
    `fill`, método **285**
    `parallelSort`, método (Java SE 8) 745
    `sort`, método **285**, 745, **817**
    `stream`, método (Java SE 8) **743**, 744
    `toString`, método 631, 814
  método `parallelSort` 287
arreglo(s) **244**, 645
  comprobación de límites **254**
  con tamaño ajustable implementación de un objeto `List` 688
  de arreglos unidimensionales 272
  de *m* por *n* **272**
  de respaldo **694**
  ignorar el elemento cero 256
  `length`, variable de instancia **246**
  multidimensional 272, 273
  pasar el elemento de un arreglo a un método 264
  pasar un arreglo a un método 264
ascendente **569**
ASCII (código estándar estadounidense para el intercambio de información), conjunto de caracteres 7, 170, 313
aserción **465**
asignación
  de uno a uno **714**
  de varios a uno **714**
asignar
  elementos de un flujo (Java SE 8) 742
    referencias de superclase y subclase a variables de la superclase y la subclase 400
  un valor a una variable 48
`asList`, método de `Arrays` **694**, 695
asociar
  derecha a izquierda 125, 134
  izquierda a derecha 134
asociatividad de los operadores **52**, 58, 134
  derecha a izquierda 52
  izquierda a derecha 58
`assert`, instrucción **466**
`AssertionError`, clase **466**
atrapar
  bloque **448**, 450, 451, 455, 458, 459
  cláusula **448**
  excepciones
    con las superclases, ejercicio 472
    usando alcances exteriores, ejercicio 472
    usando la clase `Exception`, ejercicio 472
  la excepción de una superclase 454
  o declarar, requerimiento **453**
  palabra clave **448**
  una excepción 446
atributo
  en UML 12, 78
  de un objeto 12
  de una clase 10
autoboxing 622, **687**
`AutoCloseable`, interfaz 338, **467**
  `close`, método 467
autodocumentación 48
auto-unboxing **687**
Av Pág, tecla 525
`average`
  método de la interfaz `DoubleStream` (Java SE 8) 757
  método de la interfaz `IntStream` (Java SE 8) **739**
AWT (Abstract Window Toolkit) **479**
  componentes 480
`AWTEvent`, clase 491

B

Babbage, Charles 16
backtracking **802**
barajar 257
  algoritmo 700
  y repartir cartas 307, 308
    con el método `shuffle` de `Collections` 700
barra
  de asteriscos 251, 252
  de desplazamiento 510, 542
    de un objeto `JComboBox` **507**
  de menús **475**
  de título **475**, 481
    de una ventana 478
  diagonal inversa (\) **43**
barrido **292**, **575**
base
  de datos **8**
  de un número 621
BASIC (Código de instrucciones simbólicas de uso general para principiantes) 16
`BasicStroke`, clase **556**, 584, 585
  `CAP_ROUND`, constante **585**
  `JOIN_ROUND`, constante **586**
beta
  permanente 29
  software 29
biblioteca de clases **362**
BiConsumer, interfaz funcional (Java SE 8) **755**, 762
  `accept`, método 755
Big Data **8**
Big O, notación **814**, 820, 824, 827, 833
`BigDecimal`, clase 126, 163, **346**, 781
  `add`, método **347**
  `ArithmeticException`, clase 347
  `multiply`, método **347**
  `ONE`, constante **347**
  `pow`, método **347**
  `setScale`, método 348

TEN, constante **347**
valueOf, método **347**
ZERO, constante **347**
BigInteger, clase **781**
add, método **784**
compareTo, método **782**
multiply, método **782**
ONE, constante **782**, 784
subtract, método **782**, 784
ZERO, constante **784**
binario 241
BinaryOperator, interfaz funcional (Java SE 8) 733
binarySearch, método
de Arrays **285**, 287
de Collections 696, **704**, 706
bit (dígito binario) **6**
Bjarne Stroustrup 16
Bloc de notas 18
bloque(s) **110**, 124
catch que coincida 448
de construcción 102
anidado 186
apariencia, 184
apilados 186
de instrucciones en una lambda 733
Bohm, C. 103
BOLD, constante de la clase Font **567**, 567
boolean 171
expresión **110**
promociones 211
Boolean
class **687**
boolean, tipo primitivo **110**
BorderLayout, clase **390**, **518**, 528, 529, 532, 541
CENTER, constante 390, **518**, 532, 534
EAST, constante 390, **518**, 532
NORTH, constante 390, **518**, 532
SOUTH, constante 390, **518**, 532
WEST, constante 390, **518**, 532
botón(es) **475**, **495**
central del ratón 521
de comando **495**
de estado **498**
de opción **495**, **501**
interruptores **495**
Box, clase **540**
createHorizontalBox, método **541**
boxed, método de la interfaz IntStream (Java SE 8) **762**
boxing 288
conversión **687**
BoxLayout, clase **541**
Braille, lector de pantalla 480
break, instrucción **168**, 174, 198
brillo **565**
búfer **674**
BufferedImage, clase **585**
createGraphics, método **585**
TYPE_INT_RGB, constante **585**
BufferedInputStream, clase **675**
BufferedOutputStream, clase **674**
flush, método **674**
BufferedReader, clase **675**
BufferedWriter, clase **675**
buscar 311
datos **811**
el valor mínimo en un arreglo, ejercicio 807
búsqueda secuencial de un elemento en un arreglo 813
ButtonGroup, clase **501**
add, método **504**
byte **5**, 7
Byte class **687**
byte, tipo primitivo 165
promociones 211
ByteArrayInputStream, clase 675
ByteArrayOutputStream, clase 675

C

C, lenguaje de programación 16
C++ 16
cadena(s) **39**
de caracteres 39
de formato **44**
delimitadora 623
literal **39**
vacía **490**, **599**
calculadora
de ahorro por viajes compartidos en automóvil, ejercicio 68
de impacto ambiental del carbono 33
de la frecuencia cardiaca esperada, ejercicio 99
del crecimiento de la población mundial, ejercicio 68
del índice de masa corporal, ejercicio 67
cálculo(s) 6, 58, 104
aritmético 51
del factorial con un método recursivo 781
matemáticos 15
monetarios 163, 346
precisos de punto flotante 346
cámara 15
cambiar de directorio 40
campo **7**
de una clase 8, **204**, 223
"ocultos" 223
valor inicial predeterminado **76**
canalización **673**
de flujo 738
CANCEL_OPTION, constante de JFileChooser **671**
Cancelar, botón 92
candidato
para la recolección de basura 342
para liberación 29
CAP_ROUND, constante de la clase BasicStroke **585**
capacidad de un objeto StringBuilder 611
capacity, método de la clase StringBuilder **612**
captura de lambdas **738**
carácter(es) 7
aleatorios, ejercicio 592
conjunto 66
constante **170**
de escape **43**
de espacio en blanco 610, 623, 624
al final 610
de palabra **624**
de tabulación, \t 43
especial 47, **597**
literal **597**
separador **650**
características completas 29
carga **19**
cargador de clases **18**, 484
case, palabra clave **168**
casilla de verificación **495**, 501
casino 218
caso
base **778**, 784, 785, 790
default en un switch **168**, 170, 217
Catch, bloque **256**
catch, manejador
multi-catch **449**
cd para cambiar directorios 40
ceil, método de Math 204
Celsius 551
equivalente de una temperatura en Fahrenheit 240
CENTER constante
BorderLayout **518**, 532, 534
FlowLayout **532**
centrado 530
centros de recursos de Deitel 30
cerrar una ventana 481, 485
char
arreglo 599
promociones 211
tipo primitivo **47**, 165
Character, clase 597, 620, **687**
charValue, método **622**
digit, método **621**
forDigit, método **621**
isDefined, método **620**
isDigit, método **620**
isJavaIdentifierPart, método **620**
isJavaIdentifierStart, método **620**
isLetter, método **620**
isLetterOrDigit, método **620**
isLowerCase, método **620**
isUpperCase, método **620**
static, métodos de conversión 621
toLowerCase, método **621**
toUpperCase, método **620**
CharArrayReader, clase **676**
CharArrayWriter, clase **676**
charAt, método
de la clase
String 599
StringBuilder **614**
CharSequence, interfaz **631**
charValue, método de la clase Character **622**
CHORD, constante de la clase Arc2D **585**
ciclo 113, 117
anidado dentro de un ciclo 128
condición de continuación **105**
contador 153
cuerpo 163
de ejecución de instrucciones 311
infinito **114**, 124, 157
instrucción **105**
cifrado 151
cima **120**, 710
círculo(s)
concéntricos mediante el uso
de la clase Ellipse2D.Double, ejercicio 592
del método drawArc, ejercicio 592
pequeños en UML **104**
relleno
en UML **104**
rodeado por un círculo sin relleno en UML **104**
circunferencia 66, 592
clase(s) **11**
abstracta 397, **401**, 402, 403, 422
adaptadora(s) **518**
para manejadores de eventos 522
anidada 316, **488**
método *establecer* de 324
relación entre una clase interna y su clase de nivel superior 501
archivo de **40**
base **361**
class, palabra clave 72
concreta **402**
constructor **75**, 81
predeterminado de **83**
controladora 74
de envoltura de tipos **618**
de eventos 491
de nivel superior **488**
declaración de **37**
derivada **361**
genérica **288**
interna **488**, 501, 523, 924
anónima 316, 433, 489, **507**, 523

objeto de 501
relación entre una clase interna y su clase de nivel superior 501
método *obtener* de 324
no puede extender a una clase `final`, una 420
nombre de **37**
numéricas 687
ocultamiento de datos **80**
predefinida de caracteres **624**
variable de instancia de **12**
Clases
`AbstractButton` **495**, 498
`AbstractCollection` **722**
`AbstractList` **722**
`AbstractMap` **722**
`AbstractQueue` **722**
`AbstractSequentialList` **722**
`AbstractSet` **722**
`ActionEvent` **489**, 490, 494, 541
`Arc2D` 556
`Arc2D.Double` **581**
`ArithmeticException` 347, **444**
`ArrayIndexOutOfBoundsException` 254, 257
`ArrayList<T>` **288**, 290, 688, 689, 704
`Arrays` **285**
`AssertionError` **466**
`AWTEvent` 491
`BasicStroke` **556**, 585
`BigDecimal` 126, 163, **346**, 781
`BigInteger` **781**
`Boolean` **687**
`BorderLayout` **518**, 528, 529, 532, 541
`Box` **540**
`BoxLayout` **541**
`BufferedImage` **585**
`BufferedInputStream` **675**
`BufferedOutputStream` **674**
`BufferedReader` **675**
`BufferedWriter` **675**
`ButtonGroup` **501**
`Byte` **687**
`ByteArrayInputStream` 675
`ByteArrayOutputStream` 675
`Character` 597, 615, 620, **687**
`CharArrayReader` **676**
`CharArrayWriter` **676**
`Class` 388, **418**, 484
`ClassCastException` 883
`Collections` **688**
`Collector` (Java SE 8) 745
`Color` **227**, **556**
`Component` **480**, 513, 558, 559
`ComponentAdapter` 519
`ComponentListener` 529
`Container` **480**, 511, 529, 537, 538
`ContainerAdapter` 519
`DataInputStream` **674**
`DataOutputStream` **674**
`Double` **687**
`Ellipse2D` 556
`Ellipse2D.Double` **581**
`Ellipse2D.Float` **581**
`EmptyStackException` **710**
`EnumSet` **337**
`Error` **451**
`EventListenerList` **493**
`Exception` **451**
`FileInputStream` **674**
`FileOutputStream` **674**
`FileReader` 676
`Files` **647**, 759
`Float` **687**
`FlowLayout` 483, 529
`FocusAdapter` 519
`Font` **500**, **556**, 567
`FontMetrics` **556**, **569**
`Formatter` **647**
`GeneralPath` 556, **586**
`GradientPaint` **556**, 584
`Graphics` **523**, **556**, 581
`Graphics2D` **556**, **581**, 585
`GridLayout` 529, **536**
`HashMap` **714**
`HashSet` **711**
`Hashtable` **714**
`ImageIcon` **484**
`IndexOutOfRangeException` 257
`InputEvent` **514**, 521, 525
`InputMismatchException` **445**
`InputStream` **673**
`InputStreamReader` **676**
`Integer` 478, **687**
`ItemEvent` **501**, 504
`JButton` 479, **495**, 498, 534
`JCheckBox` 479, **498**
`JColorChooser` **563**
`JComboBox` 479, **504**
`JComponent` **480**, 481, 483, 493, 505, 508, 522, 538, 556, 558
`JFileChooser` **670**
`JLabel` 479, **481**
`JList` 479, 508
`JOptionPane` **90**, **476**
`JPanel` 479, 522, 529, 538
`JPasswordField` **485**, 490
`JRadioButton` **498**, 501, 504
`JScrollPane` **510**, 513, 541, 542
`JTextArea` 528, **539**, 541
`JTextComponent` **485**, 488, 539, 541
`JTextField` 479, **485**, 489, 493, 539
`JToggleButton` **498**
`KeyAdapter` 519
`KeyEvent` **494**, 525
`Line2D` 556, **585**
`Line2D.Double` **581**
`LinearGradientPaint` 584
`LineNumberReader` **676**
`LinkedList` **688**
`ListSelectionEvent` **508**
`ListSelectionModel` **510**
`Long` **687**
`Matcher` 597, **631**
`Math` 203, 204
`MouseAdapter` 518, 519
`MouseEvent` **494**, **514**
`MouseMotionAdapter` 519, 522
`MouseWheelEvent` **515**
`NumberFormat` **346**, 1026, **1120**, 1128
`Object` 338
`ObjectInputStream` **662**
`ObjectOutputStream` **662**
`Optional`, clase (Java SE 8) 752
`OptionalDouble` **739**, 757
`OutputStream` 673
`OutputStreamWriter` **676**
`Paths` 647
`Pattern` 597, **631**
`PipedInputStream` 673
`PipedOutputStream` 673
`PipedReader` **676**
`PipedWriter` **676**
`Point` **523**
`Polygon` **556**, **578**
`PrintStream` **674**
`PrintWriter` 676
`PriorityQueue` **710**
`Properties` **718**
`RadialGradientPaint` 584
`Random` 300
`Reader` **675**
`Rectangle2D` 556
`Rectangle2D.Double` **581**
`RoundRectangle2D` 556
`RoundRectangle2D.Double` **581**, 585
`RuntimeException` **452**
`Scanner` **47**
`SecureRandom` **213**
`Short` **687**
`Stack` **708**
`StackTraceElement` **461**
`String` **92**, 597
`StringBuffer` **612**
`StringBuilder` 597, **611**
`StringIndexOutOfBoundsException` **607**
`StringReader` **676**
`StringWriter` **676**
`SystemColor` **584**
`TexturePaint` **556**, 584, 585
`Throwable` **451**, 461
`TreeMap` **714**
`TreeSet` **711**
`UnsupportedOperationException` **694**
`Vector` **688**
`WindowAdapter` 519
`Writer` **675**
`class` 7
palabra clave **37**, 72
`Class`, clase 388, **418**, 484
`getName`, método 388, **418**
`getResource`, método **484**
`ClassCastException`, clase 686
`CLASSPATH`
problema 22
variable de entorno 44
clave
de búsqueda **811**
de función 525
de ordenamiento **811**
`clear`, método
de `ArrayList<T>` 288
de `List<T>` **694**
de `PriorityQueue` **710**
`clearRect`, método de la clase `Graphics` 572
clic(s)
con el botón
central del ratón 521
izquierdo del ratón 521
con los botones izquierdo, central y derecho del ratón 519
del botón del ratón 521
del ratón 519
cliente de una clase **79**
clonar objetos
copia
en profundidad **388**
superficial **388**
`clone`, método de `Object` 388
`close`, método
de `Formatter` **654**
de la interfaz `AutoCloseable` 467
de `ObjectOutputStream` **667**
`closePath`, método de la clase `GeneralPath` **588**
COBOL (Lenguaje común orientado a objetos) 15
cocinar con ingredientes más saludables 641
código **12**
autodocumentado 48
cliente 398
de bytes **19**, 40
de liberación de recursos 455
de operación 308
de tecla virtual **528**
dependiente de la implementación 321
fuente **18**
abierto **14**
Morse 640
cofia **584**
cola 710

colección(es) **287**, **685**
  no modificable 688
  sincronizada 688
colisión(es) en una tabla hash **715**
`collect`, método de la interfaz `Stream` (Java SE 8) **745**, 745, 755, 756, 762
`Collection`, interfaz **686**, 687, 691, 696
  `contains`, método **691**
  `iterator`, método **691**
`Collections`
  clase **688**
    `addAll`, método 696
    `binarySearch`, método 696, **704**, 706
    `copy`, método 696, **703**
    `disjoint`, método 696, **706**
    `fill`, método 696, **702**
    `frequency`, método 696, **706**
    `max`, método 696, **703**
    métodos envolventes **688**
    `min`, método 696, **703**
    `reverse`, método 696, **702**
    `reverseOrder`, método **698**
    `shuffle`, método 696, **700**, 702
    `sort`, método **697**
  métodos `reverse`, `fill`, `copy`, `max` y `min` 703
`Collector`
  de flujo descendente (Java SE 8) **756**
  interfaz (Java SE 8) **745**
`Collectors`, clase (Java SE 8) **745**
  `groupingBy`, método **755**, 756, 760, 762
    `toList`, método **745**
color(es) 227, 556
  aleatorios, ejercicio 593
  de fondo 563, 565
`Color`
  clase **227**, **556**
    `getBlue`, método **560**, 562
    `getColor`, método **560**
    `getGreen`, método **560**, 562
    `getRed`, método **560**, 562
    `setColor`, método **560**
  constante 559, 562
`Color.BLACK` 227
`Color.BLUE` 227
`Color.CYAN` 227
`Color.DARK_GRAY` 227
`Color.GRAY` 227
`Color.GREEN` 227
`Color.LIGHT_GRAY` 227
`Color.MAGENTA` 227
`Color.ORANGE` 227
`Color.PINK` 227
`Color.RED` 227
`Color.WHITE` 227
`Color.YELLOW` 227
columna(s) 272
  de un arreglo bidimensional 272
coma (`,`) 160
  bandera de formato **162**
  en una lista de argumentos 44
comentario(s)
  de fin de línea (una sola línea), `//` **36**, 39
  de Javadoc **36**
  de una sola línea 39
  tradicional **36**
comilla(s)
  dobles, " 39, 43
  sencilla, carácter 597
comisión 147, 300
comisiones de ventas 300
`Comparable<T>`, interfaz 431, 603, 697
  `compareTo`, método 697
comparación
  de archivos
    con ejercicio de serialización de objetos 682
    con ejercicio de transacciones múltiples 682
    ejercicio 680
    programa 680
  de objetos `String` 600
  lexicográfica **602**, 603
`Comparator`
  en `sort` 697
  interfaz **697**, 698, 748, **748**
    `compare`, método **699**
    `thenComparing`, método (Java SE 8) **753**
  objeto 697, 703, 712, 714
`compare`, método de la interfaz `Comparator` **699**
`compareTo`, método
  de `Comparable` 697
  de la clase
    `BigInteger` **782**
    `String` 601, 603
compartimiento en un diagrama de clases de UML 77
compilación justo a tiempo **20**
compilador **10**
  con optimización 162
  justo a tiempo (JIT) **20**
compilar 40
  un programa **18**
  una aplicación con varias clases 77
`compile`, método de la clase `Pattern` **631**
completar las operaciones de E/S de disco 451
`Complex` 356
`Component`, clase **480**, 513, 558, 559, 565
  `addKeyListener`, método **525**
  `addMouseListener`, método **518**
  `addMouseMotionListener`, método **518**
  `repaint`, método **524**
  `setBackground`, método 565
  `setBounds`, método 529
  `setFont`, método **500**
  `setLocation`, método 529
  `setSize`, método 529
  `setVisible`, método **534**
`ComponentAdapter`, clase 519
componente(s) 10, 513
  de escucha registrado 494
  de GUI ligero **480**
  de software reutilizables 10, 211, 362
  de un arreglo **245**
  opacos de la GUI de Swing **522**
  pesados **480**
  reutilizable estándar **362**
`ComponentListener` interfaz 519, 529
comportamiento
  de una clase 10
composición **332**, 362, 364
  de expresiones lambda 742
comprobación
  de límites **254**
  de rangos **119**
computación en la nube **29**
computadoras en la educación 241
computarización de los registros médicos, ejercicio 99
`concat`, método de la clase `String` **608**
concatenación **208**
  de cadenas **208**, 341
concatenar cadenas 341
concordancia 758
condición **54**, 164
  de continuación de ciclo **153**, 154, 155, 156, 157, 159, 163, 164, 175
  de guardia en UML 106
  dependiente 178
  simple 176
configurar el manejo de eventos 488
confundir el operador de igualdad == con el operador de asignación = 57
conjunto
  de caracteres **7**
  de enteros, ejercicio 357
  de llamadas recursivas para `fibonacci( 3 )` 785
  exterior de llaves 256
  más interno de llaves 255
  vacío 357
consola de juegos 15
Consorcio World Wide Web (W3C) **26**
constante 343
  clave 528, **528**
  con nombre **250**
  de enumeración **221**
  de punto flotante 160
  `Math.PI` 66
constructor(es) **75**, 81
  con varios parámetros 84
  lista de parámetros 86
  llamar a otro constructor de la misma clase usando `this` 327
  predeterminado **83**, 330, 368
  sin argumentos **327**, 328
  sin un tipo de valor de retorno 83
  sobrecargado **324**
construya su propia computadora 308
`Consumer`, interfaz funcional (Java SE 8) 733, 738, 751
  `accept`, método 751
consumir
  memoria 788
  un evento **489**
contador **115**, 121, 127
`Container`, clase **480**, 511, 529, 537, 538
  `setLayout`, método **483**, 530, 534, 537
  `validate`, método **537**
`ContainerAdapter`, clase 519
`ContainerListener`, interfaz 519
`contains`, método
  de `Collection` **691**
  de la clase `ArrayList<T>` 288, **290**
`containsKey`, método de `Map` **717**
contenido dinámico **17**
conteo
  con base cero 156, **248**
  de clics 519
contexto de gráficos **558**
`continue`, instrucción **174**, 175, 198
contraseña 485
control(es) 474
  del programa **102**
Control, tecla 168
controlado por evento **485**
convención de nomenclatura
  métodos que devuelven `boolean` 171
converger en un caso base 778
conversión
  de tipos **125**
    conversión descendente **399**
    operador 66, **125**, 211
  descendente **399**, 418, 686
  explícita **125**
  implícita **125**
  unboxing **687**
convertir
  entre sistemas numéricos 621
coordenada(s)
  (0, 0) 135, 556
  horizontal **135**, **556**
  vertical **135**, **556**
  x **135**, **556**, 580
    superior izquierda 560
  y **135**, **556**, 580
    superior izquierda 560
copia
  en profundidad **388**
  superficial **388**

copiado
  de archivos 647
  de objetos
    copia en profundidad **388**
    copia superficial **388**
copo de nieve de Koch, fractal **792**
`copy`, método de `Collections` 696, **703**
corchetes, `[]` **245**
correcto en un sentido matemático 182
corrector ortográfico, proyecto 641
correo electrónico (e-mail) 25
`cos`, método de `Math` 204
coseno 204
  trigonométrico 204
`count`, método de la interfaz IntStream (Java SE 8) **739**
CPU (unidad central de procesamiento) **6**
CraigsList 26
craps (juego de casino) 218, 241, 301
crear
  archivos 647
  e inicializar un arreglo 247
  un objeto de una clase 75
`createGraphics`, método de la clase `BufferedImage` **585**
`createHorizontalBox`, método de la clase `Box` **541**
crecimiento de la población mundial, ejercicio 151
criba de Eratóstenes 306, 762
Ctrl, tecla 511, 528
<*Ctrl*>-*d* 168
<*Ctrl*>-*z* 168
cuadrícula 536
  mediante el método
    `drawLine`, ejercicio 592
    `drawRect`, ejercicio 592
  mediante la clase
    `Line2D.Double`, ejercicio 592
    `Rectangle2D.Double`, ejercicio 592
cuadro(s)
  combinado **475**, 504
  de desplazamiento **507**
  de diálogo **90**, 476
  de información sobre herramientas **480**, 483, 485
cuantificador(es)
  avaro **629**
  perezoso **629**
  reacio **629**
  utilizados en expresiones regulares **628**, 629
cuenta de ahorros 160
`CuentaDeAhorros`, clase, ejercicio 356
cuerpo
  de un ciclo 114
  de un método **38**
  de una declaración de clase **38**
  de una instrucción `if` 54
`currentTimeMillis`, método de la clase `System` 809
cursor 39, 42
  de salida 39, 42
  en pantalla 43
curva 586
  compleja 586
  de Koch, fractal **791**

D

`-d`, opción del compilador 884
`DataInput`, interfaz 674
`DataInputStream`, clase **674**
`DataOutput`, interfaz 674
  `writeBoolean`, método 674
  `writeByte`, método 674
  `writeBytes`, método 674
  `writeChar`, método 674
  `writeChars`, método 674
  `writeDouble`, método 674
  `writeFloat`, método 674
  `writeInt`, método 674
  `writeLong`, método 674
  `writeShort`, método 674
  `writeUTF`, método 674
`DataOutputStream`, clase **674**
`Date`, clase
  ejercicio 357
datos 4
  persistentes **645**
De Morgan, leyes 197
decisión **54**, 106
  lógica 4
  símbolo en UML **106**
declaración(es)
  clase **37**
  de (un) método **38**, 207
  de varias clases en un archivo de código fuente 322
  `import` **46**, 48
decremento de una variable de control **158**
`default` método en una interfaz (Java SE 8) **432**, 731, 732, 763
definir un área de dibujo personalizada 522
`delete`, método de la clase `StringBuilder` **617**
`deleteCharAt`, método de la clase `StringBuilder` **617**
delimitador de tokens **623**
Departamento de Defensa (DOD) 16
dependiente de la máquina 9
`deprecation`, bandera 46
desarrollo ágil de software **28**
desbordamiento 451
  aritmético **118**, 451
descendente **569**
descifrar 151
descomponer en tokens 623
descubrir un componente 558
desencadenar un evento 479
despachar un evento **494**
desplazamiento 506, 510
  automático 510
  (números aleatorios) **214**
    valor de **214**, 217, 218
  vertical 541
detalles específicos 398
determinar puntos C y D para el nivel 1 del "fractal Lo" 794
diagrama de actividad(es) **104**, 107, 158, 184
  de una estructura de secuencia 104
  `do...while`, instrucción 164
  en UML 114
  `for`, instrucción 158
  `if`, instrucción 106
  `if...else`, instrucción 107
  instrucción de secuencia 104
  más simple 182, 184
  `switch`, instrucción 170
  `while`, instrucción 114
`Dialog`, tipo de letra 567
`DialogInput`, tipo de letra 567
diálogo **476**
  de entrada **91**, **476**
  de mensaje **90**, **476**, 478
    tipos 478
  modal **478**, 565
  selector de colores 565
diámetro 66, 592
dibujar figuras 556
  color 560
  en la pantalla 558
`digit`, método de la clase `Character` **621**
dígito(s) 47, 622, 624
  binario (bit) **6**
  decimal **6**
  invertidos 240
`DIRECTORIES_ONLY`, constante de `JFileChooser` **670**
directorio(s) 647
  crear 647
  manipular 647
  obtener información acerca de 647
  raíz **647**
`DirectoryStream`, interfaz **647**, 774
  `entries`, método 774
disco 4, 21, 645
  duro 4, 6
`disjoint`, método de `Collections` 696, **706**
dispositivo(s)
  de almacenamiento secundario **645**
  de entrada **5**
  de salida **5**
  electrónico (inteligente) para el consumidor 17
distancia entre valores (números aleatorios) 218
`distinct`, método de la interfaz Stream (Java SE 8) **754**
distribuidor de software independiente (ISV) 386
divide y vencerás, método **201**, 202, 778
dividir el arreglo en el ordenamiento por combinación 827
división 6, 51, 52
  entre cero 21, 121, 444
`do...while`, instrucción de repetición 105, **163**, 164, 186
doble igual, `==` 57
documentar un programa **36**
dos valores más grandes 148
`Double`, clase **687**
`(double)`, conversión **125**
`double`, tipo primitivo **47**, **84**, 122
  promociones 211
`doubles`, método de la clase `SecureRandom` (Java SE 8) **762**
`DoubleStream`, interfaz (Java SE 8) **736**, 756
  `average`, método 757
  `reduce`, método 757
  `sum`, método 757
`draw`, método de la clase `Graphics2D` **584**
`draw3DRect`, método de la clase `Graphics` 572, **575**
`drawArc`, método de la clase `Graphics` **292**, **575**, 592
`drawLine`, método de la clase `Graphics` **137**, 572
`drawOval`, método de la clase `Graphics` **187**, **188**, 572, **575**
`drawPolygon`, método de la clase `Graphics` 578, **580**
`drawPolyline`, método de la clase `Graphics` 578, **580**
`drawRect`, método de la clase `Graphics` **187**, 572, 585, 592
`drawRoundRect`, método de la clase `Graphics` **573**
`drawString`, método de la clase `Graphics` **562**

E

E/S
  con búfer **674**
  mejora del rendimiento de 674
`EAST`, constante
  de la clase `BorderLayout` **518**, 532
Eclipse 18
  video de demostración 35
eco, carácter de la clase `JPasswordField` **485**
Ecofont 553
editar un programa 18

editor 18
efecto secundario **178**
eficiencia de
  búsqueda
    binaria 820
    de burbuja 836
    lineal 816
    por combinación 832
    por inserción 827
    por selección 824
eje
  x **135**, 556
  y **135**, **556**
ejecución
  del ciclo **117**
  secuencial **103**
ejecutar **20**, 40
  una aplicación 21
ejercicio(s)
  de protección de cheques 639
  de recursividad
    buscar el valor mínimo en un arreglo 807
    búsqueda binaria o lineal 837
    fractales 807
    generar laberintos al azar 809
    imprimir un arreglo 807
    laberintos de cualquier tamaño 809
    mayor común divisor 806
    método `potencia` recursivo 805
    ocho reinas 807
    palíndromos 807
    quicksort 837
    recorrer un laberinto mediante vuelta atrás recursiva 808
    tiempo para calcular números de Fibonacci 809
    visualización de la recursividad 806
elemento
  cero **245**
  de azar **213**
  de un arreglo **245**
  de una tabla 272
eliminación de duplicados 300
eliminar
  archivos 647
  directorios 647
  fugas de recursos 455
  objeto `String` duplicado 711
  palabras duplicadas, ejercicio 774
elipsis (`...`) en la lista de parámetros de un método **281**
`Ellipse2D`, clase 556
`Ellipse2D.Double`, clase **581**, 592
`Ellipse2D.Float`, clase 581
`else`, palabra clave 107
emacs 18
e-mail (correo electrónico) 25
`Empleado`
  clase que implementa `PorPagar` 426
  programa de prueba de la jerarquía de clases 414
  superclase abstracta 406
`EmpleadoAsalariado`
  clase que implementa el método `getPaymentAmount` de la interfaz `PorPagar` 428
  clase concreta que extiende a la clase `abstract` `Empleado` 408
`EmpleadoBaseMasComision`, clase que extiende a `EmpleadoPorComision` 413
`EmpleadoPorComision`, clase derivada de `Empleado` 411
`EmpleadoPorHoras`, clase derivada de `Empleado` 409
`EmptyStackException`, clase **710**
en línea, llamadas a métodos **328**
encabezado de método 73
encapsulamiento **12**
encuesta estudiantil, ejercicio 683
`endsWith`, método de la clase `String` **604**
ensamblador **9**
`ensureCapacity`, método de la clase `StringBuilder` **613**
entero(s) **45**
  arreglo de 249
  binario 150
  cociente 51
  división de **51**, **118**
  sufijo `L` 709
  valor 47
entorno de desarrollo integrado (IDE) **18**
entrada de datos 91
`entries`, método de la interfaz `DirectoryStream` 774
`enum` **221**
  constante 335
  constructor **335**
  declaración **335**
  `EnumSet`, clase 337
  palabra clave 221
  tipo **221**
  `values`, método **336**
`EnumSet`, clase **337**
  `range`, método **337**
enviar un mensaje a un objeto 11, 12
envoltura
  de línea **542**
  de objetos flujo **662**, 667
  de sincronización **721**
  de tipos, clase 618, **687**
  no modificable **721**
envolver texto en un objeto `JTextArea` 542
`EOFException`, clase **670**
`equals`, método
  de la clase
    `Arrays` **285**
    `Object` 387
    `String` 601, 603
`equalsIgnoreCase`, método de la clase `String` 601, 603
error(es)
  de compilación **36, 40**
  de desbordamiento 313
  de sintaxis **36**
  del compilador **36**
  en tiempo
    de compilación **36**
    de ejecución **21**
  fatal **110**, 313
    en tiempo de ejecución **1**
    lógico **110**
  lógico **18**, 48, **110**, 155
    en tiempo de ejecución 48
    no fatal **110**
  no fatal en tiempo de ejecución **21**
  por desplazamiento en uno **155**
  sincrónico **451**
`Error`, clase **451**
es un, relación **362**, 398
escalamiento (números aleatorios) 214
  factor de escala 214, 217, 218
escalar **263**
  valores `BigDecimal` 348
escanear imágenes 5
escribir
  en un campo de texto 485
  la equivalencia en palabras del monto de un cheque 640
escucha de eventos 432, 492, 518
  clase adaptadora de 518
  interfaz de **488**, 489, 492, 493, 494, 513, 518
escuchar eventos **489**
esfera 235
espacio
  carácter 37
  en blanco **37**, 39, 58
  vacío
    horizontal **534**
    vertical **534**
especialización **361**
especificación del lenguaje Java™ 54
especificadores de formato **44**
  `%.2f` para números de punto flotante con precisión 126
  `%b` para valores `boolean` **180**
  `%c` 66
  `%d` **49**
  `%f` 66, **88**
  `%n` (separador de líneas) **44**
  `%s` **44**
espiral 593, 783
esquema **390**
  predeterminado del panel de contenido 541
esquina superior izquierda de un componente de GUI 135, 556
establecer, método 324
estado
  de acción en UML **104**, 184
  final en UML **104**, 182
  inicial 182
    en UML **104**
estrella de cinco puntas 586
estrictamente autosimilar, fractal **791**
estructura(s)
  de datos **244**
    prefabricadas 685
    subyacentes 710
  de secuencia **103**
etiqueta **388**, **481**
  de botón **495**
  de casilla de verificación **501**
  en un `switch` **168**
etiquetado de interfaz 422, **663**
Euler 303
evaluación
  de corto circuito **178**
  de izquierda a derecha 53
evaluar al entero más cercano 238
`EventListenerList`, clase **493**
evento 432, **485**, 558
  asíncrono **451**
  clave **494**, **525**
  de ratón **494**, **513**
    manejo 514
  externo **513**
`EventObject`, método de la clase `getSource` 490
examen rápido sobre hechos del calentamiento global, ejercicio 199
excepción(es) **256**, 442
  encadenada **462**
  manejador de **256**
  manejo de 254
  no verificadas **452**
  parámetro de 257
  sin atrapar **449**
  verificada **452**
Excepciones 257
  `IndexOutOfRangeException` 257
`Exception`, clase **451**
`exists`, método de la clase `Files` **648**
`exit`, método de la clase `System` **455**, **653**
`EXIT_ON_CLOSE`, constante de la clase `JFrame` **138**
`exp`, método de `Math` 204
explorador
  de Phishing 683
  de spam 642

exponenciación 313
expresión **49**
  condicional **110**
  controladora de un `switch` **168**
  de acceso a un arreglo **245**
  de acción en UML **104**
  de creación
    de arreglos **246**
    de instancia de clase **75**, 83, 318
  entera 170
  integral constante **165**, 170
  lambda
    bloque de instrucciones 733
    composición 742
    con una lista de parámetros vacía 734
    inferencia de tipos 738
    (Java SE 8) **733**
    lista de parámetros 733
    manejador de eventos 763
    referencias a métodos 734
    tipo de 733
    tipo de objetivo **738**
    token de flecha (->) **733**
  regular **624**, 758
    * 628
    . 632
    ? 628
    \D 625
    \d 625
    \S 625
    \s 625
    \W 625
    \w 625
    ^ 628
    {*n*,} 629
    {*n*,*m*} 628
    {*n*} 628
    | 628
    + 628
extender una clase **361**
extenderse en sentido contrario a las manecillas del reloj 575
`extends`, palabra clave **137**, **365**, 376
extensibilidad 398

F

Facebook 14
factor de carga **715**
factorial 150, 196, 779
`factorial`, método 779
Fahrenheit 551
  equivalencia a grados centígrados 240
falla de constructor, ejercicio 472
`false`, palabra clave **54**, **110**
fase 120
  de inicialización 120
  de procesamiento 120
  de terminación 120
`Fecha` y `Hora`, clase, ejercicio 357
`fibonacci`, método 784
Fibonacci, serie 307, 783, 785
  definida en forma recursiva 783
`Figura`
  jerarquía de clases 363, 393
  objeto 584
figura(s) 581
  bidimensionales 556
  con tamaños aleatorios 593
  rellena 227, 585
`FiguraBidimensional`, clase 393
`FiguraTridimensional`, clase 393
filas de un arreglo bidimensional 272
`File`, clase
  `toPath`, método **671**
  utilizada para obtener la información de archivos y directorios 649
`FileInputStream`, clase 721,
`FileNotFoundException`, clase **653**
`FileOutputStream`, clase 720
`FileReader`, clase 676
`Files`, clase **647**, 759
  `exists`, método **648**
  `getLastModifiedTime`, método **648**
  `isDirectory`, método **648**
  `lines`, método (Java SE 8) **759**
  `newDirectoryStream`, método **648**
  `newOutputStream`, método **665**, **668**
  `size`, método **648**
`FILES_AND_DIRECTORIES`, constante de `JFileChooser` **670**
`FILES_ONLY`, constante de `JFileChooser` **670**
`FileWriter`, clase 676
`fill`, método de la clase
  `Arrays` **285**, 286
  `Collections` 696, **702**
  `Graphics2D` **584**, 585, 588, 593
`fill3DRect`, método de la clase `Graphics` 572, **575**
`fillArc`, método de la clase `Graphics` 291, **292**, **575**
`fillOval`, método de la clase `Graphics` **228**, **524**, 572, **575**
`fillPolygon`, método de la clase `Graphics` 578, **581**
`fillRect`, método de la clase `Graphics` **228**, **560**, 572, 585
`fillRoundRect`, método de la clase `Graphics` **573**
`filter`, método de la interfaz
  `IntStream` (Java SE 8) **741**
  `Stream` (Java SE 8) 745, 748
`FilterInputStream`, clase **674**
`FilterOutputStream`, clase **674**
filtrar
  elementos de un flujo (Java SE 8) 741
  un flujo **674**
fin
  de archivo (EOF)
    combinaciones de tecla 654
    indicador **167**
    marcador **645**
  de entrada de datos 119
  de línea (una sola línea), comentario de, `//` **36**, 39
Fin, tecla 525
`final`
  clase **420**
  método **418**
  palabra clave 179, **205**, 250, 343, 419
  variable 250
  variable local 507
`finalize`, método **338**, 388
`finally`
  bloque **448**, 454
  cláusula **454**
  palabra clave **448**
`find`, método de la clase `Matcher` **631**
`findFirst`, método de la interfaz `Stream` (Java SE 8) 752
Firefox, navegador Web 90
firma de un método **226**, 227
`first`, método de `SortedSet` **713**
`flatMap`, método de la interfaz `Stream` (Java SE 8) **760**
flecha, 104
  de desplazamiento **507**
  de transición 107, 115
  en UML **104**, 114
`float`
  promociones de tipos primitivos 211
  sufijo de literal `F` **710**
  tipo primitivo **47**, **84**
`Float`, clase **687**
`floor`, método de `Math` 204
`FlowLayout`, clase **483**, 529, 530
  `CENTER`, constante **532**
  `LEFT`, constante **532**
  `RIGHT`, constante **532**
  `setAlignment`, método **532**
flujo(s) **457**, **734**
  basado en bytes **646**
  basado en caracteres **646**
  de bytes **645**
  de entrada estándar (`System.in`) **47**, 646
  de error estándar **448**, **457** (`System.err`) 646, 674
  de salida estándar **457** (`System.out`) **39**, 646, 674
  de trabajo **104**
  del control 114, 124
    instrucción `if...else` 107
  infinito (Java SE 8) **762**
  (Java SE 8)
    asignar elementos a nuevos valores 742
    canalización **734**, 741, 742
    `DoubleStream`, interfaz **736**
    filtrar elementos 741
    infinitos **762**
    `IntStream`, interfaz **736**
    `LongStream`, interfaz **736**
    operación intermedia **734**
    operación perezosa 741, 742
    operación terminal **734**
    operaciones ansiosas 739
`flush`, método
  de la clase `BufferedOutputStream` **674**
foco **486**
`FocusAdapter`, clase 519
`FocusListener`, interfaz 519
`Font`, clase **500**, **556**, 567
  `BOLD`, constante **567**
  `getFamily`, método 566, **569**
  `getName`, método 566, **567**
  `getSize`, método 566, **567**
  `getStyle`, método 566, **569**
  `isBold`, método 566, **569**
  `isItalic`, método 566, **569**
  `isPlain`, método 566, **569**
  `ITALIC`, constante **567**
  `PLAIN`, constante **567**
`FontMetrics`, clase **556**, **569**
  `getAscent`, método 570
  `getDescent`, método 570
  `getFontMetrics`, método **569**
  `getHeight`, método 570
  `getLeading`, método 570
`for`, instrucción de repetición 105, **155**, 158, 160, 162, 186
  anidado 253, 274
  diagrama de actividades 158
  ejemplo 158
  encabezado **156**
  `for` mejorado anidado 274
  mejorado 262
`forDigit`, método de la clase `Character` **621**
`forEach`, método de la interfaz
  `IntStream` (Java SE 8) **738**
  `Map` (Java SE 8) **755**
  `Stream` (Java SE 8) 745
Formas de Java2D, paquete **212**
`format`, método
  de la clase `Formatter` 654
  de la clase `NumberFormat` **347**
  de la clase `String` **92**, **318**
formato
  de enteros decimales 49
  de hora
    estándar 319
    universal 316, 318, 319

de intercambio de gráficos (GIF) 484
de línea recta **51**
de reloj de 24 horas 316
tabular 249
`Formatter`, clase **647**, 651
`close`, método **654**
`format` método **654**
`FormatterClosedException`, clase **654**
formulación de algoritmos 115
Fortran (traductor de fórmulas) 15
fractal **791**
copo de nieve de Koch **792**
curva de Koch **791**
ejercicios 807
fractal estrictamente autosimilar **791**
"fractal Lo"
en el nivel 0 793
en el nivel 1, con los puntos C y D determinados para el nivel 2 794
en el nivel 2 794, 795
en el nivel 2, se proporcionan líneas punteadas del nivel 1 794
nivel **792**
orden **792**
profundidad **792**
propiedad de autosimilitud **791**
Fractal, interfaz de usuario 795
`frequency`, metodo de `Collections` 696, **706**
fuerza bruta 304, 305
Paseo del Caballo 305
fuga
de recursos 337, **454**
de seguridad 37, 76, 213
`Function`, interfaz funcional (Java SE 8) 733, 747
`apply`, método 747
`identity`, método 762
`@FunctionalInterface`, anotación **764**
Fundación
de software Apache 14
Eclipse 14
Mozilla 14
fusionar dos arreglos 827

G

gadgets de ventana 474
generador
de crucigramas 641
de palabras de números telefónicos, ejercicio 682
generalidades 398
de los paquetes 213
`GeneralPath`, clase 556, **586**, 592
`closePath`, método **588**
`lineTo`, método **588**
`moveTo`, método **587**
generar laberintos al azar, ejercicio 809
genéricos 686
notación de diamante **691**
`get`, método
de la clase
`ArrayList<T>` **290**
`Paths` **647**, 648
de la interfaz
`List<T>` **690**
`Map` **717**
`getActionCommand`, método de la clase `ActionEvent` **490**, 498
`getAscent`, método de la clase `FontMetrics` 570
`getAsDouble`, método de la clase `OptionalDouble` (Java SE 8) **739**, 757
`getBlue`, método de la clase `Color` **560**, 562
`getChars`, método
de la clase
`String` **599**
`StringBuilder` **614**
`getClass`
método de la clase `Object` **484**
método de `Object` 388, **418**
`getClassName`, método de la clase `StackTraceElement` **461**
`getClickCount`, método de la clase `MouseEvent` **521**
`getColor`
método de la clase
`Color` **560**
`Graphics` 560
`getContentPane`, método de la clase `JFrame` **510**
`getCurrencyInstance`, método de la clase `NumberFormat` **347**
`getDescent` método de la clase `FontMetrics` 570
`getFamily`, método de la clase `Font` 566, **569**
`getFileName`
método de la clase `StackTraceElement` **461**
método de la interfaz `Path` **648**
`getFont`, método de la clase `Graphics` **567**
`getFontMetrics`
método de la clase
`FontMetrics` **569**
`Graphics` 570
`getGreen`, método de la clase `Color` **560**, 562
`getHeight`
método de la clase
`FontMetrics` 570
`JPanel` **137**
`getIcon`, método de la clase `JLabel` **484**
`getKeyChar`, método de la clase `KeyEvent` **528**
`getKeyCode`, método de la clase `KeyEvent` **528**
`getKeyModifiersText`, método de la clase `KeyEvent` **528**
`getKeyText`, método de la clase `KeyEvent` **528**
`getLastModifiedTime`, método de la clase `Files` **648**
`getLeading`, método de la clase `FontMetrics` 570
`getLineNumber`, método de la clase `StackTraceElement` **461**
`getMessage`, método de la clase `Throwable` **461**
`getMethodName`, método de la clase `StackTraceElement` **461**
`getModifiers`, método de la clase `InputEvent` **528**
`getName`
método de la clase
`Class` 388, **418**
`Font` 566, **567**
`getPassword`, método de la clase `JPasswordField` **490**
`getPoint`, método de la clase `MouseEvent` **524**
`getProperty`, método de la clase `Properties` **718**
`getRed`, método de la clase `Color` **560**, 562
`getResource`, método de la clase `Class` **484**
`getSelectedFile`, método de la clase `JFileChooser` **671**
`getSelectedIndex`
método de la clase
`JComboBox` **508**
`JList` **511**
`getSelectedText`, método de la clase `JTextComponent` **541**
`getSelectedValues`, método de la clase `JList` **513**
`getSize`, método de la clase `Font` 566, **567**
`getSource`, método de la clase `EventObject` 490
`getStackTrace`, método de la clase `Throwable` **461**
`getStateChange`, método de la clase `ItemEvent` **508**
`getStyle`, método de la clase `Font` 566, **569**
`getText`, método de la clase `JLabel` **484**
`getWidth`, método de la clase `JPanel` **137**
`getX`, método de la clase `MouseEvent` **518**
`getY`, método de la clase `MouseEvent` **518**
GIF (formato de intercambio de gráficos) 484
gigabyte 5
GitHub 14
Google
Maps 26
Play **15**
Gosling, James 17
`Goto`
eliminación de 103
instrucción **103**
GPS, dispositivo 5
gradiente **584**
acíclica **584**
cíclica **584**
`GradientPaint`, clase **556**, 584, 593
grado(s) 575
negativos **575**
positivos **575**
graficar información 252
gráfico(s) 197, 522
bidimensionales 581
de barras 197, 251, 252
de pastel 593
ejercicio 593
de tortuga 302, 593
ejercicio 593
independientes de la plataforma 558
portables de red (PNG) 484
`Graphics`, clase **136**, 227, 291, 434, 435, **523**, **556**, 558, 581
`clearRect`, método 572
`draw3DRect`, método 572, **575**
`drawArc`, método **575**, 592
`drawLine`, método **137**, 572
`drawOval`, método 572, **575**
`drawPolygon`, método 578, **580**
`drawPolyline`, método 578, **580**
`drawRect`, método 572, 585, 592
`drawRoundRect`, método **573**
`drawString`, método **562**
`fill3DRect`, método 572, **575**
`fillArc`, método **575**
`fillOval`, método **228**, **524**, 572, **575**
`fillPolygon`, método 578, **581**
`fillRect`, método **228**, **560**, 572, 585
`fillRoundRect`, método **573**
`getColor`, método 560
`getFont`, método **567**, 567
`getFontMetrics`, método 570
`setColor`, método **228**, 585
`setFont`, método **567**
`Graphics2D`, clase **556**, **581**, 585, 588, 592
`draw`, método **584**
`fill`, método **584**, 585, 588, 593
`rotate`, método **588**
`setPaint`, método **584**
`setStroke`, método **584**
`translate`, método **588**
`GridLayout`
clase 529, **536**
que contiene seis botones 536
`group`, método de la clase `Matcher` 632
`groupingBy`, método de la clase `Collectors` (Java SE 8) **755**, 756, 760, 762
grupo de botones de opción **501**
Grupo de Expertos en Fotografía Unidos (JPEG) 484
GUI (interfaz gráfica de usuario) 432
componente **474**
herramienta de diseño **529**
portable 212

H

hablar a una computadora 5
hacer clic
  en las flechas de desplazamiento 507
  en un botón 485
hacer referencia a un objeto **81**
hardware **2**, 4, 9
`hashCode`, método de `Object` 387
hashing **714**
`HashMap`, clase **714**
  `keySet`, método**718**
`HashSet`, clase **711**
`Hashtable`
  clase **714**, 715
  persistente 718
`hasNext`, método
  de la clase `Scanner` **168**, 654
  de la interfaz `Iterator` **691**, 694
`hasPrevious`, método de `ListIterator` **694**
`headSet`, método de la clase `TreeSet` **712**
heredar **137**
herencia **12**, 137, **361**
  ejemplos 362
  `extends`, palabra clave **365**, 376
  jerarquía **362**, 403
    para todos los `MiembroDeLaComunidad` de la universidad 363
  múltiple 361
  simple **361**
herramienta de desarrollo de programas 106, 122
heurística 304
  de accesibilidad 304
hexadecimal (base 16), sistema numérico 241, 313
hilo **449**, 558
  de despacho de eventos (EDT) **558**
  sincronización 721
hipotenusa de un triángulo rectángulo 238
Hopper, Grace 15
`HORIZONTAL_SCROLLBAR_ALWAYS`, constante de la clase `JScrollPane` **542**
`HORIZONTAL_SCROLLBAR_AS_NEEDED`, constante de la clase `JScrollPane` **542**
`HORIZONTAL_SCROLLBAR_NEVER`, constante de la clase `JScrollPane` **542**
HousingMaps.com (`www.housingmaps.com`) 27
HTML (lenguaje de marcado de hipertexto) **26**
HTTP (protocolo de transferencia de hipertexto) **26**
`HugeInteger`, clase 358, ejercicio 358

I

IBM Corporation 15
`Icon` , interfaz **484**
icono 478
ID de evento **494**
IDE (entorno de desarrollo integrado) **18**
Identificador(es) **37**, 47
  nomenclatura de lomo de camello 72
  uniforme de recursos (URI) **648**
  válido 47
`identity`, método de la interfaz funcional `Function` (Java SE 8) 762
`if`, instrucción de selección simple **54**, 105, 106, 165, 186, 187
  diagrama de actividades 106
`if...else`, instrucción de selección doble 105, **106**, 122, 165, 186
  diagrama de actividades 107
ignorar el elemento cero de un arreglo 256
`IllegalArgumentException`, clase **318**
`IllegalStateException`, clase **657**
`ImageIcon`, clase **390**, **484**
implementación **12**
  abstracta 722
  de colección 721
  de una interfaz **397**, **421**, 429
  varias interfaces **515**
implementar la privacidad con la criptografía, ejercicio 151
`import`, declaración **46**, 48, 79
impresora 21
imprimir
  en varias líneas 41, 42
  un arreglo 807
    al revés, ejercicio 807
    ejercicio 807
    mediante recursividad 807
  una línea de texto **39**
"impuesto justo" 199
incrementar una variable de control **153**
incremento 160
  de una instrucción `for` 158
  de una variable de control 154
  expresión de 175
  operador de, ++ 131
  y decremento, operadores de 132
Independiente de la plataforma 19
`indexOf`
  método de la clase `ArrayList<T>` 288
  método de la clase `String` **605**
`IndexOutOfBoundsException`, clase 703
`IndexOutOfRangeException`, clase 257
indicador **48**
índice
  de arreglo fuera de límites 451
  de masa corporal (IMC) 33
  (subíndice) **242**, 254
    cero **245**
    de un `JComboBox` **507**
inferencia de tipos **691**
  con la notación <> (Java SE 7) 691
inferir
  tipos de parámetros en una lambda 738
  un tipo con la notación de diamante (<>) **290**
información
  a nivel de clase **338**
  de movimiento 5
  de orientación 5
  volátil 5
ingeniería de software 331
inicialización de arreglos bidimensionales en las declaraciones 273
inicializador de un arreglo **248**
  anidado **272**
  multidimensional 272
inicializar una variable en una declaración **47**
Inicio, tecla 525
inmutabilidad **732**
inmutable 599
`InputEvent`, clase **514**, 521, 525
  `getModifiers`, método **528**
  `isAltDown`, método 521, **528**
  `isControlDown`, método **528**
  `isMetaDown`, método 521, **528**
  `isShiftDown`, método **528**
`InputMismatchException`, clase **445**, 448
`InputStream`, clase 663, 673, 721
`InputStreamReader`, clase **676**
`insert`, método de la clase `StringBuilder` **617**
`instanceof`, operador **417**
instancia **11**
  método de (no `static`) 339
  métodos de **208**
  referencia a un método de (Java SE 8) 747
  variable de **12**, 72, 85, 204
Instrucción(es) **39**, 73, **121**
  anidadas **126**
  anidamiento de instrucciones de control **105**
  apilamiento de instrucciones de control **105**
  asistida por computadora (CAI) 241, 242
    niveles de dificultad 242
    reducción de la fatiga de los estudiantes 242
    supervisión del rendimiento de los estudiantes 242
    variación de los tipos de problemas 242
  `break` **168**, 174, 198
  `continue` **174**, 198
  de asignación 48
  de control **102**, 103, 105, 106, 787
    anidadas **126**, 185, 187, 217, 129
    anidamiento **105**, 185
    apilamiento **105**, 182
  de declaración de variables **47**
  de iteración **105**
  de repetición **104**, **105**, 113, 121, 787
    `do...while` 105, 163, 164, 186
    `for` 105, 158, 186
    `while` 105, **114**, 117, 122, 124, 153, 186, 187
  de selección **103**, **105**
    doble **105**, 128
    `if` 105, 106, 165, 186, 187
    `if` anidada 111
    `if...else` 105, **106**, 107, 122, 165, 186
    `if...else` anidada **107**, 109, 111, 113
    múltiple **105**
    `switch` 105, **165**, 169, 186
  `do...while` 105, **163**, 164, 186
  `for` 105, **155**, 158, 160, 162, 186
    anidada 253, 273, 274, 275, 279
    mejorada **262**, 274
  `if` **54**, 105, 106, 165, 186, 187
    de selección simple 106
  `if...else` 105, **106**, 107, 122, 165, 186
    anidado **107**, 109
  iteración **105**
  por línea, una 57
  profundamente anidada 185
  `return` 202, 209
  selección
    múltiple **105**
    simple **105**
  `switch` 105, **165**, 169, 186
    de selección múltiple 217
  `throw` **318**
  `try` **256**
    con recursos (`try`-with-resources) **467**
  vacía(s) **57**, 110
    un punto y coma, `;` **57**, 110, 165
  `while` 105, 114, 117, 122, 124, 153, 186, 187
`int`, tipo primitivo **47**, 122, 131, 165
  promociones 211
`IntBinaryOperator`, interfaz funcional (Java SE 8) **740**
  `applyAsInt`, método **740**
`IntConsumer`, interfaz funcional (Java SE 8) **738**
  `accept`, método **738**
`Integer`, clase 283, 478, **687**
  `parseInt`, método 283, 478
`integerPower`, método 238
integridad de los datos 331
IntelliJ IDEA 18
interacciones entre objetos 499, 503
intercambiar valores 821, 824
interés compuesto 160, 196, 197
`interface`, palabra clave **421**
interfaz(ces) **12**, 397, 422, **421**, 430
  `ActionListener` **489**, 494
  `AutoCloseable` 338, **467**
  `BiConsumer`, interfaz funcional (Java SE 8) **755**, 762
  `BinaryOperator`, interfaz funcional (Java SE 8) 733

CharSequence **631**
Collection **686**, 687, 696
Collector, interfaz funcional (Java SE 8) **745**
Comparable 431, 603, **697**
Comparator **697**, 698, 748
ComponentListener 519
Consumer, interfaz funcional (Java SE 8) 733, 738, 751
ContainerListener 519
DataInput 674
DataOutput 674
de marcado **663**
de programación de aplicaciones (API) **17**, **201**
de programación de aplicaciones de Java (API de Java) **17**, **46**, 201, 211
declaración de **421**
default, métodos (Java SE 8), 432, **432**
DirectoryStream **647**
DoubleStream, interfaz funcional (Java SE 8) **736**
FocusListener 519
funcional(es) (Java SE 8) **433**, 732, 733
  BiConsumer **755**, 762
  BinaryOperator 733
  Consumer 733, 738, 751
  Function 733, 747
  @FunctionalInterface, anotación **764**
  Predicate 733, 751
  Supplier 733
  UnaryOperator 733
Function, interfaz funcional (Java SE 8) 733, 747
gráfica de usuario (GUI) **91**, 432, **474**
  componente **91**
  herramienta de diseño **529**
Icon **484**
implementar más de una a la vez **515**
IntBinaryOperator, interfaz funcional (Java SE 8) **740**
IntConsumer, interfaz funcional (Java SE 8) **738**
IntPredicate, interfaz funcional (Java SE 8) **741**
IntStream, interfaz funcional (Java SE 8) **736**
IntUnaryOperator, interfaz funcional (Java SE 8) **742**
ItemListener **501**
Iterator **687**
KeyListener **494**, 519, 525
LayoutManager **528**, 532
LayoutManager2 **532**
List 686, **694**
ListIterator **688**
ListSelectionListener **510**
LongStream, interfaz funcional (Java SE 8) **736**
Map 686, 714
Map.Entry 760
marcadoras 422
MouseInputListener **513**, 518
MouseListener **494**, **513**, 519
MouseMotionListener **494**, **513**, 518, 519
MouseWheelListener **515**
ObjectInput **662**
ObjectOutput **662**
Path **647**
Predicate, interfaz funcional (Java SE 8) 733, 751
Queue **686**, **710**
Runnable 432
Serializable 432, **663**
Set **686**, **711**
SortedMap **714**
SortedSet **712**
static, métodos (Java SE 8) 433
Stream (Java SE 8) **734**, **744**
  Supplier, interfaz funcional (Java SE 8) 733
  SwingConstants **484**
  ToDoubleFunction, interfaz funcional (Java SE 8) **757**
  UnaryOperator, interfaz funcional (Java SE 8) 733
  WindowListener 518
interlineado **569**
internacionalización **347**
Internet **25**
  de las cosas (IoT) 27
  Explorer 90
intérprete 10
intersección
  de dos conjuntos 357
  teórica de conjuntos 357
IntPredicate, interfaz funcional (Java SE 8) **741**
  test, método **741**, **742**
Intro (o *Retorno*), tecla 39, 493
introducir datos mediante el teclado 58
ints, método de la clase SecureRandom (Java SE 8) **762**
IntStream, interfaz (Java SE 8) **736**
  average, método **739**
  boxed, método **762**
  count, método **739**
  filter, método **741**
  forEach, método **738**
  map, método **742**
  max, método **739**
  min, método **739**
  of, método **738**
  range, método **734**
  rangeClosed, método **743**
  reduce, método **739**
  sorted, método **741**
  sum, método **739**
IntUnaryOperator, interfaz funcional (Java SE 8) **742**
  applyAsInt, método **742**
invocar un método **44**, **84**, 199, 202
IOException, clase **667**
iOS 13
isAbsolute, método de la interfaz Path **648**
isActionKey, método de la clase KeyEvent **528**
isAltDown, método de la clase InputEvent **521**, **528**
isBold, método de la clase Font 566, **569**
isControlDown, método de la clase InputEvent **528**
isDefined, método de la clase Character **620**
isDigit, método de la clase Character **620**
isDirectory, método de la clase Files **648**
isEmpty, método
  ArrayList 332
  Map **718**
  Stack **710**
isItalic, método de la clase Font 566, **569**
isJavaIdentifierPart, método de la clase Character **620**
isJavaIdentifierStart, método de la clase Character **620**
isLetter, método de la clase Character **620**
isLetterOrDigit, método de la clase Character **620**
isLowerCase, método de la clase Character **620**
isMetaDown, método de la clase InputEvent **521**, **528**
isPlain, método de la clase Font 566, **569**
isSelected, método
  JCheckBox **501**
isShiftDown, método de la clase InputEvent **528**
isUpperCase, método de la clase Character **620**
ITALIC, constante de la clase Font **567**
ItemEvent, clase **501**, 504
  getStateChange, método **508**
ItemListener, interfaz **501**
  itemStateChanged, método **501**
itemStateChanged, método de la interfaz ItemListener **501**
iteración **117**, 787
  de un ciclo **153**, 175
  externa **731**, 761
  interna **732**
iteración (ciclos)
  de un ciclo for 255
iterador **685**
  bidireccional **694**
  de falla rápida 691
iterativo (no recursivo) 779
Iterator
  interfaz **687**
    hasNext, método **691**
    next, método **691**
    remove, método **691**
  método de Collection **691**

J

Jacopini, G. 103
Java
  Abstract Window Toolkit (AWT), paquete **212**
  biblioteca de clases **17**, **46**, **201**
  compilador **18**
  2D
    API 556, **581**
    formas 581
  Enterprise Edition (Java EE) **3**
  entorno de desarrollo 18, 19, 20
  HotSpot, compilador **20**
  lenguaje de programación 14
  máquina virtual (JVM) **19**, 35, 38
  Micro Edition (Java ME) **4**
  paquete
    de componentes GUI Swing **212**
    de concurrencia **212**
    de Entrada/Salida **212**
    de red **212**
    de utilerías **212**
    del lenguaje **212**
    Swing Event **212**
  SE 6
    generalidades del paquete 213
  SE 7
    inferencia de tipos con la notación <> 691
    Java Standard Edition 7 **3**
  SE 8 (Java Standard Edition 8) **3**, 245, 263, 271, 287, 321
    @FunctionalInterface, anotación 764, **764**
    API de fecha/hora 213, 287, **321**
    BinaryOperator, interfaz funcional 733
    clases internas anónimas con lambdas 508
    Collector, interfaz funcional **745**
    Collectors, clase **745**
    Consumer, interfaz funcional 733, 738, 751
    default, método en una interfaz 732, 763
    default, métodos en interfaces **432**
    doubles, método de la clase SecureRandom **762**
    efectivamente final 507
    Function, interfaz funcional 733, 747
    implementrar componentes de escucha de eventos con lambdas 491, 492
    IntBinaryOperator, interfaz funcional **740**

IntConsumer, interfaz funcional 738
interfaz(ces) funcional(es) 433, 733
IntPredicate, interfaz funcional 741
ints, método de la clase SecureRandom 762
IntUnaryOperator, interfaz funcional 742
java.util.function, paquete 732, 738
java.util.stream, pquete 736
lambda(s) 433, 633
lines, método de la clase Files 759
longs, método de la clase SecureRandom 762
método parallelSort de Arrays 287
Optional 752
OptionalDouble, clase 739
Predicate, interfaz funcional 733, 745, 748, 751
reversed, método de la interfaz Comparator 748
static, método en una interfaz 731, 732, 763
static, métodos de interfaz 433
Stream, interfaz 744
Supplier, interfaz funcional 733
ToDoubleFunction, interfaz funcional 757
UnaryOperator, interfaz funcional 733
Standard Edition (Java SE)
7 3
8 3
Standard Edition 7 (Java SE 7) 3
Standard Edition 8 (Java SE 8) 3
tipos de letra
Dialog 567
DialogInput 567
Monospaced 567
SansSerif 567
Serif 567
java, comando 19, 21, 35
java, intérprete 40
java.awt, paquete 479, 558, 559, 578, 581
java.awt.color, paquete 581
java.awt.event, paquete 212, 491, 493, 518, 528
java.awt.font, paquete 581
java.awt.geom, paquete 212, 581
java.awt.image, paquete 581
java.awt.image.renderable, paquete 581
java.awt.print, paquete 581
java.io, paquete 212, 646
java.lang, paquete 48, 203, 212, 365, 387, 597
importado en todos los programa de Java 48
java.math, paquete 126, 346, 781
java.net, paquete 212
java.nio.file, paquete 645, 646, 647, 759
java.security, paquete 213
java.sql, paquete 212
java.text, paquete 346
java.time, paquete 213, 321
java.util, paquete 46, 212, 288, 686, 708
java.util.concurrent, paquete 212
java.util.function, paquete (Java SE 8) 732, 738
java.util.prefs paquete 718
java.util.regex, paquete 597
java.util.stream, paquete (Java SE 8) 736
javac, compilador 18, 40
javacdeprecation, bandera 46
javadoc programa de utilería 36
Java2D, API 581
Javadoc, comentario 36
javax.swing, paquete 91, 212, 474, 476, 484, 493, 495, 540, 563
javax.swing.event, paquete 212, 492, 510, 518
JAX-WS, paquete 212
JButton, clase 479, 495, 498, 534
JCheckBox, botones y eventos de elementos 499
JCheckBox, clase 479, 498
isSelected, método 501
JColorChooser class 563, 565
cuadro de diálogo, ejercicio 593
showDialog, método 564
JComboBox, clase 479, 504
getSelectedIndex, método 508
que muestra una lista de nombres de imágenes 505
setMaximumRowCount, método 507
JComponent, clase 480, 481, 483, 493, 505, 508, 522, 538, 556, 558
paintComponent, método 137, 522, 556
repaint, método 559
setOpaque, método 522, 524
setToolTipText, método 483
JDBC, paquete 212
JDK 17, 40
Jerarquía
de clases 361, 402
de datos 6, 7
de figuras, ejercicio 439
JFileChooser, clase 670
CANCEL_OPTION, constante 671
FILES_AND_DIRECTORIES, constante 670
FILES_ONLY, constante 670
getSelectedFile, método 671
setFileSelectionMode, método 670
showOpenDialog, método 670
JFrame
clase 138, 229
add, método 138, 483
EXIT_ON_CLOSE 485
getContentPane, método 510
setDefaultCloseOperation, método 138, 485
setSize, método 138, 485
setVisible, método 138, 485
constante EXIT_ON_CLOSE de la clase 138
JFrame.EXIT_ON_CLOSE 485
JLabel, clase 388, 390, 479, 481
getIcon, método 484
getText, método 484
setHorizontalAlignment, método 484
setHorizontalTextPosition, método 484
setIcon, método 484
setText, método 484
setVerticalAlignment, método 484
setVerticalTextPosition, método 484
JList, clase 479, 508
addListSelectionListener, método 510
getSelectedIndex, método 511
getSelectedValues, método 513
setFixedCellHeight, método 513
setFixedCellWidth, método 513
setListData, método 513
setSelectionMode, método 510
setVisibleRowCount, método 510
JOIN_ROUND constante de la clase BasicStroke 586
JOptionPane constantes para diálogos de mensajes
JOptionPane.ERROR_MESSAGE 479
JOptionPane.INFORMATION_MESSAGE 479
JOptionPane.PLAIN_MESSAGE 479
JOptionPane.QUESTION_MESSAGE 479
JOptionPane.WARNING_MESSAGE 479
JOptionPane, clase 90, 91, 476, 477
constantes para diálogos de mensajes 479
PLAIN_MESSAGE, constante 478
showInputDialog, método 92, 477
showMessageDialog, método 91, 478
JPanel, clase 136, 137, 479, 522, 529, 538
getHeight, método 137
getWidth, método 137
JPasswordField, clase 485, 490
getPassword, método 490
JPEG (Grupo de Expertos en Fotografía Unidos) 484
JRadioButton, clase 498, 501, 504
JScrollPane, clase 510, 513, 541, 542
HORIZONTAL_SCROLLBAR_ALWAYS, constante 542
HORIZONTAL_SCROLLBAR_AS_NEEDED, constante 542
HORIZONTAL_SCROLLBAR_NEVER, constante 542
setHorizontalScrollBarPolicy, método 542
setVerticalScrollBarPolicy, método 542
VERTICAL_SCROLLBAR_ALWAYS, constante 542
VERTICAL_SCROLLBAR_AS_NEEDED, constante 542
VERTICAL_SCROLLBAR_NEVER, constante 542
JScrollPane, políticas de barras de desplazamiento 542
JTextArea, clase 528, 539, 541
setLineWrap, método 542
JTextComponent, clase 485, 488, 539, 541
getSelectedText, método 541
setDisabledTextColor, método 528
setEditable, método 488
setText, método 541
JTextField y JPasswordField 486
JTextField, clase 479, 485, 489, 493, 539
addActionListener, método 489
JToggleButton, clase 498
juego(s) de cartas 257
de Craps 301
de dados 218
jugar juegos 213
justificado a la izquierda 107, 161, 484, 530

K

Kelvin, escala de temperatura 551
kernel 13
KeyAdapter, clase 519
KeyEvent, clase 494, 525
getKeyChar, método 528
getKeyCode, método 528
getKeyModifiersText, método 528
getKeyText, método 528
isActionKey, método 528
KeyListener, interfaz 494, 519, 525
keyPressed, método 525, 528
keyReleased, método 525
keyTyped, método 525
keyPressed, método de la interfaz KeyListener 525, 528
keyReleased, método de la interfaz KeyListener 525
keySet, método
de la clase HashMap 718
de la clase Properties 721
keyTyped, método de la interfaz KeyListener 525
Keywords
abstract 402
boolean 110
break 168
case 168
catch 448
char 47
class 37, 72
continue 174
default 168

- `do` 105, **163**
- `double` **47**, **84**
- `else` 105
- `enum` **221**
- `extends` **137**, **365**, 376
- `false` **110**
- `final` 179, **205**, 250
- `finally` **448**
- `float` **47**, **84**
- `for` 105, **155**
- `if` 105
- `implements` **421**
- `import` **46**
- `instanceof` **417**
- `int` **47**
- `interface` **421**
- `new` 47, **75**, 246, 247
- `null` **81**, 246
- `private` **72**, 321, 331
- `public` **37**, 71, 72, 206, 321
- `return` **71**, **74**, 202, 209
- `static` **91**, 162, 203
- `super` **364**, 387
- `switch` 105
- `this` **73**, **322**, 339
- `throw` **458**
- `true` **110**
- `try` **447**
- `void` **38**, 73
- `while` 105, **163**

kilometraje obtenido por los automóviles 147
Kit de desarrollo
- de Java (JDK) 40
- de Java SE (JDK) 17
- de software (SDK) **29**

Koenig, Andrew 442

L

laberintos de cualquier tamaño, ejercicio 809
Lady Ada Lovelace 16
lambda (Java SE 8) **433**
LAMP **28**
lanzamiento de monedas 214, 241
`last`, método de `SortedSet` **713**
`lastIndexOf`, método de la clase `String` **605**
latín cerdo 637
`layoutContainer`, método de la interfaz `LayoutManager` **532**
`LayoutManager`, interfaz **528**, 532
- `layoutContainer`, método **532**

`LayoutManager2`, interfaz **532**
leer archivos 647
`LEFT`, constante de la clase `FlowLayout` **532**
legibilidad 36, 37, 128
`length`
- campo de un arreglo 246
- método de la clase `String` **599**
- método de la clase `StringBuilder` **612**
- variable de instancia de un arreglo **246**

lenguaje(s)
- de alto nivel **10**
- de marcado de hipertexto (HTML) **26**
- ensamblador **9**
- extensible 70
- fuertemente tipificados **134**
- máquina **9**
  - programación 308
- orientado a objetos **13**
- práctico de extracción y generación de informes (Perl) 16
- unificado de modelado (UML) 13

letra 6
- mayúscula 38, 47
- minúscula 7, 37

levantarse y arreglarse, algoritmo 102
ley de Moore **4**
liberar un recurso 455
límite de crédito en una cuenta por cobrar 147
limpieza de la pila **459**
- de llamadas a métodos 459

`Line2D`, clase 556, **585**
`Line2D.Double`, clase **581**, 592
línea 556, 571, 580
- base del tipo de letra 567
- de comandos **39**
- de espera 710
- en blanco 37, 122
- punteada en UML **105**

`LinearGradientPaint`, clase 584
líneas
- aleatorias mediante la clase `Line2D.Double`, ejercicio 592
- conectadas 578
- gruesas 581
- punteadas 581

`LineNumberReader`, clase **676**
`lines`, método de la clase `Files` (Java SE 8) **759**
`lineTo`, método de la clase `GeneralPath` **588**
`LinkedList`, clase **688**, 704, 727
- `add`, método **695**
- `addFirst`, método **696**
- `addLast`, método **695**

Linux 13, 39, 653
- sistema operativo **14**

`List`
- interfaz 686, **694**, 697, 698, 702
  - `add`, método **690**, 693
  - `addAll`, método **693**
  - `clear`, método **694**
  - `get`, método **690**
  - `listIterator`, método **694**
  - `size`, método **690**, **694**
  - `subList`, método **694**
  - `toArray`, método **695**
- método de `Properties` **720**

lista(s) 507
- de argumentos de longitud variable **281**
- de compras 113
- de parámetros **73**
  - de un método 281
  - en una lambda 733
- de selección múltiple **508**, 510, 511
- desplegable 479, **504**
- inicializadora **248**
- separada por comas 160
  - de argumentos **44**, 47
  - de parámetros 207

`ListIterator`, interfaz **688**
- `hasPrevious`, método **694**
- `previous`, método **694**
- `set`, método **694**

`listIterator`, método de la interfaz `List` **694**
`ListSelectionEvent`, clase **508**
`ListSelectionListener`, interfaz **510**
- `valueChanged`, método **510**

`ListSelectionModel`, clase **510**
- `MULTIPLE_INTERVAL_SELECTION`, constante **510**, 513
- `SINGLE_INTERVAL_SELECTION`, constante **510**, 511, 513
- `SINGLE_SELECTION`, constante **510**

literal(es)
- de cadena **597**
- de punto flotante **84**, **94**
  - `double` de manera predeterminada 84

llamada(s)
- a método(s) **11**, 202, 207
  - en la pila de ejecución del programa 786
  - realizadas dentro de la llamada `fibonacci(3)` 786
- por referencia **265**
- por valor **265**
- recursiva indirecta **778**
- síncrona **502**

llave(s)
- derecha, } **38**, 46, 117, 124
- izquierda, { **38**, 46
- ({ y }) 109, 124, 156, 165, 248
  - no requeridas 169

llegada de mensaje de red 451
llenar con color 556
`load`, método de `Properties` **721**
localización **480**
Localizador uniforme de recursos (URL) **648**
`log`, método de `Math` 205
logaritmo 204
- natural 204

Logo, lenguaje 302
lomo de camello **48**
`long`
- promociones 211
- sufijo literal `L` **709**

`Long`, clase **687**
`longs`, método de la clase `SecureRandom` (Java SE 8) **762**
`LongStream`, interfaz (Java SE 8) **736**
`lookingAt`, método de la clase `Matcher` **631**
Lord Byron 16
Lovelace, Ada 16

M

Mac OS X 13, 39, 653
`main`, método **38**, 39, 46, 79
Mandelbrot, Benoit 791
- de eventos 432, **485**
  - implementar con una lambda 763
  - lambda 763
- de excepciones **448**
  - predeterminado 461

manejar una excepción 446
manejo de eventos **485**, 488, 493
- clave 525
- origen del evento **490**

manipulación del color 558
mantenimiento de código 89
`map`
- método de la interfaz
  - `IntStream` (Java SE 8) **742**
  - `Stream` (Java SE 8) 747

`Map`, interfaz 686, **714**
- `containsKey`, método **717**
- `foreach`, método (Java SE 8) **755**
- `get`, método **717**
- `isEmpty`, método **718**
- `put`, método **717**
- `size`, método **718**

`Map.Entry`, interfaz 760
`mapToDouble`, método de la interfaz `Stream` (Java SE 8) **756**
máquina
- analítica 16
- virtual (VM) **19**

marco
- de pila **210**, 788
- de trabajo de colecciones **685**

mashups 26
- populares 1450

`Matcher`, clase 597, **631**
- `find`, método **631**
- `group`, método 632
- `lookingAt`, método **631**
- `matches`, método **631**
- `replaceAll`, método **631**
- `replaceFirst`, método **631**

`matcher`, método de la clase `Pattern` **631**
`matches`
- método de la clase `Matcher` **631**

Pattern 631
String 624
Math, clase 162, 203, 204
abs, método 204
ceil, método 204
cos, método 204
E, constante 204
exp, método 204
floor, método 204
log, método 205
max, método 204
min, método 204
PI, constante 204, 235
pow, método 162, 203, 204, 235
sqrt, método 203, 204, 210
tan, método 204
Math.PI, constante 66, 592
MathContext, clase 348
matiz 565
max
método
de Collections 696, 703
de la interfaz IntStream (Java SE 8) 739
de Math 204
maximizar una ventana 481
máximo común divisor (MCD) 240, 806
ejercicio 806
Mayús 528
mayúsculas y minúsculas
como título de libro 478, 495
estilo oración 477
MCD (mayor común divisor) 806
media 52
aritmética 52
dorada 783
palabra 313
mejora
a la clase Fecha, ejercicio 356
a la clase Tiempo2, ejercicio 356
del rendimiento del ordenamiento de burbuja 836
memoria 4, 5
búfer de 675
concesión entre espacio en memoria y tiempo de ejecución 715
fuga de 338, 454
principal 5
ubicación de 50
unidad de 5
uso de 715
menú 475, 539
Meta, tecla 521
meter en una pila 209
método(s) 11, 38
abstracto 402, 404, 406, 493
anónimo(s) 433
(Java SE 8) 733
ayudante 319
de acceso 331
de clase 203
de construcción en bloques para crear programas 11
de consulta 331
de envoltura de la clase Collections 688
de interfaz static (Java SE 8) 433
de vista de rango 694, 712
declaración 39
exponencial 204
firma 226
implícitamente final 420
lista de parámetros 73
mutador 331
natural de comparación 697
para manejar eventos de ventana 518
predicado 171, 332
que hizo la llamada (el que llama) 202
static 162
utilitario 319
variable local 73
métrica de los tipos de letra 569
altura 571
ascendente 571
descendente 571
interlineado 571
Microsoft Windows 168
MiFigura
jerarquía 434
jerarquía con MiFiguraDelimitada 435
min
método
de Collections 696, 703
de la interfaz IntStream (Java SE 8) 739
de Math 204
minimizar una ventana 481
misma probabilidad 215
modelo
de evento de delegación 492
de reanudación del manejo de excepciones 449
de software 310
de terminación del manejo de excepciones 449
de un sistema de software 480, 488, 518
modificación
de la representación de datos interna de una clase, ejercicio 356
del sistema
de cuentas por pagar, ejercicio 440
de nómina, ejercicio 439
modificador de acceso 71, 72
en UML
- (private) 78
private 72, 321, 364
protected 321, 364
public 71, 321, 364
modo de selección 510
modularización de un programa mediante métodos 202
Monospaced, tipo de letra de Java 567
monto principal en un cálculo de interés 160
mostrar
la salida 58
texto en un cuadro de diálogo 90
una línea de texto 39
MouseAdapter, clase 518, 519
mouseClicked, método de la interfaz MouseListener 514, 519
mouseDragged, método de la interfaz MouseMotionListener 514, 522
mouseEntered, método de la interfaz MouseListener 514
MouseEvent, clase 494, 514
getClickCount, método 521
getPoint, método 524
getX, método 518
getY, método 518
isAltDown, método 521
isMetaDown, método 521
mouseExited, método de la interfaz MouseListener 514
MouseInputListener, interfaz 513, 518
MouseListener, interfaz 494, 513, 519
mouseClicked, método 514, 519
mouseEntered, método 514
mouseExited, método 514
mousePressed, método 514
mouseReleased, método 514
MouseMotionAdapter, clase 519, 522
MouseMotionListener, interfaz 494, 513, 518, 519
mouseDragged, método 514, 522
mouseMoved, método 514, 522
mouseMoved, método de la interfaz MouseMotionListener 514, 522
mousePressed, método de la interfaz MouseListener 514
mouseReleased, método de la intefaz MouseListener 514
MouseWheelEvent, clase 515
MouseWheelListener, interfaz 515
mouseWheelMoved, método 515
mouseWheelMoved, método de la interfaz MouseWheelListener 515
moveTo, método de la clase GeneralPath 587
MP3, reproductor 15
muestras de colores 565
multi-catch 449
multihilos 688, 959
multinúcleo 730
MULTIPLE_INTERVAL_SELECTION, constante de la interfaz ListSelectionModel 510, 513
multiplicación, * 51, 52
multiply
método de la clase
BigDecimal 347
BigInteger 782
MySQL 28

N

navegador
90
Web 90
negación lógica, ! 179
operador NOT (!) lógico, tabla de verdad 179
negate, método de la interfaz funcional Predicate (Java SE 8) 745
nemónico 480
Netbeans 18
video de demostración 35
new Scanner(System.in), expresión 47
new, palabra clave 47, 75, 246, 247
newDirectoryStream, método de la interfaz Files 648
newOutputStream, método de la clase Files 665, 668
next, método
de Iterator 691
de Scanner 75
nextDouble, método de la clase Scanner 88
nextInt, método de la clase Random 214, 217
nextLine, método de la clase Scanner 75
Nimbus, apariencia visual 476
swing.properties lv, 476
nivel de sangría 107
nombre(s)
de clase(s)
completamente calificado 79
nomenclatura lomo de camello 72
de métodos
nomenclatura de lomo de camello 72
de un arreglo 246
de una variable 50
de variables
nomenclatura de lomo de camello 72
del paquete 79
NORTH, constante de la clase BorderLayout 518, 532
NoSuchElementException, clase 654, 657
nota en UML 104
notación
algebraica 51
de diamante (<>) 290, 290
notify, método de Object 388
notifyAll, método de Object 388
nueva línea, carácter 42
null
palabra clave 76, 81, 91, 246, 478
palabra reservada 135
NullPointerException, excepción 261
NumberFormat, clase 346
format, metodo 347
getCurrencyInstance, método 347

número(s)
aleatorio(s)
desplazar un rango **214**
diferencia entre valores 218
elemento de azar **213**
escalamiento **214**
factor de escala **214**, 217, 218
generación 257, 637
no determinísticos **213**
valor de desplazamiento **214**, 217, 218
complejo(s)357
ejercicio 356
de identificación de empleado 8
de posición 245
de punto flotante **84**, 118, 122, 124, 710
división 125
doble precisión **84**
`double`, tipo primitivo **84**
`float`, tipo primitivo **84**
precisión simple **84**
no especificado de argumentos 281
perfecto, ejercicio 240
primo 306, 762
racionales, ejercicio 357
real 47, 122

O

O(1) **815**
O(log *n*) **820**
O(*n* log *n*), tiempo **833**
O(*n*), tiempo **815**
O(*n2*), tiempo **815**
`Object`, clase 338, 361, **365**, 670
`clone`, método 388
`equals`, método 387
`finalize`, método 388
`getClass`, método 388, **418**, **484**
`hashCode`, método 387
`notify`, método 388
`notifyAll`, método 388
`toString`, método 368, 388
`wait`, método 388
`ObjectInput`, interfaz **662**
`readObject`, método **663**
`ObjectInputStream`, clase **662**, 663, 668
`ObjectOutput`, interfaz **662**
`writeObject`, método **663**
`ObjectOutputStream`, clase **662**, 663, 720
`close`, método **667**
objeto(s) 2, 10
de una clase derivada 399
envoltura (colecciones) **721**
evento 492
`Icon` de sustitución **497**
inmutable **341**
`JTextArea` que no se puede editar 539
serializado **662**
`String` inmutable 599
observaciones
de apariencia visual, generalidades xxxviii
de ingeniería de software, generalidades xxxviii
obtener, método 324, 331
ocho reinas, ejercicio 305
fuerza bruta, métodos 305
ocultamiento
de datos **80**
de información **12**, 80
ocultar 73
detalles de implementación 202, 321
un campo **223**
un cuadro de diálogo 477
`of`, método de la interfaz `IntStream` (Java SE 8) **738**
`offer`, método de `PriorityQueue` **710**
`ONE`
constante de la clase
`BigDecimal` **347**
`BigInteger` **782**, 784
opciones mutuamente exclusivas **501**
`OPEN`, constante de la clase `Arc2D` **585**
operación(es)
de carga/almacenamiento 309
de entrada/salida 104, 309
de flujo
ansiosa (Java SE 8) 739
perezoso (Java SE 8) 741, 742
sin información de estado 742
de flujo intermedias (Java SE 8)
`filter`, método de la interfaz `IntStream` **741**
`filter`, método de la interfaz `Stream` 745, 748
`flatMap`, método de la interfaz `Stream` **760**
`map`, método de la interfaz `IntStream` **742**
`map`, método de la interfaz `Stream` 747
`sorted`, método de la interfaz `IntStream` **741**
`sorted`, método de la interfaz `Stream` **745**, 748
de reducción **735**
mutable **735**
en UML **78**
física
de entrada **675**
de salida **674**
intermedia(s) 741
con estado **742**
con información de estado **742**
sin estado **742**
sin información de estado **742**
lógicas
de entrada **675**
de salida **674**
masiva **687**
terminal(es) 738
ansiosa 741
en corto circuito (Java SE 8) 752
reducción **735**
terminales de flujo (Java SE 8) **734**, 745
`average`, método de la interfaz `IntStream` **739**
`collect`, método de la interfaz `Stream` **745**, 755, 756, 762
`count`, método de la interfaz `IntStream` **739**
en cortocircuito 752
`findFirst`, método de la interfaz `Stream` 752
`mapToDouble`, método de la interfaz `Stream` **756**
`max`, método de la interfaz `IntStream` **739**
`min`, método de la interfaz `IntStream` **739**
`reduce`, método de la interfaz `IntStream` **739**
`sum`, método de la interfaz `IntStream` **739**
operador(es), **48**
^, OR exclusivo lógico booleano 176, **179**
--, predecrementar/postdecrementar **131**
--, predecremento/postdecremento 132
!, NOT lógico 176, **179**
`?:`, condicional ternario 110, 134
*=, de asignación de multiplicación 131
/=, de asignación de división 131
&, AND lógico booleano 176, **178**
&&, AND condicional **176**, 177
%=, de asignación de residuo 131
++, preincremento/postincremento 132
++, preincrementar/postincrementar 131
+=, de asignación de suma **131**
= **48**, 57
-=, de asignación de resta 131
|, OR inclusivo lógico booleano 176, **178**
||, OR condicional 176, **177**
a nivel de bits 176
AND
condicional, **&&** **176**, 178
lógico booleano, **&** 176, **178**
aritméticos 51
binario(s) **48**, 51, 179
complemento lógico, ! **179**
condicional, `?:` **110**, 134
de asignación **131**
= **48**, 57
compuesto(s) **131**, 133
compuesto de división, /= 131
compuesto de multiplicación, *= 131
compuesto de residuo, %= 131
compuesto de resta, -= 131
de comparación 431
de conversión **125**
de decremento, -- **131**, 132
de exponenciación 162
de igualdad **54**
== para comparar objetos `String` 601
incremento, ++ 131
incremento y decremento 132
lógico(s) **176**, 179
de complemento, ! **179**
multiplicación, * **51**
multiplicativos: *, / y % **125**
negación lógica, ! **179**
operadores lógicos **176**, 179, 180
OR condicional, || 176, **177**, 178
OR exclusivo lógico booleano, ^ 176, **179**
OR inclusivo lógico booleano, | **178**
postdecremento **131**
postincremento **131**
predecremento **131**
preincremento **131**
relacionales **54**
residuo 150
% **51**, 52
resta, - 52
tabla de precedencia y asociatividad 134
ternario **110**
unario **125**, 179
conversión **125**
operando **48**, 125, 308
`Optional`, clase (Java SE 8) 752
`OptionalDouble`, clase (Java SE 8) **739**, 757
`getAsDouble`, método **739**, 757
`orElse`, método **739**, 757
OR condicional, || 176, **177**
tabla de verdad 178
OR inclusivo lógico booleano, | **178**
OR exclusivo lógico booleano, ^ 176, **179**
tabla de verdad 179
`or`, método de la interfaz funcional `Predicate` (Java SE 8) **745**
orden
ascendente 285
clasificado 712, 714
de bloques `catch`, ejercicio 472
de los manejadores de excepciones 472
descendente 285
en el que deben ejecutarse las acciones **102**
natural 748
ordenamiento
con un `Comparator` 698
de burbuja 836
mejorar el rendimiento 836
de cubeta 836
de datos **811**, 820

orden descendente 697
por combinación **827**
por inserción **824**
algoritmo **824**, 827
ordenar 103
letras y eliminar duplicados, ejercicio 775
`orElse`, método de la clase `OptionalDouble` (Java SE 8) **739**, 757
origen
de datos 731
del evento **490**, 492
`OutputStream`, clase 663, 673
`OutputStreamWriter`, clase **676**
óvalo 571, 575
delimitado por un rectángulo 575
relleno con colores que cambian en forma gradual 584

P

PaaS (plataforma como un sevicio) **28**
página Web 90
`Paint`, objeto **584**
`paintComponent`
método de la clase `JComponent` **522**, 556
método de `JComponent` **137**
palabra clave **37**, 105
palabra reservada **37**, 105
`false` 106
`null` 76, **81**, 135
`true` 106
palíndromo 150, 807
ejercicio 807
panel 538
de contenido 510
`setBackground`, método **511**
de vidrio 510
pantalla 4, 5, 135, 556
multitáctil **15**
paquete(s) **45**, 201, 211
básico 41
de entrada/salida 212
de red 212
de seguridad de Java **212**
del lenguaje 212
`java.time` **321**
`java.awt` **479**, 559, 581
`java.awt.color` 581
`java.awt.event` 212, **491**, 493, 518, 528
`java.awt.font` 581
`java.awt.geom` 212, 581
`java.awt.image` 581
`java.awt.image.renderable` 581
`java.awt.print` 581
`java.io` 212, **646**
`java.lang` **48**, 203, 212, 365, 387, 597
`java.math` 126, **346**, 781
`java.net` 212
`java.nio.file` 645, **646**, 647, 759
`java.security` 213
`java.sql` 212
`java.text` **346**
`java.time` 213
`java.uti.function` (Java SE 8) 732, **738**
`java.util` **46**, 212, 288
`java.util.concurrent` 212
`java.util.prefs` **718**
`java.util.regex` 597
`java.util.stream` (Java SE 8) **736**
`javax.swing` 212, **474**, 476, 484, 495 540, 563
`javax.swing.event` 212, **492**, 493, 510, 518
paquete predeterminado **79**
predeterminado **79**
par clave/valor 715
`parallelSort`
método de la clase `Arrays` (Java SE 8) 287
método de la clase `Arrays` (Java SE 8) 745
parámetro
de excepción **448**
de operación en UML 78
76
en UML 78
paréntesis **38**, 52
anidados **52**
innecesarios 53
para forzar el orden de evaluación 134
redundantes **49**, **53**
`parseInt`
método de la clase `Integer` 478
método de `Integer` 92, 188, 283
parte
imaginaria 356
real 356
pasar
el elemento de un arreglo a un método 264
opciones a un programa 283
un arreglo a un método 264
Pascal, lenguaje de programación 15
paseo
cerrado 306, 593
completo 593
del caballo 303, 593
ejercicio 593
método de fuerza bruta 304
prueba de paseo cerrado 306
paso
de partición en quicksort 837, 838
por referencia **265**
por valor 263, **265**
`Path`, interfaz **647**
`getFileName`, método **648**
`isAbsolute`, método **648**
`toAbsolutePath`, método **648**
`toString`, método **648**
`PATH`, variable de entorno liv, lv, 40
`Paths`, clase **647**
`get`, método **647**, 648
patrón 581
de diseño **28**
de tablero de damas 66
de unos y ceros 7
`Pattern`, clase 597, **631**
`compile`, método **631**
`matcher`, método **631**
`matches`, método **631**
`splitAsStream`, método (Java SE 8) **760**
`peek`
método de la clase
`PriorityQueue` **710**
`Stack` **710**
perezoso **734**
Perl (lenguaje práctico de extracción y generación de informes) 16
persistente 6
PHP 16, **28**
`PI` 592
`PIE`, constante de la clase `Arc2D` **585**
pila **209**
de llamadas a métodos **210**
desbordamiento de pila **210**
`PipedInputStream`, clase **673**
`PipedOutputStream`, clase **673**
`PipedReader`, clase **676**
`PipedWriter`, clase **676**
píxel (“elemento de imagen”) **135**, **556**
`PLAIN`, constante de la clase `Font` **567**
`PLAIN_MESSAGE` 478
Platform como un servicio (PaaS) **28**
PNG (gráficos portables de red) 484
`Point`, clase **523**
polígono **578**, 580
cerrados 578
polilínea **578**
polimorfismo 171, 391, **396**
polinomio 53, 54
de segundo grado 53, 54
política(s) de las barras de desplazamiento **542**
horizontal 542
`poll`, método de `PriorityQueue` **710**
`Polygon`, clase **556**, **578**
`addPoint`, método 579, **581**
POO (programación orientada a objetos) **13**, 361
`pop` método de `Stack` **710**
póquer 307
`PorPagar`
declaración de la interfaz 424
diagrama de clases de UML de la jerarquía de la interfaz 423
programa de prueba de la interfaz para procesar objetos `Factura` y `Empleado` mediante el polimorfismo 430
portabilidad 558
portable **19**
poscondición **465**
posiciones de los tabuladores 43
postdecremento **132**
operador de **131**
postincremento **132**, 133
operador de **131**, 157
potencia (exponente) 204, 240
de 2 mayor que 100 114
`pow`
método de la clase
`BigDecimal` **347**
`Math` **162**, 203, 235
precedencia **52**, 58, 134, 784
de operadores aritméticos 52, 784
reglas **52**
tabla de precedencia de operadores 125
tabla de 52, 125
precisión de un número de punto flotante con formato **88**, 126
precondición **465**
predecremento **132**
operador **131**
predicate 741
`Predicate`, interfaz funcional (Java SE 8) 733, 751
`and`, método **745**
`negate`, método **745**
`or`, método **745**
preferencias, API **718**
preincrementar **132**, 133
y postincrementar 132
preincremento, operador **131**
`previous`, método de `ListIterator` **694**
primer refinamiento 127
en el refinamiento de arriba-abajo, paso a paso 120
primo 240, 727
principio(s)
de construcción de programas 190
del menor privilegio 225, **343**
`print`
método de `System.out` 42
`printf`, método de `System.out` **43**
`println`, método de `System.out` 39, 42
`printStackTrace`, método de la clase `Throwable` **461**
`PrintStream`, clase **654**, 720
`PrintWriter`, clase **654**, 676
`PriorityQueue`, clase **710**
`clear`, método **710**
`offer`, método **710**
`peek`, método **710**
`poll`, método **710**
`size`, método **710**

`private`
  campo 330
  datos 331
  modificador de acceso **72**, 321, 364
  palabra clave 331
`private static`
  miembro de clase 339
probabilidad 214
problema
  del `else` suelto **109**, 149
  del promedio de una clase 115, 116, 121, 122
procedimiento para resolver un problema 102
procesador
  de cuádruple núcleo (quad-core) 6
  de doble núcleo 6
  de palabras 597, 605
  multinúcleo **6**
procesamiento
  de datos comerciales 680
  polimórfico
    de colecciones 688
    de excepciones relacionadas 454
    de objetos `Factura` y `Empleado` 430
proceso
  controlado por eventos 558
  de diseño **13**
  de refinamiento 120
producto de entero impar 196
programa 4
  administrador de pantallas 398
  de computadora **4**
  de conversión métrica 640
  general del promedio de una clase 119
  tolerante a fallas **256**
  traductor **9**
programación
  estructurada 4, **103**, 153, 176, 182
    resumen 182
  funcional **732**
  orientada a objetos (POO) **2**, 4, **13**, 361
programador **4**
programar
  en forma específica 396
  en forma general 396, 439
promedio 52, 115, 118
promoción(es) **125**
  de argumentos **210**
  para tipos primitivos 211
  reglas 125, **210**
`Properties`, clase **718**
  `getProperty`, método **718**
  `keySet`, método 721
  `list`, método **720**
  `load`, método **721**
  `setProperty`, método **718**
  `store`, método **720**
propiedad autosimilar **791**
proporción
  de números de Fibonacci sucesivos 783
  dorada **783**
`protected`, modificador de acceso 321, 364
protector de pantalla
  con figuras, ejercicio 593
  ejercicio 592
  mediante el uso de `Timer`, ejercicio 592
  mediante la API de Java2D, ejercicio 593
  para un número aleatorio de líneas, ejercicio 593
protocolo
  de control de transmisión (TCP) **25**
  de transferencia de hipertexto (HTTP) **26**
prueba de terminación 787
`public`
  `abstract`, método 421
  `final static`, datos 421
  interfaz **316**
  método 137, 317, 321
    encapsulado en un objeto 320
    miembro de una subclase 364
    modificador de acceso 71, 72, 206, 321, 364
    palabra clave **37**, 72
    servicio **316**
  `static`
    método 339
    miembros de una clase 339
publicaciones
  comerciales 30
  técnicas 30
punta de flecha (`->`) en una lamda **733**
punto(s) **567**
  (`.`), separador **76**, 91, 162, 203, 339, 581
  activos en el código de bytes 20
  de entrada 182
  de inserción 287, 705
  de lanzamiento **445**
  de salida 182
    de una instrucción de control 105
  juego de craps, su 218
  y coma (`;`) **39**, 47, 57
`push`, método de la clase `Stack` **709**
`put`, método de la interfaz `Map` **717**
Python 16

Q

quad-core, procesador de cuádruple núcleo 6
`QUESTION_MESSAGE` 478
`Queue`, interfaz **686**, **710**
quicksort, algoritmo 837
quintillas 637
  al azar 637

R

`Racional`, clase 357
`RadialGradientPaint`, clase 584
radianes 204
radio 592
  de un círculo 239
raíz
  base de un número **621**
  cuadrada 204
RAM (memoria de acceso aleatorio) **5**
`Random`, clase 300
  `nextInt`, método **214**, 217
`range`
  método de la clase `EnumSet` **337**
  método de la interfaz `IntStream` (Java SE 8) **743**
`rangeClosed`, método de la interfaz `IntStream` (Java SE 8) **743**
rastrear eventos de ratón 515
rastreo de pila **444**
ratón 4, 474
  con tres botones 521
  con uno, dos o tres botones 521
  con varios botones 521
Re Pág, tecla 525
`Reader`, clase **675**
`readObject`
  método de
    `ObjectInput` **663**
    `ObjectInputStream` 670
realización en UML **423**
realizar
  un cálculo 58
  una acción 39
  una tarea 73
reclamar la memoria 342
recolección
  automática de basura 454
  de basura **338**, 450, 454
recorrer un arreglo **273**
recorrido de laberinto mediante el uso de la "vuelta atrás" recursiva, ejercicio 808
`Rectangle2D`, clase 556
`Rectangle2D.Double`, clase **581**
rectángulo 356, 556, 560, 572
  con relieve **575**
  delimitador **188**, **573**, 575
  redondeado 573, 585
  tridimensional 572
`Rectangulo`, clase, ejercicio 356
recursividad
  algoritmo de búsqueda
    binaria recursiva 837
    lineal recursiva 837
  evaluación recursiva 780
    de 5! 780
  generar números de Fibonacci en forma recursiva 785
  indirecta **778**
  infinita 386, **781**, 787, 789
  llamada recursiva **778**, 784, 785
  método
    `factorial` recursivo 779
    `potencia` recursivo, ejercicio 805
    recursivo 777
  paso
    de recursividad **778**, 784
    recursivo 837
  quicksort 837
  sobrecarga 788
  vuelta atrás recursiva **802**
recursos para el instructor de *Cómo programar en Java, 9/e* xxxix
redirigir un flujo estándar **646**
redondear un número de punto flotante para mostrarlo en pantalla 126
redondeo de un número 51, 118, 163, 204, 238
reducción
  mutable **745**
    Java SE 8 **745**
`reduce`
  método de la interfaz
    `DoubleStream` (Java SE 8) 757
    `IntStream` (Java SE 8) **739**
refactorización **28**
referencia(s) **81**
  a constructor (Java SE 8) 760
  a método(s) 734, **747**
    Java SE 8 747
refinamiento de arriba-abajo, paso a paso **120**, 121, 122, 127, 128
`regionMatches`, método de la clase `String` **601**
registrar el manejador de eventos **488**
registro 7, 651
  acumulador 308, 311
  de activación **210**
  de eventos 489
  de transacciones 682
regla(s)
  de anidamiento **185**
  de apilamiento **184**
  de precedencia de operadores **52**, 784
  empírica (heurística) 176
  para formar programas estructurados 182
reinventar la rueda 11, 45, 285
relación
  entre una clase interna y su clase de nivel superior 501
  jerárquica entre el método jefe y los métodos trabajadores 202
relleno
  patrón de 585
  textura 585
  rectángulo 560
    tridimensional 572
`remove`
  método
    de la clase `ArrayList<T>` 288, **290**
    de la interfaz `Iterator` **691**

repaint
método de la clase Component 524
método de la clase JComponent 559
repartir cartas 257
repetición 105, 186
controlada
por centinela 119, 121, 122, 134, 197, 310
por contador 115, 116, 124, 127, 128, 153, 155, 310, 787
definida 115
indefinida 119
replaceAll, método
de la clase
Matcher 631
String 629
replaceFirst, método
de la clase
Matcher 631
String 629
representación de un laberinto mediante un arreglo bidimensional 272, 273
requerimientos 13
de una aplicación 171
residuo 51
operador, % 51, 52, 150
resolución 135, 556
respuestas a una encuesta 254
resta 6, 51
operador, - 52
resuelve el problema de las Torres de Hanoi con un método recursivo 790
resultados de exámenes, problema 128
retorno de carro 43
retroalimentación visual 498
return
instrucción 778
palabra clave 74, 202, 209
reutilización de software 11, 202
reutilizar 11, 45
reverse
método de la clase
Collections 696, 702
StringBuilder 614
reversed, método de la interfaz Comparator (Java SE 8) 748
reverseOrder, método de Collections 698
RGB, valor(es) 227, 559, 560, 565
RIGHT, constante de la clase FlowLayout 532
Ritchie, Dennis 16
robusto 48
rombo en UML 104, 198
rotate, método de la clase Graphics2D 588
RoundingMode, enumeración 348
RoundRectangle2D, clase 556
RoundRectangle2D.Double, clase 581, 585
Ruby on Rails 17
Ruby, lenguaje de programación 17
rueda del ratón 515
Runnable, interfaz 432
RuntimeException, clase 452
ruta
absoluta 647, 649
general 586
relativa 647

S

SaaS (Software como un servicio) 28
sacar de una pila 209
Safari 90
salida(s) 39
con formato
%f especificador de formato 88
- (signo negativo) bandera de formato 161
anchura de campo 161
0, bandera 253, 318
coma (,), bandera de formato 162
justificar
a la derecha 161
a la izquierda 161
números de punto flotante 88
precisión 88
separador de agrupamiento 162
signo negativo (-), bandera de formato 161
valores boolean 180
justificada a la derecha 161
salir de una instrucción for 174
SAM, interfaz 732
sangría 107, 109
SansSerif, tipo de letra de Java 567
saturación 565
Scanner, clase 46, 47
hasNext, método 168
next, método 75
nextDouble, método 88
nextLine, método 75
SDK (Kit de desarrollo de software) 29
sección
"administrativa" de la computadora 6
de "almacén" de la computadora 6
de "embarque" de la computadora 5
de "manufactura" de la computadora 6
especial
Construya su propia computadora 308
Ejercicios avanzados de manipulación de cadenas 638
Proyectos retadores de manipulación de cadenas 641
"receptora" de la computadora 5
sector 576
secuencia 105, 184, 186, 688
de escape 43, 47, 651
\, barra diagonal inversa 43
\", comilla doble 43
\t, tabulación horizontal 43
de nueva línea, \n 43, 313
SecureRandom, clase 213, 214
documentación 214
doubles, método (Java SE 8) 762
flujos de números aleatorios (Java SE 8) 762
ints, método (Java SE 8) 762
longs, método (Java SE 8) 762
SecurityException, clase 653
segundo refinamiento 128
de arriba a abajo, paso a paso 120
seguridad 20
de tipos en tiempo de compilación 690
selección 105, 185, 186
doble 186
simple
instrucción 105, 106, 186
lista 508
seleccionar
figuras, ejercicio 593
un elemento de un menú 485
"seleccionar" cada dígito 67
seno 204
trigonométrico 204
sensible a mayúsculas y minúsculas 38
comandos de Java 22
separador de agrupamiento (salida con formato) 162
SequenceInputStream, clase 675
Serializable, interfaz 432, 663
serialización de objetos 662
serie infinita 197
Serif, tipo de letra de Java 567
servicio(s)
de una clase 321
Web 26
Amazon eCommerce 27
eBay 27
Facebook 27
Flickr 27
Foursquare 27
Google Maps 26
Groupon 27
Instagram 27
Last.fm 27
LinkedIn 27
Microsoft Bing 27
Netflix 27
PayPal 27
Salesforce.com 27
Skype 27
Twitter 26
WeatherBug 27
Wikipedia 27
Yahoo Search 27
YouTube 27
Zillow 27
Set, interfaz 686, 711, 712, 714
stream, método (Java SE 8) 760
set, método de la interfaz ListIterator 694
setAlignment, método de la clase FlowLayout 532
setBackground, método de la clase Component 291, 511, 565
setBounds, método de la clase Component 529
setCharAt, método de la clase StringBuilder 614
setColor
método
de Graphics 228
de la clase Graphics 560, 585
setDefaultCloseOperation, método de la clase JFrame 138, 485
setDisabledTextColor, método de la clase JTextComponent 528
setEditable, método de la clase JTextComponent 488
setErr, método de la clase System 646
setFileSelectionMode, método de la clase JFileChooser 670
setFixedCellHeight, método de la clase JList 513
setFixedCellWidth, método de la clase JList 513
setFont
método de la clase
Component 500
Graphics 567
setHorizontalAlignment, método de la clase JLabel 484
setHorizontalScrollBarPolicy, método de la clase JScrollPane 542
setHorizontalTextPosition, método de la clase JLabel 484
setIcon, método de la clase JLabel 484
setIn, método de la clase System 646
setLayout, método de la clase Container 483, 530, 534, 537
setLength, método de la clase StringBuilder 613
setLineWrap, método de la clase JTextArea 542
setListData, método de la clase JList 513
setLocation, método de la clase Component 529
setMaximumRowCount, método de la clase JComboBox 507
setOpaque, método de la clase JComponent 522, 524
setOut, método de System 646
setPaint, método de la clase Graphics2D 584
setProperty, método de Properties 718
setRolloverIcon, método de la clase AbstractButton 498

setScale, método de la clase BigDecimal 348
setSelectionMode, método de la clase JList **510**
setSize
  método de la clase
    Component 529
    JFrame **138**, **485**
setStroke, método de la clase Graphics2D **584**
setText
  JTextComponent 541
  método de la clase
    JLabel 390, **484**
setToolTipText, método de la clase JComponent **483**
setVerticalAlignment, método de la clase JLabel **484**
setVerticalScrollBarPolicy, método de la clase JScrollPane **542**
setVerticalTextPosition, método de la clase JLabel **484**
setVisible
  método de la clase
    Component **485**, **534**
    JFrame **138**
setVisibleRowCount, método de la clase JList **510**
seudocódigo **103**, 107, 116, 126, 128
  algoritmo 121
  primer refinamiento **120**, 127
  segundo refinamiento **120**, 128
shell **39**
  en Linux 18
  secuencia de comandos **653**
Short, clase **687**
short, tipo primitivo 165
  promociones 211
showDialog, método de la clase JColorChooser **564**
showInputDialog, método de la clase JOptionPane **92**, **477**
showMessageDialog, método de la clase JOptionPane **91**, **478**
showOpenDialog, método de la clase JFileChooser **670**
shuffle, método de la clase Collections 696, **700**, 702
signo(s)
  « y » **84**
  de dólares ($) 37, 47
  negativo (–), bandera de formato **161**
símbolo
  de estado de acción **104**
  de fusión en UML **114**
  especial 6
símbolo del sistema **18**, 39
Simpletron
  lenguaje máquina (SML) 308
  simulador 310, 313
simulación 213
  de software 308
  la tortuga y la liebre 306, 593
  lanzar monedas 241
simulador 308
  de computadora 310
simular clic
  con el botón central del ratón en un ratón con uno o dos botones 521
  con el botón derecho del ratón en un ratón con un solo botón 521
sin, método de la clase Math 204
sincronizar el acceso a una colección 688
SINGLE_INTERVAL_SELECTION, constante de la interfaz ListSelectionModel **510**, 511, 513
SINGLE_SELECTION, constante de la interfaz ListSelectionModel **510**
sintetizar las respuestas a una encuesta 254
sistema(s)
  de composición 597
  de coordenadas **135**, **556**, 558
  de reservaciones 301
  de reservaciones de una aerolínea 301
  de ventanas **480**
  incrustado 14
  numérico(s) 621
    octal (base 8) 241
  operativo **13**, 4
size, método
  de la clase
    ArrayList<T> **290**
    Files **648**
    PriorityQueue **710**
  de la interfaz
    List **690**, **694**
    Map **718**
SMS, lenguaje 643
sobrecarga de métodos **225**
sobrecargar un método **225**
sobrescribir el método de una superclase **364**, 368
software 2, **6**
  como un servicio (SaaS) **28**
  de diseño de páginas 597
  frágil **381**
  quebradizo **381**
solución iterativa del factorial 788
sort, método
  de la clase
    Arrays **285**, 745, **817**
    Collections **697**
sorted
  método de la interfaz
    IntStream (Java SE 8) **741**
    Stream (Java SE 8) **745**, 748
SortedMap, interfaz **714**
SortedSet, interfaz **712**, 713
  first, método **713**
  last, método **713**
SourceForge 14
SOUTH, constante de la clase BorderLayout **518**, 532
split, método de la clase String **623**, **629**
splitAsStream, método de la clase Pattern (Java SE 8) **760**
sqrt, método de la clase Math 203, 204, 210
Stack, pila 710
  de package java.util **708**
  isEmpty, método **710**
  peek, método **710**
  pop, método **710**
  push, método **709**
StackTraceElement, clase **461**
  getClassName, método **461**
  getFileName, método **461**
  getLineNumber, método **461**
  getMethodName, método **461**
startsWith, método de la clase String **604**
static
  campo (variable de clase) **338**
  import **342**
    sobre demanda **342**
  import individual **342**
  método 70, **91**, 162
    de una interfaz (Java SE 8) 731
    en una interfaz (Java SE 8) 732, 763
  miembro de clase 338, 339
  palabra clave 203
  variable de clase 339
store, método de Properties **720**
stream
  método de la clase Arrays (Java SE 8) **743**, 744
  método de la interfaz Set 760
Stream, interfaz (Java SE 8) **734**, **744**
  collect, método **745**, 755, 756, 762
  distinct, método **754**
  filter, método 745, 748
  findFirst, método 752
  forEach, método 745
  map, método **747**, 747
  sorted, método **745**, 748
Stream, interfaz (Java SE 8)
  flatMap, método **760**
String, clase 597
  charAt, método **599**, **614**
  compareTo, método 601, 603
  concat, método **608**
  endsWith, método **604**
  equals, método 601, 603
  equalsIgnoreCase, método 601, 603
  format, método **92**, **318**
  getChars, método **599**
  inmutable **341**
  indexOf, método **605**
  lastIndexOf, método **605**
  length, método **599**
  matches, método **624**
  regionMatches, método **601**
  replaceAll, método **629**
  replaceFirst, método **629**
  split, método **623**, **629**
  startsWith, método **604**
  substring, método 607
  toCharArray, método **610**, 807
  toLowerCase **694**
  toLowerCase, método **610**
  toUpperCase **694**
  toUpperCase, método **609**
  trim, método **610**
  valueOf, método **610**
String de moneda específico de una configuración regional 347
String, métodos de búsqueda de la clase 605
StringBuffer, clase **612**
StringBuilder, clase 597, **611**
  append, método **615**
  capacity, método **612**
  charAt, método **614**
  constructores 612
  delete, método **617**
  deleteCharAt, método **617**
  ensureCapacity, método **613**
  getChars, método **614**
  insert, método **617**
  length, método **612**
  reverse, método **614**
  setCharAt, método **614**
  setLength, método **613**
StringIndexOutOfBoundsException, clase **607**
StringReader, clase **676**
StringWriter, clase **676**
Stroke, objeto **584**, 585
Stroustrup, Bjarne 17, 442
subclase **12**, **137**, **361**
  concreta 407
  personalizada de la clase JPanel 522
subíndice (índice) **245**
subList, método de List **694**
sublista **694**
substring, método de la clase String 607
subtract, método de la clase BigInteger **782**, 784
sueldo bruto 147
sufijo F para literales float **710**
sufijo L para literales long **709**
sum
  método de la interfaz DoubleStream (Java SE 8) 757
  método de la interfaz IntStream (Java SE 8) **739**

suma 6, 51, 52
sumar los elementos de un arreglo 251
super, palabra clave 364, 387
  llamar al constructor de la superclase 378
super.paintComponent(g) 137
superclase **12**, **137**, **361**
  abstracta **401**
  constructor 368
    predeterminado 368
  directa **361**, 363
  indirecta **361**, 363
  método sobrescrito en una subclase 386
  sintaxis de llamada al constructor **378**
Supplier, interfaz funcional (Java SE 8) 733
Swing
  API de GUI 475
  componentes de GUI **474**
  paquete de componentes de GUI 212
Swing Event, Paquete 212
swing.properties, archivo lv, 476
SwingConstants, interfaz **484**
switch, instrucción de selección múltiple 105, **165**, 169, 186, 217
  case, etiqueta 168, 169
  comparación de objetos String 171
  default, caso **168**, 170, 217
  diagrama de actividad con instrucciones break 170
  expresión de control **168**
switch, lógica 171
synchronized
  palabra clave 721
System, clase
  arraycopy **285**, 286
  currentTimeMillis, método 809
  exit, método **455**, **653**
  setErr, método **646**
  setIn, método **646**
  setOut **646**
System.err (flujo de error estándar) **448**, 646, 674
System.in (flujo de entrada estándar) 646
System.out
  flujo de salida estándar **39**, 646, 674
  print, método **42**
  printf, método **43**
  println, método 39, **39**, 42
SystemColor, clase **584**

T

Tab, tecla 38
tabla(s) 272
  de asociatividad 134
  de valores 272
  de verdad **177**
    para el operador ^ 179
    para el operador ! 179
    para el operador && 177
    para el operador || 177
  hash 711, 715
tabulación horizontal 43
tailSet, método de la clase TreeSet **713**
tamaño de una variable **50**
tan, método de la clase Math 204
tangente 204
  trigonométrica 204
tareas de preparación para la terminación **338**, 388
tasa de interés 160
TCP (protocolo de control de transmisión) **25**
TCP/IP **26**
tecla de acción **525**
  de flecha 525
  modificadora 528
teclado 4, 45, 474
teléfono inteligente (smartphone) 3
temporal 125
TEN, constante de la clase BigDecimal **347**
teoría de la complejidad 785
terabyte 6
termina la repetición 114
Terminal, aplicación (OS X) 18
terminar
  con éxito 653
  un ciclo 121
test, método de la interfaz funcional IntPredicate (Java SE 8) **741**, **742**
TextEdit 18
texto
  análisis 638
  archivo **646**
  campo **92**
  de sólo lectura 481
  editor 39, 597
  fijo 49
    en una cadena de formato **44**
    o iconos que no se pueden editar 479
  seleccionado en una JTextArea 541
TexturePaint, clase **556**, 584, 585
thenComparing, método de la interfaz funcional Comparator (Java SE 8) **753**
this
  palabra clave **73**, **322**, 339
  para llamar a otro constructor de la misma clase 327
  referencia **322**
throw
  instrucción **457**
  lanzar una excepción mediante **318**, 328
  palabra clave **458**
Throwable, clase **451**, 461
  getMessage, método **461**
  getStackTrace, método **461**
  jerarquía 452
  printStackTrace, método **461**
throws
  cláusula **450**
tiempo(s)
  de ejecución
    constante **815**
    cuadrático 815
    del peor caso para un algoritmo 814
    lineal **815**
    logarítmico **820**
  para calcular números de Fibonacci, ejercicio 809
tiene un, relación **332**, **362**
Timer, clase 592
tipo **46**
  de letra
    estilo 499, **567**
    información 556
    manipulación 558
    nombre **567**
    tamaño **567**
  de una expresión lambda 733
    de una variable **50**
  de valor de retorno
    de un método **73**
  destino de una lambda (Java SE 8) **738**
  por referencia **80**, 344
  primitivo **47**, 80, 134, 210
    byte 165
    char 47, 165
    double **47**, **84**, 122
    float **47**, **84**
    int 47, 122, 131, 165
    los nombres son palabras clave 47
    pasado por valor 266
    promociones 211
    short 165
tips
  de portabilidad, generalidades xxxviii
  de rendimiento, generalidades xxxviii
  para prevenir errores, generalidades xxviii
tirar dos dados 220
tiro de dados 300
toAbsolutePath, método de la interfaz Path **648**
toArray, método de la clase List **695**, 696
toCharArray, método de la clase String **610**
toCharArray, método de la clase String **686**
toCharArray, método de String 807
ToDoubleFunction, interfaz funcional (Java SE 8) **757**
  applyAsDouble, método **757**
token de un objeto String **623**
tolerante a fallas 48, 442
toList, método de la clase Collectors (Java SE 8) **745**
toLowerCase
  método de la clase
    Character **621**
    String **610**, **694**
tomar decisiones 58
toPath, método de la clase File **671**
torres de Hanoi **789**
  para el caso con cuatro discos 789
tortuga y liebre 306, 593
  ejercicio 593
toString
  método de un objeto 208
  método de la interfaz Path **648**
toString, método
  de la clase
    ArrayList 697
    Arrays 631, 814
    Object 368, 388
total **115**, 120
  actual 120
toUpperCase
  método de la clase
    Character **620**
    String **609**, **694**
traducción **9**
transferencia de control **103**, 310, 311, 312
transición en UML **104**
transient, palabra clave **665**
translate método de la clase Graphics2D **588**
transparencia de un objeto JComponent **522**
TreeMap, clase **714**, 760
TreeSet, clase **711**, 712, 713
  headSet, método **712**
  tailSet, método **713**
TresEnRaya 358
  ejercicio 358
triángulos
  aleatorios, ejercicio 592
  generados al azar 592
trim, método de la clase String **610**
trimToSize, método de la clase ArrayList<T> 288
triples de Pitágoras 197
true **54**
  palabra reservada 106, **110**
truncados **651**
truncar 51
  parte fraccionaria de un cálculo **118**
try
  bloque **256**, 447, 459
    termina 449
  con recursos (try-with-resources), instrucción **467**
  instrucción **256**, **450**
  palabra clave **447**, 1146
tutor de mecanografía 554
TYPE_INT_RGB, constante de la clase BufferedImage **585**

U

ubicación de una variable en la memoria de la computadora 50
UEPS (último en entrar, primero en salir) **210**
último en entrar, primero en salir (UEPS) **210**
UML 105
UML (Lenguaje Unificado de Modelado) 13
  círculo relleno **104**
    rodeado por un círculo sin relleno 104
  compartimiento en un diagrama de clases 77
  condición de guardia **106**
  diagrama
    de actividad **104**, 107, 114, 158, 164
    de clases 77
  estado final **104**
  flecha 104
  línea punteada **105**
  nota **104**
  rombo 106
  símbolo de fusión **114**
una sola entrada/una sola salida, instrucciones de control **105**, 182
una sola línea (fin de línea), comentario 39
`UnaryOperator`, interfaz funcional (Java SE 8) 733
único método abstracto (SAM), interfaz 732
único punto de entrada 182
único punto de salida 182
Unicode
  conjunto de caracteres 7, 66, 135, 170, **597**, 602, 620
  valor del carácter escrito 528
unidad
  aritmética y lógica (ALU) **6**
  central de procesamiento (CPU) **6**
  de almacenamiento secundario **6**
  de entrada **5**
  de procesamiento 4
  de salida **5**
  flash 645
  lógica **5**
unión
  de dos conjuntos 357
  de línea **584**
  teórica de conjuntos 357
UNIX 39, 168, 653
`UnsupportedOperationException`, clase **694**
URI (identificador uniforme de recursos) **648**
URL (localizador uniforme de recursos) **648**
usar búsqueda binaria para localizar un elemento en un arreglo 817
utilerías, Paquete 212

V

va **654**
vaciado de computadora 311
validar datos **119**
`validate`, método de la clase `Container` **537**
valor(es)
  absoluto 204
  centinela **119**, 121, 124
  de bandera **119**
  de identidad en una reducción (Java SE 8) **740**
  de prueba **119**
  de señal **119**
  de una variable **50**
  final 154
  inicial
    de la variable de control **153**
    predeterminado **76**
  predeterminado **76**, 135
`valueChanged`, método de la interfaz `ListSelectionListener` **510**
`valueOf`
  método de la clase
    `BigDecimal` **347**
    `String` **610**
`values`, método de una `enum` **336**
Van Rossum, Guido 16
variable **45**, **46**, 47
  constante **170**, 250
    debe inicializarse 250
  de clase **204**, **338**
  de control **115**, **153**, 154, 155
  de entorno
    `CLASSPATH` 41
    `PATH` 40
    `Vector`, clase **688**
  local(es) **73**, 117, 223
    efectivamente `final` (Java SE 8) **738**
  no se puede modificar, la 343
  nombre **46**, **50**
  tipo **50**
  tipo por referencia **80**
  tamaño **50**
  valor **50**
ventana
  de comandos 21, **39**
  de terminal **39**
  padre 91, 478
ventas totales 301
Ver 475
verificación de validez 331
verificador de códigos de bytes **20**
Verificar con `assert` que un valor se encuentre dentro del rango 466
versión final 29
`VERTICAL_SCROLLBAR_ALWAYS`, constante de la clase `JScrollPane` **542**
`VERTICAL_SCROLLBAR_AS_NEEDED`, constante de la clase `JScrollPane` **542**
`VERTICAL_SCROLLBAR_NEVER`, constante de la clase `JScrollPane` **542**
`vi`, editor 18
videojuego 214
vinculación
  dinámica 401, **417**
  estática **420**
  postergada **417**
Visual Basic, lenguaje de programación 16
Visual C#, lenguaje de programación 16
Visual C++, lenguaje de programación 16
visualización de la recursividad, ejercicio 806
`void`, palabra clave **38**, 73
volumen de una esfera 235, 237
volver a lanzar
  excepciones, ejercicio 472
  una excepción **458**, 472
vuelta atrás recursiva (*backtracking*) 809

W

W3C (Consorcio World Wide Web) **26**
`wait`, método de la clase `Object` 388
`WEST`, constante de la clase `BorderLayout` **518**, 532
`while`, instrucción de repetición 105, **114**, 117, 122, 124, 153, 186
  diagrama de actividad en UML 114
widgets 474
`WindowAdapter`, clase 519
`WindowListener`, interfaz 518, 519
Windows 13, 168, 653
  sistema operativo **13**
World Wide Web (WWW) **26**
  navegador 90
`writeBoolean`, método de la interfaz `DataOutput` 674
`writeByte`,
  método de la interfaz `DataOutput` 674
`writeChar`, método de la interfaz `DataOutput` 674
`writeChars`, método de la interfaz `DataOutput` 674
`writeDouble`, método de la interfaz `DataOutput` 674
`writeFloat`, método de la interfaz `DataOutput` 674
`writeInt`, método de la interfaz `DataOutput` 674
`writeLong`, método de la interfaz `674` 752
`writeObject`, método
  de la clase `ObjectOutputStream` 667
  de la interfaz `ObjectOutput` **663**
`Writer`, clase **675**
`writeShort`, método de la interfaz `DataOutput` 674
`writeUTF`, método de la interfaz `DataOutput` 674
www 28

Z

`ZERO`
  constante de la clase
    `BigDecimal` **347**
    `BigInteger` **784**
Zuckerberg, Mark 28
Zynga 5

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

En esta edición nos enfocamos en las mejores prácticas de ingeniería de software, tomando como base nuestro reconocido método de "código activo", donde los conceptos se presentan en el contexto de programas funcionales, completos, que se ejecutan en las versiones recientes de Windows®, OSX® y Linux®, en lugar de hacerlo a través de fragmentos de código.

Este libro ofrece una introducción clara, simple, atractiva y entretenida para un curso sobre programación en Java; y entre las características novedosas que se presentan sobresalen las siguientes:

- Amplia cobertura de los fundamentos, que incluye dos capítulos sobre instrucciones de control.
- Enfoque en ejemplos reales.
- Ejemplo práctico opcional sobre interfaces gráficas de usuario (GUI) y gráficos.
- Introducción a las clases y objetos desde los primeros capítulos del libro.
- Instrucción *try* con recursos y la interfaz *AutoClosable de Java SE7.*
- Secciones modulares opcionales sobre el lenguaje y las características de biblioteca de Java SE 7, Java SE 8 y una mezcla de ambos.
- Introducción opcional en línea al desarrollo de aplicaciones Android basado en Java.
- Archivos, flujos y serialización de objetos.
- Capítulos en línea *opcionales*, varios de ellos en español y otros más en inglés, para cursos avanzados y a nivel profesional.
- Tratamiento de otros temas, como recursividad, búsqueda, ordenamiento, colecciones genéricas, estructuras de datos, applets, multimedia, bases de datos/JDBC™, desarrollo de aplicaciones Web y servicios Web.
- Caso de estudio opcional sobre Diseño orientado a objetos, y una implementación en Java de un cajero automático ATM.

Cada parte de código está acompañado con ejemplos de ejecuciones actuales. Todo el código fuente en inglés está disponible en **www.deitel.com/books/jhtp10/**. El código en español puede descargarlo desde el sitio web de este libro en:

www.pearsonenespañol.com/deitel

www.pearsonenespañol.com

ISBN 978-607-32-3802-1

90000

9 786073 238021