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

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.

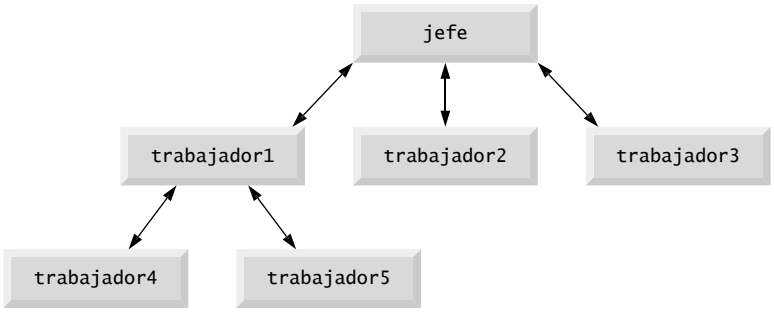

Fig. 6.1 | Relación jerárquica entre el método jefe y los métodos trabajadores.

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
```

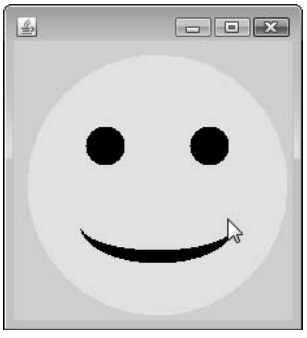

Fig. 6.12 | Aplicación de prueba que muestra una cara sonriente.

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.

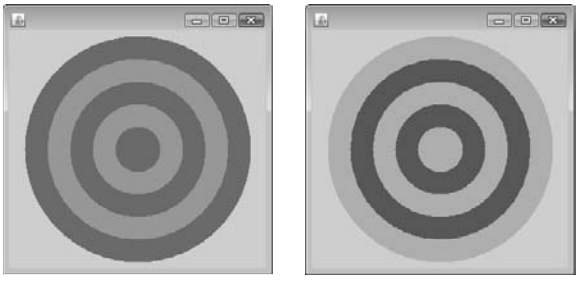

Fig. 6.13 | Un tiro al blanco con dos colores alternantes al azar.

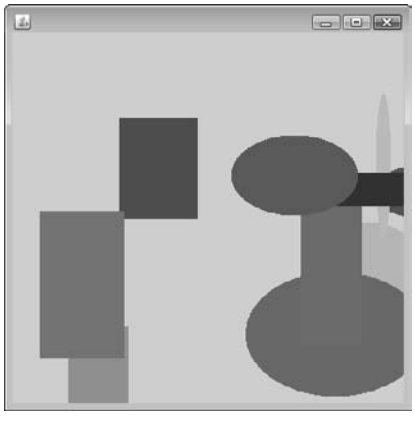

Fig. 6.14 | Figuras generadas al azar.

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.