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

3

Sus servidores públicos le sirven bien.

—Adlai E. Stevenson

Nada puede tener valor sin ser un objeto de utilidad.

—Karl Marx

Objetivos

En este capítulo aprenderá a:

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

Plan general

3.1 Introducción

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

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

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

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

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

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

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

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

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

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

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

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

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

```
public class Cuenta
```

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

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

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

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

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

La variable de instancia `nombre`

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

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

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

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

Buena práctica de programación 3.1

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

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

La mayoría de las declaraciones de variables de instancia van precedidas por la palabra clave `private` (como en la línea 7). Al igual que `public`, la palabra clave `private` es un *modificador de acceso*. Las variables o los métodos declarados con el modificador de acceso `private` son accesibles sólo para los métodos de la clase en la que se declaran. Por lo tanto, la variable `nombre` sólo puede utilizarse en los métodos de cada objeto `Cuenta` (`establecerNombre` y `obtenerNombre` en este caso). Pronto verá que esto presenta poderosas oportunidades de ingeniería de software.

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

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

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

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

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

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

Los parámetros son variables locales

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

El cuerpo del método `establecerNombre`

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

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

Buena práctica de programación 3.2

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

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

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

El método `obtenerNombre` (líneas 16 a 19)

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

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

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

La instrucción `return` en la línea 18 de la figura 3.1 regresa el valor `String` de la variable de instancia `nombre` a quien hizo la llamada. Por ejemplo, cuando el valor se devuelve a la instrucción en las líneas 25 y 26 de la figura 3.2, la instrucción usa ese valor para mostrar el nombre en pantalla.

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

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

La clase controladora `PruebaCuenta`

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

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

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

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

```
El nombre inicial es: null

Introduzca el nombre:
Alfonso Romero

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

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

Objeto `Scanner` *para recibir entrada del usuario*

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

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

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

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

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

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

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

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

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

Tip para prevenir errores 3.1

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

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

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

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

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

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

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

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

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

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

3.2.3 Compilación y ejecución de una aplicación con varias clases

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

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

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

```
javac *.java
```

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

```
java PruebaCuenta
```

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

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

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

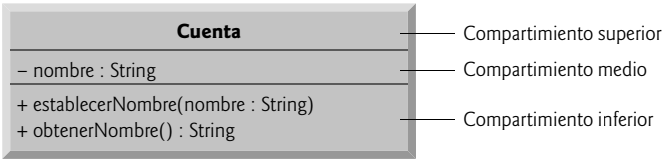

Fig. 3.3 | Diagrama de clases de UML para la clase `Cuenta` de la figura 3.1.

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

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

Tipos de valores de retorno
UML indica el *tipo de valor de retorno* de una operación colocando dos puntos y el tipo de valor de retorno *después* de los paréntesis que le siguen al nombre de la operación. El método `obtenerNombre` de la clase `Cuenta` (figura 3.1) tiene un tipo de valor de retorno `String`. El método `establecerNombre` *no* devuelve un valor (porque devuelve `void` en Java), por lo que el diagrama de clases de UML *no* especifica un tipo de valor de retorno después de los paréntesis de esta operación.

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

3.2.5 Observaciones adicionales sobre la clase `PruebaCuenta`

El método `static main`
En el capítulo 2, cada clase que declaramos tenía un método llamado `main`. Recuerde que `main` es un método especial que *siempre* es llamado automáticamente por la Máquina Virtual de Java (JVM) a la hora de ejecutar una aplicación. Es necesario llamar a la mayoría de los otros métodos para decirles de manera *explícita* que realicen sus tareas.

Las líneas 7 a la 27 de la figura 3.2 declaran el método `main`. Una parte clave para permitir que la JVM localice y llame al método `main` para empezar la ejecución de la aplicación es la palabra clave `static` (línea 7),

la cual indica que `main` es un método `static`. Un método `static` es especial, ya que puede llamarse *sin tener que crear primero un objeto de la clase en la cual se declara ese método*; en este caso, la clase `PruebaCuenta`. En el capítulo 6 analizaremos los métodos `static` con detalle.

Observaciones sobre las declaraciones `import`

Observe la declaración `import` en la figura 3.2 (línea 3), la cual indica al compilador que el programa utiliza la clase `Scanner`. Como vimos en el capítulo 2, las clases `System` y `String` están en el paquete `java.lang`, que se importa de manera implícita en *todo* programa de Java, por lo que todos los programas pueden usar las clases de ese paquete *sin tener que* importarlas de manera explícita. *La mayoría* de las otras clases que utilizará en los programas de Java *deben* importarse de manera *explícita*.

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

La declaración `import` en la línea 3 *no* es obligatoria si hacemos referencia a la clase `Scanner` en este archivo como `java.util.Scanner`, que incluye el *nombre completo del paquete y de la clase*. Esto se conoce como el **nombre de clase completamente calificado**. Por ejemplo, la línea 10 de la figura 3.2 también podría escribirse como

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

Observación de ingeniería de software 3.1

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

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

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

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

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

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

El proceso de declarar variables de instancia con el modificador de acceso `private` se conoce como *ocultamiento de datos*, u *ocultamiento de información*. Cuando un programa crea (instancia) un objeto de la clase `Cuenta`, la variable `nombre` se *encapsula* (oculta) en el objeto, y sólo está accesible para los métodos de la clase de ese objeto.

Observación de ingeniería de software 3.2

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

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

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

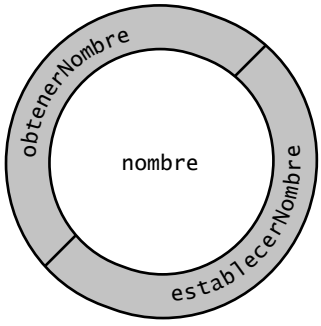

Fig. 3.4 | Vista conceptual de un objeto `Cuenta` con su variable de instancia `private` llamada `nombre` encapsulada y con la capa protectora de métodos `public`.

3.3 Comparación entre tipos primitivos y tipos por referencia

Los tipos en Java se dividen en tipos primitivos y **tipos por referencia**. En el capítulo 2 trabajó con variables de tipo `int`: uno de los tipos primitivos. Los otros tipos primitivos son `boolean`, `byte`, `char`, `short`, `long`, `float` y `double`, cada uno de los cuales veremos en este libro. En el apéndice D hay un resumen de ellos. Todos los tipos no primitivos son *tipos por referencia*, por lo que las clases, que especifican los tipos de objetos, son tipos por referencia.

Una variable de tipo primitivo puede almacenar sólo *un* valor de su tipo declarado a la vez. Por ejemplo, una variable `int` puede almacenar un entero a la vez. Cuando se asigna otro valor a esa variable, el nuevo valor sustituye su valor inicial, el cual se *pierde*.

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

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

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

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

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

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

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

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

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

3.4 La clase Cuenta: inicialización de objetos mediante constructores

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

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

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

Cuando usted declara una clase, puede proporcionar su propio constructor para especificar una *inicialización personalizada* para los objetos de su clase. Por ejemplo, tal vez quiera especificar el nombre para un objeto `Cuenta` al momento de crear este objeto, como en la línea 10 de la figura 3.6:

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

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

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

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

Declaración del constructor de `Cuenta`

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

Tip para prevenir errores 3.2

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

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

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

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

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

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

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

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

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

Los constructores no pueden devolver valores

Una importante diferencia entre los constructores y los métodos es que *los constructores no pueden devolver valores*, por lo cual *no pueden* especificar un tipo de valor de retorno (ni siquiera void). Por lo general, los constructores se declaran como public; más adelante en el libro explicaremos cuándo usar constructores private.

Constructor predeterminado

Recuerde que la línea 13 de la figura 3.2

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

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

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

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

Agregar el constructor al diagrama de clases de UML de la clase `Cuenta`
El diagrama de clases de UML de la figura 3.7 modela la clase `Cuenta` de la figura 3.5, la cual tiene un constructor con un parámetro llamado `nombre`, de tipo `String`. Al igual que las operaciones, UML modela a los constructores en el *tercer* compartimiento de un diagrama de clases. Para diferenciar a un constructor de las operaciones de una clase, UML requiere que se coloque la palabra "constructor" entre los signos « y » antes del nombre del constructor. Es costumbre listar los constructores *antes* de otras operaciones en el tercer compartimiento.

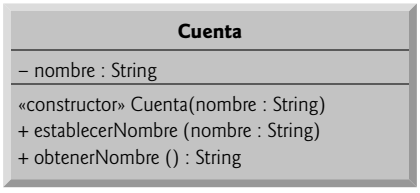

Fig. 3.7 | Diagrama de clases de UML para la clase `Cuenta` de la figura 3.5.

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

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

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

La mayoría de los programadores representan los números de punto flotante con el tipo `double`. De hecho, Java trata de manera predeterminada a todos los números de punto flotante que escribimos en el código fuente de un programa (como 7.33 y 0.0975) como valores `double`. Dichos valores en el código fuente se conocen como **literales de punto flotante**. En el apéndice D, Tipos primitivos, podrá consultar los rangos precisos de los valores para los tipos `float` y `double`.

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

Nuestra siguiente aplicación contiene una versión de la clase Cuenta (figura 3.8) que mantiene como variables de instancia el nombre *y* el saldo de una cuenta bancaria. Un banco ordinario da servicio a *muchas* cuentas, cada una con su *propio* saldo, por lo que la línea 8 declara una variable de instancia de tipo double llamada saldo. Cada instancia (es decir, objeto) de la clase Cuenta contiene sus *propias* copias de nombre y saldo.

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

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

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

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

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

La clase tiene un *constructor* y cuatro *métodos*. Puesto que es común que alguien abra una cuenta para depositar dinero de inmediato, el constructor (líneas 11 a 19) recibe ahora un segundo parámetro llamado `saldoInicial` de tipo `double`, el cual representa el *saldo inicial* de la cuenta. Las líneas 17 y 18 aseguran que `saldoInicial` sea mayor que `0.0`. De ser así, el valor de `saldoInicial` se asigna a la variable de instancia `saldo`. En caso contrario, `saldo` permanece en `0.0`, su *valor inicial predeterminado*.

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

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

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

El método `obtenerSaldo` (líneas 29 a la 32) permite a los *clientes* de la clase (es decir, otras clases cuyos métodos llamen a los métodos de esta clase) obtener el valor del `saldo` de un objeto `Cuenta` específico. El método especifica el tipo de valor de retorno `double` y una lista de parámetros *vacía*.

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

Observe una vez más que las instrucciones en las líneas 18, 25 y 31 utilizan la variable de instancia `saldo`, aun y cuando *no* se declaró en *ninguno* de los métodos. Podemos usar `saldo` en estos métodos, ya que es una *variable de instancia* de la clase.

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

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

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

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

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

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

Escriba el monto a depositar para cuenta1: 25.53

sumando 25.53 al saldo de cuenta1

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

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

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

sumando 123.45 al saldo de cuenta2

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

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

Cómo mostrar los saldos iniciales de los objetos `Cuenta`

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

Formato de los números de punto flotante para visualizarlos en pantalla

Cada uno de los saldos se imprime en pantalla mediante `printf`, con el especificador de formato `%.2f`. El **especificador de formato** `%f` se utiliza para imprimir valores de tipo `float` o `double`. El `.2` entre `%` y `f` representa el número de *lugares decimales* (2) que deben imprimirse a la *derecha* del punto decimal en el número de punto flotante; a esto también se le conoce como la **precisión** del número. Cualquier valor de punto flotante que se imprima con `%.2f` se *redondeará* a la *posición de las centenas*; por ejemplo, 123.457 se redondearía a 123.46 y 27.33379 se redondearía a 27.33.

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

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

Error común de programación 3.1

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

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

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

Código duplicado en el método `main`

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

Observación de ingeniería de software 3.4

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

Diagrama de clases de UML para la clase `Cuenta`

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

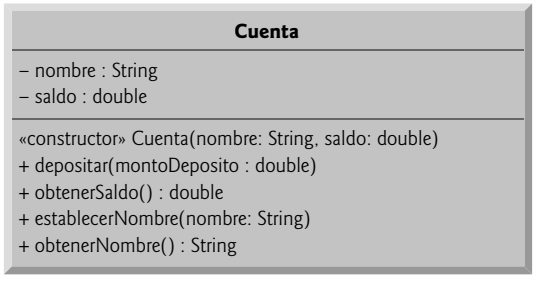

Fig. 3.10 | Diagrama de clases de UML para la clase `Cuenta` de la figura 3.8.

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

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

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

Este ejemplo práctico de GUI y gráficos aparece en 10 secciones breves (vea la figura 3.11). Cada sección presenta nuevos conceptos y proporciona ejemplos con capturas de pantalla que muestran interacciones de ejemplo y resultados. En las primeras secciones, usted creará sus primeras aplicaciones gráficas. En las secciones posteriores, utilizará los conceptos de programación orientada a objetos para crear una aplicación que dibuja una variedad de figuras. Cuando presentemos de manera formal a las GUI en el capítulo 12, utilizaremos el ratón para elegir con exactitud qué figuras dibujar y en dónde dibujarlas. En el capítulo 13 agregaremos las herramientas de la API de gráficos en 2D de Java para dibujar las figuras con distintos grosores de línea y rellenos. Esperamos que este ejemplo práctico le sea informativo y divertido.

Ubicación	Título – Ejercicio(s)
Sección 3.6	Uso de cuadros de diálogo: entrada y salida básica con cuadros de diálogo
Sección 4.15	Creación de dibujos simples: mostrar y dibujar líneas en la pantalla
Sección 5.11	Dibujo de rectángulos y óvalos: uso de figuras para representar datos
Sección 6.13	Colores y figuras rellenas: dibujar un tiro al blanco y gráficos aleatorios
Sección 7.17	Dibujo de arcos: dibujar espirales con arcos
Sección 8.16	Uso de objetos con gráficos: almacenar figuras como objetos
Sección 9.7	Mostrar texto e imágenes mediante el uso de etiquetas: proporcionar información de estado
Sección 10.10	Dibujo con polimorfismo: identificar las similitudes entre figuras
Ejercicio 12.17	Expansión de la interfaz: uso de componentes de la GUI y manejo de eventos
Ejercicio 13.31	Caso de estudio de GUI y gráficos: Agregar Java 2D

Fig. 3.11 | Resumen del ejemplo práctico de GUI y gráficos en cada capítulo.

Cómo mostrar texto en un cuadro de diálogo

Los programas que hemos presentado hasta ahora muestran su salida en la *ventana de comandos*. Muchas aplicaciones utilizan ventanas, o **cuadros de diálogo** (también llamados **diálogos**) para mostrar la salida. Por ejemplo, los navegadores Web como Chrome, Firefox, Internet Explorer, Safari y Opera muestran las páginas Web en sus propias ventanas. Los programas de correo electrónico le permiten escribir y leer mensajes en una ventana. Por lo general, los cuadros de diálogo son ventanas en las que los programas muestran mensajes importantes a los usuarios. La clase `JOptionPane` cuenta con cuadros de diálogo prefabricados, los cuales permiten a los programas mostrar ventanas que contengan mensajes; a dichas ventanas se les conoce como **diálogos de mensaje**. La figura 3.12 muestra el objeto `String` "`Bienvenido a Java`" en un diálogo de mensaje.

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

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

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

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

Introducción de los métodos `static`

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

NombreClase.nombreMétodo(*argumentos*)

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

Introducción de texto en un cuadro de diálogo

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

```
1  // Fig. 3.13: DialogoNombre.java
2  // Entrada básica con un cuadro de diálogo.
3  import javax.swing.JOptionPane;
4
5  public class DialogoNombre
6  {
7     public static void main(String[] args)
8     {
9        // pide al usuario que escriba su nombre
10       String nombre =JOptionPane.showInputDialog("Cual es su nombre?");
11
12       // crea el mensaje
13       String mensaje =
14          String.format("Bienvenido, %s, a la programacion en Java!", nombre);
15
16       // muestra el mensaje para dar la bienvenida al usuario por su nombre
17       JOptionPane.showMessageDialog(null, mensaje);
18    } // fin de main
19 } // fin de la clase DialogoNombre
```

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

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

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

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

Las líneas 13 y 14 utilizan el método `static String` llamado **`format`** para devolver un objeto `String` que contiene un saludo con el nombre del usuario. El método `format` es similar al método `System.out.printf`, excepto que `format` devuelve el objeto `String` con formato, en vez de mostrarlo en una ventana de comandos. La línea 17 muestra el saludo en un cuadro de diálogo de mensaje, como hicimos en la figura 3.12.

Ejercicio del ejemplo práctico de GUI y gráficos

3.1 Modifique el programa de suma que aparece en la figura 2.7 para usar la entrada y salida en un cuadro de diálogo con los métodos de la clase `JOptionPane`. Como el método `showInputDialog` devuelve un objeto `String`, debe convertir el objeto `String` que introduce el usuario a un `int` para usarlo en los cálculos. El método `parseInt`

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

3.7 Conclusión

En este capítulo aprendió a crear sus propias clases y métodos, a crear objetos de esas clases y a llamar métodos de esos objetos para realizar acciones útiles. Declaró variables de instancia de una clase para mantener los datos de cada objeto de la clase, y declaró sus propios métodos para operar sobre esos datos. Aprendió cómo llamar a un método para decirle que realice su tarea y cómo pasar información a un método en forma de argumentos cuyos valores se asignan a los parámetros del mismo. Vio la diferencia entre una variable local de un método y una variable de instancia de una clase, y que sólo las variables de instancia se inicializan en forma automática. También aprendió a utilizar el constructor de una clase para especificar los valores iniciales para las variables de instancia de un objeto. Vio cómo crear diagramas de clases en UML que modelen los métodos, atributos y constructores de las clases. Por último, aprendió acerca de los números de punto flotante (números con puntos decimales); aprendió cómo almacenarlos con variables del tipo primitivo `double`, cómo recibirlos en forma de datos de entrada mediante un objeto `Scanner` y cómo darles formato con `printf` y el especificador de formato `%f` para fines de visualización. [En el capítulo 8 comenzaremos a representar las cantidades monetarias en forma precisa con la clase `BigDecimal`]. Tal vez también ya haya comenzado el ejemplo práctico opcional de GUI y gráficos, con el que aprenderá a escribir sus primeras aplicaciones de GUI. En el siguiente capítulo empezaremos nuestra introducción a las instrucciones de control, las cuales especifican el orden en el que se realizan las acciones de un programa. Utilizará estas instrucciones en sus métodos para especificar el orden en el que deben realizar sus tareas.

Resumen

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

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

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

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

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

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

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

Sección 3.2.3 Compilación y ejecución de una aplicación con varias clases

- El comando `javac` puede compilar varias clases a la vez. Sólo debe listar los nombres de archivo del código fuente después del comando, separando cada nombre de archivo del siguiente mediante un espacio. Si el directorio que contiene la aplicación incluye sólo los archivos de esa aplicación, puede compilar todas sus clases con el comando `javac *.java`. El asterisco (`*`) en `*.java` indica que deben compilarse todos los archivos en el directorio actual que terminen con la extensión de nombre de archivo "`.java`".

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

- En UML, cada clase se modela en un diagrama de clases (pág. 77) en forma de rectángulo con tres compartimientos. El compartimiento superior contiene el nombre de la clase, centrado horizontalmente y en negrita. El compartimiento intermedio contiene los atributos de la clase, que corresponden a las variables de instancia en Java. El compartimiento inferior contiene las operaciones de la clase (pág. 78), que corresponden a los métodos y los constructores en Java.
- UML representa a las variables de instancia como un nombre de atributo, seguido de dos puntos y el tipo del atributo.
- En UML, los atributos privados van precedidos por un signo menos (-).
- Para modelar las operaciones, UML lista el nombre de la operación, seguido de un conjunto de paréntesis. Un signo más (+) enfrente del nombre de la operación indica que ésta es una operación pública en UML (es decir, un método `public` en Java).
- Para modelar un parámetro de una operación, UML lista el nombre del parámetro, seguido de dos puntos y el tipo del parámetro entre los paréntesis que van después del nombre de la operación.
- Para indicar el tipo de valor de retorno de una operación, UML coloca dos puntos y el tipo de valor de retorno después de los paréntesis que siguen del nombre de la operación.
- Los diagramas de clases de UML no especifican tipos de valores de retorno para las operaciones que no devuelven valores.
- Al proceso de declarar variables de instancia como `private` se le conoce como ocultamiento de datos o de información.

Sección 3.2.5 Observaciones adicionales sobre la clase `PruebaCuenta`

- Debe llamar a la mayoría de los métodos distintos de `main` de manera explícita para decirles que realicen sus tareas.
- Una parte clave para permitir que la JVM localice y llame al método `main` para empezar la ejecución de la aplicación es la palabra clave `static`, la cual indica que `main` es un método `static` que puede llamarse sin tener que crear primero un objeto de la clase en la cual se declara ese método.
- La mayoría de las clases que utilizará en los programas de Java deben importarse de manera explícita. Hay una relación especial entre las clases que se compilan en el mismo directorio. De manera predeterminada, se considera que dichas clases se encuentran en el mismo paquete; a éste se le conoce como el paquete predeterminado. Las clases en el mismo paquete se importan implícitamente en los archivos de código fuente de las demás clases del mismo paquete. No se requiere una declaración `import` cuando una clase en un paquete usa otra clase en el mismo paquete.
- No se requiere una declaración `import` si siempre hace referencia a una clase con su nombre de clase completamente calificado, que incluye tanto el nombre de su paquete como de su clase.

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

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

Sección 3.3 Comparación entre tipos primitivos y tipos por referencia

- En Java, los tipos se dividen en dos categorías: tipos primitivos y tipos por referencia. Los tipos primitivos son `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` y `double`. Todos los demás tipos son por referencia, por lo cual, las clases que especifican los tipos de los objetos, son tipos por referencia.
- Una variable de tipo primitivo puede en un momento dado almacenar exactamente un valor de su tipo declarado.
- Las variables de instancia de tipos primitivos se inicializan de manera predeterminada. Las variables de los tipos `byte`, `char`, `short`, `int`, `long`, `float` y `double` se inicializan con `0`. Las variables de tipo `boolean` se inicializan con `false`.
- Las variables de tipos por referencia (llamadas referencias; pág. 81) almacenan la ubicación de un objeto en la memoria de la computadora. Dichas variables hacen referencia a los objetos en el programa. El objeto al que se hace referencia puede contener muchas variables de instancia y métodos.
- Las variables de instancia del tipo por referencia se inicializan de manera predeterminada con el valor `null`.

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

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

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

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

- Un número de punto flotante (pág. 84) es un número con un punto decimal. Java proporciona dos tipos primitivos para almacenar números de punto flotante en la memoria: `float` y `double` (pág. 84).
- Las variables de tipo `float` representan números de punto flotante de precisión simple, y tienen siete dígitos significativos. Las variables de tipo `double` representan números de punto flotante de precisión doble. Éstas requieren el doble de memoria que las variables `float` y proporcionan 15 dígitos significativos. Tienen aproximadamente el doble de precisión de las variables `float`.
- Las literales de punto flotante (pág. 84) son de tipo `double` de manera predeterminada.
- El método `nextDouble` de `Scanner` (pág. 88) devuelve un valor `double`.
- El especificador de formato `%f` (pág. 88) se utiliza para mostrar valores de tipo `float` o `double`. El especificador de formato `%.2f` especifica que se deben mostrar dos dígitos de precisión a la derecha del punto decimal (pág. 88), en el número de punto flotante.
- El valor predeterminado para una variable de instancia de tipo `double` es `0.0`, y el valor predeterminado para una variable de instancia de tipo `int` es `0`.

Ejercicios de autoevaluación

3.1 Complete las siguientes oraciones:

a) Cada declaración de clase que empieza con la palabra clave _____ debe almacenarse en un archivo que tenga exactamente el mismo nombre de la clase, y que termine con la extensión de nombre de archivo `.java`.

b) En la declaración de una clase, la palabra clave _____ va seguida inmediatamente por el nombre de la clase.

c) La palabra clave _____ solicita memoria del sistema para almacenar un objeto, y después llama al constructor de la clase correspondiente para inicializar ese objeto.

d) Cada parámetro debe especificar un(a) _____ y un(a) _____.

e) De manera predeterminada, se considera que las clases que se compilan en el mismo directorio están en el mismo paquete, conocido como _____.

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

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

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

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

3.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

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

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

3.4 Explique el propósito de un parámetro de un método. ¿Cuál es la diferencia entre un parámetro y un argumento?

Respuestas a los ejercicios de autoevaluación

3.1 a) `public`. b) `class`. c) `new`. d) tipo, nombre. e) paquete predeterminado. f) `float`, `double`. g) de precisión doble. h) `nextDouble`. i) modificador. j) `void`. k) `nextLine`. l) `java.lang`. m) declaración `import`. n) número de punto flotante. o) simple. p) `%f`. q) primitivo, por referencia.

3.2 a) Falso. Por convención, los nombres de los métodos empiezan con una primera letra en minúscula y todas las palabras subsiguientes en el nombre empiezan con una letra en mayúscula. b) Verdadero. c) Verdadero. d) Falso. Una variable de tipo primitivo no puede usarse para invocar a un método; se requiere una referencia a un objeto para invocar a los métodos de ese objeto. e) Falso. Dichas variables se llaman variables locales, y sólo se pueden utilizar en el método en el que están declaradas. f) Verdadero. g) Falso. Las variables de instancia de tipo primitivo se inicializan de manera predeterminada. A cada variable local se le debe asignar un valor de manera explícita. h) Verdadero. i) Verdadero. j) Verdadero. k) Falso. Dichas literales son de tipo `double` de manera predeterminada.

3.3 Una variable local se declara en el cuerpo de un método, y sólo puede utilizarse a partir del punto en el que se declaró, hasta el final de la declaración del método. Una variable de instancia se declara en una clase, pero no en el cuerpo de alguno de los métodos de la clase. Además, las variables de instancia están accesibles para todos los métodos de la clase (en el capítulo 8 veremos una excepción a esto).

3.4 Un parámetro representa la información adicional que requiere un método para realizar su tarea. Cada parámetro requerido por un método está especificado en la declaración del método. Un argumento es el valor actual para un parámetro del método. Cuando se llama a un método, los valores de los argumentos se pasan a los parámetros correspondientes del método, para que éste pueda realizar su tarea.

Ejercicios

3.5 ***(Palabra clave `new`)*** ¿Cuál es el propósito de la palabra clave `new`? Explique lo que ocurre cuando se utiliza en una aplicación.

3.6 ***(Constructores predeterminados)*** ¿Qué es un constructor predeterminado? ¿Cómo se inicializan las variables de instancia de un objeto, si una clase sólo tiene un constructor predeterminado?

3.7 ***(Variables de instancia)*** Explique el propósito de una variable de instancia.

3.8 ***(Uso de clases sin importarlas)*** La mayoría de las clases necesitan importarse antes de poder ser usadas en una aplicación ¿Por qué cualquier aplicación puede utilizar las clases `System` y `String` sin tener que importarlas primero?

3.9 ***(Uso de una clase sin importarla)*** Explique cómo podría un programa utilizar la clase `Scanner` sin importarla.

3.10 ***(Métodos* establecer *y* obtener*)*** Explique por qué una clase podría proporcionar un método *establecer* y un método *obtener* para una variable de instancia.

3.11 ***(Clase `Cuenta` modificada)*** Modifique la clase `Cuenta` (figura 3.8) para proporcionar un método llamado `retirar`, que retire dinero de un objeto `Cuenta`. Asegúrese de que el monto a retirar no exceda el saldo de `Cuenta`. Si lo hace, el saldo debe permanecer sin cambio y el método debe imprimir un mensaje que indique "`El monto a retirar excede el saldo de la cuenta`". Modifique la clase `PruebaCuenta` (figura 3.9) para probar el método `retirar`.

3.12 ***(La clase `Factura`)*** Cree una clase llamada `Factura` que una ferretería podría utilizar para representar una factura para un artículo vendido en la tienda. Una `Factura` debe incluir cuatro piezas de información como variables de instancia: un número de pieza (tipo `String`), la descripción de la pieza (tipo `String`), la cantidad de artículos de ese tipo que se van a comprar (tipo `int`) y el precio por artículo (`double`). Su clase debe tener un constructor que inicialice las cuatro variables de instancia. Proporcione un método *establecer* y un método *obtener* para cada variable de instancia. Además, proporcione un método llamado `obtenerMontoFactura`, que calcule el monto de la factura (es decir, que multiplique la cantidad de artículos por el precio de cada uno) y después devuelva ese monto como un valor `double`. Si la cantidad no es positiva, debe establecerse en `0`. Si el precio por artículo no es positivo, debe establecerse en `0.0`. Escriba una aplicación de prueba llamada `PruebaFactura`, que demuestre las capacidades de la clase `Factura`.

3.13 ***(La clase `Empleado`)*** Cree una clase llamada `Empleado`, que incluya tres variables de instancia: un primer nombre (tipo `String`), un apellido paterno (tipo `String`) y un salario mensual (`double`). Su clase debe tener un constructor que inicialice las tres variables de instancia. Proporcione un método *establecer* y un método *obtener* para cada variable de instancia. Si el salario mensual no es positivo, no establezca su valor. Escriba una aplicación de prueba llamada `PruebaEmpleado`, que demuestre las capacidades de la clase `Empleado`. Cree dos objetos `Empleado` y muestre el salario *anual* de cada objeto. Después, proporcione a cada `Empleado` un aumento del 10% y muestre el salario anual de cada `Empleado` otra vez.

3.14 ***(La clase `Fecha`)*** Cree una clase llamada `Fecha`, que incluya tres variables de instancia: un mes (tipo `int`), un día (tipo `int`) y un año (tipo `int`). Su clase debe tener un constructor que inicialice las tres variables de instancia, y debe asumir que los valores que se proporcionan son correctos. Proporcione un método *establecer* y un método *obtener* para cada variable de instancia. Proporcione un método `mostrarFecha`, que muestre el mes, día y año, separados por barras diagonales (`/`). Escriba una aplicación de prueba llamada `PruebaFecha`, que demuestre las capacidades de la clase `Fecha`.

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

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

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

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

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

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

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

Marcando la diferencia

3.16 ***(Calculadora de la frecuencia cardiaca esperada)*** Mientras se ejercita, puede usar un monitor de frecuencia cardiaca para ver que su corazón permanezca dentro de un rango seguro sugerido por sus entrenadores y doctores. De acuerdo con la Asociación Estadounidense del Corazón (AHA) (`www.americanheart.org/presenter.jhtml?identifier=4736`), la fórmula para calcular su *frecuencia cardiaca máxima* en pulsos por minuto es 220 menos su edad en años. Su *frecuencia cardiaca esperada* tiene un rango que está entre el 50 y el 85% de su frecuencia cardiaca máxima. [*Nota*: estas fórmulas son estimaciones proporcionadas por la AHA. Las frecuencias cardiacas máxima y esperada pueden variar con base en la salud, condición física y sexo del individuo. **Siempre debe consultar un médico o a un profesional de la salud antes de empezar o modificar un programa de ejercicios**]. Cree una clase llamada `FrecuenciasCardiacas`. Los atributos de la clase deben incluir el primer nombre de la persona, su apellido y fecha de nacimiento (la cual debe consistir de atributos independientes para el mes, día y año de nacimiento). Su clase debe tener un constructor que reciba estos datos como parámetros. Para cada atributo debe proveer métodos *establecer* y *obtener*. La clase también debe incluir un método que calcule y devuelva la edad de la persona (en años), un método que calcule y devuelva la frecuencia cardiaca máxima de esa persona, y otro método que calcule y devuelva la frecuencia cardiaca esperada de la persona. Escriba una aplicación de Java que pida la información de la persona, cree una instancia de un objeto de la clase `FrecuenciasCardiacas` e imprima la información a partir de ese objeto (incluyendo el primer nombre de la persona, su apellido y fecha de nacimiento), y que después calcule e imprima la edad de la persona en (años), frecuencia cardiaca máxima y rango de frecuencia cardiaca esperada.

3.17 ***(Computarización de los registros médicos)*** Un tema relacionado con la salud que ha estado últimamente en las noticias es la computarización de los registros médicos. Esta posibilidad se está tratando con mucho cuidado, debido a las delicadas cuestiones de privacidad y seguridad, entre otras cosas. [Trataremos esas cuestiones en ejercicios posteriores]. La computarización de los registros médicos puede facilitar a los pacientes el proceso de compartir sus perfiles e historiales médicos con los diversos profesionales de la salud que consulten. Esto podría mejorar la calidad del servicio médico, ayudar a evitar conflictos de fármacos y prescripciones erróneas, reducir los costos y, en emergencias, podría ayudar a salvar vidas. En este ejercicio usted diseñará una clase "inicial" llamada `PerfilMedico` para una

persona. Los atributos de la clase deben incluir el primer nombre de la persona, su apellido, sexo, fecha de nacimiento (que debe consistir de atributos separados para el día, mes y año de nacimiento), altura (en centímetros) y peso (en kilogramos). Su clase debe tener un constructor que reciba estos datos. Para cada atributo, debe proveer los métodos *establecer* y *obtener*. La clase también debe incluir métodos que calculen y devuelvan la edad del usuario en años, la frecuencia cardiaca máxima y el rango de frecuencia cardiaca esperada (vea el ejercicio 3.16), además del índice de masa corporal (BMI; vea el ejercicio 2.33). Escriba una aplicación de Java que pida la información de la persona, cree una instancia de un objeto de la clase `PerfilMedico` para esa persona e imprima la información de ese objeto (incluyendo el primer nombre de la persona, apellido, sexo, fecha de nacimiento, altura y peso), y que después calcule e imprima la edad de esa persona en años, junto con el BMI, la frecuencia cardiaca máxima y el rango de frecuencia cardiaca esperada. También debe mostrar la tabla de valores del BMI del ejercicio 2.33.