CAPÍTULO

# 3 Objetos y clases

En este capítulo se comienza a estudiar la *programación orientada a objetos*. Una componente fundamental de la programación orientada a objetos es la especificación, implementación y uso de objetos. En el Capítulo 2 vimos algunos ejemplos de objetos, incluyendo las cadenas y los ficheros, que son parte de la librería obligatoria de Java. También vimos que estos objetos tienen un estado interno que se puede manipular aplicando el operador punto para seleccionar un método. En Java, el estado y la funcionalidad de un objeto vienen dados por la definición de una *clase*. Un objeto es una instancia de una clase.

En este capítulo veremos:

- Cómo usa Java las clases para conseguir *encapsulación* y *ocultamiento de información*.
- Cómo se implementan y se documentan de forma automática las clases.
- Cómo se agrupan las clases en paquetes.

## 3.1 ¿Qué es la programación orientada a objetos?

La programación orientada a objetos parece estar surgiendo como el paradigma dominante de los noventa. En esta sección se estudian algunos elementos de la orientación a objetos que soporta Java y se mencionan algunos de los principios de la programación orientada a objetos.

El corazón de la programación orientada a objetos es el *objeto*. Un objeto es un tipo de datos con estructura y estado. Cada objeto tiene definidas operaciones que pueden acceder a ese estado o manipularlo. Como ya hemos visto, en Java los objetos se distinguen de los valores de los tipos primitivos, pero ésta es más bien una característica de Java que de la programación orientada a objetos. Además de ejecutar acciones generales, podemos hacer lo siguiente:

Los *objetos* son entidades con estructura y estado. Cada objeto define operaciones que pueden acceder a ese estado o manipularlo.

- Crear nuevos objetos, posiblemente con un valor inicial.
- Copiar o comprobar la igualdad.
- Realizar E/S sobre esos objetos.

Un objeto es una *unidad atómica*: los usuarios generales del objeto no pueden diseccionar sus partes.

El *ocultamiento de información* convierte los detalles de la implementación, incluyendo las componentes de un objeto, en inaccesibles.

La *encapsulación* consiste en el agrupamiento de datos y de las operaciones que se aplican sobre ellos para formar un agregado, junto con el ocultamiento de la implementación del agregado.

Una *clase* en Java consta de *atributos* que almacenan datos, y de *métodos* que se aplican a instancias de la clase.

También vimos el objeto como una *unidad atómica* que el usuario no debería pretender diseccionar. La mayoría de nosotros ni siquiera pensaría en jugar con los bits que representan a un número en coma flotante, y consideraríamos completamente ridículo el intentar incrementar nosotros mismos un número en coma flotante alterando su representación interna.

El principio de atomicidad se conoce como *ocultamiento de la información*. El usuario no tiene acceso directo a las partes del objeto o a su implementación; solamente se puede acceder a ellas de forma indirecta a través de métodos proporcionados junto con el objeto. Podemos pensar que cada objeto viene con el aviso «No abrir —partes no utilizables por el usuario». En la vida real, la mayoría de la gente que intenta arreglar las cosas a pesar de tales avisos, termina produciendo más perjuicio que beneficio. En este sentido, la programación imita al mundo real. El agrupamiento de datos y las operaciones que se aplican sobre ellos, forman un agregado, mientras que el ocultamiento de los detalles del agregado, se conoce como *encapsulación*.

Un objetivo importante de la programación orientada a objetos es soportar la reutilización de código. Al igual que los ingenieros usan las misma componentes repetidamente en sus diseños, los programadores deberían ser capaces de reutilizar objetos en lugar de implementarlos repetidamente. Cuando tenemos una implementación del objeto exacto que necesitamos usar, la reutilización es simple. El problema está en utilizar un objeto disponible cuando no es exactamente el que se necesita, sino solamente uno muy parecido.

Los lenguajes orientados a objetos proporcionan varios mecanismos para satisfacer este objetivo. Uno es el uso de código *genérico*. Si la implementación es idéntica independientemente del tipo de los objetos sobre los que se aplica, no hay necesidad de rescribir completamente el código: en lugar de ello, se escribe el código genéricamente, de forma que funcione para cualquier tipo. Por ejemplo, la lógica usada para ordenar un vector de objetos es independiente del tipo de los objetos que se están ordenando, de modo que se podría usar un algoritmo genérico.

El mecanismo de *herencia* nos permite extender la funcionalidad de un objeto. En otras palabras, podemos crear tipos nuevos con las propiedades del tipo original restringidas (o extendidas). La herencia nos permite avanzar un largo trecho hacia el objetivo deseado de reutilización de código.

Otro principio importante de la programación orientada a objetos es el *polimorfismo*. Un tipo referencia polimórfico puede referenciar objetos de varios tipos diferentes. Cuando se aplican métodos al tipo polimórfico, se selecciona automáticamente la operación apropiada para el objeto referenciado en ese momento. En Java, esto se implementa como parte de la herencia. El polimorfismo nos permite implementar clases que comparten una lógica común. Como se estudia en el Capítulo 4, esto se aplica reiteradamente en las librerías de Java. El uso de la herencia para crear estas jerarquías distingue la programación orientada a objetos de la más simple *programación basada en objetos*.

En Java, los algoritmos genéricos se implementan como parte de la herencia. En el Capítulo 4 se estudia la herencia y el polimorfismo. En este capítulo, describimos cómo Java usa las clases para conseguir encapsulación y ocultamiento de información.

Un *objeto* en Java es una instancia de una clase. Una *clase* es similar a una estructura en C o un registro en Pascal o Ada, excepto por el hecho de que hay dos mejoras importantes. En primer lugar, las componentes pueden ser funciones o

datos, las cuales se conocen respectivamente por *métodos* y *atributos*. En segundo lugar, se puede restringir la visibilidad de dichas componentes. Puesto que los métodos que manipulan el estado del objeto son componentes de la clase, se accede a ellos mediante el operador punto, al igual que a los atributos. En la terminología de la programación orientada a objetos, cuando hacemos una llamada a un método, estamos pasando un mensaje al objeto.

## 3.2 Un ejemplo sencillo

La funcionalidad se incorpora a través de los *métodos*, que pueden verse como componentes adicionales a través de las cuales se manipula el estado del objeto.

Recuerde que cuando está diseñando una clase, es importante ser capaz de ocultar los detalles internos al usuario de la clase. Esto se consigue de dos formas. En primer lugar, la clase puede definir su funcionalidad mediante componentes de la clase, llamadas métodos. Algunos de estos métodos describen cómo se crea e inicializa una instancia de la estructura, otros cómo se realiza las comprobaciones de igualdad o cómo se lleva a cabo la salida. Otras funciones serían específicas a la estructura particular. La idea es que los atributos internos que representan el estado del objeto no deberían ser manipulados directamente por el usuario de la clase, sino solamente a través de los métodos. Esta idea se puede reforzar ocultando componentes al usuario. Para hacer esto podemos especificar que se almacenen en una sección *privada*. El compilador hará cumplir la regla que convierte a las componentes de la sección privada en inaccesibles para el usuario del objeto. En general, todas las componentes de datos deberían ser privadas.

Las componentes *públicas* son visibles para rutinas que no pertenecen a la clase; las componentes *privadas* no lo son.

En la Figura 3.1 se muestra una declaración de clase para un objeto `CeldaEntero`[1].La declaración consta de dos partes: una pública y otra privada. La sección *pública* representa la porción visible al usuario del objeto. Puesto que esperamos ocultar los datos, en general solamente debería aparecer métodos y constantes en la sección pública. En nuestro ejemplo, tenemos métodos que leen y escriben en el objeto `CeldaEntero`. La sección *privada* contiene los datos: ésta es invisible para el usuario del objeto. Se debe acceder al atributo `valorAlmacenado` a través de las rutinas públicas `leer` y `escribir`; `main` no puede acceder directamente a ella. En la Figura 3.2 encontramos una representación gráfica de la situación.

```
1  // Clase CeldaEntero
2  // int leer( )              -->  Devuelve el valor almacenado
3  // void escribir( int x )   -->  Se almacena x
4
5  public class CeldaEntero
6  {
7          // Métodos públicos
8      public int leer( ) {return valorAlmacenado;}
9      public void escribir( int x ) {valorAlmacenado = x;}
10
11         // Representación interna privada del dato
12     private int valorAlmacenado;
13 }
```

**Figura 3.1** Una declaración completa de una clase `CeldaEntero`.

[1] Recuerde que las clases deben guardarse en ficheros del mismo nombre que la clase. Luego `CeldaEntero` debe guardarse en el fichero *CeldaEntero.java*.


**Figura 3.2** Componentes de `CeldaEntero`: `leer` y `escribir` son accesibles pero `valorAlmacenado` está oculto.

Las componentes declaradas como privadas no son visibles para las rutinas que no son de la clase.
Un *atributo* es una componente que almacena datos; un *método* es una componente que lleva a cabo una acción.

La Figura 3.3 muestra cómo se usan los objetos de la clase `CeldaEntero`. Puesto que `leer` y `escribir` son componentes de la clase `CeldaEntero`, se puede acceder a ellas usando el operador punto. También se podría acceder al atributo `valorAlmacenado` usando el operador punto, pero debido a que es privada, el acceso en la línea 14 sería ilegal, si ésta no estuviera comentada.

A continuación resumimos la terminología. La clase define *componentes*, que pueden ser *atributos* (datos) o *métodos* (funciones). Los métodos pueden actuar sobre los atributos y pueden llamar a otros métodos. El modificador de visibilidad `public` significa que la componente es accesible para cualquiera a través del operador punto. El modificador de visibilidad `private` significa que la componente solamente es accesible a través de otros métodos de esa clase. Si no hay modificador de visibilidad, tenemos acceso amistoso, el cual se estudia en la Sección 3.5.4. Hay un cuarto modificador conocido como `protected`, que se estudia en el Capítulo 4.

```
1  //Ejercicio con la clase CeldaEntero
2
3  public class TestCeldaEntero
4  {
5      public static void main( String [ ] args )
6      {
7          CeldaEntero m = new CeldaEntero( );
8
9          m.escribir( 5 );
10         System.out.println( "Contenidos de la celda: " + m.leer( ) );
11
12         // La siguiente línea sería ilegal si no estuviera comentada
13         // porque valorAlmacenado es una componente privada
14     // m.valorAlmacenado = 0;
15     }
16 }
```

**Figura 3.3** Una rutina simple de comprobación para mostrar cómo se accede a los objetos de la clase `CeldaEntero`.

## 3.3 Javadoc

La *especificación de una clase* describe lo que se puede hacer sobre un objeto. La *implementación* representa los detalles internos de cómo se satisface la especificación.

Cuando se diseña una clase, la *especificación de la clase* representa el diseño de la clase y nos dice qué se puede hacer sobre un objeto. La *implementación* representa los detalles internos de cómo se consigue esto. En lo que respecta al usuario, estos detalles no son importantes. En muchos casos, la implementación representa información del propietario que el diseñador de la clase no desea compartir. Sin embargo, la especificación debe estar compartida; en caso contrario, no se podría usar la clase.

En muchos lenguajes, los objetivos de compartir la especificación pero con ocultamiento de la implementación se consigue colocando especificación e implementación en ficheros fuente separados. Por ejemplo, en C++ la interfaz de clase se coloca en un archivo con extensión `.h`, y la implementación de la clase en un archivo `.cpp`. En el fichero `.h`, la interfaz de clase muestra los métodos (proporcionando las cabeceras de los métodos) implementados por la clase.

El programa *javadoc* genera automáticamente documentación para las clases.

Java tiene un enfoque distinto. Es fácil ver que se puede documentar automáticamente una lista de métodos en una clase, con signaturas y tipos del resultado, a partir de la implementación. Java usa esta idea: el programa *javadoc*[2], proporcionado en todos los sistemas Java, se puede ejecutar para generar automáticamente documentación para las clases. La salida de *javadoc* es un conjunto de ficheros HTML que se puede ver o imprimir con un navegador.

El fichero de implementación también puede añadir comentarios *javadoc* que comiencen con el token `/**`. Dichos comentarios se añaden automáticamente, de una forma uniforme y consistente, a la documentación producida por *javadoc*.

Algunas marcas de *javadoc* son `@author`, `@param`, `@return` y `@exception`. Se usan en los comentarios *javadoc*.

Hay también varias marcas especiales que se pueden incluir en los comentarios *javadoc*. Algunas de ellas son `@author`, `@param`, `@return` y `@exception`. En la Figura 3.4 se muestra el uso de los comentarios *javadoc* en la definición de clases. En la línea 3, se usa la marca `@author`. Esta marca debe preceder a la definición de la clase. La línea 10 ilustra el uso de la marca `@return`, y la 19 el de la marca `@param` . Estas marcas deben aparecer antes de la declaración de un método. El primer token que sigue a la marca `@param` es el nombre del parámetro. La marca `@exception` no se ilustra, pero tiene la misma sintaxis que `@param`.

```
1  /**
2   * Clase para simular una celda de memoria para un entero
3   * @author Mark A. Weiss
4   */
5
6  public class CeldaEntero
7  {
8      /**
9       * Obtiene el valor almacenado.
10      * @return el valor almacenado.
11      */
12     public int leer( )
13     {
14         return valorAlmacenado;
15     }
16
17     /**
18      * Almacena un valor.
19      * @param x el número a almacenar.
20      */
21     public void escribir( int x )
22     {
23         valorAlmacenado = x;
24     }
25
26     private int valorAlmacenado;
27 }
```

**Figura 3.4** Declaración de `CeldaEntero` con comentarios *javadoc*.

[2] *N. del T.:* La Figura 3.5 corresponde a la versión inglesa de la Figura 3.4.

En la Figura 3.5 se muestra la salida que resulta de la ejecución de *javadoc*. Ejecute *javadoc* proporcionando el nombre (incluida la extensión `.java`) del fichero fuente. Para obtener dibujos para los constructores, métodos, etc., debe tener un subdirectorio `images` en el directorio donde se generan los ficheros HTML

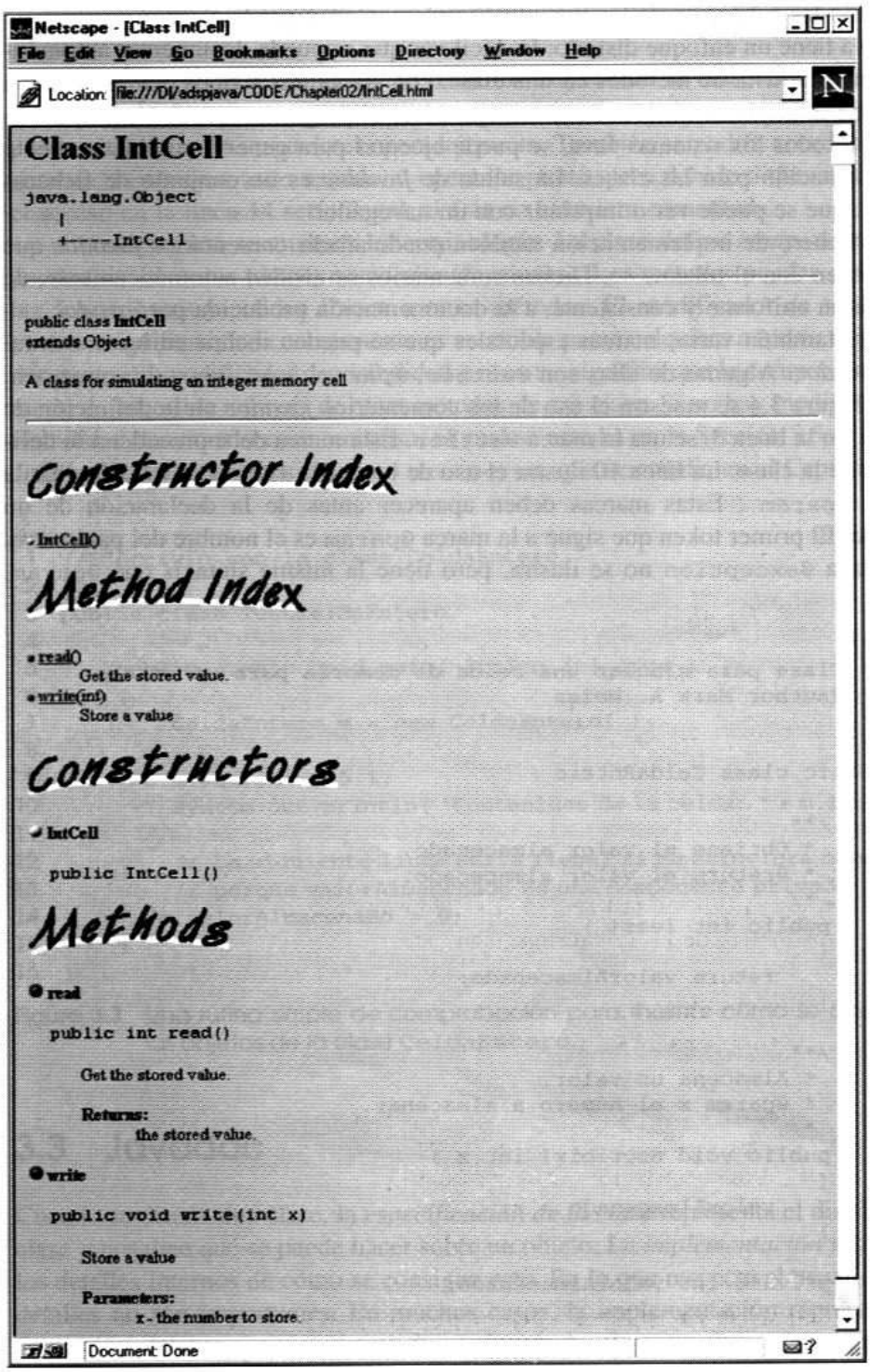


**Figura 3.5** Salida de *javadoc* para la Figura 3.4.

(este subdirectorio es parte de la distribución JDK y está replicado en el código que se suministra en la red).

La salida de *javadoc* está formada exclusivamente por comentarios, excepto en el caso de las cabeceras de los métodos. El compilador no comprueba que dichos comentarios estén implementados. A pesar de ello, nunca nos quedaremos cortos al valorar la importancia de disponer de documentación apropiada. *javadoc* hace más fácil la tarea de generar documentación con un buen formato.

## 3.4 Métodos básicos

Algunos métodos son comunes a todas las clases. En esta sección se estudia los métodos *modificadores* y *de acceso*, y tres métodos especiales: los constructores, `toString` y `equals`. También se habla de los métodos estáticos, uno de los cuales es `main`.

### 3.4.1 Constructores

Un *constructor* nos dice cómo se declara e inicializa un objeto.

El constructor por defecto consiste en una aplicación atributo a atributo de las correspondientes inicializaciones por defecto.

Como se ha mencionado anteriormente, una propiedad básica de los objetos es que se pueden definir, posiblemente con un valor inicial. En Java, los métodos que controlan cómo se crea e inicializa un objeto reciben el nombre de *constructores*. Gracias a la sobrecarga, en una clase se puede definir diversos constructores.

Si no se proporciona ningún constructor, como en el caso de la clase `CeldaEntero` de la Figura 3.1, se genera un constructor por defecto que inicializa cada atributo usando los correspondientes valores por defecto. Esto significa que los atributos de tipo primitivo se inicializan a cero y los tipos referencia se inicializan a la referencia `null`. Por tanto, en el caso de `CeldaEntero`, la componente `valorAlmacenado` es 0.

Para escribir un constructor, proporcionamos un método con el mismo nombre que la clase y sin resultado. En la Figura 3.6 hay dos constructores: uno comienza en la línea 7 y el otro en la línea 15. Usando estos constructores, podemos construir objetos `Fecha` de las dos formas siguientes:

```
Fecha f1 = new Fecha( );
Fecha f2 = new Fecha( 15, 4, 1998 );
```

Observe que una vez que se escribe un constructor, ya no se utiliza un constructor por defecto sin parámetros. Si desea volver a contar con él, tiene que escribirlo explícitamente. Así, el constructor de la línea 7 es obligatorio para poder construir el objeto que referencia `f1`.

### 3.4.2 Métodos modificadores y de acceso

Los atributos de una clase se declaran normalmente como privados. En consecuencia, las rutinas que no pertenecen a la clase no pueden acceder directamente a

ellos. Sin embargo, algunas veces nos gustaría examinar el valor de un atributo, e incluso podríamos querer cambiarlo directamente.

Un método que examina, pero no cambia, el estado de un objeto es un *método de acceso*. Un método que cambia el estado es un *modificador*.

Una alternativa para hacer esto es declarar los atributos como públicos. Ésta es, sin embargo, una elección pobre, pues viola el principio de ocultamiento de información. En su lugar podemos proporcionar métodos para examinar y cambiar cada atributo. Un método que examina pero no cambia el estado de un objeto es un *método de acceso*. Un método que cambia el estado es un *modificador* (porque modifica el estado del objeto).

Casos particulares de métodos de acceso y modificadores trabajan solamente sobre un atributo. Estos métodos de acceso usualmente tendrán nombres que empiezan por `obtener`[3], como `obtenerMes`, mientras que los modificadores tendrán nombres que comienzan por `cambiar`[4], como `cambiarMes`.

La ventaja de usar un modificador es que de esta forma podemos asegurar que los cambios realizados en el estado del objeto son consistentes. Así, un modificador que cambie el atributo `dia` en un objeto `Fecha` nos garantizará que solamente se generan fechas legales.

### 3.4.3 Salida y `toString`

Podemos escribir un método `toString`, que devuelva un valor de tipo `String` representando el estado del objeto.

Normalmente queremos mostrar el estado de un objeto usando `print`. Esto se consigue definiendo un método `toString`[5] en la clase. Dicho método devuelve un valor de tipo `String` apto para producir una salida. Como ejemplo, en la Figura 3.6 se muestra una implementación básica del método `toString` para la clase `Fecha`.

### 3.4.4 `equals`

Se puede definir un método `equals` para comprobar si dos referencias describen el mismo valor.

El método `equals` se usa para comprobar si dos referencias describen el mismo valor. La signatura es siempre la siguiente

```
public boolean equals( Object lder )
```

Observe que el parámetro es del tipo `Object`, en lugar del tipo de la clase. El método `equals` sobre una clase `NombreClase` se implementará de modo que devuelva `true` sólo cuando `lder` es una instancia de `NombreClase` y, si tras la conversión a `NombreClase`, todos los atributos primitivos son iguales (usando `==`) y todos los atributos referencia son iguales (usando una aplicación componente a componente de `equals`).

El parámetro de `equals` es de tipo `Object`.

En la Figura 3.6 se proporciona un ejemplo de cómo se implementa `equals`. El operador `instanceof` se estudia en la Sección 3.6.3.

---

[3] *N. del T.:* `get` en inglés.

[4] *N. del T.:* `set` en inglés.

[5] *N. del T.:* Los métodos `toString`, `equals` y `compareTo` no se han traducido ya que dichos métodos están predefinidos para el tipo `Object`. Las definiciones dadas en este libro sobreescriben a los predefinidos.

```
1  // Clase Fecha básica para ilustrar algunas características de
2  // Java sin comprobaciones de error ni comentarios javadoc
3
4  public class Fecha
5  {
6          // Constructor de cero parámetros
7      public Fecha( )
8      {
9          dia = 1;
10         mes = 1;
11         anyo = 1998;
12     }
13
14         // Constructor de tres parámetros
15     public Fecha( int elDia, int elMes, int elAnyo )
16     {
17         dia = elDia;
18         mes = elMes;
19         anyo = elAnyo;
20     }
21
22         // Devuelve true si dos valores son iguales
23     public boolean equals( Object lder )
24     {
25         if( ! ( lder instanceof Fecha ) )
26             return false;
27         Fecha lderFecha = ( Fecha ) lder;
28         return lderFecha.dia == dia && lderFecha.mes == mes &&
29                lderFecha.anyo == anyo;
30     }
31
32         // Conversión a String
33     public String toString( )
34     {
35         return dia + "/" + mes + "/" + anyo;
36     }
37
38         // Atributos
39     private int dia;
40     private int mes;
41     private int anyo;
42 }
```

**Figura 3.6** Una clase `Fecha` básica que ilustra el uso de constructores y los métodos `equals` y `toString`.

### 3.4.5 Métodos `static`

Un *método estático* es un método que no necesita un objeto que lo controle. El método estático más común es `main`. En las clases `String`, `Integer` y `Math` se puede encontrar otros métodos estáticos. Algunos ejemplos son `String.valueOf`, `Integer.parseInt`, `Math.sin` y `Math.max`. El acceso a los métodos estáticos utiliza las mismas reglas de visibilidad que los atributos estáticos.

*Un método estático es un método que no necesita un objeto que lo controle.*

Recuerde del Capítulo 1 que algunos atributos de la clase usan el modificador `static`. Usándolo, en conjunción con la palabra `final`, obtenemos las constantes. Sin la palabra `final`, tenemos *atributos estáticos*, que tienen otro significado distinto, estudiado en la Sección 3.6.4.

### 3.4.6 `main`

Cuando se usa el comando *java* para ejecutar el intérprete, se llama al método `main` de la clase del fichero referenciado por el comando. Por tanto, cada clase puede tener su propia función `main` sin ningún problema. Sin embargo, aunque así se puede probar la funcionalidad, colocar `main` en una clase proporciona a esta función más visibilidad que la que estaría permitida en general. Por tanto, llamadas de `main` a métodos no públicos tendrán éxito en la prueba, aunque serán ilegales en un entorno más general.

## 3.5 Paquetes

Un *paquete* se usa para organizar una colección de clases.

Los paquetes se usan para organizar una serie de clases relacionadas. Cada paquete consta de un conjunto de clases. Dos clases en un mismo paquete tienen restricciones ligeramente menores de visibilidad entre ellas que si estuvieran en paquetes distintos.

Java proporciona varios paquetes predefinidos, entre ellos `java.applet`, `java.awt`, `java.io`, `java.lang` y `java.util`. El paquete `java.lang` incluye, entre otras, las clases `Integer`, `Math`, `String` y `System`. Algunas clases del paquete `java.util` son `Date`, `Random` y `StringTokenizer`. El paquete `java.io` se usa para la E/S e incluye las clases de canales vistas en la Sección 2.6. El paquete `java.applet` incluye clases usadas para diseñar applets. El paquete `java.awt` incluye clases que forman el Abstract Window Toolkit[6], que se usa para generar interfaces gráficas de usuario.

Una clase `C` en un paquete `P` se especifica como `P.C`. Por ejemplo, podemos tener un objeto `Date` construido con la fecha y hora actuales como estado inicial usando

```
Java.util.Date hoy = new java.util.Date( );
```

Observe que incluyendo un nombre de paquete evitamos conflictos con clases nombradas de forma idéntica en otros paquetes.

### 3.5.1 La directiva `import`

La *directiva* `import` se usa para proporcionar una abreviatura de un nombre de clase totalmente cualificado.

Usar un nombre completo con el paquete y la clase puede resultar pesado. Para evitar esto, usamos la directiva `import`.

```
import NombrePaquete.NombreClase;
import NombrePaquete.*;
```

Con la primera forma, se puede usar `NombreClase` como abreviatura de un nombre de clase totalmente cualificado. Con la segunda, todas las clases del paquete se pueden abreviar con el nombre de la clase correspondiente.

[6] *N. del T.:* En español: herramienta de ventanas abstractas. En lo sucesivo se usará sin traducir, puesto que sus siglas, awt, se usan abundantemente en el texto, así como en el nombre del paquete.

Por ejemplo, con estas directivas `import`,

```
import java.util.Date;
import java.io.*;
```

podemos usar

```
Date hoy = new Date( );
FileReader elFichero = new FileReader( nombre );
```

El uso de las directivas `import` ahorra escritura. Y puesto que se ahorra más con la segunda forma, la verá usada muy a menudo. Sin embargo, las directivas `import` tienen dos desventajas. En primer lugar, la abreviatura hace difícil saber, a partir de la lectura del código, qué clase se está usando cuando hay varias directivas `import`. Además, la segunda forma puede permitir abreviaturas para clases que no se pretendía abreviar, e introducir conflictos de nombres que deberán ser resueltos escribiendo nombres de clase completamente cualificados.

El uso descuidado de las directivas `import` puede producir conflictos de nombres.

Suponga que usamos

```
import java.util.* // Paquete de librería
import util.*;     // Paquete definido por el usuario
```

con la intención de importar la clase `java.util.Random` y un paquete que hemos escrito nosotros mismos. Entonces, si tuviéramos nuestra propia clase `Date`, la directiva `import` generaría un conflicto con `java.util.Date`, de forma que para referirnos a estas clases deberíamos utilizar los nombres totalmente cualificados. Y lo que es más, si estamos usando una clase de uno de estos paquetes, no sabremos si procede del paquete de librería o de nuestro propio paquete. Habríamos evitado ese problema si hubiéramos usado la primera forma:

```
import java.util.Random;
```

Las directivas `import` deben aparecer antes del comienzo de la declaración de una clase. Vimos un ejemplo en la Figura 2.11. Además, el paquete `java.lang` se importa de forma automática. Por ello podemos usar abreviaturas como `Math.max`, `Integer.parseInt`, `System.out`, etc.

El paquete `java.lang.*` se importa automáticamente.

### 3.5.2 La instrucción `package`

Para indicar que una clase es parte de un paquete, debemos hacer dos cosas. En primer lugar, debemos incluir la *instrucción* `package` como primera línea, antes de la definición de clase. En segundo lugar, debemos colocar el código en un subdirectorio apropiado.

La *instrucción* `package` indica que una clase es parte de un paquete. Debe preceder a la definición de la clase.

En este libro, usamos los tres paquetes mostrados en la Figura 3.7. Otros programas, incluyendo los programas de prueba y las aplicaciones de la Parte III del libro, son clases aisladas que no forman parte de un paquete.

| Paquete | Uso |
|---|---|
| `EstructurasDatos` | Clases que implementan estructuras de datos y ordenación. |
| `Excepciones` | Clases de excepciones. |
| `Soporte` | Varias clases de soporte e interfaces. |

**Figura 3.7** Paquetes definidos en este libro.

En la Figura 3.8 se muestra un ejemplo de cómo se usa la instrucción `package`. Tenemos un método estático `pausaLarga` que simplemente se duerme durante mil millones de segundos (aproximadamente dos semanas). Este método es útil porque cuando algunos entornos integrados ejecutan aplicaciones de consola desde dentro de sus entornos, cierran la salida de la consola tan pronto como termina el programa. Esto puede hacer difícil ver la salida. El método `pausaLarga` mantiene la consola sin cerrar en tales casos.

```
1  package Soporte;
2
3  public class Saliendo
4  {
5      // Suspende la ejecución del programa actual durante un tiempo
6      public static void pausaLarga( )
7      {
8          try
9            { Thread.sleep( 1000000000 ); }
10         catch( InterruptedException e ) { }
11     }
12 }
```

**Figura 3.8** Una clase `Saliendo` con un único método estático, que forma parte del paquete `Soporte`.

### 3.5.3 La variable de entorno `CLASSPATH`

La variable `CLASSPATH` especifica los ficheros y directorios donde se debe buscar para encontrar las clases.

Los paquetes se buscan en los lugares indicados por la variable `CLASSPATH`. ¿Qué significa esto? A continuación se muestran dos posibles asignaciones de valores a la variable `CLASSPATH`, la primera para un sistema Windows 95, y la segunda para un sistema Unix:

```
SET CLASSPATH=.;C:\cafe\java\lib\
setenv CLASSPATH .:$HOME/java:/usr/java/lib
```

En ambos casos, la variable `CLASSPATH` lista los directorios (o ficheros comprimidos) que contienen los ficheros de las clases de los paquetes. Por ejemplo, si

su variable CLASSPATH tiene un valor incorrecto, no podrá compilar ni siquiera el programa más trivial porque no se encontrará java.lang.

Una clase de un paquete P debe estar en un directorio P, el cual se encontrará buscando en la lista de CLASSPATH. El directorio actual (directorio .) siempre está en la variable CLASSPATH, luego si va a trabajar en un único directorio principal, simplemente puede crear subdirectorios en él. Lo más probable, sin embargo, es que quiera crear un directorio java separado y después crear subdirectorios en él. Entonces tendría que aumentar la variable CLASSPATH para incluir el directorio java. En la definición Unix previa se hace esto al añadir $HOME/java a CLASSPATH. Dentro del directorio java, cree subdirectorios llamados Soporte, EstructurasDatos y Excepciones. Coloque el código de la clase Saliendo en el subdirectorio Soporte.

Una clase de un paquete P debe estar en un directorio P, el cual se encontrará buscando en la lista de CLASSPATH.

Una aplicación puede usar entonces el método pausaLarga escribiendo

```
Soporte.pausaLarga( );
```

o simplemente usando pausaLarga, si se proporciona una directiva import apropiada.

### 3.5.4 Reglas de visibilidad amistosa dentro de un paquete

Los paquetes tienen varias reglas de visibilidad importantes. En primer lugar, si no se especifica ningún modificador de visibilidad para un atributo, entonces el atributo es *amistoso*. Esto significa que solamente es visible para las demás clases del mismo paquete. Este tipo de visibilidad es mayor que la de la parte privada (que es invisible incluso para otras clases dentro del mismo paquete) pero menor que la de la parte pública (que es visible también para clases que no pertenecen a ese paquete).

Los atributos sin modificadores de visibilidad son *amistosos*, lo que significa que solamente son visibles para otras clases del mismo paquete.

En segundo lugar, solamente las clases públicas de un paquete se pueden usar fuera del paquete. Ésta es la razón por la que siempre hemos usado el cualificador public delante de las clases. Las clases no se pueden declarar con la palabra private. El acceso amistoso se extiende a las clases también. Si una clase no se declara con la palabra public, entonces solamente las demás clases del paquete pueden acceder a ella; se trata de una *clase amistosa*. En la Parte IV veremos que las clases amistosas se pueden usar sin violar por ello los principios de ocultamiento de información. De hecho hay situaciones en las que las clases amistosas pueden sernos de mucha utilidad.

Las clases amistosas son visibles solamente para otras clases del mismo paquete.

Todas la clases que no forman parte de un paquete, pero se pueden alcanzar a través de la variable CLASSPATH, se consideran parte de un mismo paquete por defecto. Como resultado el acceso amistoso se aplica entre todas ellas. Ésta es la razón por la que la visibilidad no queda afectada si se omite el modificador public en las clases que no pertenecen a ningún paquete. Sin embargo, este uso del acceso amistoso es bastante pobre. Evite el acceso amistoso excepto en casos como los que aparecen en algunas de las implementaciones de estructuras de datos de la Parte IV.

### 3.5.5 Compilación separada

Los ficheros fuente en Java se pueden compilar en cualquier orden.

Cuando un programa consta de varios ficheros `.java`, cada fichero se debe compilar por separado. Normalmente colocamos cada clase en su fichero `.java` correspondiente. Como resultado, obtenemos una colección de ficheros `.class`. Como se ha mencionado antes, cuando se escribe el comando *java* para ejecutar el intérprete, se llama al método `main` de la clase situada en el fichero al que se hace referencia en el comando. Los ficheros fuente en Java se pueden compilar en cualquier orden, siempre que los ficheros `.class` estén disponibles cuando se ejecute el intérprete.

## 3.6 Construcciones adicionales

Tres palabras clave adicionales son `this`, `instanceof` y `static`. La palabra `this` tiene varios usos en Java; dos de ellos se estudia en esta sección. La palabra `instanceof` también tiene varios usos; aquí la usaremos para asegurar que una conversión de tipo puede tener éxito. La palabra `static` tiene también varios usos. Ya hemos estudiado los *métodos estáticos*. Esta sección cubre los *atributos estáticos* y el *inicializador estático*.

### 3.6.1 La referencia `this`

`this` es una referencia al objeto actual. Se puede usar para enviar el objeto actual, como un todo, a algún otro método.

El primer uso de `this` es como referencia al objeto actual. Piense en la referencia `this` como en un dispositivo de orientación que, en cualquier instante de tiempo, le dice dónde está. Un uso importante de la referencia `this` es el manejo del caso especial de autoasignación.

Un ejemplo de ello es un programa que copia un fichero en otro. Un algoritmo normal comienza borrando el fichero destino. Si no se realiza ninguna comprobación para asegurar que los ficheros fuente y destino son de hecho distintos, entonces podría borrarse el fichero fuente —difícilmente una característica deseable. Cuando se manejan dos objetos, de forma que sobre uno de ellos se escribe un valor y del otro se lee algún valor, deberíamos comprobar primero si nos encontramos en este caso especial, que se conoce como *aliasing*.

El *aliasing* es una situación especial que aparece cuando el mismo objeto aparece con más de un rol.

Como segundo ejemplo, supongamos que tenemos una clase `Cuenta` con un método `transferenciaFinal`. Este método mueve todo el dinero de una cuenta a otra. En principio, es una rutina sencilla de escribir:

```
// Transferir todo el dinero de lder a la cuenta actual
public void transferenciaFinal( Cuenta lder )
{
    dolares += lder.dolares;
    lder.dolares = 0;
}
```

Sin embargo, considere lo que sucede si hacemos:

```
Cuenta cuenta1;
Cuenta cuenta2;
   ...
cuenta2 = cuenta1;
cuenta1.transferenciaFinal( cuenta2 );
```

Puesto que estamos transfiriendo de una cuenta a ella misma, no debería haber cambio en dicha cuenta. Sin embargo, la última instrucción en `transferenciaFinal` provoca que la cuenta se vacíe. Una forma de evitar esto es usar un test de alias:

```
// Transferir todo el dinero de lder a la cuenta actual
public void transferenciaFinal( Cuenta lder )
{
    if( this == lder )    // Test de alias
        return;
    dolares += lder.dolares;
    lder.dolares = 0;
}
```

### 3.6.2 La abreviatura `this` para constructores

Se puede usar `this` para hacer una llamada a otro constructor de la misma clase.

Muchas clases tienen varios constructores que se comportan de una forma similar. Podemos usar `this` dentro de un constructor para llamar a otro de los constructores de clase. Una alternativa al constructor de cero parámetros `Fecha` de la Figura 3.6 sería

```
public Fecha( )
{
    this( 1, 1, 1998); // Llama al constructor de tres parámetros
}
```

Son posibles usos más complicados, pero siempre la llamada a `this` debe ser la primera instrucción en el constructor; después puede aparecer otras instrucciones.

### 3.6.3 El operador `instanceof`

El operador `instanceof` se usa para comprobar si una expresión es una instancia de una clase dada.

El operador `instanceof` lleva a cabo una comprobación en tiempo de ejecución. El resultado de

```
exp instanceof NombreClase
```

es `true` si `exp` es una instancia de `NombreClase`, y `false` en caso contrario. Si `exp` es `null`, el resultado es siempre `false`. El operador `instanceof` se usa normalmente para realizar una conversión de tipo y es `true` cuando ésta puede llevarse a cabo con éxito.

### 3.6.4 Atributos estáticos

Los *atributos estáticos* son esencialmente variables globales con ámbito de clase.

Los atributos estáticos se usan cuando necesitamos que una variable esté compartida por todas las componentes de una clase. Normalmente se trata de una constante simbólica, pero no necesariamente. Cuando una variable se declara `static` en una clase, se crea una única instancia de la variable. No forma parte de ninguna instancia de la clase. En su lugar, se comporta como una variable global pero con el ámbito de la clase. En otras palabras, en la declaración

```
public class Ejemplo
{
    private int x;
    private static int y;
}
```

cada objeto `Ejemplo` almacena su propia `x`, pero hay solamente una `y` compartida.

Un uso común de un atributo estático es para representar una constante. Por ejemplo, la clase `Integer` define el valor `MAX_VALUE` como

```
public final static int MAX_VALUE = 2147483647;
```

Si una constante no fuera un atributo estático, entonces cada instancia de `Integer` tendría un atributo llamado `MAX_VALUE`, lo cual supondría un absurdo gasto de espacio. En lugar de ello, hay una única variable llamada `MAX_VALUE`. Cualquiera de los métodos de `Integer` puede acceder a ella usando el identificador `MAX_VALUE`. También se puede acceder a ella desde un objeto `Integer` usando `Obj.MAX_VALUE`, como con cualquier otro atributo. Observe que esto está permitido solamente porque `MAX_VALUE` es pública. Finalmente, se puede acceder a `MAX_VALUE` usando el nombre de la clase, como en `Integer.MAX_VALUE` (de nuevo permitido por ser pública). Esto no estaría permitido si se tratara de un atributo no estático.

También sin el cualificador `final`, los atributos estáticos son útiles. Supongamos que queremos llevar la cuenta del número de objetos `CeldaEntero` que hemos construido. Lo que necesitamos es una variable estática. Declaramos en la clase `CeldaEntero`

```
private static int instanciasActivas = 0;
```

Entonces podríamos incrementar `instanciasActivas` en el constructor. Sin el modificador `static`, tendríamos un comportamiento incorrecto, puesto que cada objeto `CeldaEntero` tendría su propio atributo, por lo que `instanciasActivas` se incrementaría de 0 a 1 en el constructor.

Observe que dado que un atributo estático no tiene objeto que lo controle, sino que es compartido por todas las instancias de la clase, cualquier método estático de la clase puede acceder a él y modificarlo, si la visibilidad lo permite. Un método estático solamente puede acceder a un atributo no estático, que es parte de cada instancia de la clase, si se proporciona un objeto controlador.

### 3.6.5 Inicializadores estáticos

Un *inicializador estático* es un bloque de código que se usa para inicializar atributos estáticos.

Los atributos estáticos se inicializan cuando se carga la clase. A veces la inicialización es compleja. Por ejemplo, supongamos que necesitamos un vector estático que almacena las raíces cuadradas de los primeros 100 enteros. Sería deseable que esos valores se calcularan automáticamente. Una posibilidad es proporcionar un método estático y obligar al programador a llamarlo antes de usar el vector.

Una alternativa es usar un *inicializador estático*. En la Figura 3.9 se muestra un ejemplo, donde el inicializador se extiende de la línea 5 a la 9. La forma más sencilla de inicializador estático consiste en colocar el código de la inicialización de los atributos estáticos en un bloque precedido por la palabra clave `static`. El inicializador estático debe aparecer a continuación de la declaración del atributo estático.

```
1  public class Raices
2  {
3      private static double raicesCuadradas[ ] = new double[ 100 ];
4
5      static
6      {
7          for( int i = 0; i < raicesCuadradas.length; i++ )
8              raicesCuadradas[ i ] = Math.sqrt( ( double ) i );
9      }
10
11     // Resto de la clase
12 }
```

**Figura 3.9** Ejemplo de un inicializador estático.

## Resumen

En este capítulo se han descrito las clases y los paquetes de Java. La clase es el mecanismo de Java utilizado para crear nuevos tipos referencia; los paquetes se usan para agrupar clases similares. Para cada clase podemos

- definir la construcción de objetos,
- proporcionar ocultamiento de información y atomicidad, y
- definir métodos para manipular objetos.

Las clases constan de dos partes: la especificación y la implementación. La especificación le dice al usuario de la clase lo que hace la clase; la implementación lo hace. La implementación frecuentemente contiene código de propietario y en algunos casos se distribuye solamente como un fichero `.class`. Sin embargo, la especificación es de conocimiento público. En Java, usando *javadoc* se puede generar una especificación que lista los métodos de la clase a partir de la implementación.

El ocultamiento de información se puede lograr usando la directiva `private`. La inicialización de los objetos está controlada por las funciones constructoras, y las componentes del objeto se pueden examinar y modificar mediante funciones de acceso y funciones modificadoras, respectivamente. Otros métodos auxiliares, como `toString` y `equals`, también pueden ser escritos.

Las características estudiadas en este capítulo implementan los aspectos fundamentales de la programación orientada a objetos. En el siguiente capítulo se estudiará la herencia, la cual es otro de los pilares de la programación orientada a objetos.

## Elementos del juego

**acceso amistoso** Las componentes que no tengan modificadores de visibilidad solamente podrán ser accedidas desde los métodos de las clases del mismo paquete.

**aliasing** Caso especial en el que el mismo objeto aparece con más de un rol.

**atributo** Componente de los objetos de una clase que almacena datos.

**atributo estático** Esencialmente es una variable global con ámbito de clase.

**clase** Consta de atributos y métodos que se aplican a instancias de la clase.

**clase amistosa** Clase no pública a la que solamente pueden acceder otras clases del mismo paquete.

**compilación separada** Cada clase se compila por separado, en cualquier orden.

**constructor** Nos dice cómo se declara e inicializa un objeto. El constructor por defecto es una inicialización por defecto componente a componente, tal que los atributos de tipo primitivo se inicializan a cero y los atributos referencia a `null`.

**directiva `import`** Se usa para proporcionar abreviaturas de nombres de clase totalmente cualificados.

**encapsulación** Agrupación de datos y operaciones que se aplican sobre ellos para formar un agregado, a la vez que se oculta la implementación del agregado.

**especificación de una clase** Describe su funcionalidad, pero no la implementación.

**implementación** Representa los detalles internos de cómo se satisfacen las especificaciones. En lo que respecta al usuario, la implementación no es importante.

**inicializador estático** Bloque de código que se usa para inicializar los atributos estáticos.

**instrucción `package`** Indica que una clase es miembro de un paquete. Debe preceder a la definición de la clase.

***javadoc*** Genera automáticamente documentación para las clases.

**llamada al constructor `this`** Usado para hacer una llamada a otro constructor de la misma clase.

**marca *javadoc*** Entre ellas tenemos `@author`, `@param`, `@return` y `@exception`. Se usan dentro de comentarios *javadoc*.

**método** Función proporcionada como componente adicional que, si no es estática, opera sobre instancias de la clase.

**método de acceso** Método que examina un objeto pero no cambia su estado.

**método `equals`** Se puede implementar para comprobar si dos referencias describen el mismo valor. El parámetro formal es siempre de tipo `Object`.

**método `toString`** Devuelve un valor del tipo `String` basado en el estado del objeto.

**modificador** Método que cambia el estado de un objeto.

**objeto** Entidad con estructura y estado, para el que están definidas operaciones que pueden acceder o manipular el estado.

**ocultamiento de información** Convierte en inaccesibles los detalles de implementación, incluyendo las componentes de un objeto.

**operador `instanceof`** Comprueba si una expresión es una instancia de una clase.

**paquete** Usado para organizar una colección de clases.

**privado** Componente no visible para los métodos que no pertenecen a la clase.

**programación basada en objetos** Usa las características de la encapsulación y el ocultamiento de información de los objetos pero no usa la herencia.
**programación orientada a objetos** Se distingue de la programación basada en objetos en que hace uso de la herencia para formar jerarquías de clases.
**público** Componente que es visible para los métodos que no pertenecen a la clase.
**referencia `this`** Referencia al objeto actual. Se puede usar para enviar el objeto actual, como un todo, a otro método.
**unidad atómica** En relación a un objeto, sus partes no pueden ser diseccionadas por los usuarios generales del objeto.
**variable `CLASSPATH`** Especifica los directorios y ficheros donde se debe buscar para encontrar las clases.

## Errores comunes

1. No se puede acceder a las componentes privadas desde fuera de la clase. Recuerde que, por defecto, las componentes de la clase son amistosas, por lo que son visibles solamente dentro del paquete.
2. Use `public class` en lugar de `class` a menos que esté escribiendo una clase de ayuda que luego vaya a deshechar.
3. El parámetro formal de `equals` debe ser del tipo `Object`. En caso contrario, aunque el programa se compilaría correctamente, hay casos en los que se usará un `equals` por defecto (que siempre devuelve `false`).
4. Los métodos estáticos no pueden acceder a componentes no estáticas que no tengan un objeto controlador.
5. Un error común es olvidar el `.*` en la segunda forma de escribir una directiva `import`.
6. Las clases que forman parte de un paquete deben colocarse en un directorio que se llame igual que el paquete y que debe poderse alcanzar a través de la variable `CLASSPATH`.
7. `this` es una referencia final y no se puede modificar.

## En Internet

Los siguientes ficheros están disponibles:

| | |
|---|---|
| **Date.java** | Traducido por `Fecha.java`, contiene la clase `Date`, traducida como `Fecha` en la Figura 3.6. Se encuentra en el directorio **Chapter03**. |
| **Exiting.java** | Traducido por `Saliendo.java`, contiene el método `longPause`, traducido por `pausaLarga` en la Figura 3.8. Se encuentra en el directorio **Supporting**, traducido por `Soporte`. |
| **IntCell.java** | Traducido por `CeldaEntero.java`, contiene la clase `IntCell`, traducida por `CeldaEntero` en la Figura 3.4. Se encuentra en el directorio **Chapter03**. También se encuentra allí la salida producida por *javadoc*. |
| **Squares.java** | Traducida por `Raices.java`, contiene el inicializador estático de la Figura 3.9. Se encuentra en el directorio **Chapter03**. |

**TestIntCell.java** Traducido por `TestCeldaEntero.java`, contiene un `main` que prueba `IntCell` (`CeldaEntero`) que se muestra en la Figura 3.3. Se encuentra en el directorio **Chapter03**.

## Ejercicios

### Cuestiones breves

**3.1.** ¿Qué es el ocultamiento de información? ¿Qué es la encapsulación? ¿Cómo soporta Java estos conceptos?
**3.2.** Explique las secciones pública y privada de una clase.
**3.3.** Describa la función del constructor.
**3.4.** ¿Qué sucede si una clase no proporciona constructores?
**3.5.** ¿Qué es el acceso amistoso en un paquete?
**3.6.** ¿Cómo se lleva a cabo la salida en una clase `NombreClase`?
**3.7.** Describa las dos formas de escribir una directiva `import` que permitan que la clase `pausaLarga` sea usada sin proporcionar el nombre del paquete `Soporte`.

### Problemas teóricos

**3.8.** ¿Por qué no se puede declarar como privadas las clases?
**3.9.** Supongamos que el método `main` de la Figura 3.3 formaba parte de la clase `CeldaEntero`.

*a*) ¿Todavía funcionaría el programa?
*b*) ¿Se podría quitar el comentario a la línea comentada sin generar un error?

### Problemas prácticos

**3.10.** Un cerrojo con combinación tiene las siguientes propiedades básicas: la combinación (una secuencia de tres números) está oculta; el cerrojo se puede abrir proporcionando la combinación; y la combinación se puede cambiar, pero solamente por alguien que conoce la combinación actual. Diseñe una clase con métodos públicos `abrir` y `cambiarComb`, y atributos privados para almacenar la combinación. La combinación debería asignarse en el constructor. Deshabilite la copia de cerrojos con combinación.

### Prácticas de programación

**3.11.** Escriba una clase que implemente los números racionales. Sus atributos serán dos variables de tipo `long` que almacenarán el numerador y denominador. Almacene los números racionales en forma simplificada, con el denominador siempre no negativo. Proporcione un conjunto razonable de constructores, los métodos `sumar`, `restar`, `multiplicar` y `dividir`,

así como `toString`, `equals` y `compareTo` (que se comporta como el de la clase `String`). Asegúrese de que `toString` trata correctamente el caso en que el denominador es cero.

**3.12.** Implemente una clase `Fecha` simple. Debería poder representar cualquier fecha desde el 1 de enero de 1.800 al 31 de diciembre del 2.500; restar dos fechas; incrementar una fecha en un número de días; y comparar dos fechas usando `equals` y `compareTo`. Una `Fecha` se representa internamente como el número de días a partir de una cierta fecha de inicio, que aquí es el comienzo de 1.800. Esto convierte en triviales todos los métodos excepto `toString`.

La regla para los años bisiestos es que un año es bisiesto si es divisible por cuatro y no es divisible por 100 a menos que también lo sea por 400. Por tanto 1.800, 1.900 y 2.100 no son años bisiestos, pero el 2.000 sí lo es. El constructor debe comprobar la validez de la fecha, al igual que debe hacerlo `toString`. La `Fecha` podría no ser válida si un incremento o decremento provocan que se salga fuera del rango.

Una vez ha decidido la especificación, puede hacer una implementación. La parte difícil es la conversión entre las representaciones interna y externa de la fecha. A continuación se explica un posible algoritmo.

Cree dos atributos estáticos de tipo vector. El primer vector, `diasHastaPrimeroDeMes`, contendrá el número de días hasta el primero de cada mes en un año no bisiesto. Luego contiene 0, 31, 59, 90 y así sucesivamente. El segundo vector, `diasHasta1Enero`, contendrá el número de días hasta el primero de cada año, comenzando con `primerAnyo`. Luego contiene 0, 365, 730, 1.095, 1.460, 1.826 y así sucesivamente porque 1.800 no es un año bisiesto, pero 1.804 sí lo es. Su programa debería inicializar este vector usando un inicializador estático. Entonces puede usar el vector para convertir la representación interna en la externa.

**3.13.** Implemente una clase `Complejos` de números complejos. Recuerde que un número complejo consta de una parte real y una imaginaria. Soporte las mismas operaciones que la clase de los racionales, cuando tengan sentido (por ejemplo `compareTo` no tiene sentido). Añada métodos de acceso para extraer las partes real e imaginaria.

**3.14.** Implemente una clase `TipoEntero` que soporte un conjunto razonable de constructores, `sumar`, `restar`, `multiplicar`, `dividir`, `equals`, `compareTo` y `toString`. Mantenga `TipoEntero` como un vector lo suficientemente largo. Para esta clase, la operación difícil es la división, seguida de cerca por la multiplicación.

## Bibliografía

Se puede encontrar más información en las referencias, al final del Capítulo 1.