CAPITULO 5. COMPORTAMIENTO MAS SOFISTICADO

5.1 Documentación de las clases de biblioteca

La **biblioteca de clases estándar de Java** contiene muchas clases que son muy útiles. Es importante saber cómo se usa la biblioteca.

Otro factor importante es la posibilidad de usar esas clases sin mirar su código fuente, sin ser necesario saber cómo fueron implementadas para usar su funcionalidad. Todo lo que necesitamos saber es el nombre de la clase, los nombres de los métodos, los parámetros y los tipos de retorno de los métodos y saber qué hacen estos métodos.

La misma cuestión es cierta para otras clases no estándar en proyectos de software grandes.

5.3 Lectura de documentación de clase

5.3.1 Comparar interfaz e implementación

La documentación incluye diferentes piezas de información, entre otras: el nombre de la clase, una descripción general del propósito de la clase, una lista de los constructores y los métodos de la clase, los parámetros y los tipos de retorno de cada constructor y de cada método y una descripción del propósito de cada constructor y cada método. Toda esta información reunida recibe el nombre de *interfaz* de la clase.

La **interfaz** de una clase describe lo que es capaz de hacer dicha clase y la manera en que se puede usar sin mostrar su implementación. La interfaz no muestra el código con que está implementada la clase. Estamos nuevamente frente a la abstracción en acción.

El código completo que define una clase se denomina la **implementación** de dicha clase.

También se utiliza la terminología interfaz referida a métodos individuales. La interfaz de un método consiste en su *signatura* y un comentario. La signatura de un método incluye, en este orden: un modificador de acceso, (public, private,…), el tipo de retorno del método, el nombre del método y una lista de parámetros. La interfaz de un método proporciona todos los elementos necesarios para saber cómo usarlo.

5.3.2 Usar métodos de clases de biblioteca

Se dice que un **objeto es inmutable** si su contenido o su estado no puede ser cambiado una vez que se ha creado. Los objetos String son un ejemplo de objetos inmutables.

5.3.3 Comprobar la igualdad de cadenas

Con variables de objeto, el operador (==) evalúa si ambos operandos hacen referencia al mismo objeto, no si sus valores son iguales. Esta es una diferencia importante. La solución para este problema es usar el método equals, que deben tener todas las clases, ya que al menos lo heredan de la Clase Object que es superclase de todas las demás.

5.4 Agregar comportamiento aleatorio

Aleatorio y pseudo-aleatorio: Las computadoras operan de una manera bien definida y determinística que se apoya en el hecho de que todo cálculo es predecible y repetible, en consecuencia existe poco espacio para un comportamiento realmente aleatorio. Se han propuesto muchos algoritmos para producir secuencias semejantes a los números aleatorios. Estos números no son típicamente números aleatorios verdaderos, aunque siguen reglas muy complicadas. Estos números se conocen como números *pseudo-aleatorios*. En Java la generación de números pseudoaleatorios ha sido implementada en una clase de la biblioteca.

5.4.1 La clase Random

La biblioteca de clases de Java contiene una clase de nombre Random capaz de generar datos pseudoaleatorios. Ejemplo de uso:

```
Random miObjetoRandom;
miObjetoRandom = new Random();
int indice = miObjetoRandom.nextInt () ;
System.out.println(indice);
```

5.4.2 Números aleatorios en un rango limitado

Lo más frecuente es que necesitemos números aleatorios dentro de un rango determinado. La clase Random posee el método nextInt, pero con un parámetro para especificar el rango de números que queremos usar.

int nextInt(int n)
Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.

5.4.4 Lectura de documentación de clases parametrizadas

En la ayuda, algunos nombres de las clases que aparecen en la lista de la documentación tienen un formato ligeramente diferente, tal es el caso de ArrayList<E>. Las clases similares a éstas se denominan «clases parametrizadas» o «clases genéricas». La información contenida entre los símbolos de menor y de mayor nos dice que, cuando usemos estas clases deberemos suministrar uno o más nombres de tipos entre dichos símbolos, para completar la definición.

Por lo tanto, si busca en la lista de métodos de ArrayList<E> verá métodos tales como: boolean add (E o) o E get (int index)

Estas signaturas nos indican que el tipo de objetos que podemos agregar u obtener de un ArrayList depende del tipo usado para parametrizarlo.

5.5 Paquetes y la sentencia import

Las clases de Java almacenadas en la biblioteca de clases no están disponibles automáticamente para su uso, tal como las otras clases del proyecto actual. Para poder disponer de ellas, debemos explicitar en nuestro código que queremos usar una clase de la biblioteca. Esta acción se denomina *importación de la clase* y se implementa mediante la sentencia *import*. La sentencia *import* tiene la forma general: import *nombre -de -clase -calificado;*

Java utiliza *paquetes* (*packages*) para acomodar las clases de la biblioteca en grupos que permanecen juntos. Los paquetes pueden estar anidados, es decir, los paquetes pueden contener otros paquetes.

El *nombre completo* o *nombre calificado* de una clase es el nombre de su paquete, seguido por un punto y por el nombre de la clase, por ejemplo: java.util.ArrayList y java.util.Random.

Java también nos permite importar paquetes completos con sentencias de la forma import nombre -del - paquete. *;, por lo que la siguiente sentencia importaría todas las clases del paquete java. util: import java.util.*;

Existe una excepción a esta regla: algunas clases se usan tan frecuentemente que casi todas las clases deberían importarlas. Estas clases se han ubicado en el paquete java.lang y este paquete se importa automáticamente dentro de cada clase. La clase String es un ejemplo de una clase ubicada en java.lang.

5.6 Usar mapas para las asociaciones

5.6.1 Concepto de mapa

Un **mapa** es una colección que almacena pares clave/valor como entradas. Los valores se pueden buscar suministrando la clave. Un mapa puede almacenar un número flexible de entradas. En un *Map* cada entrada no es un único objeto sino un *par* de objetos. Este par está compuesto por un objeto *clave* y un objeto *valor*.

En lugar de buscar las entradas en esta colección mediante un índice entero usamos el objeto *clave* para buscar el objeto *valor*. Un mapa puede organizarse de manera tal que resulte fácil buscar en él un valor para una clave. En consecuencia, los mapas son ideales para una única forma de búsqueda, en la que conocemos la clave a buscar y necesitamos conocer solamente el valor asociado a esta clave.

5.6.2 Usar un HashMap

Un HashMap es una implementación particular de un Map. Los métodos más importantes de la clase HashMap son put y get. El método put inserta una entrada en el mapa y el método get recupera el valor correspondiente a una clsve determi nada.

```
HashMap<String , String> agenda = new HashMap<String ,String) ();
agenda.put ("Lisa Jones" , (402) 4536 4674 " );
String numero = agenda.get ( "Lisa Jones");
```

Si a un HashMap se le proporciona un par clave, valor, con una clave ya existente se reemplaza el valor anterior de esa clave. Si se intenta acceder al valor correspondiente a una clave inexistente, devuelve null; también se pueden almacenar valores nulos, o incluso guardar una clave null, por lo que existe el método boolean containsKey(E clave), que nos informa si un mapa contiene una determinada clave. El método int size() devuelve el número de pares clave, valor almacenados.

5.7 **Usar conjuntos**

La biblioteca estándar de Java incluye diferentes variantes de conjuntos, implementados en clases diferentes. Una de ellas es HashSet. Un **conjunto** es una colección que almacena cada elemento individual una sola vez como máximo. No mantiene un orden específico. En un conjunto, el ingresar un elemento por segunda vez simplemente no tiene ningún efecto.

```
import java.util.HashSet;
HashSet<String> miConjunto
miConjunto.add( "uno");
miConjunto.add("dos");

for(String miCon junto) {
   Hacer algo con cada elemento
}
```

List, Map y Set. Cuando tratamos de comprender la forma en que se usan las diferentes clases de colecciones, la segunda parte del nombre es la mejor indicación de los datos que almacenan, y la primera palabra describe la forma en que se almacenan. Generalmente estamos más interesados en el "qué" (la segunda parte) antes que en el "cómo". De modo que un TreeSet debiera usarse de manera similar a un HashSet, mientras que un TreeMap debiera usarse de manera similar a un HashMap.

5.8 Dividir cadenas String.split

El método split puede dividir una cadena en distintas subcadenas y las devuelve en un array de cadenas. El parámetro del método split establece la clase de caracteres de la cadena original que producirá la división en palabras.

5.10 Escribir documentación de clase

La **documentación** de una clase debiera ser suficientemente detallada como para que otros programadores puedan usarla sin tener que leer su implementación. El sistema Java incluye una herramienta denominada javadoc que se puede utilizar para generar la interfaz que describa nuestros archivos fuente.

5.10.2 Elementos de la documentación de una clase

La documentación de una clase debe incluir como mínimo:

- el nombre de la clase;
- un comentario que describa el propósito general y las características de la clase;
- un número de versión;
- el nombre del autor (o autores);
- la documentación de cada constructor y de cada método.

La documentación de cada constructor y de cada método debe incluir:

- el nombre del método;
- el tipo de retorno;
- los nombres y tipos de los parámetros;
- una descripción del propósito y de la función del método;
- una descripción de cada parámetro;
- una descripción del valor que devuelve.

Además, cada proyecto debiera tener un comentario general, frecuentemente guardado en un archivo de nombre "Leeme" o "ReadMe".

En Java, los comentarios de estilo javadoc se escriben con un símbolo especial de al comienzo (/**):

```
/**
Este es un comentario javadoc
*/
```

El símbolo de inicio de un comentario debe tener dos asteriscos para que javadoc lo reconozca. Este tipo de comentario, ubicado inmediatamente antes de la declaración de clase es interpretado como un comentario de clase. Si el comentario está ubicado directamente arriba de la signatura de un método, es considerado como un comentario de método.

En Java y mediante javadoc, se dispone de varios símbolos especiales para dar formato a la documentación. Estos símbolos comienzan con el símbolo @ e incluyen entre otros:

@version
@autor
@param
@return

5.11 Comparar público con privado

Los modificadores de acceso son las palabras clave como public o private que aparecen al comienzo de las declaraciones de campos y de las signaturas de los métodos. Los **modificadores de acceso** definen la visibilidad de un campo, de un constructor o de un método. Los elementos públicos son accesibles dentro de la misma clase o fuera de ella: los elementos privados son accesibles solamente dentro de la misma clase.

Los campos, los constructores y los métodos pueden ser públicos o privados; a la mayoría de los campos suelen ser privados y la mayoría de los constructores y de los métodos suelen ser públicos.

La interfaz de una clase es el conjunto de detalles que necesita ver otro programador que utilice dicha clase. La interfaz puede verse como la *parte pública* de una clase.

La implementación es la sección de una clase que define precisamente cómo funciona la clase. También nos referimos a la implementación como la *parte privada* de una clase.

El usuario de una clase no necesita conocer su implementación. En realidad, existen buenas razones para *evitar que un usuario conozca* la implementación (o por lo menos, que use ese conocimiento). Este principio se denomina *ocultamiento de la información.*

La palabra clave *public* declara que un elemento de una clase (un campo o un método) forma parte de la interfaz (es decir, es visible públicamente); la palabra clave *private* declara que un elemento es parte de la implementación (es decir, permanece oculto para los accesos externos).

5.11.1 Ocultamiento de la información

El **ocultamiento de la información** es un principio que establece que los detalles internos de implementación de una clase deben permanecer ocultos para las otras clases. Asegura una mejor modularización de la aplicación.

En muchos lenguajes de programación orientados a objetos, el interior de una clase (su implementación) permanece oculta para las otras clases. Hay dos aspectos en este punto: primero, un programador que hace uso de una clase *no necesita conocer* su interior; segundo, a un usuario *no se le permite conocer* los detalles internos.

El primer principio, *necesidad de conocer*, tiene que ver con la abstracción y la modularización. Si necesitáramos conocer todos los detalles internos de todas las clases que queremos usar, no terminaríamos nunca de implementar sistemas grandes.

El segundo principio, *no se permite conocer*, es diferente. También tiene que ver con la modularización pero en un contexto diferente. El lenguaje de programación no permite el acceso a una sección privada de una clase mediante sentencias en otra clase. Esto asegura que una clase no dependa de cómo está implementada exactamente otra clase. Este punto es muy importante para el trabajo de mantenimiento. Una tarea muy común de mantenimiento de un programa es la modificación o extensión de la implementación de una clase para mejorarlo o para solucionar defectos. Idealmente, las modificaciones en la implementación de una clase no debieran generar la necesidad de cambiar también las otras clases. Esta característica se conoce como *acoplamiento:* si se cambia una parte de un programa no debiera ser necesario hacer cambios en otras partes del programa, cuestión que se conoce como alto y bajo acoplamiento. El bajo acoplamiento es bueno porque hace que el trabajo de mantenimiento del programador sea mucho más fácil: en lugar de comprender y modificar muchas clases, deberá comprender y modificar sólo una clase.

Para ser más precisos, la regla de que a un usuario «no se le debe permitir conocer el interior de una clase» no se refiere al programador de otras clases sino a la clase en sí misma. Generalmente, no es un problema el hecho de que un programador conozca los detalles de implementación, pero una clase no debiera “conocer” (o depender) de los detalles internos de otras clases.

La palabra clave *private* refuerza el ocultamiento de la información al impedir el acceso a esta parte de la clase desde otras clases. Esto asegura el bajo acoplamiento y hace que la aplicación resulte más modular y más fácil de mantener.

5.11.2 Métodos privados y campos públicos

En general los métodos de las clases son públicos, aunque no siempre es así. A veces existen métodos de apoyo dentro de una clase que son usados por otros métodos de dicha clase; normalmente esas subtareas no tienen la finalidad de ser invocadas directamente desde el exterior de la clase pero se las ubica como métodos separados con la intencionalidad de lograr que la implementación de una clase sea más fácil de leer. En este caso, tales métodos deben ser privados. Otra buena razón para tener un método privado es cuando una tarea necesita ser usada (como una subtarea) en varios métodos de una clase. En lugar de escribir el código varias veces, podemos escribirlo una única vez en un solo método privado y luego invocar este método desde diferentes lugares de la clase.

En Java, los campos también pueden ser declarados privados o públicos. La declaración de los campos como públicos rompe con el principio de ocultamiento de la información. Hace que una clase que depende de esa información sea vulnerable a operaciones incorrectas, si se modifica la implementación. Una razón más para mantener los campos como privados reside en que permiten que un objeto crezca manteniendo el control sobre su estado. Si el acceso a los campos privados se canaliza a través de métodos de acceso y de modificación, entonces un objeto tiene la habilidad de asegurar que el campo nunca se configura con un valor que resulte inconsistente con su estado. Este nivel de integridad no es posible si los campos son públicos. Abreviando, los campos debieran ser siempre privados.

5.13 Variables de clase y constantes

5.13. La palabra clave static

La palabra clave static se usa en la sintaxis de Java para definir variables de clase.

```
public class MiClase
{
<acceso> static <tipo> <identificador>;
…
}
```

Las clases pueden tener campos: estos campos se conocen como **variables de clase** o **variables estáticas**. En todo momento, existe exactamente una copia de una variable de clase, independientemente del número de instancias que se hayan creado de dicha clase.

El código de la clase puede acceder (leer y asignar) a esta clase de variable de la misma forma en que accede a las variables de instancia. Se puede acceder a la variable de clase desde cualquiera de las instancias de la clase; como resultado, los objetos comparten esta variable.

Java también soporta *métodos de clase*, es decir métodos a los que se accede directamente desde la clase y que no necesitan una instancia para ser llamados.

5.13.2 Constantes

Un uso frecuente de la palabra clave static se produce en la declaración de *constantes*. Las constantes son similares a las variables pero no pueden cambiar su valor durante la ejecución de una aplicación. En Java, las constantes se definen con la palabra clave *final*:

```
private final int TOPE = 10;
```

En esta sentencia definimos una constante de nombre TOPE con el valor 10. Observamos que las declaraciones de constantes son similares a las declaraciones de campos pero con dos diferencias:

- deben incluir la palabra clave final antes del nombre del tipo y
- deben ser inicializadas con un valor en el momento de su declaración (o en los constructores de la clase).

Cualquier intento posterior de cambiar un campo constante dará por resultado un mensaje de error en tiempo de compilación. Por convención, las constantes se escriben frecuentemente con letras mayúsculas.

En la práctica, es muy frecuente el caso en que las constantes se relacionen con todas las instancias de una clase. En esta situación declaramos *constantes de clase*. Las constantes de clase son campos de clase constantes. Se declaran usando una combinación de las palabras clave *static* y *final*.

```
private static final int TOPE = 10;
```