Objetos, clases y métodos. Parte II

1 Encapsulación

En el paradigma de la programación orientada a objetos (POO), la encapsulación se refiere a la posibilidad de utilizar una clase sabiendo lo básico para poderla usar y prescindir de sus detalles de implementación. Consiste en agrupar, siguiendo un proceso de abstracción, todos los datos (es decir, atributos) y operaciones/acciones (es decir, métodos) relacionados en una misma clase de manera que sea sencillo reutilizar la clase y ocultar los detalles (es decir, atributos y métodos).

La encapsulación lleva consigo otras características de la POO, como son la **reutilización de código** y la **ocultación de atributos y métodos**.

2 Reutilización de código

La encapsulación permite que, al tener todos los atributos y métodos que están relacionados entre sí centralizados en una clase, sea fácil poder reaprovechar este código en diferentes programas simplemente instanciando un objeto de dicha clase.

No nos importa cómo está implementada la clase por dentro (por ejemplo, cuántos bucles hay, si el código está optimizado o no, etc.), solo nos interesa saber qué nos permite hacer y cómo se utiliza. Suponemos que la clase está bien construida y sus métodos funcionan correctamente. Si esto depende de nosotros, hemos de ser precisos en la programación de los métodos para asegurar su correcto funcionamiento.

Por ejemplo, si tenemos una clase llamada *SortedArrayInteger*, nos interesa saber que tiene cinco métodos que realizan lo siguiente:

1) añadir un elemento (que será un entero) en una posición dada,
2) eliminar el elemento de una posición dada,
3) consultar el elemento ubicado en una posición dada,
4) decirnos el número de elementos que hay en el *array*, y
5) devolver el *array* ordenado ascendente o descendentemente según el valor del parámetro *boolean* que se le pasa. Cómo está implementada la ordenación (bucles, condicionales, etc.), no nos interesa. Si la clase *SortedArrayInteger* funciona bien, la podremos usar en cualquier programa. Es decir, la podremos reutilizar.

En el diseño de una clase hay que pensar muy bien qué atributos y métodos va a tener la clase para cubrir necesidades actuales y dejar la puerta abierta para futuros usos o mejoras. Por ejemplo, si está previsto que la clase que se va a diseñar se pueda usar más allá del problema que se está resolviendo en ese momento es decir, pensamos que se va a reutilizar, debemos tener una mayor capacidad de abstracción.

3 Ocultación

Cuanto menos información se dé de una clase (es decir, cuanto más pequeña es la parte visible de una clase para las otras clases que pueden interactuar con ella), menos probable es que se den problemas y más sencillo es asegurar ciertos factores de calidad como la reusabilidad, la portabilidad y la facilidad de uso de las clases.

La encapsulación garantiza la integridad de los datos que contiene el objeto mediante el mecanismo de ocultación. La **ocultación** de información es un mecanismo muy útil y una consecuencia directa de la encapsulación.

De manera informal, podemos decir que el mecanismo de ocultación permite ocultar todo aquello que no interesa que se sepa o se tenga acceso.

Al concepto de ***ocultación*** también nos podemos referir como "**visibilidad** de los miembros de una clase (atributos y métodos)" o "**restricción de acceso** a atributos y métodos".

El programador que utilice la clase debe conocer y poder acceder directamente a los atributos y métodos estrictamente necesarios, nada más. Así se evita el acceso a los miembros de la clase por cualquier otro medio distinto a los especificados.

Ejemplo:

Imaginemos que queremos implementar el método *hablar(String texto)* de la clase *Persona*. Puede que sea un método difícil de codificar y que sea conveniente dividir su implementación en diferentes métodos más simples. Estos nuevos métodos en los que está descompuesto el método *hablar(String texto)*no tienen por qué ser conocidos por el programador que usará la clase *Persona*, es decir, se los podríamos ocultar y no cambiaría nada para él. Usando la terminología de la programación orientada a objetos diremos que el método *hablar(String texto)* será *público*, mientras que el resto de métodos que utiliza serán *privados.*

3.1 Niveles de ocultación

Cuando hablamos de **visibilidad/ocultación** de un atributo o método de una clase, hay que entenderla en relación con el **resto de clases**.

Tiene que quedar claro que **una clase siempre puede acceder a todos los atributos y métodos definidos en ella**.

Otra idea importante es que la **interacción entre dos clases se hace mediante la interacción entre objetos de esas dos clases**. Esta interacción se hace mediante **mensajes**.

La encapsulación define los niveles de acceso para los miembros de una clase. Podemos restringir el acceso a los atributos y/o a los métodos en diferente medida. En general, los lenguajes de POO definen tres niveles de acceso:

1) **Público (*public*)**: cuando un atributo y/o método es público en una clase, cualquier otra clase puede acceder a él directamente y utilizarlo. Conviene que los atributos no sean públicos.
2) **Protegido (*protected*)**: cuando un atributo y/o método es protegido en una clase, cualquier otra clase hija (o subclase) de esa clase puede acceder a él y utilizarlo. Es decir, una clase que hereda puede acceder y usar atributos y métodos que han sido declarados como "protegidos" en la clase padre.
3) **Privado (*private*)**: cuando un atributo y/o método es privado en una clase, ninguna otra clase puede acceder a él. Sólo puede ser accedido desde dentro de la propia clase que lo define. Desde fuera de la clase a través de los métodos ***getters***.

Hay lenguajes que definen más niveles de acceso. Por ejemplo, C# define el nivel *internal* y Java define el nivel *package.*

Si no indicamos la visibilidad de un miembro de la clase, el compilador da una visibilidad por defecto a los atributos y métodos. En Java, esta visibilidad por defecto es ***package****:* cuando un atributo y/o método tiene este nivel de acceso sólo se puede usar dentro del paquete en el que está declarado.

Niveles de acceso definidos en Java:

Nivel de acceso / Acceso desde	La misma clase	Package	Subclase	Otros / mundo
Public	x	x	x	x
Protected	x	x	x	
Default (package)	x	x		
Prívate	x			

4 Miembros de objeto y miembros de clase

Como vemos en los ejemplos, cada instancia u objeto creado puede hacer uso de los métodos de la clase, pero el comportamiento de cada método dependerá del estado de la instancia.

Asimismo, los atributos de cada instancia son los de la clase *Persona*, pero sus valores son los asignados a la instancia. Por ello, se suele decir que al instanciar una clase, se están creando atributos y métodos de la instancia u objeto.

Cada objeto tiene su propio espacio de memoria donde guarda sus datos o propiedades (los valores concretos de sus atributos). Pero de las funciones miembro (métodos) sólo existe una copia para todos los objetos que pertenecen a una determinada clase.

Sin embargo, si un **atributo** de una determinada clase se declara **`static`**, significa que sólo existirá en memoria una copia de ese dato, la cual será compartida por todos los objetos de esa clase. Además este dato existe aunque no se haya declarado ninguna referencia ni instanciado ningún objeto de la clase.

Un método declarado **`static`** no es un método del objeto sino de la clase. Esto permite que se pueda usar dicho método sin tener declarado ningún objeto. La llamada a un método **`static`** se puede hacer poniendo:

```
NombreClase. MetodoEstatico();
```

O:

```
NombreObjeto. MetodoEstatico();
```

Aunque la segunda forma de invocar al método **`static`** puede resultar engañosa, ya que su actuación no es sobre el objeto particular.

Dentro de un **método estático** sólo se pueden usar los **miembros** de la clase que sean **estáticos**. En cambio, **un atributo estático** puede ser **accedido** tanto **desde** un **método estático** como **no estático**.

Así pues, a los atributos y métodos que van precedidos por la palabra *static*, además de llamarles atributos y métodos estáticos, se les denomina también **elementos de la clase**, no de la instancia, puesto que no pertenecen a un objeto (instancia de clase) en concreto, sino a la clase como entidad.

Ejemplo:

```
public class Persona {

      private String nombre;
      private Integer edad;
      private Integer altura;
```

```
	private static Integer contador=0;

	Persona(){
		nombre="NombreDadoPorConstructor";
		contador++;
	}

	Persona(String unNombre, Integer unaEdad, Integer unAltura){
		nombre=unNombre;
		edad=unaEdad;
		altura=unAltura;
		contador++;
	}

	public static Integer getContador() {
		return contador;
	}

	public Integer getAltura() {
		return altura;
	}
	public void setAltura(Integer altura) {
		this.altura = altura;
	}
	public String getNombre() {
		return nombre;
	}
	public void setNombre(String unNombre) {
		nombre = unNombre;
	}
	public Integer getEdad() {
		return edad;
	}
	public void setEdad(Integer unaEdad) {
		if (checkEdadOK(unaEdad)){
			edad = unaEdad;
		}else{
			edad=0;
		}
	}
	private Boolean checkEdadOK(Integer edad){
		Boolean resultado=false;
		if(edad>=0){
			resultado=true;
		}
		return resultado;
	}
	public String hablar(String frase){
		return "soy " + nombre + ": " + frase;
	}

}
```

5 Paso de parámetros. Paso por valor y paso por referencia.

Los parámetros sirven para pasar información desde un método que llama a otro método, es decir, desde el método llamante al método llamado.

El encabezado del método llamado y las llamadas al mismo desde el método llamante deben coincidir en cuanto al número, tipo y orden de los parámetros y argumentos, pues lo contrario es un error de sintaxis

En los lenguajes de programación suelen existir dos formas de pasar los parámetros a los métodos. Paso por valor (dónde se realiza una copia de las variables) o paso por referencia (dónde se pasa una referencia a la variable original).

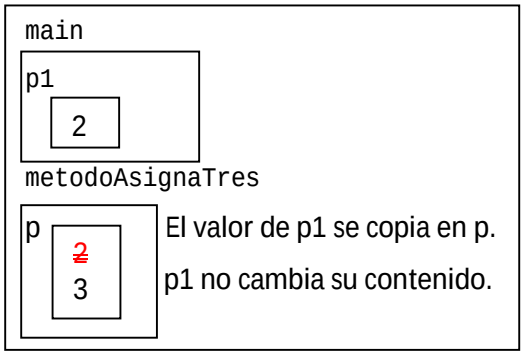
En Java solo hay paso de parámetros por valor, pero hay gente habla del paso de parámetros por referencia. Veamos con ejemplos por qué parece que realizamos un paso de parámetros por referencia en Java.

En el caso de los **tipos de datos primitivos**, en Java se realiza una copia, cualquier modificación de su valor que se haga dentro del método no tendrá efecto una vez se salga del mismo

```
public class Prueba {

	public static void metodoAsignaTres (int p) {
		p=3;
	}

	public static void main(String[] args) {
		int p1=2;
		metodoAsignaTres (p1);
		System.out.println(p1); //p1 = 2
	}
}
```



En el ejemplo anterior no tenemos la posibilidad de modificar el método `metodoAsignaTres` para que el parámetro se pase por referencia.

Pero si pasamos un **objeto** como parámetro lo que pasamos es una referencia al objeto, y por lo tanto se realiza una copia de la referencia. Así tenemos dos variables diferentes apuntando al mismo objeto.

Creamos un proyecto llamado Prueba con las siguientes clases:

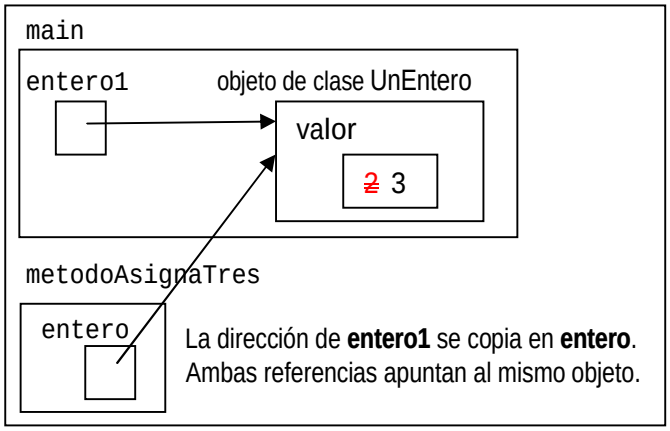
```
public class UnEntero {
	private int valor;

	public int getValor() {
		return valor;
	}

	public void setValor(int valor) {
		this.valor = valor;
	}
}

public class Prueba {

	public static void metodoAsignaTres (UnEntero entero) {
		entero.setValor(3);
	}
```



```
	public static void main(String[] args) {
		UnEntero entero1 = new UnEntero();
		entero1.setValor(2);
		System.out.println(entero1.getValor());());//muestra 2
		metodoAsignaTres (entero1);
		System.out.println(entero1.getValor());//muestra 3
	}
}
```

Hemos instanciado el objeto, le hemos asignado el valor de 2 al atributo de la clase y el método lo cambia a 3.

En la variable `entero` del método `metodoAsignaTres` se copia el contenido de la variable `entero1` del método `main` que es la referencia del objeto original. Por tanto los cambios que se hacen en el `metodoAsignaTres` modifican el objeto original. Esto hace que tengamos la impresión de que se ha hecho paso por referencia de los objetos.