UD06 – PROGRAMACIÓN ORIENTADA A OBJETOS

Contenido

1 Programación Orientada a Objetos

La **programación estructurada** estudia la forma de organizar la programación de manera que establece una serie de reglas para generar programas legibles y fáciles de mantener, obteniendo software de calidad.

La **programación modular** puede entenderse como el siguiente paso en la evolución de la programación tras la programación estructurada. Si la programación estructurada establece los criterios para obtener código de mayor calidad en cuanto a mantenimiento y simplicidad, la programación modular evoluciona en el sentido de dar más independencia al código de forma que pueda atomizarse para facilitar su reutilización, lo que también redunda en una mejora del mantenimiento y aligera el cuerpo principal de los programas.

La evolución de la programación en cuanto a la capacidad de abstracción y reutilización de código se ve reflejada en la **programación orientada a objetos, OOP**. Ésta, se basa en la idea de encapsular estado y comportamiento en objetos, definiendo previamente las "clases" de objetos. En general, la programación se resuelve comunicando dichos objetos a través de mensajes. Suelen permitir herencia y polimorfismo, y se siguen utilizando estructuras de control. La forma de programar en OOP sería más cercana a como expresaríamos las cosas en la vida real que otros tipos de programación al implementar los programas identificando propiedades y comportamiento de los objetos con atributos y métodos de dichos objetos en la realidad. Su principal ventaja es la reutilización de código y su facilidad para pensar soluciones a determinados problemas dado su alto nivel de abstracción.

La programación orientada a objetos (**Object-Oriented Programming, OOP**) es un **paradigma** de programación basado en el concepto de "**objetos**", los cuales pueden contener datos, a menudo llamados **atributos**; y código, en forma de procedimientos, normalmente llamados **métodos**.

Hay una gran diversidad de lenguajes orientados a objetos, pero los más populares están basados en **clases**; un objeto es una **instancia** de una clase la cual también determina su **tipo**.

Muchos de los lenguajes de programación más usados, tales como C++, Java, Python, etc., son lenguajes multiparadigma que soportan OOP en mayor o menor grado, típicamente en combinación con programación imperativa y procedimental. Podemos incluir entre los lenguajes orientados a objetos más significativos a los siguientes: Java, C++, C#, Python, Kotlin, PHP, Ruby, Perl, Object Pascal, Objective-C, Dart, Swift o Scala.

Los lenguajes de programación orientados a objetos comparten con sus predecesores no orientados a objetos las siguientes características:

- Variables, que pueden almacenar información formateada en un pequeño número de tipos de datos como enteros, caracteres alfanuméricos etc. Estos tipos pueden incluir estructuras de datos como cadenas, listas o tablas entre otros.
- Procedimientos y funciones, habitualmente llamados métodos. Estos toman valores de entrada, generan valores de salida y manipulan datos.
- La programación estructura incluye estructuras secuenciales, repetitivas (bucles) y condicionales (if).
- La programación modular proporciona la capacidad de agrupar procedimientos en ficheros y módulos con el propósito de organizarlos adecuadamente.

La OOP utiliza los conceptos de encapsulamiento, abstracción, polimorfismo y herencia. Se popularizó a finales de los años 80 y durante los 90 con C++ y Java y su uso es muy extenso.

Abordaremos estos conceptos a lo largo de esta unidad didáctica.

2 Clases y objetos

Un **objeto** es un conjunto de software que contiene estado y comportamiento. Los objetos de software se utilizan para modelar objetos del mundo real que podemos encontrar en el día a día. Veremos cómo representar estado y comportamiento dentro de un objeto, introduciremos el concepto de encapsulación y explicaremos los beneficios que aporta esta solución al diseño de software.

Una **clase** es un modelo o prototipo a partir del cual se crean los objetos; una clase modela el estado y el comportamiento de un objeto del mundo real.

2.1 Características de los objetos

Los objetos son clave para entender la programación orientada a objetos. Si miramos a nuestro alrededor encontramos muchos ejemplos de objetos en el mundo real: el ordenador, el monitor, el teclado o el ratón, por ejemplo.

Los objetos del mundo real comparten tres características **estado**, **comportamiento** e **identidad**. Por ejemplo, los coches tienen estado; marcha actual, velocidad actual, y un comportamiento; velocidad máxima, caja de cambios de 6 velocidades, frenos, control de crucero, etc. También tienen identidad: dos coches exactamente de mismo modelo serán 2 vehículos diferentes, cada uno con su número de bastidor y su cerradura diferente del otro, teniendo, eso sí, prestaciones idénticas.

Los objetos de software son conceptualmente similares a los objetos del mundo real: también tienen estado, comportamiento e identidad. Un objeto almacena su estado en campos o atributos (las variables que contenga) y expone su comportamiento a través de métodos (funciones y procedimientos). Los métodos operan en el estado interno de un objeto y proporciona el mecanismo principal de comunicación de objeto a objeto. A ocultar el estado interno y exigir que toda la interacción con el objeto se realice a través de los métodos se le conoce como **encapsulamiento** de datos, uno de los principios fundamentales de la orientación a objetos.

Al atribuir el estado y proporcionar métodos para cambiar ese estado, el objeto mantiene el control de cómo el mundo exterior puede usarlo.

La identidad se establece principalmente por el puntero a la dirección de memoria que almacena físicamente los datos del objeto.

La agrupación de código en objetos de software individuales proporciona una serie de beneficios, entre los que se incluye:

1. Modularidad: el código fuente de un objeto se puede escribir y mantener independientemente del código fuente de otros objetos. Una vez creado, un objeto se puede pasar fácilmente dentro del sistema.
2. Ocultar información: al interactuar solo con los métodos de un objeto, los detalles de su implementación interna permanecen ocultos del mundo exterior.

3. Reutilización de código: si ya existe un objeto (tal vez escrito por otro desarrollador de software), podemos usar ese objeto en nuestro programa.
4. Facilidad de conexión y depuración: si un objeto en particular resulta problemático, simplemente podemos eliminarlo de nuestra aplicación y reemplazarlo por otro objeto diferente. Esto es análogo a la solución de problemas mecánicos en el mundo real. Si se rompe una pieza de nuestro vehículo podremos sustituirla por un recambio y no reemplazar el vehículo entero. Imaginemos que tenemos un programa que descarga y sube ficheros a un servicio FTP, es muy común que utilicemos una librería (conjunto de objetos) que se encarguen del protocolo de comunicación con el FTP, si tenemos problemas con esta librería podremos sustituirla por otra que sea más robusta.

2.2 Clases

En el mundo real encontraremos muchos objetos individuales todos del mismo tipo. Una industria automovilística fabricará miles de coches de la misma marca y modelo, siguiendo los mismos diseños y planos y con los mismos componentes. En términos de orientación a objetos diremos que un coche es una *instancia* de la *clase de objetos* conocidos como coches. Una **clase** es un modelo a partir del cual se crean los objetos individuales.

Veamos un ejemplo con Java, definimos la clase *Car*, para ello debemos utilizar la palabra clave class, el nombre que le demos (empezando por mayúsculas, UpperCamelCase), en nuestro caso Car, y a continuación entre llaves {} el código de nuestra clase. En el ejemplo hemos incluido los campos atributos velocidad (speed) y marcha (gear), éstos determinan el estado de los objetos que se creen de esta clase. El comportamiento vendrá dado por los métodos changeGear, speedUp, applyBrackets y printStates que permiten modificar y mostrar el estado de los objetos.

```
public class Car {

	double speed = 0.0;
	int gear = 0;

	void changeGear(int newValue) {
		gear = newValue;
	}

	void speedUp(int increment) {
		speed += increment;
	}

	void applyBrakes(int decrement) {
		speed-=decrement;
	}

	void printStates() {
		System.out.println("gear: " + gear
				+ ", speed: " + speed);
	}
}
```

La clase Car no contiene un método *main*, esto es porque no es una aplicación completa, es solo un modelo para los coches que podamos *usar* en una aplicación. La responsabilidad de crear y usar los nuevos objetos *Car* pertenece a alguna otra clase en nuestra aplicación.

La siguiente clase *CarDemo* crea dos objetos *Car* separados e invoca a sus métodos, esta clase sí tiene método *main* lo que nos permite ejecutarla como una aplicación de Java.

```
public class CarDemo {

	public static void main(String[] args) {
		// creamos dos objetos
		// diferentes de la clase coche
		Car car1 = new Car();
		Car car2 = new Car();

		// cambios su estado
		// invocando a sus métodos
		car1.changeGear(1);
		car1.speedUp(10);
		car1.speedUp(10);
		car1.changeGear(2);
		car1.speedUp(10);
		car1.speedUp(10);
		car1.changeGear(3);
		car1.speedUp(10);

		car2.changeGear(1);
		car2.speedUp(20);
		car2.changeGear(2);
		car2.speedUp(20);
		car2.changeGear(3);
		car2.speedUp(20);
		car2.applyBrakes(15);

		//mostramos sus estados
		car1.printStates();
		car2.printStates();
	}
}
```

3 Definición de clases

Ya hemos visto que la sintaxis básica para la definición de clases en Java utiliza la palabra reservada ***class*** de la siguiente forma:

```
modificador class Nombre {
	modificador nombreAtributo;
	modificador nombreMetodo();
}
```

3.1 Modificadores

Una clase puede definirse como pública o privada por ejemplo en función de desde dónde se va a poder acceder a ella. Veamos los **modificadores de acceso** disponibles en Java:

- **public**. Una clase pública es accesible desde cualquier clase en cualquier paquete.
- **private**. Las clases privadas sólo son accesibles desde el archivo en el que se declaran.
- **protected**. Son clases privadas para aquellas que no heredan de ella o están fuera del paquete al que pertenece la clase.
- **Sin modificador (package)**. Este tipo de clases es visible en todo el paquete al que pertenece.

Modificador	Clase	Package	Subclase	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
(sin modificador)	Y	Y	N	N
private	Y	N	N	N

Los modificadores de acceso public, private, protected y package se aplican también a atributos y métodos.

Para los atributos es común que se declaren como privados y se acceda a ellos a través de métodos.

3.2 Constructores

Un constructor es un método especial de una clase, éste se ejecuta de manera automática en el momento de crear un objeto de la clase correspondiente.

Un constructor se identifica con el mismo nombre que el nombre de la clase, no retorna ningún valor (tampoco se declara como void).

Todas las clases tienen al menos un constructor aun cuando no hayamos definido ninguno. En ese caso se crea uno por defecto.

Podemos crear tantos constructores como necesitemos, con los parámetros que necesitemos en cada caso.

Los constructores tienen como objetivo principal la inicialización de las variables o atributos de la clase.

3.3 Instanciación de objetos

Para instanciar un objeto de una clase utilizaremos la palabra reservada **new** de la siguiente manera:

```
nombre_de_clase nombre_de_instancia = new nombre_clase(params);
```

La palabra reservada ***new*** se encarga de la reserva de memoria y devuelve una referencia a la dirección de memoria donde será almacenado el objeto.

Con la primera aparición de *nombre_de_clase* nos referimos al tipo de objeto que vamos a instanciar. La segunda aparición se refiere al constructor que vamos a utilizar en función de los parámetros que pasemos. El constructor por defecto (implícito o explícito) no tiene parámetros.

El *nombre_de instancia* se refiere al nombre de la variable con el que nos referiremos al objeto o instancia.

3.4 Acceso a atributos

Es muy común en Java definir los atributos como privados y acceder a ellos a través de métodos tanto para lectura como para escritura:

```
// atributo
private String name;

// método de acceso de lectura
public getName(){
        return this.name;
}

// método de acceso de escritura
public setName(String name){
        this.name = name;
}
```

La utilización de métodos getters y setters (u otros) para acceder a los atributos permite encapsular estos atributos (ocultación). Esta técnica permite proteger los atributos y aplicar validaciones o modificaciones al acceder a los atributos en caso de ser necesario.

3.5 Palabra reservada *this*

Usaremos la palabra reservada ***this*** para hacer referencia al objeto que estamos programando. Es muy útil, ya que cuando la anteponemos a algún atributo o método estamos inequívocamente indicando al compilador que ese elemento es del objeto en el que nos encontramos.

Permite, como en el ejemplo, pasar parámetros con el mismo nombre que los atributos de la clase de manera que podemos distinguir claramente cuál es el parámetro del método y cuál es el atributo de la clase.

La palabra reservada ***this*** tiene otra función dentro de los métodos constructores. A veces puede ocurrir que un método constructor tenga que ejecutar el mismo código que otro método constructor ya diseñado. En esta situación sería interesante poder llamar el constructor existente, con los parámetros adecuados, sin tener que copiar el código del constructor ya diseñado, y eso nos lo facilita la palabra reservada *this* utilizada como nombre de método: *this([lista de parámetros])*.

```
public class Person {
	//atributos: nombre, apellidos y DNI
	String name;
	String surname;
	String idCard;

	// constructor con nombre y apellidos
	public Person(String name, String surname) {
		super();
		this.name = name;
		this.surname = surname;
	}

	// constructor con nombre, appellidos y DNI
	public Person(String name, String surname, String idCard) {
		// llamada al constructor con nombre y apellidos
		// evitando repetir código
		this(name, surname);
		this.idCard = idCard;
	}
}
```

La palabra reservada *this* como método para llamar un constructor en el diseño de otro constructor sólo se puede utilizar en la primera sentencia del nuevo constructor. Al finalizar la llamada de otro constructor mediante *this*, se continúa con la ejecución de las instrucciones que haya después de la llamada *this (…)*.

3.6 Destructores

Un destructor es un método opuesto a un constructor, en lugar de crear un objeto lo destruye liberando la memoria de nuestra computadora.

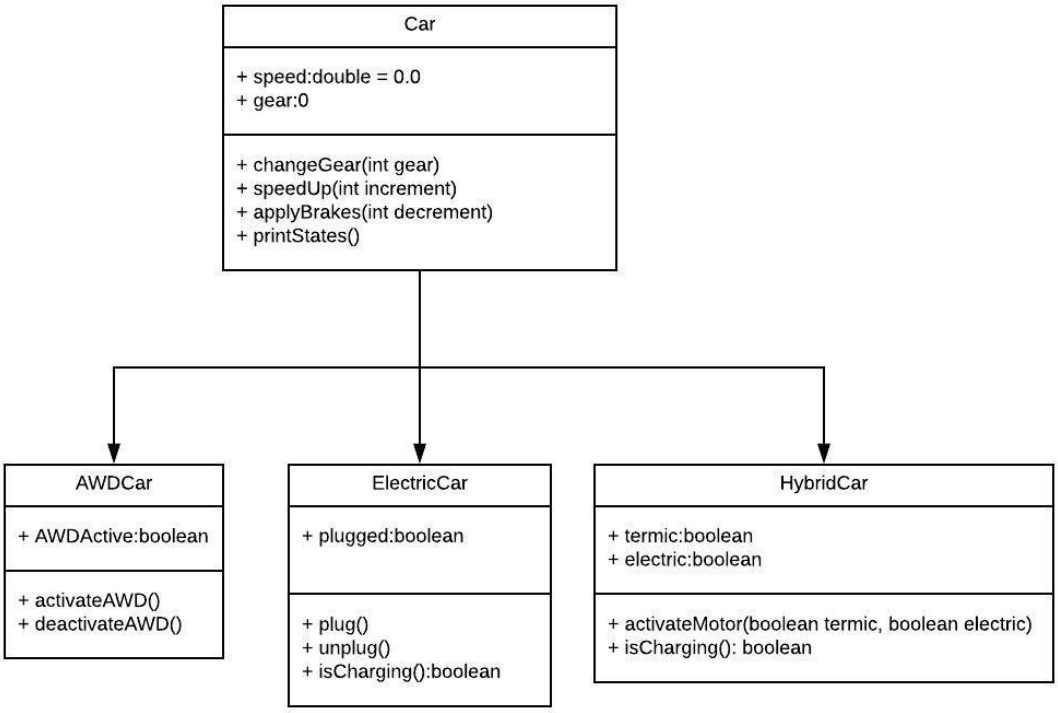
En Java no existen los destructores, el *Garbage Collector*, se encarga de liberar de la memoria los objetos que ya no se utilizan (que no tengan referencias en ejecución).

En las clases de Java hay un método heredado llamado finalize, que es llamado por el GC para la destrucción del objeto. El convenio es crear un método close cuando una clase necesita limpiar explícitamente sus recursos (por ejemplo, conexiones a ficheros o bases de datos) e invocar este método cuando dejamos de utilizarlo.

4 Herencia

Diferentes objetos tienen a menudo características comunes. Por ejemplo, los 4x4 (todo terreno o AWD), los coches híbridos o los coches eléctricos, comparten las características de un coche: marcha y velocidad, cambio de marcha acelerador y frenos. Además, cada uno de estos tipos tienen distintos atributos que los distinguen de los demás: por ejemplo, la tracción a las 4 ruedas en el caso de los 4x4.

La programación orientada a objetos permite que las clases hereden el estado y el comportamiento de otras clases. En este ejemplo, la clase Car se convierte ahora en la superclase de AWDCar, HybridCar y ElectricCar. En el lenguaje de programación Java, cada clase puede tener una superclase directa, y cada superclase tiene el potencial para un número ilimitado de subclases:



La sintaxis en Java para crear subclases es simple. En la declaración de la clase debemos usar la palabra clave ***extends*** seguida del nombre de la clase de la que vamos a heredar. Por herencia la clase hija tendrá disponible los atributos y métodos de la clase padre y además incluir los suyos propios. Como en el código de la clase hija no aparecen los atributos y método heredados tendremos que ser cuidados con la documentación de las clases para poder disponer de toda la información detallada cuando apliquemos herencia.

Vemos un ejemplo:

```
package ud06;

public class AWDCar extends Car {

	boolean AWDActive = false;

	void activateAWD() {
		AWDActive = true;
	}

	void deactivateAWD() {
		AWDActive = false;
	}

	@Override
	void printStates() {
		System.out.println("gear: " + gear
				+ ", speed: " + speed + ", AWD: " + AWDActive);
	}

}
```

Podemos sobreescribir un método de la clase padre para cambiar su comportamiento en la clase hija con la palabra clave @Override. Ya estudiaremos esta funcionalidad en detalle a lo largo del curso.

Vemos que podemos usar tanto el estado como el comportamiento de la clase padre con solo heredar de ella:

```
public class CarDemo {

	public static void main(String[] args) {

		AWDCar car = new AWDCar();
		car.activateAWD();
		car.changeGear(1);
		car.speedUp(15);
		car.printStates();

	}

}
```

5 Interfaces

Como hemos visto, los objetos definen su interacción con el mundo exterior a través de los métodos que exponen. Los métodos constituyen la *interfaz* del objeto con el mundo exterior. Si buscamos ejemplos en el mundo real podemos fijarnos en el botón de encendido de un televisor. Pulsando este botón podemos encender y apagar el televisor. Además, este botón está presente en multitud de dispositivos además del televisor; en un móvil, en una tablet, en un ordenador, en un monitor, etc. Por lo tanto, podríamos decir que todos los dispositivos que cuentan con un botón on/off implementan esta interfaz. Por supuesto, cada dispositivo implementa esta interfaz de una manera diferente.

De manera general, una *interfaz* es un grupo de métodos relacionados que definen un comportamiento y que permiten la interacción de ese comportamiento del objeto que los implementa con el mundo exterior.

En OOP, más específicamente en Java, una *interfaz* viene definida por un nombre y por un conjunto de métodos con el cuerpo de código vacío.

```
interface interfaceName {
   void method1(type param1);
   type method2(type param1, type param2);
}
```

Así en el ejemplo anterior podríamos extraer la interfaz *GearBox* con el método *changeGear(int newValue)*:

```
interface GearBox {
        void changeGear(int newValue);
}
```

Entonces hacer que la clase *Car* implemente esta interfaz:

```
class Car implmements GearBox {
        int gear = 0;
        //implementación concreta de GearBox para la clase Car
        void changeGear(int newValue){
                gear = newValue;
        }
        …
}
```

Implementar una interfaz permite a una clase declarar de manera más formal el comportamiento que proporciona. Las interfaces proporcionan un "*contrato*" de comportamiento entre las clases y el mundo exterior y este *contrato* está forzado por el compilador. Si declaramos una clase que implementa una interfaz dada, todos los métodos definidos por la interfaz deben aparecer en el código fuente de la clase para que esta clase pueda compilar.

6 Paquetes

Un paquete (*package*) es un espacio de nombres (namespace) que organiza un conjunto de clases e interfaces. Conceptualmente podemos pensar que los paquetes son similares a las carpetas de un ordenador; guardas tus documentos en una carpeta, tus imágenes en otra, tus programas en otra, etc. Una aplicación escrita en Java podría estar compuesta de cientos o incluso miles de clases individuales, tiene sentido organizar las clases e interfaces relacionadas en paquetes.

La plataforma de Java proporciona una enorme cantidad de librerías (bibliotecas) de clases que podemos utilizar para desarrollar nuestras propias aplicaciones. Cada librería está compuesta por un conjunto de paquetes y son conocidas como APIs, "Application Programming Interface".

Los paquetes de las APIs de Java representan las tareas más comunes de programación. Así, por ejemplo, el objeto *String* contiene estado y comportamiento para las cadenas de caracteres o la clase *File* permite trabajar fácilmente con ficheros en el sistema de archivos del ordenador; leer, modificar, mover o borrar ficheros. Hay literalmente miles de clases que nos ayudarán a centrarnos en el diseño de nuestra aplicación y no en la infraestructura necesaria para hacer nuestra aplicación funcionar.

La API de Java contiene el listado completo de todos los paquetes, clases, interfaces, campos y métodos proporcionados por la plataforma Java SE.

7 Variables y métodos estáticos

Una variable estática es una variable de clase, su valor será común a todas las instancias de esta, pudiendo ser modificada por todas ellas. Para declarar un atributo como estático utilizamos el modificador *static*.

En el siguiente ejemplo declaramos una clase *Person*, con un atributo *name* y un atributo *counter*. Declaramos un constructor que acepta el parámetro *name* que da nombre a la persona que representa y por cada instancia que se cree de la clase *Person* aumenta uno el contador. De esta forma tendremos el recuento de todas las instancias de la clase Person que se han creado en el programa.

```
import java.util.Scanner;

public class Person {
	private String name; // atributo de instancia privado
	private static int counter = 0; // atributo de clase privado

	// constructor que recibe el nombre esto anula el constructor por defecto implícito
	public Person(String name) {
		super();
		this.name = name;
		counter++;
	}

	// getter Name, lectura del atributo
	public String getName() {
		return name;
	}

	// setter Name, escritura del atributo
	public void setName(String name) {
		this.name = name;
	}

	// lectura del atributo de clase Counter
	public static int getCounter() {
		return counter;
	}

	// método main
	public static void main(String[] args) {
		// instanciamos varios objetos de la clase
		Person juanCarlos = new Person("Juan Carlos");
		Person jesus = new Person("Jesús");
		Person celia = new Person("Celia");

		// mostramos el número de instancias creadas
		System.out.println("Se han creado " + Person.getCounter() + " instancias de
la clase Person");
	}
}
```

También podemos declarar métodos estáticos, accesibles directamente desde la clase sin necesidad de instanciar ningún objeto. En el ejemplo anterior el método getCounter es un método estático. Los atributos y métodos estáticos son accesibles desde métodos de clase y de instancia. En cambio, los atributos y métodos de instancia no son accesibles desde los métodos de clase. Es decir, desde un método estático no podremos acceder a atributos y métodos no estáticos.

8 Método *main*

Cada programa necesita un lugar por el que comenzar su ejecución; en Java la ejecución comienza en el método *main*.

La signatura más común para el método *main* es:

```
public static void main(String[] args) {}
```

Esta es la que hemos visto y utilizado hasta ahora, y es la forma en la que el IDE autocompleta el método por nosotros. Vamos ahora a prestar atención al detalle de esta signatura:

- *public* – modificador de acceso que indica visibilidad global.
- *static* – modificador que indica que el método puede ser accedido directamente desde la clase, sin necesidad de instanciar un objeto.
- *void* – no devuelve ningún valor de retorno
- *main* – nombre del método, es el identificador que la máquina virtual de java JVM busca para ejecutar un programa Java.

El parámetro *args* representa un array con los parámetros de entrada que podemos pasar al programa para su ejecución.