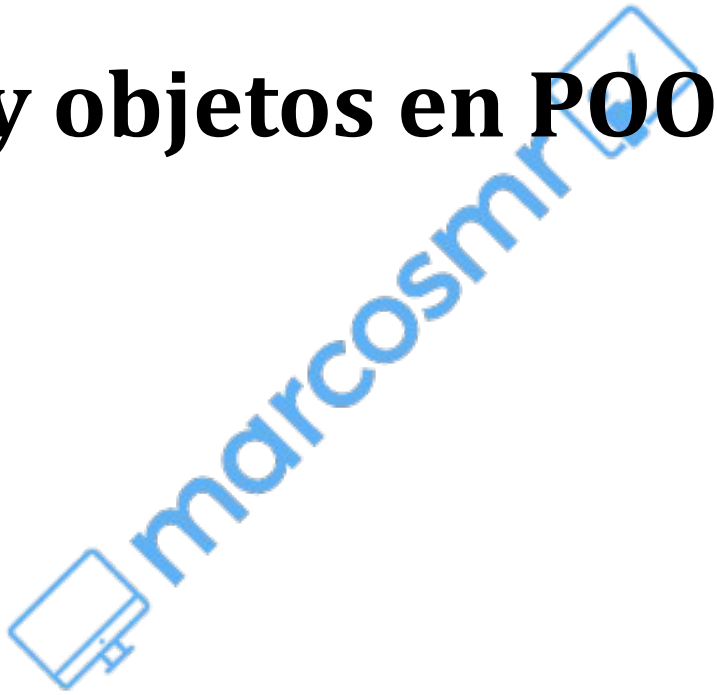
Clases y objetos en POO



ÍNDICE

1. Introducción

La programación orientada a objetos (POO) está centrada en los datos y las operaciones sobre dichos datos. Para llevarla a cabo se implementa lo que se conoce como **clase**. Una clase es una agrupación de datos (atributos) y de métodos que operan sobre esos datos. Nos van a permitir describir la información en relación a sus datos y a los métodos que permiten operar sobre dichos datos de forma autónoma, es decir, crear tipos de datos con su propia funcionalidad.

Conocemos, por ejemplo, los tipos de datos primitivos: `int`, `float`, `boolean`, etcétera. Sin embargo, puede que en nuestra aplicación nos interese trabajar con un nivel conceptual más elevado: el concepto "Coordenada GPS". Hasta este momento no tenemos ningún tipo que nos permita modelizarlo. ¿Qué podemos hacer?

- Identificar qué información es importante en una coordenada GPS: longitud y latitud.
- Definir qué funcionalidad es importante que tenga una coordenada GPS: saber en qué hemisferio está.
- **Implementar una clase con la información y la funcionalidad requerida**.

Otro ejemplo de clase puede ser un coche, el cual tiene unas características (atributos) y una funcionalidad propias. Por ejemplo:

- Información: color, marca, modelo y matrícula. Se implementa en **atributos**.
- Funcionalidad: desplazarse o pararse. Se implementan mediante **métodos**.

La clase `Coche` se responsabilizará de manejar todo aquello que tenga que ver con el coche.

Durante el presente tema, nos vamos a centrar en estudiar la POO haciendo uso de Java como lenguaje de programación para los ejemplos.

2. Definición de clases

La definición de una clase engloba la especificación de los atributos de la clase así como de sus métodos. La estructura es la siguiente:

```
[public] class nombreClase {
     // Definición de atributos y métodos
     //…
}
```

- La palabra `public` es opcional: si no se pone, la clase tiene la visibilidad por defecto. En Java, por defecto, significa que la clase únicamente es visible para las demás clases del mismo paquete (*package*). Si queremos que sea accesible desde cualquier otro código, deberemos poner `public`.
- Todos los métodos y atributos de la clase deben ser definidos dentro del bloque `{...}` de la propia clase.

2.1 Clase básica: únicamente contiene información

Supongamos que queremos crear la clase de coordenadas GPS de la que hemos hablado en la introducción. ¿Cómo podemos hacerlo? Una alternativa es la siguiente:

```
public class CoordenadaGPS {

      private double latitud;
      private double longitud;
}
```

2.2 Clase con funcionalidad

La anterior clase nos permite almacenar información. Sin embargo, en las clases lo habitual es que también haya funcionalidad o comportamiento (lógica de negocio de la clase). Si queremos ampliar dicha clase con un método que nos diga si la coordenada GPS pertenecen al hemisferio norte podemos implementar lo siguiente:

```java
public class CoordenadaGPS {
	private double latitud;
	private double longitud;

	public boolean estaEnHemisferioNorte() {
		return latitud > 0; // ¿Cómo consideramos el == 0?
	}
}
```

Es importante fijarse que tantos los atributos como el método el método están declarados como `public` para poder ser accedidos desde el exterior de la propia clase.

3. Objetos

Un objeto (o instancia de una clase) es un ejemplar concreto de una clase. Las clases son como tipos de datos o plantillas y los objetos son instancias concretas de un tipo determinado (de una determinada clase): plantilla rellenada. El símil es similar a un molde de galletas (plantilla) y la propia galleta (objeto).

Por ejemplo, si queremos almacenar las coordenadas GPS de Madrid y Roma debemos declarar dos objetos, cada uno para una ciudad, que sean de la clase `CoordenadaGPS`.

```java
public static void main(String[] args) {
	CoordenadaGPS coordenadasMadrid;
	CoordenadaGPS coordenadasRoma;
}
```

Para hacer uso de los métodos del objeto, debemos poner *`nombreObjeto.nombreMétodo();`*

4. Constructor

Hasta el momento, hemos definido una clase (`CoordenadaGPS`) y hemos declarado objetos (instancias) de dicha clase pero, en realidad, no hemos creado ningún objeto, únicamente los hemos declarado. Eso nos lleva a que no le estamos sacando mucha utilidad porque no estamos estableciendo datos para dichos objetos. Si nos fijamos, ninguno de los dos objetos creados anteriormente, para Madrid y Roma, tienen datos, es decir, la longitud y latitud correspondiente a cada ciudad.

Para crear un objeto de una determinada clase debemos utilizar la palabra reservada `new` y usar uno de los constructores de la clase. ¿Pero qué es eso del constructor? ¡Vamos a ello!.

Un constructor es un método al que se llama cuando deseamos crear un objeto de una clase. La principal misión del constructor es reservar memoria e inicializar los atributos miembro de la clase.

Los constructores no tienen valor de retorno (ni siquiera void) y su nombre es el mismo que el de la clase.

4.1 Sin parámetros

El constructor más básico es aquel que permite crear un objeto sin darle valor a los atributos que contiene. Los valores de los atributos serán aquellos asignados por defecto por el lenguaje de programación. En el caso de los `double` en Java se les asigna un valor de 0.0

Este constructor se crea automáticamente en nuestra clase si no existe ningún otro pero si queremos implementarlo específicamente podemos hacerlo de la siguiente forma:

```
public class CoordenadaGPS {
	private double latitud;
	private double longitud;

	public CoordenadaGPS() {
		//...
	}

	public boolean estaEnHemisferioNorte() {
		return latitud > 0;
	}
}
```

Supongamos que, por defecto, queremos que al crear un nuevo objeto de la clase `CoordenadaGPS` la latitud sea de 40.4530541 y la longitud sea de -3.6883445. Podemos hacerlo así:

```
	public CoordenadaGPS() {
		latitud = 40.4530541;
		longitud = -3.6883445;
	}
```

Finalmente, una vez que ya hemos implementado nuestro constructor, lo que debemos hacer es instanciar un objeto de la clase de la siguiente forma. Con ello **creamos** un objeto de la clase `CoordenadaGPS`.

```
CoordenadaGPS misCoordenadas = new CoordenadaGPS();
```

4.2 Con parámetros

Lo normal es que nos interese crear un objeto con unos valores específicos, es decir, personalizados. Por ejemplo, con las coordenadas GPS reales de Madrid y Roma. Para ello, debemos implementar un constructor con parámetros en nuestra clase `CoordenadaGPS`.

```java
public class CoordenadaGPS {
	private double latitud;
	private double longitud;

	public CoordenadaGPS(double lat, double lon) {
		latitud = lat;
		longitud = lon;
	}

	public boolean estaEnHemisferioNorte() {
		return latitud > 0;
	}
}
```

Podemos tener tantos constructores como deseemos pero no pueden tener la misma interfaz de entrada (también conocido como "firma del método"). ¿Eso qué quiere decir? Que tiene que haber diferencia en el número de parámetros, tipos de parámetros y/o en el orden de los parámetros que reciben un constructor y otro. Por ejemplo, no podemos crear dos constructores que reciban dos números enteros. Desde un punto de vista más coloquial podemos decir que "*lo que está dentro de los paréntesis, sin tener en cuenta el nombre de los parámetros, debe ser distinto*" entre constructores de la misma clase.

¿Cómo podríamos instanciar nuestras coordenadas, para ambas ciudades, haciendo uso del constructor con parámetros? De la siguiente forma:

```java
CoordenadaGPS coordenadasMadrid = new CoordenadaGPS(40.4167754, -3.7037902);
CoordenadaGPS coordenadasRoma = new CoordenadaGPS(41.9027835, 12.4963655);
```

4.3 this

4.3.1 Referencia a variables o atributos

En el constructor con parámetros, los nombres de los parámetros pueden coincidir con las variables/atributos de la propia clase. ¿Qué ocurre si en vez de llamar "lat" al parámetro le llamamos "latitud"? Nos quedaría un código, en el constructor, similar a:

```java
latitud = latitud;
```

Podemos darnos cuenta que eso puede llevar a confusión: cuando hago referencia a "latitud", ¿estoy hablando del atributo de la clase o del parámetro del constructor? Para solventar esta cuestión, podemos hacer uso de la palabra reservada **this** y dejar nuestro constructor de la siguiente forma:

```
	public CoordenadaGPS(double latitud, double longitud) {
		this.latitud = latitud;
		this.longitud = longitud;
	}
```

4.3.2 Referencia a constructores

La palabra reservada **this** también la podemos utilizar para llamar desde un constructor a otro. Supongamos que en nuestra clase `CoordenadaGPS` tenemos los dos siguientes constructores:

```
public CoordenadaGPS() {
	latitud = 40.4530541;
	longitud = -3.6883445;
}

public CoordenadaGPS(double lat, double lon) {
	latitud = lat;
	longitud = lon;
}
```

Con el primer constructor, con parámetros, asignamos la latitud y la longitud con los valores pasados por parámetro. En el segundo constructor, sin parámetros, *seteamos* unos valores por defecto. Dado que ya tenemos un constructor con parámetros que nos asigna la latitud y la longitud, el segundo constructor (sin parámetros) lo podemos implementar así:

```
public CoordenadaGPS() {
	this(40.4530541, -3.6883445);
}
```

5. Getters & setters

Perfecto, ya hemos conseguido declarar nuestra clase e instanciar varios objetos de la misma. Cada uno de los objetos con sus valores personalizados al habérselos pasados por constructor. ¿Pero cómo puedo saber el valor que tiene un atributo una vez creado el objeto? ¿Cómo puedo saber, en este momento, la latitud de Madrid? ¿Se puede modificar la longitud de Roma una vez creado el objeto? En este punto es donde entra los *getters* y *setters* de Programación Orientada a Objetos (POO).

- Un ***getter*** es un método que nos devuelve el valor que tiene, en un momento dado, un atributo de la clase. Generalmente, son públicos (`public`) o privados (`private`) en función de lo que nos interese. Lo habitual es que sean públicos.
- Un ***setter*** es un método que nos permite modificar o asignar valor a un atributo de la clase. Suelen ser públicos (`public`) o privados (`private`) en función de lo que nos interese. Lo habitual es que sean públicos.

De esta manera, las buenas prácticas indican que, en general, los atributos deben ser privados y los *getters/setters* deben ser públicos para manejar dichos atributos.

Estupendo, todo claro. Ahora toca aprender a implementarlos en nuestra clase de `CoordenadaGPS`.

```
public class CoordenadaGPS {
	private double latitud;
	private double longitud;

	public CoordenadaGPS(double latitud, double longitud) {
		this.latitud = latitud;
		this.longitud = longitud;
	}
```

```
	public double getLatitud() {
		return latitud;
	}

	public void setLatitud(double latitud) {
		this.latitud = latitud;
	}

	public double getLongitud() {
		return longitud;
	}

	public void setLongitud(double longitud) {
		this.longitud = longitud;
	}

	public boolean estaEnHemisferioNorte() {
		return latitud > 0;
	}
}
```

Muy bien, tenemos nuestra clase con sus atributos, con el constructor con parámetros, los *getters* y *setters*, así como un método que nos dice si las coordenadas pertenecen al hemisferio norte. ¿Cómo podemos hacer uso de todo ello? Vamos a ver todo en un ejemplo completo implementando el siguiente código en nuestra clase principal:

```
private static void muestraInformacion(CoordenadaGPS coordenadas)
{
	String hemisferio = (coordenadas.estaEnHemisferioNorte()) ? "norte" : "sur";

	System.out.printf("La ciudad tiene una latitud de %f y una longitud de %f. Está en el hemisferio %s.%n",
	coordenadas.getLatitud(),
	coordenadas.getLongitud(),
	hemisferio);
}
```

```
public static void main(String[] args)
{
     System.out.println("a) Creación de un objeto con las coordenadas GPS de Madrid haciendo uso del
constructor con parámetros");
     CoordenadaGPS coordenadasMadrid = new CoordenadaGPS(40.416775, -3.703790);

     System.out.println("\nb) Mostramos la información por pantalla");
     muestraInformacion(coordenadasMadrid);

     System.out.println("\nc) Modificamos la longitud a -1.234567 y volvemos a mostrar la información");
     coordenadasMadrid.setLongitud(-1.234567);
     muestraInformacion(coordenadasMadrid);
}
```

La salida, por consola, del anterior código es la siguiente:

```
a) Creación de un objeto con las coordenadas GPS de Madrid haciendo uso del constructor con parámetros

b) Mostramos la información por pantalla
La ciudad tiene una latitud de 40,416775 y una longitud de -3,703790. Está en el hemisferio norte.

c) Modificamos la longitud a -1.234567 y volvemos a mostrar la información
La ciudad tiene una latitud de 40,416775 y una longitud de -1,234567. Está en el hemisferio norte.
```

6. Variables o atributos miembro de la clase. Estáticos

En nuestro ejemplo de la clase `CoordenadaGPS` hemos visto que tenemos dos atributos miembro de objeto, es decir, los valores de latitud y longitud son personalizados para cada objeto. El objeto *`coordenadasMadrid`* tiene sus propios valores de latitud y longitud y son diferentes de los valores de latitud y longitud de la variable (objeto) *`coordenadasRoma`*.

Sin embargo, puede que nos interese que una variable sea compartida por todos los objetos de la clase, es decir, que cualquier objeto de dicha clase pueda modificar el valor de una variable y que el resto de objetos de dicha clase puedan ver esa modificación.

Para entenderlo mejor, vamos a hacer un ejemplo académico en el que vamos a tener un atributo denominado "`cuantos`" que va a ir almacenando cuántos objetos de `CoordenadaGPS` hemos creado. Dicho atributo es lo que se conoce como miembro de clase y necesita ser declarado con la palabra reservada **`static`** (la cual ya conocemos).

6.1 Declaración de una variable o atributo miembro de la clase

Como acabamos de ver, necesitamos hacer uso de la palabra reservada **`static`** para hacer la declaración del atributo `cuantos`.

```
public class CoordenadaGPS {
	private double latitud;
	private double longitud;

	public static int cuantos;
	// ...
}
```

6.2 Uso de una variable o atributo miembro de clase

En este momento hemos declarado el atributo estático (**`static`**) pero para que cumpla su función (almacenar cuántos objetos de `CoordenadaGPS` se han creado) debemos incrementar su valor en una unidad cada vez que se cree un objeto. ¿Cuál es el lugar ideal para hacerlo? El constructor (o constructores). ¡Vamos a ello! Nuestra clase quedaría así:

```
public class CoordenadaGPS {
	private double latitud;
	private double longitud;
	public static int cuantos;
```

```
	public CoordenadaGPS(double latitud, double longitud) {
		this.latitud = latitud;
		this.longitud = longitud;
		cuantos++;
	}

	public double getLatitud() {
		return latitud;
	}

	public void setLatitud(double latitud) {
		this.latitud = latitud;
	}

	public double getLongitud() {
		return longitud;
	}

	public void setLongitud(double longitud) {
		this.longitud = longitud;
	}

	public boolean estaEnHemisferioNorte()
	{
		return latitud > 0;
	}
}
```

Dado que las variables miembro de clase (y las constantes) pertenecen a una clase y no a un objeto (instancias de dicha clase) no es necesario crear un objeto de la clase para poder hacer uso de ellas. En nuestro caso, al haberla declarado como pública, podemos acceder a su valor mediante `CoordenadaGPS.cuantos` En nuestro programa principal podemos tener el siguiente código que hace uso del atributo `cuantos`.

```
public class App {

	private static void muestraInformacion(CoordenadaGPS coordenadas) {
		String hemisferio = (coordenadas.estaEnHemisferioNorte()) ? "norte" : "sur";

		System.out.printf("La ciudad tiene una latitud de %f y una longitud de %f. Está en el
hemisferio %s.%n",
					coordenadas.getLatitud(),
					coordenadas.getLongitud(),
					hemisferio);
		System.out.println("-> Hasta el momento, hemos creado " + CoordenadaGPS.cuantos + "
coordenada(s) GPS");
	}

	public static void main(String[] args) {
		System.out.println("a) Creación de un objeto con las coordenadas GPS de Madrid haciendo uso
del constructor con parámetros");
		CoordenadaGPS coordenadasMadrid = new CoordenadaGPS(40.416775, -3.703790);
		muestraInformacion(coordenadasMadrid);

		System.out.println("\nb) Creación de un objeto con las coordenadas GPS de Roma haciendo uso
del constructor con parámetros");
		CoordenadaGPS coordenadasRoma = new CoordenadaGPS(41.902783, 12.496365);
		muestraInformacion(coordenadasRoma);
	}
}
```

No obstante, una solución más elegante para acceder a su valor es crear un método *getter* estático que nos devuelva el valor de dicho atributo. La salida por pantalla será:

```
a) Creación de un objeto con las coordenadas GPS de Madrid haciendo uso del constructor con parámetros
La ciudad tiene una latitud de 40,416775 y una longitud de -3,703790. Está en el hemisferio norte.
-> Hasta el momento, hemos creado 1 coordenada(s) GPS

b) Creación de un objeto con las coordenadas GPS de Roma haciendo uso del constructor con parámetros
La ciudad tiene una latitud de 41,902782 y una longitud de 12,496365. Está en el hemisferio norte.
-> Hasta el momento, hemos creado 2 coordenada(s) GPS
```

Podemos ver que los valores de latitud y longitud están personalizados para las coordenadas de cada ciudad. Sin embargo, el atributo `cuantos` es compartido por los dos objetos ya que es un atributo de la clase y su gestión es a nivel de clase, no de objeto.

6.3 Resumen

En resumen, este tipo de atributos/variables/constantes son propios de la clase y no de cada objeto. Para ello, debemos declararlos con la palabra reservada **`static`**.

Los elementos estáticos se suelen utilizar para definir constantes comunes para todos los objetos de la clase o atributos que solamente tienen sentido para toda la clase. Para llamarlos se suele utilizar el nombre de la clase (no es imprescindible, pues se puede utilizar también el nombre de cualquier objeto), porque de esta forma su sentido queda más claro.

Los atributos miembro **`static`** se crean en el momento en que pueden ser necesarios: cuando se va a crear el primer objeto de la clase, en el momento en el que se llama a un método **`static`** o en cuanto se utiliza una variable **`static`** de dicha clase. Lo importante es que los atributos miembro **`static`** se inicializan siempre antes que cualquier objeto de la clase.

7. Constantes

Una constante es un dato que no puede cambiar su valor. En Java ya sabemos que debemos declararlo con la palabra reservada **`final`**. Por ejemplo, si queremos crear una constante estática en nuestra clase `CoordenadaGPS`, que indique la versión de dicha clase mediante una cadena de texto, podemos implementar el siguiente código:

```
public class CoordenadaGPS {
	private static final String VERSION = "v1.0";
	// ...
}
```

8. Paquetes

8.1 Concepto

Un paquete (*package*) es una agrupación de clases que tienen un sentido lógico como grupo. Podemos hacer el símil con una carpeta del disco duro (paquete) que agrupa todos los PDFs de apuntes (clases) de un determinado módulo. El usuario puede crear sus propios paquetes.

Para que una clase pase a formar parte de un paquete llamado ejercicios, hay que introducir en ella la siguiente sentencia como primera sentencia del fichero (sin comentarios ni líneas en blanco):

```
package ejercicios;
```

Los nombres de los paquetes se suelen escribir con minúsculas, para distinguirlos de las clases, que deben empezar por mayúscula. El nombre de un paquete suele constar de varios nombres unidos por puntos. Por ejemplo, la clase `Scanner` está en el paquete `java.util` y por eso tenemos que importarla de la siguiente forma `import java.util.Scanner;`

Todas las clases que forman parte de un paquete deben estar en el mismo directorio. Los nombres compuestos de los paquetes están relacionados con la jerarquía de directorios en que se guardan las clases. Es recomendable que los nombres de las clases de Java sean únicos en Internet.

8.2 Finalidad

Los paquetes se utilizan con las finalidades siguientes:

- Para agrupar clases relacionadas.
- Para evitar conflictos de nombres (se recuerda que el dominio de nombres de Java es la Internet). En caso de conflicto de nombres entre clases importadas, el compilador obliga a cualificar en el código los nombres de dichas clases con el nombre del paquete.
- Para ayudar en el control de la accesibilidad de clases y miembros.

8.3 Funcionamiento

Con la sentencia `import nombrePaquete.nombreClase;` se puede importar la clase *nombreClase* del paquete *nombrePaquete*. Recuerda que el nombre del paquete puede estar formado por varias palabras separadas por puntos.

Con la sentencia `import nombrePaquete.*;` podemos importar todas las clases del paquete *nombrePaquete*.

En un programa de Java, una clase puede ser referida con su nombre completo (el nombre del paquete más el de la clase, separados por un punto). También se pueden referir con el nombre completo las variables y los métodos de las clases. Esto se puede hacer siempre de modo opcional, pero es incómodo. Por ejemplo, para usar la clase `LocalDateTime` podemos importar el paquete correspondiente (`import java.time.LocalDateTime;`) y posteriormente incluir la siguiente línea en nuestro código: `LocalDateTime fechaHora;`

Eso es lo recomendable ya que nos permite reutilizar código y no tener que escribir el nombre cualificado de las clases que queremos utilizar. La alternativa, sin tener que hacer la importación de la clase, es usar el nombre completo cualificado de la clase en nuestro código:

```
java.time.LocalDateTime fechaHora;
```

La sentencia `import` permite abreviar los nombres de las clases, variables y métodos, evitando el tener que escribir continuamente el nombre del paquete importado.

Por defecto, se importan los paquetes `java.lang` y el paquete actual (las clases del directorio actual).

9. Cuestiones a tener en cuenta

Cuando trabajamos dentro del paradigma POO tenemos que saber a qué se refieren los siguientes conceptos y tener claras las siguientes cuestiones:

- ✔ **Clases**. Las clases son el centro de la Programación Orientada a Objetos.
- ✔ **Encapsulación**.
  - Las clases pueden ser declaradas como públicas (`public`), pudiendo ser usadas desde cualquier lugar del código; o como `package` (que es el valor por defecto y hace que permiten que sean accesibles solamente para otras clases del mismo paquete).
  - Los atributos miembro y los métodos pueden ser `public`, `private` (únicamente permite acceder desde dentro de la propia clase) o `protected` (desde la propia clase o desde clases que deriven/hereden de ella… ya lo estudiaremos) y `package`. De esta forma se puede controlar el acceso y evitar un uso inadecuado.
- ✔ **Herencia**. Una clase puede derivar de otra (`extends`), y en ese caso hereda todos sus atributos y métodos. Una clase derivada puede añadir nuevos atributos/variables y métodos y/o redefinir los atributos/variables y métodos heredados. Ya lo estudiaremos.
- ✔ **Polimorfismo**. Los objetos de distintas clases pertenecientes a una misma jerarquía o que implementan una misma interface pueden tratarse de una forma

general e individualizada, al mismo tiempo. Esto facilita la programación y el mantenimiento del código. Ya lo estudiaremos junto con la herencia.

A continuación se enumeran algunas características importantes que debemos tener en cuenta a la hora de utilizar clases y objetos:

- ✔ Todos los objetos deben pertenecer a una clase.
- ✔ En un fichero podemos definir varias clases, pero en un fichero no puede haber más que una clase `public`. Este fichero se debe llamar como la clase public que contiene con extensión *.java. Nosotros no vamos a hacer eso, es decir, vamos a crear un fichero por cada clase.
- ✔ Con algunas excepciones, lo habitual es escribir una sola clase por fichero.
- ✔ Si una clase contenida en un fichero no es public, no es necesario que el fichero se llame como la clase. Nosotros vamos a llamar a la clase con el mismo nombre del fichero.
- ✔ Por defecto:
  - Las clases las vamos a crear como públicas.
  - Los atributos los vamos a crear como privados.
  - Los métodos los crearemos públicos si ofrecen funcionalidad hacia el exterior de la clase; o privados si lo que implementan es lógica interna de la clase.