

**PROGRAMACIÓN**

# **UP5. INTRODUCCIÓN A LA POO**



**1º CFGS DAW**

Curso 2024-25



# Introducción a la POO

## 5.1. Introducción a la POO

Durante la década de los 80 y principios de los 90, el paradigma de programación estructurada (la que hemos visto hasta ahora) fue claramente el dominante a la hora de realizar programas más o menos complejos. Sin embargo, dicho paradigma presentaba algunas carencias a la hora de abarcar problemas complejos.

La programación orientada a objetos (*POO*) supone un nuevo paradigma frente a la programación estructurada o modular. En esta, los datos y operaciones se encuentran separados (por un lado las variables y por otro las instrucciones), y los métodos se van pasando de unos a otros los datos que necesitan, mediante parámetros. En la *POO*, los datos y métodos se agrupan, formando así un bloque que contiene todo que se denomina **clase**. Así, parece más lógica y sencilla la relación entre los datos e instrucciones que componen un mismo programa.

Por ejemplo, si tuviera un programa para manipular un triángulo...

- En la **programación modular**, para calcular su área, necesitaría recoger y guardar dos valores en variables (base y altura), y luego pasárselas al método que calculará el área y nos devolverá el resultado para imprimirlo:

```
public class Triangulo {  
  
    public static void main(String[] args) {  
        double base = leerValor("Introduce la base del triángulo: ");  
        double altura = leerValor("Introduce la altura del triángulo: ");  
        double area = calcularArea(base, altura);  
        mostrarResultado(area);  
    }  
  
    public static double leerValor(String mensaje) {  
        Scanner scanner = new Scanner(System.in);  
        System.out.print(mensaje);  
        return scanner.nextDouble();  
    }  
}
```

```

17     public static double calcularArea(double base, double altura) {
18         return (base * altura) / 2;
19     }
20
21     public static void mostrarResultado(double area) {
22         System.out.println("El área del triángulo es: " + area);
23     }

```

- Si traducimos lo anterior a la *POO*, necesitaríamos 2 variables (o características) asociadas a la clase *Triangulo* (a partir de ahora, llamadas ATRIBUTOS), junto a los métodos necesarios para calcular el área:

```

1 public class Triangulo2 {
2
3     double base;
4     double altura;
5
6     public Triangulo2(double base, double altura) {
7         this.base = base;
8         this.altura = altura;
9     }
10
11    public double calcularArea() {
12        return (base * altura) / 2;
13    }
14
15    public void mostrarResultado() {
16        System.out.println("El área del triángulo es: " + calcularArea());
17    }
18}

```

Y desde *main*:

```

public class Main {
    public static void main(String[] args) {

        Triangulo2 triangulo = new Triangulo2(5, 2);
        System.out.println("Has introducido base = " + triangulo.base +
        triangulo.mostrarResultado());
    }
}

```

$$\left. \begin{array}{c} 8 \\ 9 \end{array} \right| \quad \}$$

## 5.1.1. Clases y objetos

---

Una **clase** representa al conjunto de objetos que tienen unas características comunes. Si volvemos al ejemplo del triángulo, sabemos que hay de distintos tipos: rectángulo, equilátero, isósceles... Pero todos tienen algo en común: base y altura. Pero en este caso, la POO va más allá, y nos permite hacer analogía más allá de la lógica y las matemáticas. Podríamos definir, por ejemplo, una clase **Persona** con su edad, nombre, DNI, etc.

Los datos que resumen dichas características comunes y las operaciones que se pueden hacer con ellos reciben el nombre de **atributos** y **métodos**, respectivamente, de modo que los métodos operan sobre los atributos de la clase. Podría entenderse una clase como una plantilla con las características y comportamientos comunes de los objetos, lo que permite definirlos y crearlos.

```
1 public class Persona {  
2     //atributos  
3     String nombre;  
4     String dni;  
5     int edad;  
6 }
```

### ¿Y los objetos?

Como ya hemos ido hablando, las clases son un concepto estático definido en el programa fuente, mientras que **los objetos son instancias de una clase**. Son entes dinámicos que ocupan memoria en la ejecución de un programa. Por ejemplo, en la clase *Persona*, tendríamos una serie de atributos o variables que puedan tener en común las personas (edad, nombre, DNI, etc), y unos métodos u operaciones que puede hacer (hablar, andar, comer). Después podemos tener varios objetos de esa clase, por ejemplo, uno con nombre = Andrés y edad = 20, otro con nombre = Silvia y edad = 34, etc.

```
1      CLASE  
2 +-----+  
3 |  Persona  |  
4 +-----+  
5 | - nombre: String|  
6 | - dni: String  |  
7 | - edad: int    |  
8
```

```

8
9 | +-----+
10| | + mostrarInfo() |
11| +-----+
12
13          OBJETOS
14 +-----+ +-----+
15| persona1: Persona | persona2: Persona   |
16| +-----+ +-----+
17| nombre = "Juan" | nombre = "Ana García" |
18| dni = "12345678A" | dni = "87654321B"   |
19| edad = 30       | edad = 25        |
+-----+ +-----+

```

En definitiva, es como definir un tipo de dato (clase) y varias variables de ese tipo (objetos), con sus respectivos valores.

```

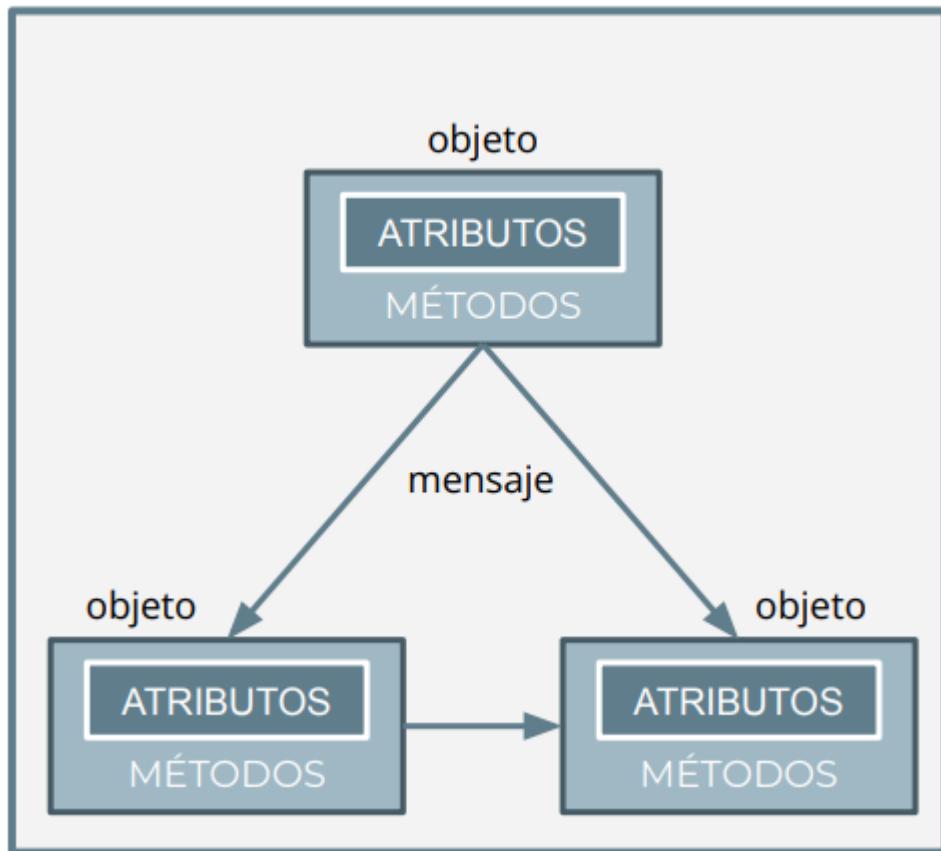
1 Persona persona1 = new Persona("Juan", "12345678A", 30);
2 Persona persona2 = new Persona("Ana García", "87654321B", 25);

```

Vale... ¿pero cómo se crea una clase?

Toda clase (u objeto) puede estar compuesta por los siguientes elementos:

- **Atributos o campos:** son las variables o características propias de una clase u objeto. Por ejemplo, como ya sabemos, las personas tienen un nombre, por lo que el nombre sería un atributo de la clase *Persona*. Los atributos que tienen las clases y objetos pueden ser tipos simples (enteros, reales, booleanos) o compuestos (por ejemplo, arrays, u objetos del tipo de otras clases).
- **Mensajes.** Un objeto no funciona de forma aislada en un programa, sino que tiene que comunicarse con otros. Para ello, se utilizan los mensajes. En Java, los "mensajes" son básicamente las llamadas a los métodos que permiten que los objetos se comuniquen entre sí.
- **Métodos.** Un método indica qué debe hacer un objeto cuando recibe cierto mensaje. Además, permite enviar mensajes a otros objetos. Cuando se envía un mensaje a un objeto, es decir, cuando se invoca a un método de dicho objeto, es necesario saber qué hace ese mensaje/método, pero no cómo lo hace. Estructuralmente, la definición y programación equivale a lo que en programación modular se denominaban procedimientos o funciones.



- **Constructores.** Los constructores son un tipo de métodos que permiten crear objetos de una clase. Para ello, permiten reservar memoria para ellos y guardar los datos de sus atributos y métodos en memoria. Los vemos a continuación.
- **Destructores.** Los destructores son un tipo de métodos que permiten borrar objetos de una clase, liberando la memoria que estaban ocupando. Algunos lenguajes de programación como Java no requieren de destructores para liberar memoria, ya que disponen de mecanismos que periódicamente eliminan de la memoria aquellos objetos que ya no se están usando. Otros lenguajes como C++ sí que dejan en manos del programador la responsabilidad de liberar la memoria de objetos que ya no necesite, cuando lo considere oportuno.

## 5.1.2. Constructores

---

Cuando se construye un objeto es necesario dar un valor inicial a sus atributos. Es por ello que existe un método especial en cada clase, llamado constructor, que es ejecutado de forma automática cada vez que es instanciada una clase (realizamos un *new*).

**El constructor se llama igual que la clase y no devuelve nunca ningún valor a través del *return*.** La invocación al método constructor devolverá una instancia de dicho objeto.

```
1 public class Persona {  
2     String nombre; //atributo nombre  
3     String dni; //atributo dni  
4     int edad; //atributo edad  
5  
6     //constructor de la clase Persona  
7     public Persona(String nombre, String dni, int edad) {  
8         this.nombre = nombre;  
9         this.dni = dni;  
10        this.edad = edad;  
11    }  
12}  
13}
```

Y desde el *main*:

```
1 public class Main {  
2     public static void main(String[] args) {  
3         //creación de dos objetos Persona  
4         Persona persona1 = new Persona("Juan Pérez", "12345678A", 30);  
5         Persona persona2 = new Persona("Ana García", "87654321B", 25);  
6  
7         System.out.println(persona1.nombre);  
8         System.out.println(persona2.nombre);  
9     }  
10}
```

La palabra reservada *this* en JAVA

Dentro de un método o de un constructor, **this** hace referencia al objeto actual (instancia actual):

```
public class Punto {  
    public int x;  
    public int y;  
    public Punto(int a, int b) {  
        x = a;  
        y = b;  
    }  
}
```

```
public class Punto {  
    public int x;  
    public int y;  
    public Punto(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

## Sobrecarga de constructores

Una clase puede tener varios constructores, que se diferencia por el tipo y su número de parámetros. Igual que pasaba con los métodos, pueden tener la misma cantidad de parámetros, pero como mínimo, uno de ellos tiene que ser de un tipo distinto. O directamente tener parámetros distintos, como en el siguiente caso:

```
1  public class Persona {  
2      //atributos con visibilidad protegida  
3      String nombre;  
4      String dni;  
5      int edad;  
6  
7      //constructor parametrizado de la clase Persona  
8      public Persona(String nombre, String dni, int edad) {  
9          this.nombre = nombre;  
10         this.dni = dni;  
11         this.edad = edad;  
12     }  
13  
14     //constructor parametrizado pero sin el atributo edad  
15     public Persona(String nombre, String dni) {  
16         this.nombre = nombre;  
17         this.dni = dni;  
18     }  
19  
20 }
```

Y desde el *main*:

```
1 public class Main {  
2     public static void main(String[] args) {  
3         //creación de dos objetos Persona  
4         Persona persona1 = new Persona("Juan Pérez", "12345678A", 30);  
5         Persona persona2 = new Persona("Ana García", "87654321B", 25);  
6         Persona persona_sin_edad = new Persona("Carlos Díaz", "54789651", 0);  
7  
8         System.out.println(persona1.nombre);  
9         System.out.println(persona2.nombre);  
10        System.out.println(persona_sin_edad.nombre + " " + persona_sin_edad.dni);  
11    }  
12 }
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" -cp . Main  
Juan Pérez  
Ana García  
Carlos Díaz 0
```

Además, en Java existen los llamados **constructores por defecto** (sin parámetros) que inicializan un objeto a:

- En caso de ser tipos primitivos, a su valor por defecto.
- *Strings* y referencias a objetos a *null*.

```
1 public class Persona {  
2     //atributos con visibilidad protegida  
3     String nombre;  
4     String dni;  
5     int edad;  
6  
7     //constructor parametrizado de la clase Persona  
8     public Persona(String nombre, String dni, int edad) {  
9         this.nombre = nombre;  
10        this.dni = dni;  
11        this.edad = edad;  
12    }  
13  
14     //constructor por defecto de la clase Persona  
15     public Persona() {  
16    }
```

```
17 }  
18 }  
19 }
```

Este constructor por defecto es útil si necesitamos crear objetos de *Persona* que serán inicializados más tarde, o si los atributos secundarios no son necesarios al momento de la creación del objeto.

```
1 public class Main {  
2     public static void main(String[] args) {  
3         //creación de dos objetos Persona  
4         Persona persona1 = new Persona("Juan Pérez", "12345678A", 30);  
5         Persona persona2 = new Persona("Ana García", "87654321B", 25);  
6         Persona persona3 = new Persona();  
7  
8         System.out.println(persona1.nombre);  
9         System.out.println(persona2.nombre);  
10        System.out.println(persona3.nombre);  
11    }  
12 }
```

```
"C:\Program Files\Java\jdk-23\bin\ja  
Juan Pérez  
Ana García  
null
```

### 5.1.3. Visibilidad

---

Tanto los atributos como los métodos y constructores de las clases y objetos tienen una determinada visibilidad, que indica qué otros objetos pueden acceder a dichos atributos o métodos. Dependiendo del tipo de acceso, tenemos fundamentalmente tres tipos de visibilidad:

- (+) **Pública (public)**: se puede acceder a ellos desde cualquier objeto del sistema
- (-) **Privada (private)**: sólo pueden ser accedidos desde el propio objeto
- (#) **Protegida (protected)**: se puede acceder a ellos desde el propio objeto o cualquier subtipo del mismo (mediante herencia).

Lo más habitual y **recomendable es que los atributos tengan una visibilidad privada** (o protegida, en el caso de la herencia que veremos en el tema siguiente), **y que los métodos, salvo aquellos de uso interno de cada clase, sean públicos.**

```
1 public class Persona {  
2     //atributos con visibilidad protegida  
3     private String nombre;  
4     private String dni;  
5     private int edad;  
6  
7     //constructor de la clase Persona  
8     public Persona(String nombre, String dni, int edad) {  
9         this.nombre = nombre;  
10        this.dni = dni;  
11        this.edad = edad;  
12    }  
13  
14    //método para mostrar la información de la persona  
15    public void mostrarInformacion() {  
16        System.out.println("Nombre: " + nombre);  
17        System.out.println("DNI: " + dni);  
18        System.out.println("Edad: " + edad);  
19    }  
20}
```

De esta forma, desde el *main* ya no podremos acceder a los atributos directamente para, por ejemplo, imprimirlas sin pasar por el método *mostrarInformacion()* obligatoriamente:

```
public class Main {
    public static void main(String[] args) {
        Persona persona1 = new Persona(nombre: "Juan", dni: "12345678A", edad: 30);
        Persona persona2 = new Persona(nombre: "Ana García", dni: "87654321B", edad: 25);

        System.out.println(persona1.nombre);
    }
}
```

'nombre' has private access in 'org.example.Persona'  
Make 'Persona.nombre' package-private Alt+Mayús+Intro More actions... Alt+Intro

© org.example.Persona  
private String nombre  
untitled

```
1 public class Main {
2     public static void main(String[] args) {
3         //creación de dos objetos Persona
4         Persona persona1 = new Persona("Juan Pérez", "12345678A", 30);
5         Persona persona2 = new Persona("Ana García", "87654321B", 25);
6
7         //mostrando la información de las dos personas mediante el método
8         persona1.mostrarInformacion();
9         System.out.println();
10        persona2.mostrarInformacion();
11    }
12 }
13 }
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe"
Nombre: Juan Pérez
DNI: 12345678A
Edad: 30

Nombre: Ana García
DNI: 87654321B
Edad: 25

Process finished with exit code 0
```

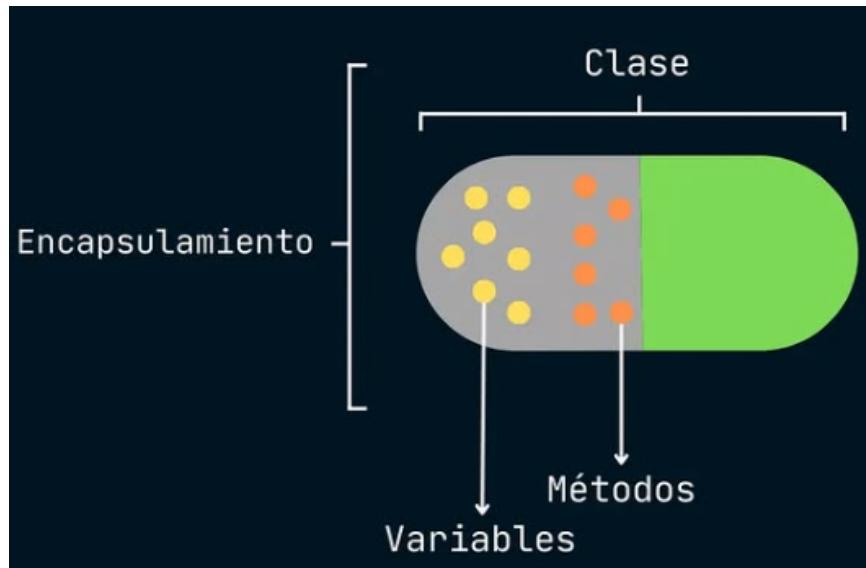
Se establece así para que el acceso y la modificación de los valores de los atributos solamente pueda hacerse de forma controlada desde dentro de la misma clase, y no desde cualquier parte del proyecto. En eso consiste el **encapsulamiento** que veremos a continuación.

Vale... ¿y por qué pones también "-", "+" y "#"?

En los diagramas UML (Lenguaje Unificado de Modelado) que haremos dentro de unos días, los modificadores de acceso se representan con símbolos. El modificador **private** se representa con un signo **-**, el **public** con un **+** y el **protected** con un **#**. Para que vayas familiarizándote:

```
1 | +-----+
2 | | Persona |
3 | +-----+
4 | | - nombre |
5 | | - dni |
6 | | - edad |
7 | +-----+
8 | | + mostrarInfo() |
9 | +-----+
```

## 5.1.4. Encapsulamiento (getters/setters)



Al integrar tanto métodos como atributos dentro de clases y objetos, los estamos encapsulando. El encapsulamiento es un buen mecanismo que **permite acceder a las propiedades o atributos (para leerlos o para modificarlos) sólo a través de ciertas vías, y no directamente, controlando así accesos o intentos de modificación no deseables**. Consiste en ocultar (hacer privados) los atributos de la clase, y mostrar (hacer públicos) unos métodos especiales que sirvan para acceder a ellos de forma controlada (los llamados *getters* y *setters*).

```
1  public class Persona {  
2      //atributos privados  
3      private String nombre;  
4      private String dni;  
5      private int edad;  
6  
7      //constructor de la clase Persona  
8      public Persona(String nombre, String dni, int edad) {  
9          this.nombre = nombre;  
10         this.dni = dni;  
11         this.edad = edad;  
12     }  
13  
14     //métodos getter para acceder a los atributos  
15     public String getNombre() {  
16         return nombre;  
17     }  
18  
19     public String getDni() {
```

```

20         return dni;
21     }
22
23     public int getEdad() {
24         return edad;
25     }
26
27     //métodos setter para modificar los atributos
28     public void setNombre(String nombre) {
29         this.nombre = nombre;
30     }
31
32     public void setDni(String dni) {
33         this.dni = dni;
34     }
35
36     public void setEdad(int edad) {
37         this.edad = edad;
38     }
39
40     //método para mostrar la información de la persona
41     public void mostrarInformacion() {
42         System.out.println("Nombre: " + nombre);
43         System.out.println("DNI: " + dni);
44         System.out.println("Edad: " + edad);
45     }
46 }
```

Por otro lado, mediante la ocultación separamos la implementación interna de un objeto de su uso. Así, el usuario sólo necesita saber los mensajes que necesita para comunicarse con el objeto y qué hace cada uno, pero no cómo lo hace. Por tanto, el objeto se comporta como una caja negra, consiguiendo mayor abstracción.

```

public class Main {
    public static void main(String[] args) {
        //creación de dos objetos Persona
        Persona persona1 = new Persona("Juan Pérez", "12345678A", 30);
        Persona persona2 = new Persona("Ana García", "87654321B", 25);

        //mostrando la información de las dos personas mediante el método
        persona1.mostrarInformacion();
        System.out.println();
        persona2.mostrarInformacion();
```

```
12 |         //ejemplo de uso de los getters y setters
13 |         persona1.setNombre("Juan Ruiz");
14 |         System.out.println("Nuevo nombre de persona1: " + persona1.getN
15 |     }
16 | }
```

# Ejercicios

---



## Ejercicio 2

---

Añade cuatro métodos a la clase *Persona* para realizar las siguientes acciones:

- a) *Obtener profesión.*
  - b) *Obtener ciudad de procedencia.*
  - c) Modificar profesión.
  - d) Modificar ciudad de referencia.
- 



## Ejercicio 1

---

Añade dos atributos a la clase *Persona* para guardar la ciudad de residencia y la profesión de cada una.

Modifica también el constructor parametrizado para adaptarlo a los nuevos atributos.

---



## Ejercicio 3

---

Actualiza el pseudo-diagrama UML para incorporar los cambios realizados hasta ahora sobre la clase *Persona*.

---

# Batería de ejercicios sobre clases y objetos

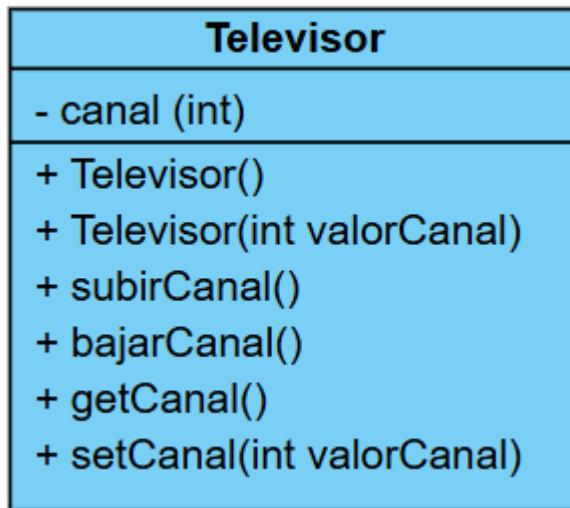
---



## CLASE TELEVISOR

---

1. Crea una clase de tipo *Televisor* que cumpla con el siguiente diagrama:



y teniendo en cuenta que el televisor tiene entre 1 y 99 canales.

2. Amplía la clase *Televisor* con un atributo adicional para el volumen (*int*). Añade también los getters y los setters.

3. Cuando creamos una instancia de *Televisor*, esta ya tendrá el volumen por defecto en el nivel 5, y el canal por defecto en el 1.

4. Por lo que respecta a la implementación de los métodos que modifican el canal y el volumen, no tendremos en cuenta los valores negativos ni tampoco el valor máximo(100). Deberemos controlar dichos métodos para que no hagan nada en esos casos concretos.

5. Al modificar tanto el volumen como el canal se deben mostrar mensajes indicando el nuevo nivel de volumen o el nuevo número de canal. Por ejemplo, en caso de modificar el volumen:

*"Volumen: valorVolumen"*

o en caso de retroceder un canal:

*"Canal: valorCanal"*

6. Para comprobar el funcionamiento, desde una clase principal *AplicacionTv* tenemos que:

- Crear instancias (objetos) de *Televisor*.
- Cambia los canales y el volumen para comprobar que todo funciona de forma correcta, y que cada objeto de tipo *Televisor* tiene su propio estado.



## CLASE PACIENTE

Crea una clase llamada *Paciente* que cumpla las siguientes condiciones:

1. Sus atributos son: nombre, edad, DNI, sexo (H hombre, M mujer), peso y altura. Si quieras añadir algún atributo puedes hacerlo. OJO: recuerda que no queremos que se acceda directamente a ellos.
2. Todos los atributos menos el DNI podrán admitir valores nulos según su tipo (0 números, cadena vacía para String, etc.). Sexo será "X". Usa una constante para ello.
3. Método *generarDNI()*: genera un número aleatorio de 8 cifras, y a partir de este número, su letra correspondiente. La forma de calcularlo es la siguiente:
  - El número del DNI obtenido aleatoriamente se divide entre 23, y se coge el resto de esa división.
  - Cada resto tiene asignada una letra específica según una tabla predeterminada. Usa el siguiente vector como referencia para el orden:

```
letras[] = {'T', 'R', 'W', 'A', 'G', 'M', 'Y',
'F', 'P', 'D', 'X', 'B', 'N', 'J', 'Z',
'S', 'Q', 'V', 'H', 'L', 'C', 'K', 'E'};
```

Este método será invocado cuando se construya el objeto. No debe ser visible al exterior, ya que se trata de un cálculo interno cada vez que se instancia un objeto desde otra clase.

4. Se implantarán varios constructores:

- Un constructor con todos los atributos como parámetros.
- Un constructor por defecto.

Prueba de estas dos formas...

```

public Persona2() {
    this.sexo = SEXO_DEF;
    generarDni();
}

public Persona2() {
    this("", 0, SEXO_DEF, 0, 0);
}

```

¿Funciona igual? ¿Por qué?

- Un constructor con el nombre, edad y sexo, el resto por defecto.

5. Los métodos que se implementarán son:

- ***calcularIMC():*** calculará si la persona esta en su peso ideal (peso en kg)/(altura^2 en metros). Usa el método Math.pow
  - Si esta fórmula devuelve un valor menor que 20, la función devuelve un -1.
  - Si devuelve un número entre 20 y 25 (incluidos), significa que esta por debajo de su peso ideal la función devuelve un 0 y,
  - Si devuelve un valor mayor que 25 significa que tiene sobrepeso, la función devuelve un 1.

Usa constantes para devolver estos valores.

- ***esMayorDeEdad():*** indica si es mayor de edad, devuelve un booleano.
- ***comprobarSexo(char sexo):*** comprueba que el sexo introducido es correcto. Si no es correcto, será "X". No será visible al exterior.
- ***mostrarInfoPaciente():*** devuelve toda la información del objeto.
- Métodos ***get/set*** de cada parámetro, excepto del atributo DNI.

6. Ahora, crea una clase ejecutable que haga lo siguiente:

- Pide por teclado el nombre, la edad, sexo, peso y altura.
- Crea 3 objetos de la clase anterior:
  - El primer objeto obtendrá las anteriores variables pedidas por teclado.
  - El segundo objeto obtendrá todos los anteriores menos el peso y la altura.
  - El último por defecto. Para este último, utiliza los métodos set para darle un valor a los atributos después de crearlo.
- Para cada objeto, el programa principal deberá comprobar si está en su peso ideal, tiene sobrepeso o por debajo de su peso ideal con un mensaje.
- Indicar para cada objeto si es mayor de edad.
- Por último, mostrar la información de cada objeto.



Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0  
<<http://creativecommons.org/licenses/by-sa/4.0/>>

## toString y static

### 5.2. El método `toString()` y uso de `static` en la POO

En Java, al trabajar con clases y objetos, hay conceptos clave que nos permiten gestionar tanto el comportamiento de las clases como la forma en que representamos a los objetos en nuestros programas. Dos de esos conceptos fundamentales son los componentes estáticos (`static`) y el método `toString()`.

## 5.2.1. El método `toString()`

---

En Java, la clase `Object` (que es la madre de todas las clases) proporciona un **método llamado `toString()`**. Este método devuelve una representación en forma de cadena de texto (`String`) del objeto, de forma que podemos imprimir el nombre de la clase y el valor *hash* del objeto, lo cual no es muy útil para los usuarios del programa. Es lo mismo que nos pasaba cuando intentábamos imprimir *arrays*:

```
1 public class Main {  
2     public static void main(String[] args) {  
3         String nombres[] = {"Pepe", "Juan"};  
4         System.out.println(nombres);  
5  
6         Persona persona = new Persona("Juan", 30);  
7         System.out.println(persona);  
8  
9     }  
10 }  
11 }
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" .  
[Ljava.lang.String;@7b23ec81  
org.example.Persona@5f184fc6
```

Es decir, al imprimir el objeto `persona`, Java llama automáticamente al método `toString()`. Como se puede ver, esta salida no es útil para describir los datos del objeto. Por esta razón, es importante sobrescribirlo para proporcionar una representación más significativa de nuestros objetos. Por ejemplo:

```
"C:\Program Files\Java\jdk-23\bin\java.exe"  
[Pepe, Juan]  
Persona [nombre=Juan, edad=30]
```

### Re-implementación del Método `toString()`

Imaginemos que tenemos una clase `Persona` y queremos sobrescribir el método `toString()` para mostrar de forma legible los detalles relevantes de una persona. La forma de hacerlo es la siguiente:

```
1 class Persona {  
2     String nombre;  
3     int edad;  
4  
5     Persona(String nombre, int edad) {  
6         this.nombre = nombre;  
7         this.edad = edad;  
8     }  
9  
10    @Override  
11    public String toString() {  
12        return "Persona [nombre=" + nombre + ", edad=" + edad + "]";  
13    }  
14}
```

### ¿Qué es ese `@Override`?

En este caso, `@Override` es una anotación de Java que se escribe encima del método a sobrescribir (`toString()`), e indica que un método en una subclase (`Persona`) está sobrescribiendo (o redefiniendo) un método de su clase madre (`Object`). Lo veremos más en profundidad cuando hablamos de Herencia en el próximo tema.

Si ejecutamos ahora el `main` anterior, obtendríamos la siguiente salida que ya es legible y proporciona información útil sobre el objeto:

```
1 public class Main {  
2     public static void main(String[] args) {  
3  
4         Persona persona = new Persona("Juan", 30);  
5         System.out.println(persona);  
6     }  
7 }  
8 }
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-  
Persona [nombre=Juan, edad=30]
```



### Ejercicio de ejemplo

---

Abre tu proyecto creado recientemente y reemplaza en todas las Clases los métodos de tipo *mostrarInfo()* creados hasta ahora por el método *toString()*.

Modifica también los programas principales que usaban dichas Clases para adaptarlos al nuevo método.

---

## 5.2.2. Métodos y atributos estáticos (static)

En Java, la palabra reservada *static* se utiliza para declarar atributos o métodos que pertenecen a la clase, pero no a las instancias de la clase. Los elementos estáticos son compartidos por todas las instancias de la clase, y no es necesario crear un objeto para acceder a ellos.

### Atributos estáticos

Teniendo en cuenta lo que acabamos de comentar, podemos decir que un atributo estático pertenece a la clase y no a los objetos creados a partir de la clase. Por lo tanto, cambiar el valor de un atributo estático afecta a todas las instancias de la clase.

```
1 public class Contador {  
2  
3     public static int contador = 0;  
4  
5     Contador() {  
6         contador++;  
7     }  
8  
9 }  
10  
11 public class Main {  
12     public static void main(String[] args) {  
13  
14         Contador obj1 = new Contador();  
15         Contador obj2 = new Contador();  
16         Contador obj3 = new Contador();  
17  
18         System.out.println("Objetos creados: " + Contador.contador); //  
19     }  
20 }  
21 }
```

### Métodos estáticos

Como ya sabemos del tema anterior, un método estático puede ser invocado sin crear una instancia (objeto) de la clase. Pero en la POO, deberemos cumplir ciertas reglas si

queremos crearlos como *static*:

- No deben acceder a atributos o métodos de instancia directamente.
- Sólo pueden usar atributos y métodos estáticos (o parámetros, si es que reciben alguno).
- No pueden usar la palabra clave *this*.

```
public class Punto {  
    ...  
    //atributo estático de la clase  
    private static int cantidadPuntos;  
    //cada instancia incrementa este atributo  
    public Punto ( ) {  
        cantidadPuntos++;  
    }  
    //método estático que retorna un atributo estático  
    public static int getPuntos( )  
        return cantidadPuntos;  
    }  
    ...  
}  
  
public class App {  
    public static void main (String[] args) {  
        //se crean instancias  
        Punto p1 = new Punto ( );  
        Punto p2 = new Punto ( );  
        Punto p3 = new Punto ( );  
        //accedemos al método estático para ver el  
        //número de instancias de tipo Punto creadas  
        System.out.println(Punto.getPuntos( ));  
    }  
}
```

En resumen, **solamente se usan como veníamos haciendo hasta ahora: sin aplicar ni usar nada característico de la POO.**

### Cuándo usar *static*...

- Atributos estáticos: cuando se necesita compartir un valor común entre todas las instancias de la clase. Por ejemplo: contadores, variables globales para valores por defecto, etc.
- Métodos estáticos: cuando el método no depende de atributos de instancia. Por ejemplo, métodos de utilidad como cálculos matemáticos, manipulación de fechas, etc.
- Constantes: cuando los valores deben ser fijos y accesibles globalmente.



### Ejercicio de ejemplo

Crea una clase *Estudiante* con:

- Un atributo estático *contadorEstudiantes* para contar cuántos estudiantes han sido creados.

- **Atributos:** *nombre*, *curso*, *NIA* y *email*. El NIA se debe generar automáticamente de forma correlativa cada vez que se cree o instancie un estudiante nuevo.
- Dos constructores: uno parametrizado con todos los atributos (menos el NIA) y otro que solamente reciba *nombre*.
- Un método estático ***obtenerTotalEstudiantes()*** que devuelva el valor del contador.
- Un método estático ***validarCorreo()*** para validar el formato del correo electrónico.

Usa la siguiente instrucción:

```
correo.matches("[A-Za-z0-9+_.-]+@alu.edu.gva.es$")
```

donde:

- ^ : indica el inicio de la cadena.
- [A-Za-z0-9+\_.-]+ : acepta letras (mayúsculas y minúsculas), números, y los caracteres +, ., , y -.
- El + al final significa "una o más veces".
- @alu.edu.gva.es: especifica que el correo debe terminar exactamente con este dominio.
- \$ : indica el final de la cadena.

- 
- Crea una constante ***CORREO\_FORMAT*** de tipo *static* con el contenido del *matches* anterior y actualízalo para usar la nueva constante.
  - Sobrescribe el método ***toString()*** para mostrar la info de cada estudiante.

# Ejercicio de ampliación sobre clases/objetos y métodos static

---

Crea una clase llamada *Libro* con las siguientes características:

## Atributos:

- *título*
- *autor*
- *id* (número autogenerado como LIB001, LIB002,... cada vez que se crea uno nuevo).
- *disponible* (booleano): indica si el libro está prestado o no.
- *cantidadLibros* (estático): lleva el conteo del número total de libros creados.
- *librosDisponibles* (estático): rastrea el número de libros que no están prestados.

## Constructores:

- Un constructor que inicialice el título y el autor. El atributo *id* se debe calcular automáticamente y el atributo *disponible* debe inicializarse como *true*.
- Incrementa los contadores estáticos de libros *cantidadLibros* y *librosDisponibles* al crear un nuevo libro.

## Métodos:

- *calcularId()*
- *prestar()*: cambia el estado de *disponible* a *false* y muestra el mensaje "El libro ha sido prestado con éxito.". Además, deberemos actualizar la variable estática *librosDisponibles* (-1).

En caso de que el libro ya esté prestado, muestra un mensaje indicando que no se puede prestar.

- *devolver()*: cambia el estado de disponible a *true* y muestra el mensaje "El libro ha sido devuelto con éxito.". Además, deberemos actualizar la variable estática *librosDisponibles* (+1).

Si no estaba prestado, muestra un mensaje indicando que no se puede devolver.

- *estaDisponible()*: devuelve el valor de disponible.
- Método estático *getTotalLibros()*: devuelve la cantidad total de libros creados.
- Método estático *getLibrosDisponibles()*: devuelve la cantidad total de libros creados.
- Sobrescribe *toString()* para mostrar la info de cada *Libro*.

```
Total de libros creados: 2
Libros disponibles: 2
El libro 'Cien años de soledad' ha sido prestado.
Libros disponibles después del préstamo: 1
El libro 'Cien años de soledad' ha sido devuelto.
Libros disponibles después de la devolución: 2
```

---



Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0  
<http://creativecommons.org/licenses/by-sa/4.0/>

## 5.3. Relaciones simples entre clases

### 5.3. Relaciones simples entre clases

Normalmente, las clases que definimos en un programa para resolver un problema no están aisladas, sino que se relacionan entre sí. Por ejemplo, imaginemos que estamos gestionando las matrículas de alumnos en un centro educativo...

Podríamos tener una clase *Alumno* con sus atributos (número de expediente, nombre, etc.) y una clase *Asignatura* con los datos que nos interesen de las asignaturas que oferta el centro (código de la asignatura, nombre, horas, etc.). Si dejamos estas clases sueltas y nos limitamos a crear objetos de cada una (es decir, los diferentes alumnos y asignaturas del centro educativo) no tendremos forma de saber qué asignaturas tiene cada alumno.

Otro ejemplo sería la clase *Tutor*, que no tiene sentido si no tiene a un *Grupo* como referencia:



Para ello, deberemos relacionar las clases entre sí, de forma que puedan "verse" y desde un objeto de una clase poder saber los datos de otro (u otros) objeto(s) de la otra. Veamos las opciones de las que disponemos para relacionarlas a continuación.

## 5.3.1. Asociaciones/agregaciones

---

El tipo de relación más frecuente es la **asociación**, que es un tipo de relación entre clases que se podría definir como "*tener un*".

En el ejemplo anterior, cada *Alumno* inscrito en el centro educativo tiene unas asignaturas matriculadas. Por lo tanto, se puede dar una asociación entre la clase *Alumno* y la clase *Asignatura*, que básicamente se plasmaría añadiendo en una de las dos clases (o en las dos, dependiendo de lo que nos interese) un atributo del tipo de la otra.

Lo mismo para el *Tutor* y el *Grupo* visto anteriormente. Un ejemplo en *Java*:

```
1 public class Tutor {  
2     private String nombre;  
3     private String especialidad;  
4     private Grupo grupo;  
5  
6 }  
7  
8 public class Grupo {  
9     private String nombre_grupo;  
10  
11 }
```

o al revés:

```
1 public class Tutor {  
2     private String nombre;  
3     private String especialidad;  
4  
5 }  
6  
7 public class Grupo {  
8     private String nombre_grupo;  
9     private Tutor tutor;  
10  
11 }
```

Dada la asociación,



se debe especificar:

- En el centro, el nombre de la asociación.
- Con una flecha se especifica la navegabilidad. Partiendo del nombre de la asociación y los métodos de las clases, se debe poder establecer cuál es la clase origen y cuál el destino.
- La navegabilidad indica el sentido de las interacciones entre objetos: a qué clase pertenecen los objetos que pueden llamar operaciones y a qué clase los objetos que reciben estas llamadas.
- Si no se especifica navegabilidad (sin flecha) se tratará de una asociación bidireccional, y ambas clases podrán llamar a métodos de la otra. La vemos en el siguiente apartado.
- En cada extremo se indica la cardinalidad (como en BBDD), que debe especificar con cuantas instancias de una de las clases puede estar enlazada una instancia de la otra clase en un momento determinado de la ejecución del programa.

Diferentes cardinalidades y su significado	
1	Sólo un enlace
0...1	Ninguno o un enlace
2,4	Dos o cuatro enlaces
1...5	Entre 1 y 5 enlaces
1...*	Entre 1 y un número indeterminado, es decir, más de 1
*	Un número indeterminado. Es equivalente a 0...*

- Por lo tanto, un objeto de la clase *Tutor* siempre tiene un objeto de la clase *Grupo* enlazado (ya que un tutor siempre tutoriza a un grupo).
- No se puede dar el caso de que un tutor no tutorice a ningún grupo. La inversa también es cierta, todo grupo es tutorizado por algún tutor.
- El tutor puede llamar operaciones sobre el grupo, pero no al revés. Esto tiene sentido, ya que es el tutor el que controla al grupo.

Como ves, la **navegabilidad** y la **cardinalidad** son imprescindibles, ya que la decisión que se tome en estos aspectos dentro de la etapa de diseño tendrá implicaciones directas sobre la implementación.

Más ejemplos:



Con el siguiente símbolo ya se dice cuál es la relación entre los objetos de ambas clases, por lo que se pueden omitir el nombre y la función en los descriptores de la asociación.



- Una instancia cualquiera de la clase *Página* puede enlazar hasta un número indeterminado de objetos diferentes de la clase *Cita*.
- Dado un objeto cualquiera de la clase *Cita*, sólo estará enlazado a un único objeto *Página*. Por lo tanto, no se puede tener una misma cita en dos páginas diferentes (pero sí tener dos citas diferentes y de contenido idéntico, con los mismos valores para los atributos, cada una a una página diferente).
- Tampoco puede haber citas que no estén escritas en ninguna página.



Dado un cliente, este puede estar registrado en más de una sucursal, pero al menos siempre lo estará en una. Nunca se puede dar el caso de un cliente dado de alta en el sistema pero que no esté registrado en ninguna sucursal.



## Ejercicio

Crea una relación entre la clase *Estudiante* y la clase *Libro* implementadas anteriormente. Hazlo así:

- a) Crea un atributo de tipo Estudiante llamado ***estudiantePrestado*** para almacenar al estudiante que tiene el libro prestado. Cuando se crea un libro, este atributo tendrá un valor nulo.
- b) Los métodos ***prestar()*** y ***devolver()*** gestionarán esta relación.

- En el caso de ***prestar()***, deberemos añadir un parámetro a recibir por el método para poder guardar la info en el atributo ***estudiantePrestado***.
- En el caso de ***devolver()***, no modificaremos los parámetros y volveremos a poner el valor del atributo ***estudiantePrestado*** a nulo.

```
El libro 'Cien años de soledad' ha sido prestado a Juan Pérez  
El libro 'Don Quijote de la Mancha' ha sido prestado a María López  
El libro 'Cien años de soledad' ha sido devuelto por Juan Pérez
```

- c) Dibuja el diagrama de la relación creada.
-

## 5.3.2. Asociaciones bidireccionales

---

En el caso del ejercicio anterior, donde la clase *Estudiante* no contiene información sobre los libros que tiene prestados, la relación es una asociación unidireccional desde *Libro* hacia *Estudiante*. Esto significa que la clase *Libro* conoce a la clase *Estudiante* mediante el atributo *estudiantePrestado*, pero sin embargo, la clase *Estudiante* no sabe qué libros tiene asociados. Sólo almacena su propia información (*nombre, curso, nia, email*).

Teniendo en cuenta esto... ¿es correcto que *Estudiante* no contenga info de *Libro*?

Pues depende del diseño y los requisitos del sistema.

- Será correcto si sólo necesitamos saber qué estudiante tiene un libro prestado (vista desde el lado de *Libro*). Es decir, queremos mantener la simplicidad y no estamos preocupados por mostrar todos los libros que un estudiante tiene prestados.
- No, no es correcto si lo que queremos es que el *Estudiante* pueda rastrear los libros que tiene prestados. En este caso tendríamos que hacer una asociación bidireccional.

¿Cómo cambiar a una relación a bidireccional?

Si lo que queremos es que la relación sea bidireccional (el estudiante también conoce los libros que tiene prestados), podemos añadir un atributo *libroPrestado* de tipo *Libro* a la clase *Estudiante*, junto a sus correspondientes *getters/setters*:

```
1 private Libro libroPrestado;
2
3 public Libro getLibroPrestado() {
4     return libroPrestado;
5 }
6
7 public void setLibroPrestado(Libro libroPrestado) {
8     this.libroPrestado = libroPrestado;
9 }
```

y en la clase *Libro*, añadir lógica al método *prestar(Estudiante estudiante)* para controlar si el estudiante al que le estamos intentando asignar un libro ya tiene alguno reservado (suponemos que solamente puede reservar uno).

Por ejemplo:

```
1 public void prestar(Estudiante estudiante) {  
2     if (prestado) {  
3         System.out.println("El libro ya está prestado a " + estudianteP  
4     } else if (estudiante.getLibroPrestado() != null) {  
5         System.out.println("El estudiante ya tiene un libro prestado.")  
6     } else {  
7         this.prestado = true;  
8         this.estudiantePrestado = estudiante;  
9         estudiante.setLibroPrestado(this); // Relación bidireccional  
10        System.out.println("El libro '" + titulo + "' ha sido prestado")  
11    }  
12}
```

Ahora, tanto *Libro* como *Estudiante* están vinculados, por lo que ya podremos consultar la información desde ambos lados.



## Ejercicio 1

- 
- a) Actualiza las clases *Estudiante* y *Libro* para incorporar la relación de bidireccionalidad.
  - b) Modifica también el diagrama de relación entre ambas clases.
- 



## Ejercicio 2

- 
- a) Crea una nueva clase llamada *Editorial* (*nombre, pais*).
  - b) Esta nueva clase puede representar la *Editorial* que publica los libros, por lo tanto, crea una asociación donde cada *Libro* "conozca" a su editorial.
  - c) Añade la clase *Editorial* al diagrama de relación.
- 



## Ejercicio 3 (asociaciones ternarias)

---

Una clase *Prestamo* podría modelar la relación entre un *Libro* y un *Estudiante* cuando se presta un libro. Esto introduce lo que llamamos una asociación ternaria (entre tres clases).

- a) Crea una nueva clase *Prestamo* con los siguientes atributos: uno de tipo *Estudiante*, otro de tipo *Libro* y, por último, una de tipo *LocalDateTime* que contenga la *fecha* del préstamo realizado. Solamente debe haber un constructor que reciba todos los atributos necesarios por parámetro.
  - b) Modifica el método *prestar()* de la clase *Libro* para que se cree un objeto de tipo *Prestamo* cada vez que se realice uno. Inicializa el parámetro *fecha* con la función *LocalDate.now()*.
  - c) Añade la clase *Prestamo* al diagrama de relación entre clases.
-

## BONUS: Paso por valor y por referencia

---

Como has podido observar al modificar la clase *Libro*, hemos necesitado pasar como parámetro a un objeto de tipo *Estudiante* para poder prestar y devolver libros mediante la relación de bidireccionalidad. Pero, ¿te has dado cuenta de lo que realmente hemos hecho?

Igual que nos pasaba con los *arrays*, cuando pasamos un *Objeto* por parámetro **NO** estamos realizando una copia, sino que se pasa por referencia (dirección de memoria): es decir, que **voy a afectar directamente al objeto** y no a una copia.

En nuestro caso, hemos modificado el objeto de tipo *Estudiante* original para "setear" su atributo relacionado con el *Libro*:

```
1 public void prestar(Estudiante estudiante){  
2  
3     if(disponible && estudiante.getLibroPrestado() == null) {  
4         disponible = false;  
5         System.out.println("El libro " + getTitulo() + " ha sido pr  
6         librosDisponibles--;  
7         estudiantePrestado = estudiante;  
8         estudiante.setLibroPrestado(this); // MODIFICAMOS EL OBJETO  
9     } else if (estudiante.getLibroPrestado() != null) {  
10         System.out.println("El estudiante " + estudiante.getNom  
11     }else{  
12         System.out.println("El libro " + getTitulo() + " no se p  
13     }  
14 }  
15 }
```

Por lo tanto:

- En *Java*, los tipos primitivos (int, char, boolean,...) siempre se pasan por valor a los métodos.
- Los objetos y *arrays* siempre se pasan por referencia a los métodos.

---

### Clases inmutables

Por otra parte, existen casos especiales como los *Strings*, ya que estos siguen siendo objetos, pero de tipo inmutables. Es decir, una vez se ha creado una *String*, no se pueden modificar ninguno de sus caracteres sin crear una nueva instancia. Cualquier operación que se realice en un objeto *String*, como concatenar, cambiar mayúsculas o minúsculas, etc, genera un nuevo objeto *String* con los cambios realizados.

```
String s = "Hola";
s = s + " Mundo!"; // Crea un nuevo objeto con el contenido "Hola Mundo!"
```

En Java existen dos formas de crear un objeto de la clase *String*: mediante una asignación literal, y mediante el uso del constructor *new String()*:

```
String cadena1 = "Hola Mundo";
String cadena2 = new String("Hola Mundo");
```

La principal diferencia entre estas dos formas es que, cuando se crea una *String* mediante una asignación literal, la JVM busca en su pool de *Strings* si existe una *String* igual y, si existe, retorna su dirección en memoria. Si no existe, la JVM crea una nueva *String* en el

pool de *Strings* y retorna su dirección en memoria.

En cambio, cuando se crea una *String* mediante el constructor *new String()*, se crea un nuevo objeto *String* en la memoria, independientemente de si existe o no una *String* igual en el pool de *Strings*. Por lo tanto, **es recomendable utilizar asignaciones literales para crear *Strings* porque son más eficientes**, ya que evitan la creación de objetos innecesarios.

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

    int num = 1;
    String nombre = "String inmutable";
    Punto p1 = new Punto(1,1);

    modificarPunto(p1);
    modificarEntero(num);
    modificarString(nombre);

    System.out.println("Punto.x = " + p1.getX());    // Punto.x = 999
    System.out.println("num = " + num);                // num = 1
    System.out.println("nombre = " + nombre);          // nombre = String inmutable
}

public static void modificarPunto(Punto p)    { p.setX(999); }
public static void modificarEntero(int n)     { n = 999; }
public static void modificarString(String s)  { s = "String modificada"; }
```

### 5.3.3. Composiciones

La **composición** es una relación fuerte "*parte de*", en la cual una clase contiene otra, y si la clase contenedora se elimina, los objetos dentro también desaparecen.

Este tipo de asociación se representa de forma idéntica a una *agregación*, sólo que en este caso el rombo es de color negro.



#### Ejemplo

👉 Podríamos tener el siguiente escenario...

- Una Casa tiene Habitaciones.
- Una Habitación no puede existir sin una Casa.
- Si la Casa se destruye, todas sus Habitaciones desaparecen.

Esto podría implementarse en *Java* de la siguiente manera:

- Clase *Habitacion*:

```
1 class Habitacion {  
2     private String nombre;  
3  
4     public Habitacion(String nombre) {  
5         this.nombre = nombre;  
6     }  
7  
8     public String getNombre() {  
9         return nombre;  
10    }  
11  
12    @Override  
13    public String toString() {  
14        return nombre;  
15    }  
16}
```

- Clase *Casa*:

```
1 class Casa {  
2     private String direccion;  
3     private List<Habitacion> habitaciones; // COMPOSICIÓN: la Casa tiene Habitaciones  
4  
5     public Casa(String direccion) {  
6         this.direccion = direccion;  
7         this.habitaciones = new ArrayList<>();  
8     }  
9  
10    public void agregarHabitacion(String nombre) {  
11        Habitacion habitacion = new Habitacion(nombre); //solamente la Casa crea las Habitaciones  
12        habitaciones.add(habitacion);  
13    }  
14  
15    public void mostrarHabitaciones() {  
16        System.out.println("Casa en " + direccion + " tiene las siguientes Habitaciones:");  
17    }  
18}
```

- *Main* de prueba:

```
1 public class Main {  
2     public static void main(String[] args) {  
3  
4         Casa casa = new Casa("Calle Mayor 123");  
5  
6         casa.agregarHabitacion("Dormitorio");  
7         casa.agregarHabitacion("Cocina");  
8         casa.agregarHabitacion("Baño");  
9  
10        casa.mostrarHabitaciones();  
11    }  
12}  
13}
```

- La clase Casa contiene Habitaciones dentro de sí misma.
- Las Habitaciones se crean dentro de Casa, no fuera de ella.
- Si eliminamos la Casa, sus Habitaciones también desaparecen.



## Ejercicios

---

- a) Añade el atributo *metrosCuadrados* y sus correspondientes getters y setters a la clase *Habitación*.
- b) Modificar el método *mostrarHabitaciones()* para incluir el número total de habitaciones y su tamaño correspondiente.

```
Casa en Plaza España 123 tiene 3 habitaciones:  
- Dormitorio (12.5 m2)  
- Cocina (8.2 m2)  
- Baño (5.0 m2)
```

- c) Crear un método *public Habitacion getHabitacionMasGrande()* que devuelva la habitación más grande de la Casa.
- d) Modifica el método *agregarHabitacion()* para evitar que se añadan habitaciones con el mismo nombre.
- e) Implementa un método *eliminarHabitacion(String nombre)*, que elimine una habitación (si existe).
- f) Añadir una clase *Propietario* y asociarla a la *Casa* mediante composición.

--> *Propietario* debe tener los atributos *nombre* y *edad*.  
--> La *Casa* debe contener un *Propietario* (un propietario no existe sin una casa).  
--> Modifica *Casa* para que el propietario sea obligatorio y mostrar su información en *mostrarHabitaciones()*.

- g) Crear una clase *Electrodomestico* y asociarla a la *Habitacion* mediante composición. (Si una *Casa* es destruida, todas sus *Habitaciones* y los *Electrodomésticos* en ellas deben ser destruidos también).

--> *Electrodomestico* tendrá los atributos *nombre* y *consumo* energético (kWh).  
--> Cada habitación contiene una lista de electrodomésticos.

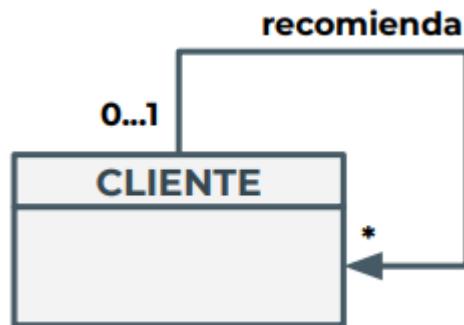
- h) Crea los métodos *agregarElectrodomestico()* y *mostrarElectrodomesticos()* en la clase *Habitacion*. Evita añadir electrodomésticos con el mismo nombre dentro de la misma habitación.

- i) Crea un método *public double calcularConsumo()* en la clase *Habitación*, que calcule el consumo total en esa estancia.

- j) Crea un método en la clase *Casa* que haga uso del que acabas de crear en el apartado anterior, para mostrar qué habitación consume más energía dentro de la casa.

## 5.3.4. Asociaciones reflexivas

Una asociación reflexiva ocurre cuando una clase tiene una relación con ella misma. En otras palabras, un objeto de la clase puede estar asociado con otro objeto de la misma clase.



### Un ejemplo...

Imaginemos un *Empleado* que tiene un *Supervisor*, y el *Supervisor* también puede ser un *Empleado*. Es decir, un *Empleado* puede ser supervisado por otro *Empleado*, creando una relación reflexiva entre objetos de la misma clase.

```
class Empleado {
    private String nombre;
    private String puesto;
    private Empleado supervisor; //RELACIÓN REFLEXIVA

    public Empleado(String nombre, String puesto) {
        this.nombre = nombre;
        this.puesto = puesto;
        this.supervisor = null; //No tiene supervisor inicialmente
    }

    //Método para asignar supervisor
    public void asignarSupervisor(Empleado supervisor) {
        this.supervisor = supervisor;
    }

    //Método para obtener supervisor
    public Empleado getSupervisor() {
        return supervisor;
    }
}
```

```
22     public void mostrarInfo() {
23         System.out.println("Empleado: " + nombre + " - Puesto: " + puest
24         if (supervisor != null) {
25             System.out.println("Supervisor: " + supervisor.nombre);
26         } else {
27             System.out.println("Este empleado no tiene supervisor.");
28         }
29     }
30 }
```

Con el siguiente *Main*:

```
1  public class Main {
2      public static void main(String[] args) {
3
4          Empleado empleado1 = new Empleado("Juan", "Desarrollador");
5          Empleado empleado2 = new Empleado("Ana", "Gerente");
6          Empleado empleado3 = new Empleado("Pedro", "Supervisor");
7
8          empleado1.asignarSupervisor(empleado2); //Juan es supervisado
9          empleado2.asignarSupervisor(empleado3); //Ana es supervisada por Pedro
10
11         empleado1.mostrarInfo();
12         empleado2.mostrarInfo();
13         empleado3.mostrarInfo();
14
15     }
16 }
```

```
Empleado: Juan - Puesto: Desarrollador
Supervisor: Ana
Empleado: Ana - Puesto: Gerente
Supervisor: Pedro
Empleado: Pedro - Puesto: Supervisor
Este empleado no tiene supervisor.
```



## Ejercicio

---

Modifica el ejemplo anterior para que, además de un supervisor, los empleados puedan tener compañeros de trabajo que también sean de tipo *Empleado*.

---



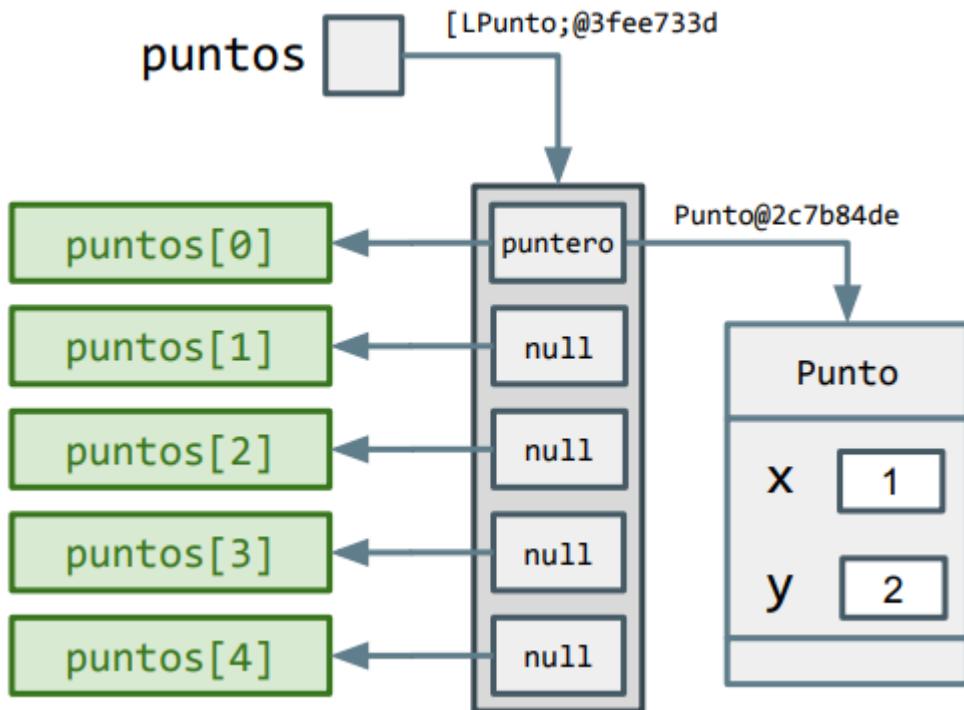
Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0  
<<http://creativecommons.org/licenses/by-sa/4.0/>>

## BONUS. ArrayList de objetos

### BONUS. ArrayList de objetos

El uso de vectores o *arrays* no tiene por qué restringirse a elementos de tipo primitivo. Por ejemplo, pueden crearse referencias a *arrays* de la clase *Punto*:

```
Punto[] puntos = new Punto[5];
puntos[0] = new Punto(1,2);
```



Pero en la *POO* no es común ceñirnos a los *arrays* tradicionales, sino usar otro tipo de listas "dopadas" para manipular mejor nuestros objetos. Estas listas son las llamadas ***ArrayList***.

Un *ArrayList* permite almacenar elementos en memoria de manera dinámica. La principal diferencia con los *arrays* es que el número de elementos que almacena no está limitado por un número fijado al inicio.

La declaración de un *ArrayList* se hace según el siguiente formato:

```
1 | ArrayList<nombreTipoClase> nombreDeLista = new ArrayList<>();
```

Entre <> indicamos la clase o tipos básicos de los objetos que se almacenarán:

```
ArrayList<Integer> listaNums = new ArrayList<>();  
ArrayList<Boolean> listaBooleans = new ArrayList<>();  
ArrayList<Double> listaDoubles = new ArrayList<>();  
ArrayList<String> listaPalabras = new ArrayList<>();  
ArrayList<Punto> listaPuntos = new ArrayList<>();
```

```
1 | ArrayList<String> listaPaises = new ArrayList<>();
```

## Métodos típicos de los *ArrayList*

- El método *add* posibilita añadir elementos, que se colocan después del último elemento que existiera en el *ArrayList*.

```
ArrayList<String> listaPaises = new ArrayList<>();  
listaPaises.add("España"); //Ocupa la posición 0  
listaPaises.add("Francia"); //Ocupa la posición 1  
listaPaises.add("Portugal"); //Ocupa la posición 2
```

```
1 | ArrayList<String> listaPaises = new ArrayList<>();  
2 | listaPaises.add("España"); //Ocupa la posición 0  
3 | listaPaises.add("Francia"); //Ocupa la posición 1  
4 | listaPaises.add("Portugal"); //Ocupa la posición 2
```

Es posible insertar un elemento en una determinada posición, desplazando el elemento que se encontraba en esa posición, y todos los siguientes, una posición más. Se utiliza el método *add* indicando como primer parámetro el número de la posición:

```
...  
//El orden hasta ahora es: España, Francia, Portugal  
listaPaises.add(1, "Italia");  
//El orden ahora es: Italia, España, Francia, Portugal
```

```
1 ArrayList<String> listaPaises = new ArrayList<>();  
2 listaPaises.add("España");  
3 listaPaises.add("Francia");  
4 listaPaises.add("Portugal");  
5 //El orden hasta ahora es: España, Francia, Portugal  
6 listaPaises.add(1, "Italia");  
7 //El orden ahora es: España, Italia, Francia, Portugal
```

- El método **get** permite obtener el elemento almacenado en una determinada posición:

```
System.out.println(listaPaises.get(3));  
//Siguiendo el ejemplo anterior, mostraría: Portugal
```

```
1 ArrayList<String> listaPaises = new ArrayList<>();  
2 listaPaises.add("España");  
3 listaPaises.add("Francia");  
4 listaPaises.add("Portugal");  
5 //El orden hasta ahora es: España, Francia, Portugal  
6 listaPaises.add(1, "Italia");  
7 //El orden ahora es: España, Italia, Francia, Portugal  
8 System.out.println(listaPaises.get(3));  
9 //Siguiendo el ejemplo anterior, mostraría: Portugal
```

Si tuviéramos un *ArrayList* con otro tipo de datos, como objetos que hayamos creado:

```
ArrayList<Punto> puntos = new ArrayList<>();  
puntos.add(new Punto());  
puntos.add(new Punto(1));  
puntos.add(new Punto(1,3));  
  
puntos.get(2).mostrarDatos(); //sobre la instancia con  
coordenadas(1,3), se invocará al método mostrarDatos
```

- Para eliminar un determinado elemento se emplea el método **remove** al que se le puede indicar por parámetro un valor *int* con la posición a suprimir:

```
ArrayList<String> listaPaises = new ArrayList();
listaPaises.add("España");
listaPaises.add("Francia");
listaPaises.add("Portugal");
//listaPaises → España, Francia, Portugal
listaPaises.add(1, "Italia");
//listaPaises → España, Italia, Francia,
Portugal
listaPaises.remove(2);
//listaPaises → España, Italia, Portugal
listaPaises.remove("Portugal");
//listaPaises → España, Italia
```

```
1 | ArrayList<String> listaPaises = new ArrayList();
2 | listaPaises.add("España");
3 | listaPaises.add("Francia");
4 | listaPaises.add("Portugal");
5 | //listaPaises → España, Francia, Portugal
6 | listaPaises.add(1, "Italia");
7 | //listaPaises → España, Italia, Francia,
8 | Portugal
9 | listaPaises.remove(2);
10 | //listaPaises → España, Italia, Portugal
11 | listaPaises.remove("Portugal");
12 | //listaPaises → España, Italia
```

o bien, se puede **especificar el elemento a eliminar** si es encontrado en la lista:

```

ArrayList<Punto> puntos;
puntos = new ArrayList();
Punto p1 = new Punto(1,2);
Punto p2 = new Punto(1);
Punto p3 = new Punto(1,3);
Punto p4 = new Punto(3,3);
puntos.add(p1); //p1
puntos.add(p2); //p1,p2
puntos.add(p3); //p1,p2,p3
puntos.add(1, p4); //p1,p4,p2,p3
puntos.remove(2); //p1,p4,p3
puntos.remove(p1); //p4,p3

```

- El método ***indexOf*** retorna un valor *int* con la posición que ocupa el elemento que se indica por parámetro. Si no lo encuentra, devuelve -1.

```

String paisBuscado = "Francia";
int pos = listaPaises.indexOf(paisBuscado);
if(pos != -1)
    System.out.println(paisBuscado + " encontrado en la posición: " + pos);
else System.out.println(paisBuscado + " no se ha encontrado");

```

```

1  ArrayList<String> listaPaises = new ArrayList();
2  listaPaises.add("España");
3  listaPaises.add("Francia");
4  listaPaises.add("Portugal");
5  //listaPaises → España, Francia, Portugal
6  listaPaises.add(1, "Italia");
7  //listaPaises → España, Italia, Francia,
8  Portugal
9  listaPaises.remove(2);
10 //listaPaises → España, Italia, Portugal
11 listaPaises.remove("Portugal");
12 //listaPaises → España, Italia
13 String paisBuscado = "Francia";
14 int pos = listaPaises.indexOf(paisBuscado);
15 if(pos != -1)
16     System.out.println(paisBuscado + " encontrado en la posición: " + pos)
17 else System.out.println(paisBuscado + " no se ha encontrado");

```

- Podemos recorrer un *ArrayList* con un bucle **for** como lo haríamos con los *arrays* convencionales. Para obtener el número de elementos, se utiliza el método **size()**.

```
for(int i = 0; i < listaPaises.size(); i++) {
    System.out.println(listaPaises.get(i));
}
```

```
1 ArrayList<String> listaPaises = new ArrayList();
2 listaPaises.add("España");
3 listaPaises.add("Francia");
4 listaPaises.add("Portugal");
5 //listaPaises → España, Francia, Portugal
6 listaPaises.add(1, "Italia");
7 //listaPaises → España, Italia, Francia, Portugal
8
9 for(int i = 0; i < listaPaises.size(); i++) {
10     System.out.println(listaPaises.get(i));
11 }
```

Como ya sabes, el **foreach** nos permite recorrer los elementos de un *array* sin utilizar un índice. Aunque la flexibilidad es menor que el bucle *for* normal, es muy útil para recorrer de forma típica un *ArrayList*.

```
for (String s : listaPaises) {
    System.out.println(s);
}
```

```
System.out.println(listaPaises.toString());
```

- **set (posición, nuevoValor)**: modifica un elemento.

```
1 ArrayList<String> listaPaises = new ArrayList();
2 listaPaises.add("España");
3 listaPaises.add("Francia");
4 listaPaises.add("Portugal");
5 listaPaises.add(1, "Italia");
```

```
7 |         listaPaises.set(2,"Polonia");
8 |         System.out.println(listaPaises.get(2));
```

- **clear()**: borra todo el contenido de la lista.

```
1 |         ArrayList<String> listaPaises = new ArrayList();
2 |         listaPaises.add("España");
3 |         listaPaises.add("Francia");
4 |         listaPaises.add("Portugal");
5 |         listaPaises.add(1, "Italia");
6 |
7 |         listaPaises.clear();
8 |         System.out.println("El tamaño de la lista es " + listaPaises.size());
```

- Object **clone()**: retorna una copia de la lista.

```
1 |         ArrayList<String> listaPaises = new ArrayList();
2 |         listaPaises.add("España");
3 |         listaPaises.add("Francia");
4 |         listaPaises.add("Portugal");
5 |         listaPaises.add(1, "Italia");
6 |
7 |         Object listaPaises_copia = listaPaises.clone();
8 |         System.out.println(listaPaises_copia.toString());
```

- boolean **contains(Object o)**: retorna *true* si se encuentra el elemento en la lista, y *false* en caso contrario.

```
1 |         ArrayList<String> listaPaises = new ArrayList();
2 |         listaPaises.add("España");
3 |         listaPaises.add("Francia");
4 |         listaPaises.add("Portugal");
5 |         listaPaises.add(1, "Italia");
6 |
7 |
8 |         if(listaPaises.contains("Suiza")){
9 |             System.out.println("Tenemos a Suiza.");
10 |         }else{
```

```
12 |         System.out.println("No tenemos a Suiza.");
    }
```

- **boolean isEmpty()**: retorna *true* si la lista está vacía.

```
1     ArrayList<String> listaPaises = new ArrayList();
2     listaPaises.add("España");
3     listaPaises.add("Francia");
4     listaPaises.add("Portugal");
5     listaPaises.add(1, "Italia");
6
7     //listaPaises.clear();
8
9     if(listaPaises.isEmpty()){
10         System.out.println("La lista está vacía");
11     }else{
12         System.out.println("La lista no está vacía");
13     }
```

- **int lastIndexOf(Object o)**: devuelve el índice de la última ocurrencia del elemento. Si el elemento no se encuentra, devuelve -1.

```
1     ArrayList<String> listaPaises = new ArrayList();
2     listaPaises.add("España");
3     listaPaises.add("Francia");
4     listaPaises.add("Portugal");
5     listaPaises.add(1, "Italia");
6
7     System.out.println(listaPaises.lastIndexOf("Francia")); //2
8     System.out.println(listaPaises.lastIndexOf("Suiza")); // -1
```

- **addAll(Arrays.asList(elemento1, elemento2, elemento3,...))**: añadir más de un elemento a la lista.

```
ArrayList<String> listaPaises = new ArrayList();
listaPaises.add("España");
listaPaises.add("Francia");
listaPaises.add("Portugal");
listaPaises.add(1, "Italia");
```

```
7     listaPaises.addAll(Arrays.asList("Polonia","Suiza","Alemania"))
8     System.out.println(listaPaises.toString());
```

También se puede hacer desde un método a parte, usando **varargs**:

```
1  public class Main {
2      public static void main(String[] args) {
3
4          ArrayList<String> listaPaises = new ArrayList();
5          listaPaises.add("España");
6          listaPaises.add("Francia");
7          listaPaises.add("Portugal");
8          listaPaises.add(1, "Italia");
9
10         agregarVariosPaises(listaPaises,"Polonia","Suiza","Alemania");
11         System.out.println(listaPaises.toString());
12
13     }
14
15     public static void agregarVariosPaises(ArrayList<String> lista, Str
16         lista.addAll(Arrays.asList(paises));
17     }
18
19 }
```

# Ejercicios sobre ArrayList

---



## Ejercicios "lista de la compra"

---

- a) Crea una nueva lista de la compra y verifica si está vacía. Luego, añade un elemento y verifícalo de nuevo.
- b) Añade 5 elementos más, todos a la vez. Muestra el contenido de la lista tras hacerlo.
- c) Muestra el tercer elemento de la lista.
- d) Cambia el segundo elemento de la lista por "Queso".
- e) Comprueba si un producto específico (por ejemplo, "Leche") está en la lista.
- f) Implementa un control para que antes de añadir un nuevo elemento (*add*) se verifique si ya existe en la lista para evitar duplicados. Haz lo mismo para cuando se modifique un elemento (*set*).
- g) Muestra el número de elementos en la lista.
- h) Clona la lista actual.
- i) Limpia la lista clonada y verifica que está vacía.
- j) Compara las dos listas y verifica que no sean iguales con el método *equals()*.

***boolean sonIguales = lista1.equals(lista2);***

- k) Añade dos elementos a la lista clonada.
- l) Dada la lista de la compra original, muestra los elementos que faltan en la lista clonada. Usa el método *removeAll()*;

***ArrayList<String> faltantes = new ArrayList<>(original);***

***faltantes.removeAll(clonada);***

- m) Crea otra lista del mismo tamaño que la original y llénala con números enteros aleatorios [0,10], simulando los precios correspondientes a cada producto. Muestra las dos listas.

# Actividad. Incorporar ArrayList a clases Estudiante y Editorial

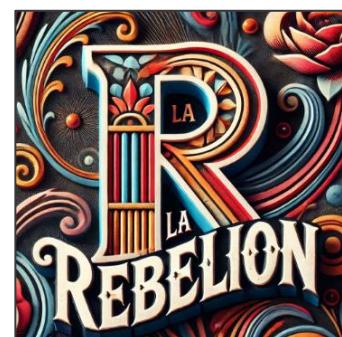
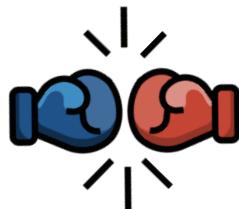
---

- a) Implementa los *ArrayList* para que un objeto de tipo Estudiante pueda almacenar más de un libro. Es decir, que pueda pedir prestado más de un libro.
- b) Adapta el constructor, los getters, los setters y el *toString* a la estructura nueva.
- c) Crea dos métodos nuevos: uno para insertar libros nuevos en el *ArrayList* (*anyadirLibro*) y otro para borrar un libro de la lista (*borrarLibro*).
- d) Modifica la condición del método *prestar()* de la clase Libro para que antes de asignarle un libro a un estudiante se compruebe si este ya tiene prestado el mismo libro (ya existe en la lista). Modifica también el camino feliz del método para que en lugar de llamar al set, se llame al nuevo método *anyadirLibro()*.
- e) Modifica el método *devolver()* de la clase Libro para que llame al método *eliminarLibro()* creado anteriormente.
- f) Cambia la relación entre Libro y Editorial para que tengan una relación bidireccional. Es decir, añade lógica para que Editorial contenga información de los libros que tiene publicados en un atributo de tipo *ArrayList*.
- g) Adapta los getters, los setters y el *toString* de la clase ***Editorial*** a la estructura nueva. Crea dos métodos nuevos, uno para añadir libros a la lista y otro para borrarlos.
- h) Modifica el constructor de la clase ***Libro*** para que cada vez que se cree uno nuevo, se añada a la lista de libros publicados de la clase ***Editorial***.



Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0  
<<http://creativecommons.org/licenses/by-sa/4.0/>>

## PRÁCTICA 1. "EL FORMIGUERO" Y "LA REBELIÓN"



### → INTRODUCCIÓN

El mundo de la televisión últimamente parece estar en guerra. Algunos “se rebelan” contra otros, en directo, los otros desmienten las informaciones... Y los telespectadores, lejos de analizarlo de forma crítica, se hacen palomitas para ver según qué programa en función de su ideología.

### → PROBLEMA A RESOLVER

Jordi está harto de la división entre “las dos Españas”, y decide intentar poner un poco de paz creando una aplicación que recoja datos diarios sobre estos programas para dar objetividad a las distintas informaciones que van apareciendo.

#### Paso 1. Construcción de las clases que van a interactuar entre ellas

La aplicación que necesita desarrollar nuestro protagonista constará de los siguientes elementos:

- Una clase que contendrá al programa principal llamado **“AppProgramas”**.
- Una clase **Cadena**, con los siguientes atributos:
  - *nombre (String)*
  - *listaProgramas (ArrayList<Programa>)*

Por defecto, se crea sin programas y se van agregando conforme se van *creando* programas relacionados con nuestra cadena.

- Una clase **Programa**, con los siguientes atributos:
  - *nombre (String)*
  - *cadena (Cadena)*
  - *temporadas (int)*
  - *listaEmpleados (ArrayList<Empleado>)*
  - *listaInvitados (ArrayList<Invitado>)*

- *director (Empleado)* – créalo en el mismo constructor y añádelo a la listaEmpleados

Por defecto, se crean con 0 temporadas y no tendremos empleados (excepto el director) ni invitados hasta que se vayan añadiendo conforme se va *contratando o invitando*.

- Una clase ***Empleado***, con los siguientes atributos:

- *id (String autogenerado EP001, EP002,...EP014,...).*
- *nombre (String)*
- *cargo (String)* – valores posibles: “**director**”, “**técnico**”, “**presentador**”, “**colaborador**”. Cualquiera que no sea uno de esos valores, **no se debe permitir añadirlo** como cargo y lo dejaremos con el valor por defecto “**pte**”.
- *director (Empleado)* – debe coincidir con el *director* del *Programa*. **Si el cargo ya es “director”, este atributo debe ser nulo.**

- Una clase ***Invitado***, con los siguientes atributos:

- *nombre (String)*
- *profesión (String)*
- *fecha\_visita (LocalDate)* – por defecto el día que se crea el *Invitado*
- *temporada (int)*

#### **A tener en cuenta...**

- Las clases de tipo *Empleado* e *Invitado* se relacionan mediante composición con la clase *Programa*, de forma que si un objeto de tipo *Programa* se destruye, desaparecen también sus empleados e invitados asociados.
- La clase *Cadena* y la clase *Programa* tienen relación de tipo asociación/agregación bidireccional, es decir, se conocen la una a la otra y ambas existen por sí solas. De forma que, si el día de mañana el programa “*La rebelión*” vuelve a su *Cadena* original, esta sigue manteniendo todas sus características y solamente necesitaremos modificar el valor del atributo *cadena* en la clase *Programa* y eliminar de la *listaProgramas* en la clase *Cadena* dicho programa.
- Ten en cuenta que debes implementar los getters, setters y sobrescribir el método *toString()*. En caso de que alguna clase tenga atributos de tipo *ArrayList*, debes implementar también los métodos responsables de añadir y borrar elementos a la lista.

#### Paso 2. Métodos extras a implementar

- a) Modifica el constructor de *Invitados* para que llame al set de *fecha\_visita* con **una fecha que le vamos a preguntar al usuario** para guardar cuándo tienen que ir al *Programa*.

Usa estas instrucciones como guía:

```
//crea un LocalDate a partir de un año, mes y día  
LocalDate fecha = LocalDate.of(2025, 03, 15);  
//con el formato montado, ya podemos usarlo en constructores o métodos que esperen datos LocalDate  
Persona persona = new Persona(fecha);
```

- b) Método **invitadosTemporada(int temporada)** que muestre **cuántos Invitados** han acudido al *Programa* dada una temporada. Muestra también sus **nombres** y **profesiones**.
- c) Método **int vecesInvitado(String nombre)** que devuelva las veces que ha ido un Invitado al *Programa*.
- d) Método **rastrearInvitado(String nombre)** que haga uso del método creado en el apartado anterior e imprima las veces que ha ido un Invitado al *Programa*, además de mostrar también en qué **fechas** y **temporadas**.
- e) Método **boolean buscarInvitado(String nombre)** para que dado un *Invitado* sea capaz de buscar si ha acudido a un *Programa*.
- f) Método **invitadoAntes(String nombre)** que use el método implementado en el apartado anterior para que en caso de haber devuelto *true* buscando en dos *Programas* distintos, muestre en cuál ha estado antes.

Usa estas instrucciones como guía:

```
//dadas dos fechas  
LocalDate fecha1 = LocalDate.of(2023, 5, 10);  
LocalDate fecha2 = LocalDate.of(2024, 1, 31);  
//las comparamos para saber cuál es mayor  
if (fecha1.isBefore(fecha2)) {  
    System.out.println("fecha1 es ANTES que fecha2");  
}
```

## Ejemplo de funcionamiento:

```
Cadena [nombre='Antena 3', listaProgramas=[]]

Programa{nombre='El Hormiguero', cadena=Antena 3, director=Empleado{nombre='Director1', id='EP001', cargo='director', director=null}, temporadas=0, listaEmpleados=[Empleado{nombre='Director1', id='EP001', cargo='director', director=null}], listaInvitados=[]}

Cadena [nombre='Antena 3', listaProgramas=[Programa{nombre='El Hormiguero', cadena=Antena 3, director=Empleado{nombre='Director1', id='EP001', cargo='director', director=null}, temporadas=0, listaEmpleados=[Empleado{nombre='Director1', id='EP001', cargo='director', director=null}], listaInvitados=[]}]]]

Programa{nombre='El Hormiguero', cadena=Antena 3, director=Empleado{nombre='Director1', id='EP001', cargo='director', director=null}, temporadas=0, listaEmpleados=[Empleado{nombre='Director1', id='EP001', cargo='director', director=null}, Empleado{nombre='Pablo Motos', id='EP002', cargo='presentador', director=Empleado{nombre='Director1', id='EP001', cargo='director', director=null}}], listaInvitados=[]}

[Empleado{nombre='Director1', id='EP001', cargo='director', director=null}, Empleado{nombre='Pablo Motos', id='EP002', cargo='presentador', director=Empleado{nombre='Director1', id='EP001', cargo='director', director=null}}]

Introduce el año en el que acudirá el invitado Aitana:
2025
Introduce el mes:
12
Introduce el día:
12

[Invitado{nombre='Aitana', profesion='cantante', fecha_visita=2025-12-12, temporada=1}]
```

Para el siguiente *main* de ejemplo:

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

    //Creamos una cadena de tv
    Cadena antena3 = new Cadena("Antena 3");
    System.out.println(antena3);

    //Creamos un programa
    Programa el_hormiguero = new Programa("El Hormiguero",antena3,"Director1");
    System.out.println(el_hormiguero);
    System.out.println(antena3);

    //Insertamos empleados en el programa
    el_hormiguero.insertarEmpleado("Pablo Motos","presentador",null);
    System.out.println(el_hormiguero);

    //Ver empleados del programa
    System.out.println(el_hormiguero.getListaEmpleados());

    //Insertamos invitados en el programa
    el_hormiguero.insertarInvitado("Aitana","cantante",1);

    //Ver invitados del programa
    System.out.println(el_hormiguero.getListalInvitados());

}
```

## ➔ REALIZACIÓN DE LA PRÁCTICA

Sigue los siguientes pasos para realizar la práctica. ¡Ve probando tu trabajo de vez en cuando para evitar que nos volvamos locos si hubiera algún error!

1. Programa en Java la aplicación requerida
2. Plan de pruebas. Realiza las pruebas necesarias para comprobar que el programa funciona bien
3. Diagrama UML

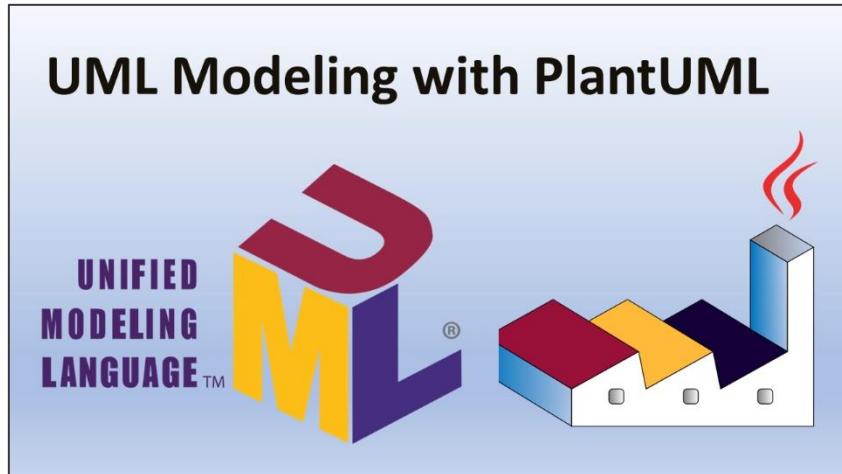


### ENTREGA

**REALIZA UN INFORME EN PDF CON LA INFO GENERADA Y LOS PASOS SEGUIDOS PARA REALIZAR ESTA PRÁCTICA. EXPLICA TU CÓDIGO.**

**SÚBELO TODO A LA TAREA DE AULES DISPONIBLE. ADEMÁS, PEGA LA URL DE TU PROYECTO EN GITHUB.**

## PRÁCTICA 2. MODELADO DE CLASES CON *PLANTUML*



<https://plantuml.com/es/>

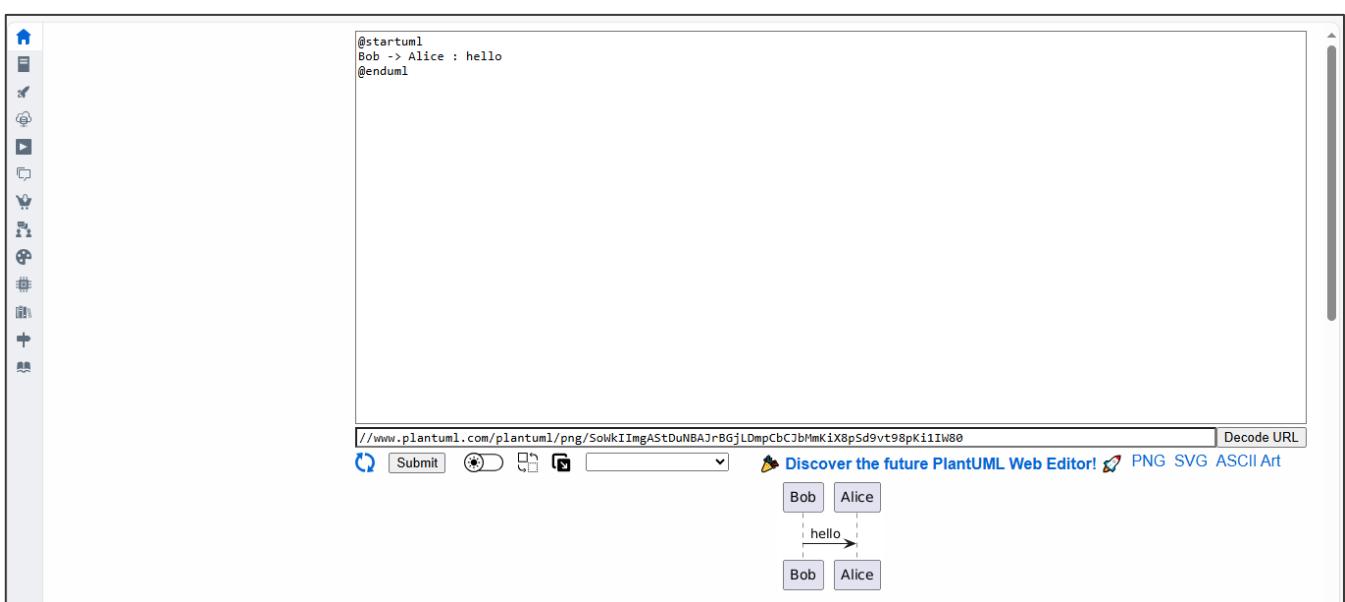
### → INTRODUCCIÓN: PlantUML de un vistazo

**PlantUML** es una herramienta gratuita muy versátil que facilita la creación rápida de una amplia gama de tipos de diagramas. Se encuentra en código abierto en el siguiente repositorio de *GitHub*: [plantuml/plantuml: Generate diagrams from textual description](https://github.com/plantuml/plantuml)

Esta herramienta utiliza un lenguaje sencillo que los usuarios pueden redactar sin demasiado esfuerzo. Para ver info detallada sobre las capacidades de este lenguaje y su sintaxis, se puede consultar la [Guía de Referencia del Lenguaje PlantUML](#), pero tranquilo, que no lo escribiremos nosotros.

### → INSTALACIÓN

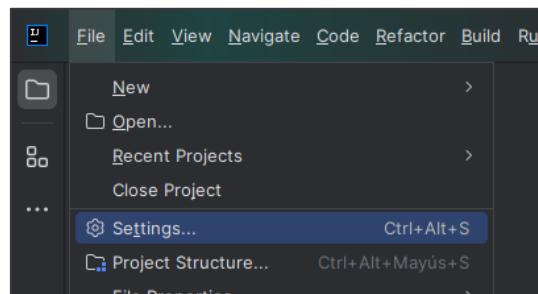
Hay varias formas de consumir esta plataforma, ya que permite la instalación local, el uso de su [servidor online](#) y la instalación de plugins en los IDE más importantes.



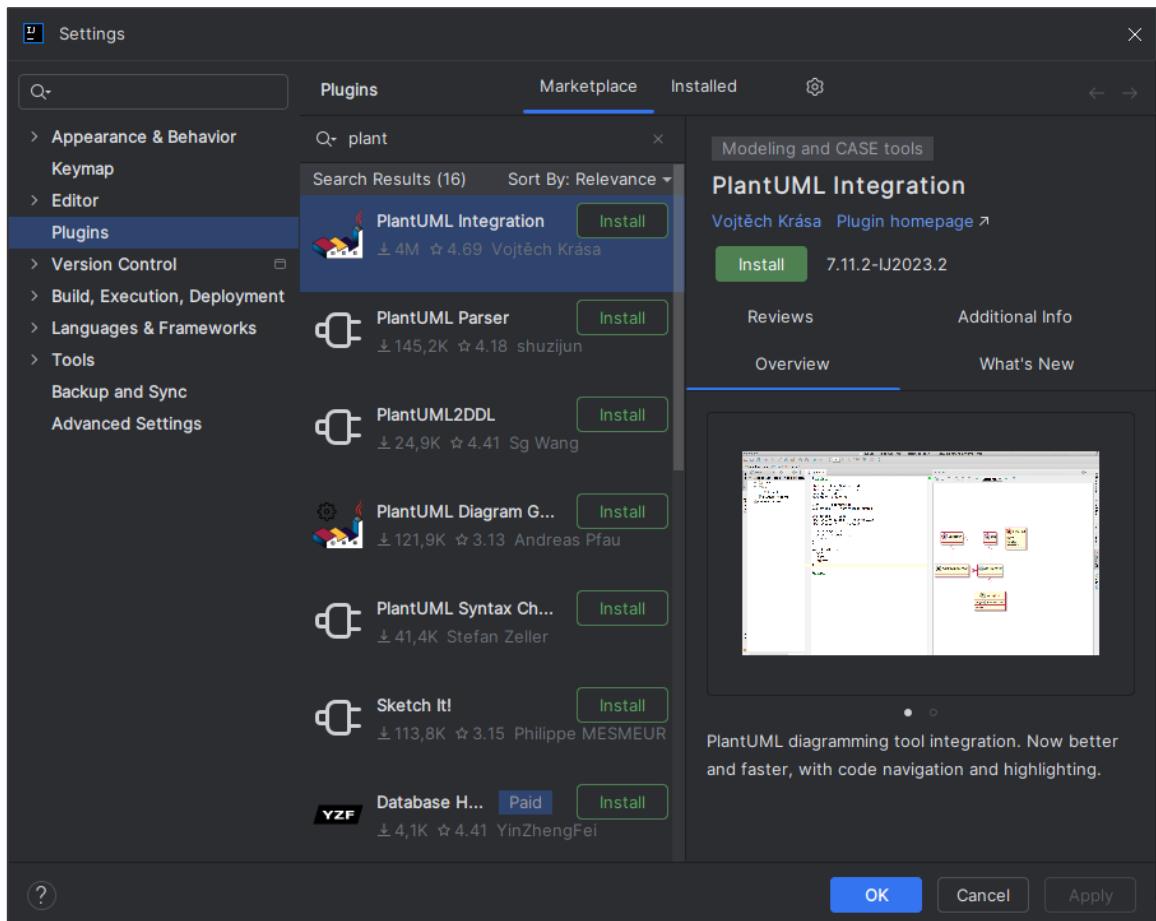
En nuestro caso, dado que las clases a modelar estarán en un paquete de proyecto *Maven* de *IntelliJ IDEA*, sería conveniente instalar el plugin correspondiente.

## 1. Instalación del plugin *PlantUML Integration*

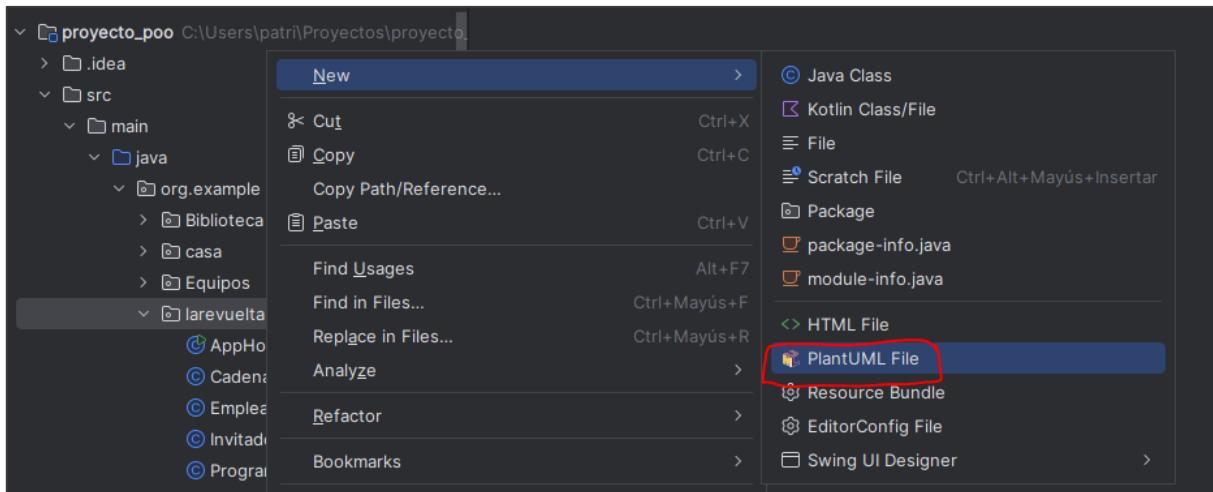
- Ve a tu *IntelliJ IDEA* y accede a **File → Settings** del menú superior:



- Busca la pestaña **Plugins** y escribe en el buscador el que necesitamos:



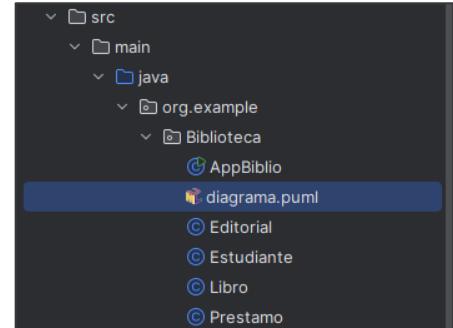
- Pulsa el botón **Install** y reinicia el IDE.
- Una vez hecho, si nos posicionamos sobre cualquier elemento del proyecto que tengamos abierto y hacemos *click* derecho para crear un nuevo componente, nos debe aparecer **PlantUML File** como opción:



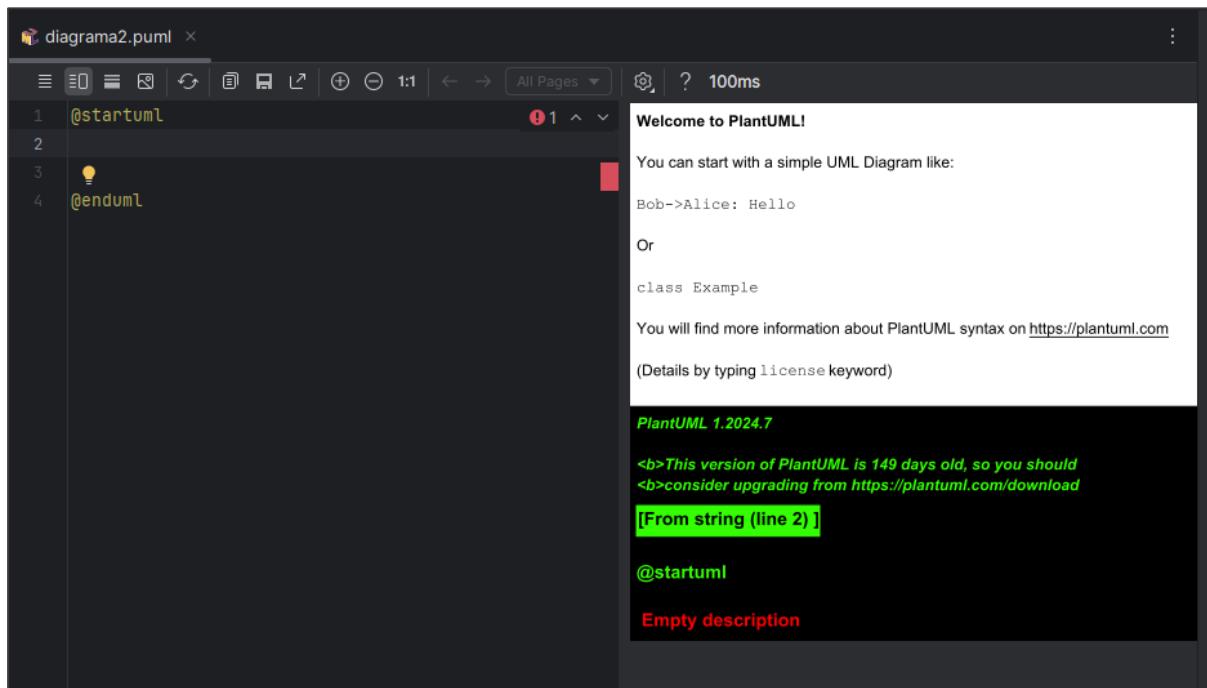
## 2. Creación de archivos .puml y generación de diagramas de clases

Siguiendo por donde nos hemos quedado en el apartado anterior, dirígete a alguno de los paquetes donde tengas los ejercicios sobre POO realizados durante las últimas semanas. Para este ejemplo, cogeremos el de la Biblioteca, que posee relaciones de varios tipos (asociación/agregación y composición).

- Creamos un archivo **.puml** en el paquete que contiene las clases relacionadas con nuestro proyecto de la Biblioteca:



- Nos aparecerá en la pantalla una vista previa del plugin. A la izquierda, el editor para escribir el código *PlantUML* entre las etiquetas `@startuml` y `@enduml` y a la derecha, una vista previa de lo que vamos añadiendo:

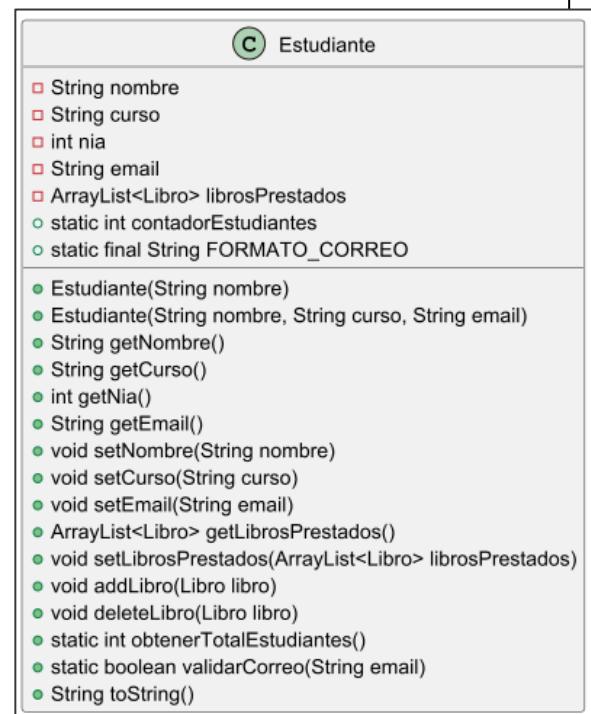


- Vamos a empezar a crear código *PlantUML*. Por ejemplo, para nuestra clase *Estudiante*, se generaría:

```
@startuml
```

```
class Estudiante {
    - String nombre
    - String curso
    - int nia
    - String email
    - ArrayList<Libro> librosPrestados
    + static int contadorEstudiantes
    + static final String FORMATO_CORREO
    + Estudiante(String nombre)
    + Estudiante(String nombre, String curso, String email)
    + String getNombre()
    + String getCurso()
    + int getNia()
    + String getEmail()
    + void setNombre(String nombre)
    + void setCurso(String curso)
    + void setEmail(String email)
    + ArrayList<Libro> getLibrosPrestados()
    + void setLibrosPrestados(ArrayList<Libro> librosPrestados)
    + void addLibro(Libro libro)
    + void deleteLibro(Libro libro)
    + static int obtenerTotalEstudiantes()
    + static boolean validarCorreo(String email)
    + String toString()
}
```

```
@enduml
```



The screenshot shows a code editor with two panes. The left pane contains the PlantUML code for the 'Estudiante' class, identical to the one above. The right pane shows the generated UML class diagram, which is identical to the one shown above.

```
1 @startuml
2
3 class Estudiante {
4     - String nombre
5     - String curso
6     - int nia
7     - String email
8     - ArrayList<Libro> librosPrestados
9     + static int contadorEstudiantes
10    + static final String FORMATO_CORREO
11    + Estudiante(String nombre)
12    + Estudiante(String nombre, String curso, String email)
13    + String getNombre()
14    + String getCurso()
15    + int getNia()
16    + String getEmail()
17    + void setNombre(String nombre)
18    + void setCurso(String curso)
19    + void setEmail(String email)
20    + ArrayList<Libro> getLibrosPrestados()
21    + void setLibrosPrestados(ArrayList<Libro> librosPrestados)
22    + void addLibro(Libro libro)
23    + void deleteLibro(Libro libro)
24    + static int obtenerTotalEstudiantes()
```

¿Te has asustado? Normal... Porque esto nos lo va a hacer *ChatGPT* 🤖

### 3. Uso de IA para generar código *PlantUML*

Como ya hemos visto, la facilidad de usar esta herramienta no sería tal si tuviéramos que escribir el código de nuestras clases todo el rato... Pero vamos a hacerlo más sencillo.

- Abre *ChatGPT* (o *Copilot*) y escribe un *prompt* para que nos genere automáticamente el código *PlantUML* de nuestras clases, así como sus relaciones. Le pediremos algo así:

hola copi, conoces plantuml?

¡Hola! Sí, conozco PlantUML. Es una herramienta de código abierto que permite crear diagramas UML (Lenguaje Unificado de Modelado) a partir de una descripción textual. Esto significa que puedes escribir un simple archivo de texto con la especificación de tus diagramas, y PlantUML se encargará de generar el diagrama gráfico correspondiente.

PlantUML es bastante versátil y admite varios tipos de diagramas, como diagramas de clases, secuencia, de actividades, de casos de uso, de componentes y más. Es muy útil para desarrolladores de software y arquitectos que necesitan visualizar sus diseños de una manera clara y precisa.

¿Estás interesado en aprender a usar PlantUML o ya lo has estado utilizando?

puedes generar el código de un diagrama de relación entre varias clases que te voy a dar?

¡Por supuesto! Dame los detalles de las clases y sus relaciones, y te generaré el código para el diagrama UML en PlantUML. Puedes incluir los nombres de las clases, sus atributos, métodos y cómo se relacionan entre sí (por ejemplo, herencia, asociación, composición, etc.). ¡Estoy listo para ayudarte!

```
package org.example.Biblioteca;

import java.util.ArrayList;

/**
 * Representa a un estudiante que tiene libros prestados.
 * Este estudiante puede pedir prestados libros de la biblioteca.
 */
public class Estudiante {

    public static int contadorEstudiantes = 0;
    public static final String FORMATO_CORREO = "[A-Za-z0-
9+_.]+@alu.edu.gva.es$";

    private String nombre;
```

...

```

        getPais() + " librosPublicados=" + getLibrosPublicados() + " ]";
    }

}

```

Aquí tienes el código de PlantUML para generar el diagrama de relaciones entre las clases `Estudiante`, `Libro`, `Prestamo` y `Editorial`:

```

Plantuml

@startuml

package org.example.Biblioteca {

    class Estudiante {
        - String nombre
        - String curso
        - int nia
    }
}

```

**En general, el resultado del modelado de las clases será correcto. Pero fíjate...**

**Para crear las relaciones**, al final del código, escribe unas líneas tal que así:

```

Estudiante "1" o-- "0..*" Libro : presta >

Estudiante "1" -- "0..*" Prestamo : realiza >

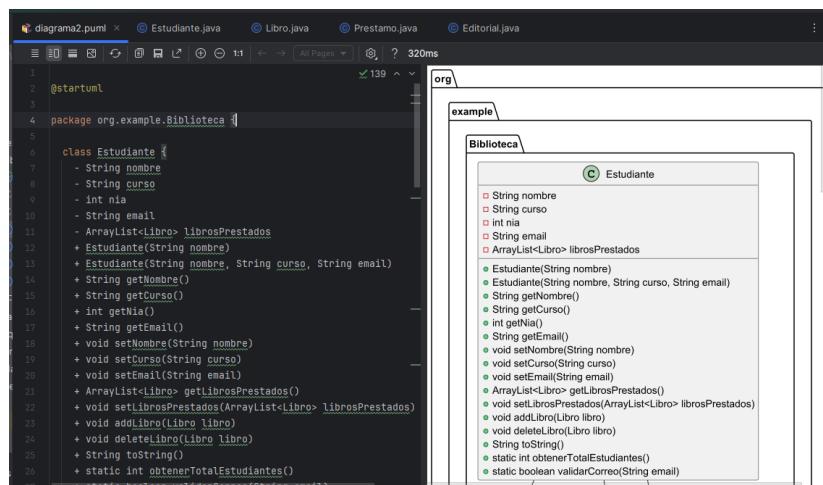
Libro "1" -- "1" Editorial : pertenece a >

Prestamo "1" -- "1" Libro : incluye >

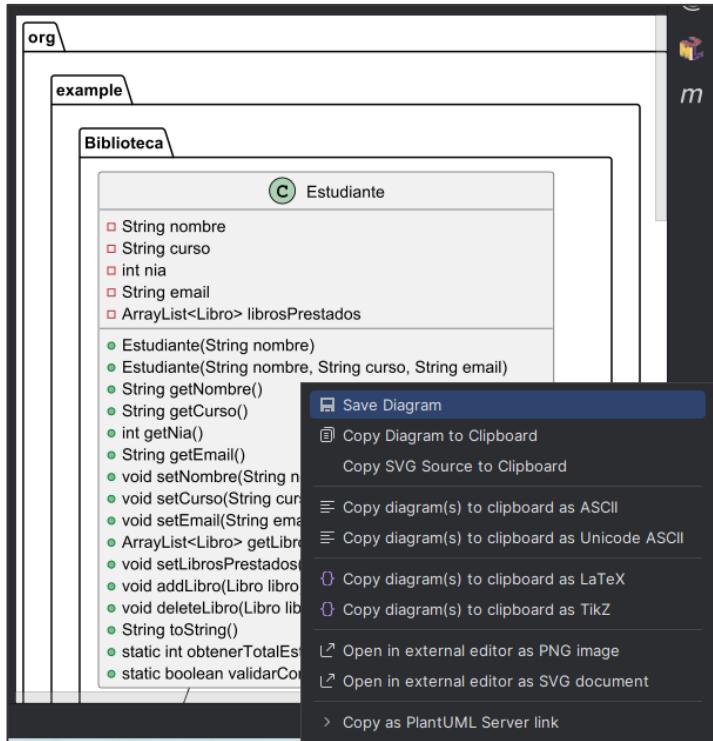
Prestamo "1" -- "1" Estudiante : realizado por >

```

**Esta parte siempre deberemos revisarla**, ya que tiene por costumbre confundir cuando una relación es de asignación/agregación (utiliza el símbolo `o`) y de composición (utiliza el símbolo `*`). **Si detectamos algún error, deberemos guiarlo hasta que devuelva el resultado que queremos.**



- Para guardar o exportar el diagrama tenemos varias opciones. Haz *click* derecho sobre el diagrama para verlas:



De todas las que hay, normalmente usaremos **Save Diagram** para guardarla en PNG o simplemente la copiaremos para pegarla en algún informe de prácticas.

Otro uso curioso es que *PlantUML* nos permite subirlo a su servidor para poder tenerlo accesible fuera de *IntelliJ IDEA* mediante una URL. Para ello, usaremos la opción “**Copy as PlantUML Server link**” y veremos que nos abre la plataforma online vista anteriormente, con nuestro diagrama precargado.

[PlantUML Web Server](#)

## → REALIZACIÓN DE LA PRÁCTICA

Sigue los siguientes pasos para realizar la práctica. A partir de ahora, haz capturas de todo y ve pegándolas en un documento de texto. ¡Guarda el documento de vez en cuando para evitar que se borre el avance de tu trabajo si se cierra el editor de textos u ocurre cualquier problema en tu equipo!

1. Realiza el mismo proceso visto anteriormente para generar el diagrama del ejercicio **Biblioteca** (*Estudiante – Libro – Prestamo – Editorial*). Guárdalo en formato **PNG** en la misma localización que tengas las clases.
2. Genera el diagrama del ejercicio **Casa – Habitacion – Electrodoméstico**. Guárdalo en formato **PNG** en la misma localización que tengas las clases.
3. Actualiza el repositorio remoto para que se reflejen las modificaciones realizadas y guardemos nuestros archivos **.puml** y diagramas también en **GitHub**. Pega aquí la URL:



## ENTREGA

\*\*CONVIERTE EL DOCUMENTO DE TEXTO GENERADO SIGUIENDO TODOS LOS PASOS A PDF Y SÚBELO A TRAVÉS DE LA ENTREGA DISPONIBLE EN AULES.

## PRÁCTICA 3. LIBRERÍAS DE AYUDA PARA LA IMPLEMENTACIÓN DE LA POO:

### LOMBOK



[Project Lombok](#)

#### → Introducción a Lombok en Java

Lombok es una librería de Java que ayuda a reducir el código repetitivo en las clases, proporcionando **anotaciones (@)** para generar código automáticamente como constructores, *getters*, *setters*, método *toString()* y otros. Su uso mejora la legibilidad y mantenimiento del código, lo que lo convierte en una opción popular en muchos proyectos.

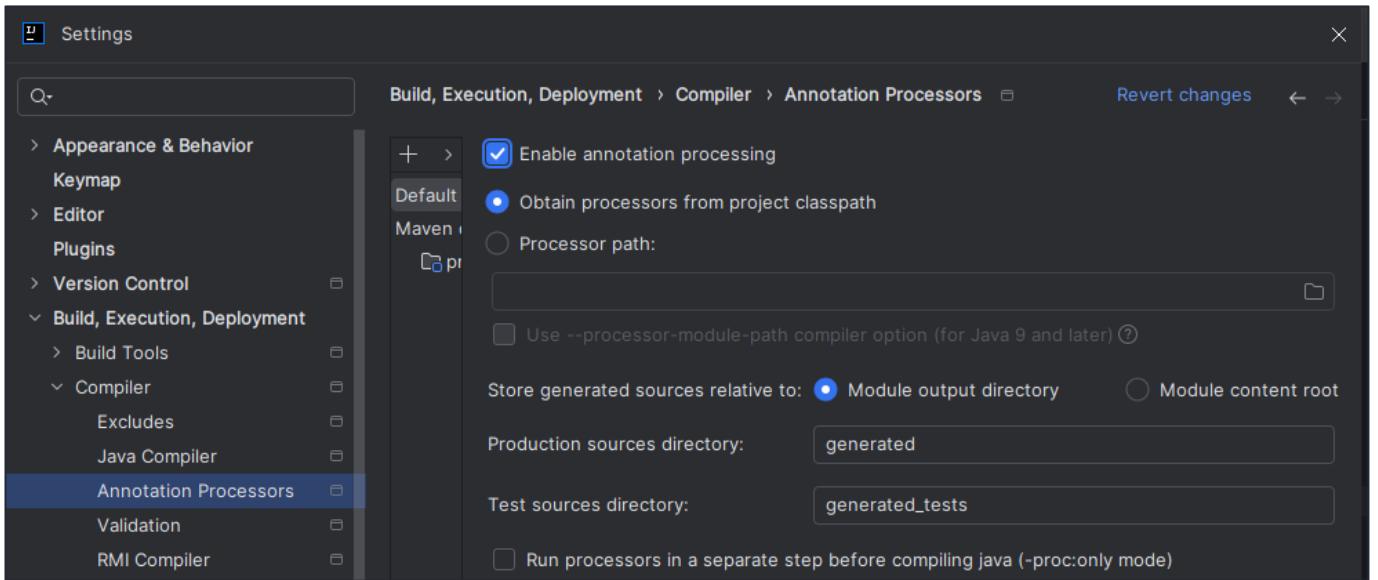
#### → INSTALACIÓN

Para utilizar [Lombok en un proyecto Maven](#), es necesario agregar la siguiente dependencia en el archivo *pom.xml*:

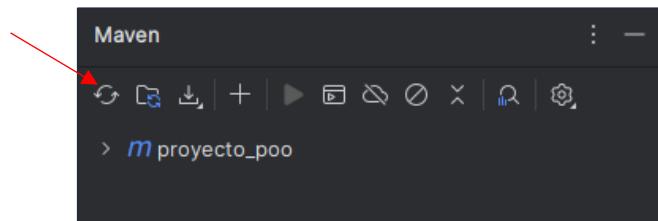
```
<dependencies>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.36</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

A screenshot of the IntelliJ IDEA IDE. On the left, the project structure shows a package named "org.example" containing sub-packages "Biblioteca", "casa", "Equipos", and "larevuelta", along with classes "Instituto" and "Main". Below these are "resources", "test", "target", and ".gitignore". A red arrow points from the ".gitignore" file to the "pom.xml" file, which is currently selected and has a blue background. On the right, the code editor displays the XML content of the "pom.xml" file. A red box highlights the section of the XML where the Lombok dependency is defined, starting from line 14. The dependency tag contains the group ID "org.projectlombok", artifact ID "lombok", version "1.18.36", and scope "provided". The rest of the XML file includes the POM header, project information, and properties like "maven.compiler.source" set to "23".

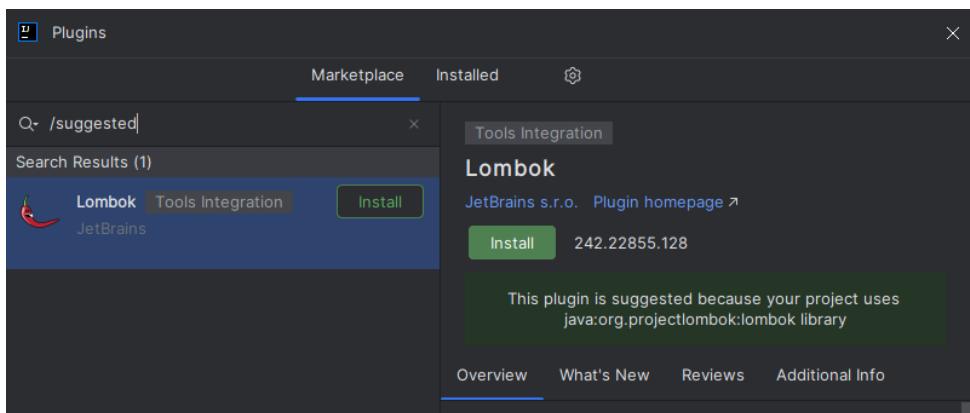
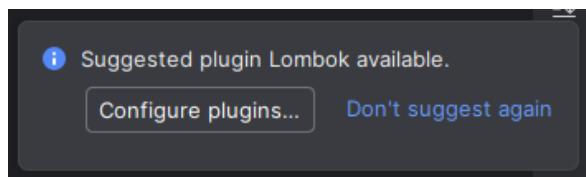
Vamos a activar el procesamiento de anotaciones. Lo habilitaremos en **Settings → Build, Execution, Deployment → Compiler → Annotation Processors**



Fuerza la actualización de *Maven*: Ve a **View → Tool Windows → Maven** y haz clic en el botón de **Reload All Maven Projects** (ícono de actualización).



Una vez hecho, nos recomendará instalar el Plugin de Lombok:



Dale a **Install** y reinicia *IntelliJ IDEA*.

## → ANOTACIONES PRINCIPALES DE LOMBOK

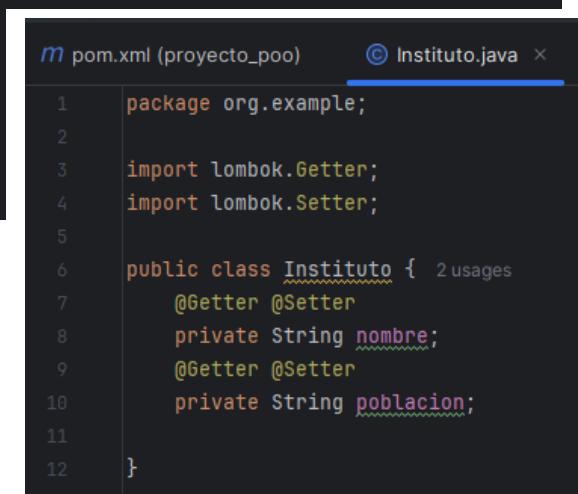
Una vez realizada toda la configuración, vamos a probar que *Lombok* funciona como se espera.

## 1. @Getter y @Setter

Estas anotaciones “generan” los métodos *getters* y *setters* para los atributos de una clase:

```
import lombok.Getter;
import lombok.Setter;

public class Instituto {
    @Getter @Setter
    private String nombre;
    @Getter @Setter
    private String poblacion;
}
```

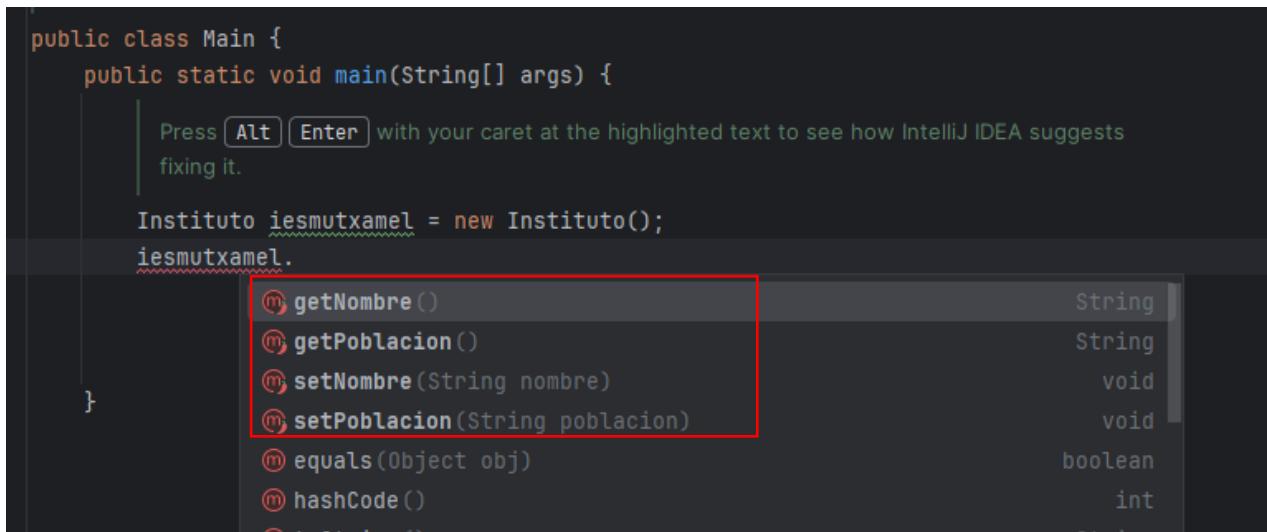


The screenshot shows two tabs: 'pom.xml (proyecto\_poo)' and 'Instituto.java'. The 'Instituto.java' tab is active, displaying the generated code. The code includes imports for lombok.Getter and lombok.Setter, and a class definition for Instituto with two private attributes: nombre and poblacion, each annotated with @Getter and @Setter.

```
m pom.xml (proyecto_poo) © Instituto.java ×
1 package org.example;
2
3 import lombok.Getter;
4 import lombok.Setter;
5
6 public class Instituto { 2 usages
7     @Getter @Setter
8     private String nombre;
9     @Getter @Setter
10    private String poblacion;
11
12 }
```

Y decimos “generan” entre comillas, porque **en realidad no se van a ver en nuestra clase**. Son “invisibles”.

Vamos a probarlo generando un objeto de tipo *Instituto* desde el *main* y comprobando que se puede llamar a los métodos *set* y *get* de cada atributo de la clase (sin que a priori “existan”):

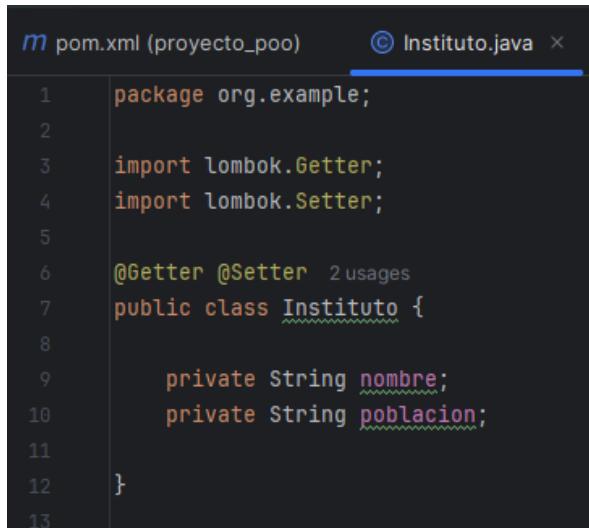


The screenshot shows a code editor with a Main class. In the main method, there is a line of code: 'Instituto iestmutxamel = new Instituto();'. Below this, the cursor is at 'iestmutxamel.' and a code completion dropdown is open. The dropdown shows several methods: getNombre(), getPoblacion(), setNombre(String nombre), setPoblacion(String poblacion), equals(Object obj), hashCode(), and toString(). The 'setNombre()' and 'setPoblacion()' methods are highlighted with a red box. To the right of the methods, their return types and parameters are listed: String, String, void, void, boolean, int, and String respectively.

```
public class Main {
    public static void main(String[] args) {
        Instituto iestmutxamel = new Instituto();
        iestmutxamel.
    }
}
```

Method	Type
getNombre()	String
getPoblacion()	String
setNombre(String nombre)	void
setPoblacion(String poblacion)	void
equals(Object obj)	boolean
hashCode()	int
toString()	String

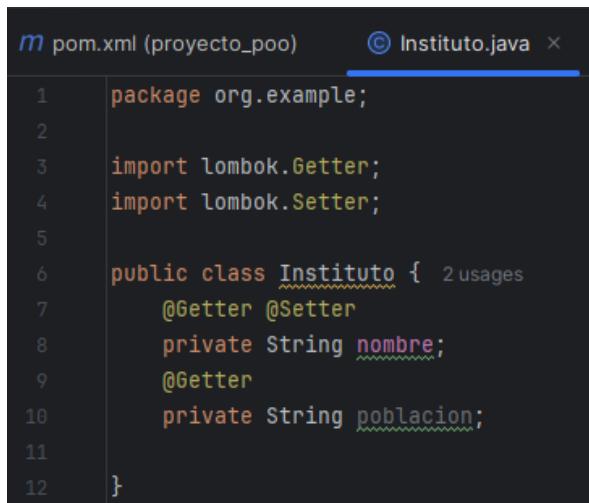
Observa que hemos puesto las etiquetas **@Getter** y **@Setter** encima de cada atributo, pero podríamos habérnoslo ahorrado simplemente especificándolo encima de la clase:



```
m pom.xml (proyecto_poo) © Instituto.java ×
1 package org.example;
2
3 import lombok.Getter;
4 import lombok.Setter;
5
6 @Getter @Setter 2 usages
7 public class Instituto {
8
9     private String nombre;
10    private String poblacion;
11
12 }
13
```

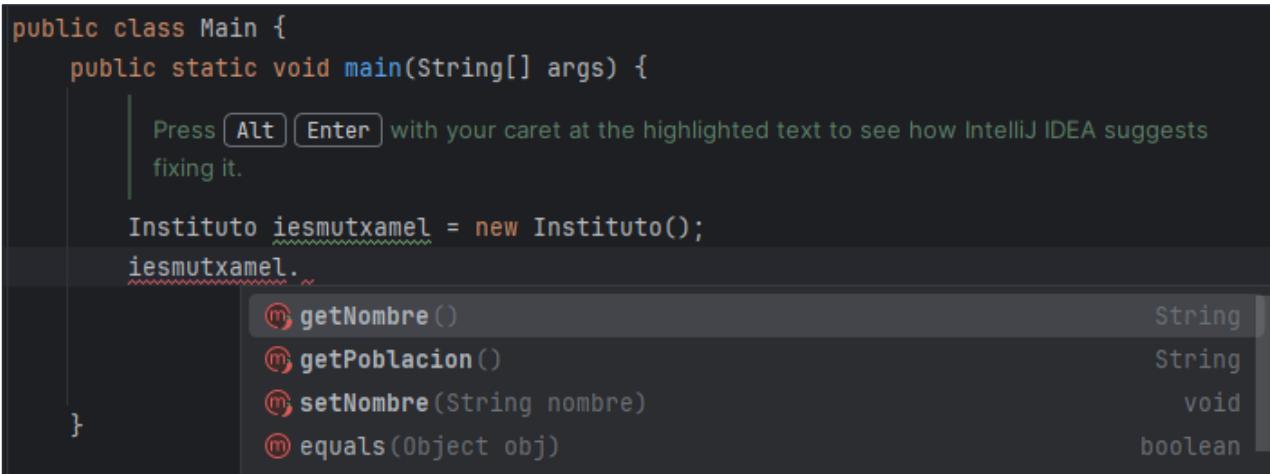
Esto significa que queremos los **getter** y los **setter** de todos y cada uno de los atributos de la clase.

Por esas mismas, si por algún casual no quisiéramos que algún atributo tuviera **setter**, con no indicárselo sería suficiente para que no se autogenera. Por ejemplo, si no queremos permitir modificar la población:



```
m pom.xml (proyecto_poo) © Instituto.java ×
1 package org.example;
2
3 import lombok.Getter;
4 import lombok.Setter;
5
6 public class Instituto { 2 usages
7     @Getter @Setter
8     private String nombre;
9     @Getter
10    private String poblacion;
11
12 }
```

Y desde el *main* comprobaremos como la opción de *settear* la población ya no aparece:



```
public class Main {
    public static void main(String[] args) {
        Instituto iesmutxamel = new Instituto();
        iesmutxamel.
    }
}
```

Press **Alt** **Enter** with your caret at the highlighted text to see how IntelliJ IDEA suggests fixing it.

getNombre()	String
getPoblacion()	String
setNombre(String nombre)	void
equals(Object obj)	boolean

## 2. @ToString

Genera automáticamente el método `toString()`:

```
import lombok.ToString;

@ToString
public class Instituto {
    @Getter @Setter
    private String nombre;
    @Getter
    private String poblacion;
}
```

```
m pom.xml ( proyecto_poo )  ☐ Instituto.java ×
1 package org.example;
2
3 import lombok.Getter;
4 import lombok.Setter;
5 import lombok.ToString;
6
7 @ToString 2 usages
8 public class Instituto {
9     @Getter @Setter
10    private String nombre;
11    @Getter
12    private String poblacion;
13
14 }
15
```

Desde el `main`:

```
public class Main {
    public static void main(String[] args) {
        Instituto iesmutxamel = new Instituto();
        iesmutxamel.
        ↓
        ⚡ getNombre()           String
        ⚡ getPoblacion()        String
        ⚡ setNombre(String nombre) void
        ⚡ toString()             String
        ⚡ equals(Object obj)    boolean
    }
}
```

## 3. @NoArgsConstructor y @AllArgsConstructor

Estas anotaciones generan constructores los constructores típicos (por defecto y parametrizado):

```
import lombok.NoArgsConstructor;
import lombok.AllArgsConstructor;

@NoArgsConstructor //Constructor sin parámetros
@AllArgsConstructor //Constructor con todos los atributos
@ToString
@Getter @Setter
public class Instituto {
    private String nombre;
    private String poblacion;
    private int codigo;
}
```

```
m pom.xml ( proyecto_poo )  ☐ Instituto.java ×
1 package org.example;
2
3 import lombok.*;
4
5 @NoArgsConstructor 2 usages
6 @AllArgsConstructor
7 @ToString
8 @Getter @Setter
9 public class Instituto {
10
11    private String nombre;
12    private String poblacion;
13    private int codigo;
14
15 }
```

Es decir, desde el `main` ahora podremos crear un objeto de tipo

`Instituto` de dos formas: con un constructor por defecto y con un constructor parametrizado con todos los atributos de la clase:

```

public class Main {
    public static void main(String[] args) {
        Press Alt Enter with your caret at the highlighted text to see how IntelliJ IDEA suggests fixing it.

        Instituto iesmutxamel = new Instituto(nombre: "IES MUTXAMEL", poblacion: "MUTXAMEL", codigo: 1);
        Instituto iessanvicente = new Instituto();

        iessanvicente.|_
            ⚭ getCodigo() int
            ⚭ getPoblacion() String
            ⚭ getNombre() String
            ⚭ setCodigo(int codigo) void
            ⚭ setNombre(String nombre) void
            ⚭ setPoblacion(String poblacion) void
            ⚭ toString() String
            ⚭ equals(Object obj) boolean
    }
}

```

## ➔ CONTROL DE PARÁMETROS NO NULOS CON LOMBOK

En Java, cuando trabajamos con Programación Orientada a Objetos (*POO*), a veces es importante asegurarnos de que los parámetros que recibe un método o constructor no sean nulos. Hay varias formas de hacer esto:

### 1. Uso de *Objects.requireNonNull*

Como ya sabemos, Java proporciona una clase “madre” llamada *Objects*. Esta tiene el método *requireNonNull()*, el cual lanza una excepción de tipo *NullPointerException* si el argumento es *null*.

Por ejemplo, si en nuestra clase *Instituto* tuviéramos un *setNombre(String nombre)* como el que sigue:

```

@AllArgsConstructor
@RequiredArgsConstructor
@ToString
@Getter @Setter
public class Instituto {

    private String nombre;
    private String poblacion;
    private int codigo;

    public void setNombre(String nombre) {
        this.nombre = Objects.requireNonNull(nombre, "El nombre no puede ser nulo");
    }
}

```

e intentaremos modificar el nombre de un instituto por un valor nulo:

The screenshot shows the IntelliJ IDEA interface. In the top-left, the project structure shows a package named 'org.example' containing a 'main' directory with a 'java' folder. Inside 'java' are several files: 'Biblioteca.java', 'casa.java', 'Equipos.java', 'larevuelta.java', 'Instituto.java', and 'Main.java'. Below these are 'resources' and 'src' folders. The code editor on the right displays the 'Main.java' file:

```
public class Main {
    public static void main(String[] args) {
        Instituto iesmutxamel = new Instituto(nombre: "IES MUTXAMEL", poblacion: "MUTXAMEL", codigo: 1);
        Instituto iessanvicente = new Instituto();
        iesmutxamel.setNombre(null);
    }
}
```

A tooltip in the code editor says: "Press Alt + Enter with your caret at the highlighted text to see how IntelliJ IDEA suggests fixing it." The terminal window below shows the output of running the program:

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\lib\idea_rt.jar=61501:C:\Program
Exception in thread "main" java.lang.NullPointerException Create breakpoint : El nombre no puede ser nulo
at java.base/java.util.Objects.requireNonNull(Objects.java:246)
at org.example.Instituto.setNombre(Instituto.java:19)
at org.example.Main.main(Main.java:14)

Process finished with exit code 1
```

Pruébalo tú...

## 2. Uso de validaciones manuales

Podríamos usar también una **condición if** para lanzar un mensaje por pantalla y no dejar realizar la modificación:

```
@AllArgsConstructor
@RequiredArgsConstructor
@ToString
@Getter @Setter
public class Instituto {

    private String nombre;
    private String poblacion;
    private int codigo;

    public void setNombre(String nombre) {
        if (nombre == null) {
            System.out.println("El nombre no puede ser nulo");
        } else{
            this.nombre = nombre;
        }
    }
}
```

```

    public class Main {
        public static void main(String[] args) {
            Instituto iesmutxamel = new Instituto( nombre: "IES MUTXAMEL", poblacion: "MUTXAMEL", codigo: 1);
            Instituto iessanvicente = new Instituto();
            iesmutxamel.setNombre(null);
        }
    }

```

The Run tab shows the command: "C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\lib\idea\_rt.jar=61546:C:\Program...". The output window shows: "El nombre no puede ser nulo" (The name cannot be null) and "Process finished with exit code 0".

**Pruébalo tú...**

### 3. Anotaciones `@NonNull` en Lombok

Pero la forma que más nos interesa, en este caso, es la que nos proporciona *Lombok* con la marca `@NonNull`:

```

@RequiredArgsConstructor
@ToString
@Getter @Setter
public class Instituto {

    @NonNull
    private String nombre;
    private String poblacion;
    private int codigo;

    public void setNombre(@NonNull String nombre) {
        this.nombre = nombre;
    }

}

```

Si volvemos al *main* e intentamos lanzar lo mismo, obtendremos otro *NullPointerException*:

```

    public class Main {
        public static void main(String[] args) {
            Instituto iesmutxamel = new Instituto( nombre: "IES MUTXAMEL", poblacion: "MUTXAMEL", codigo: 1);
            // Instituto iessanvicente = new Instituto();
            iesmutxamel.setNombre(null);
            //System.out.println(iesmutxamel);
        }
    }

```

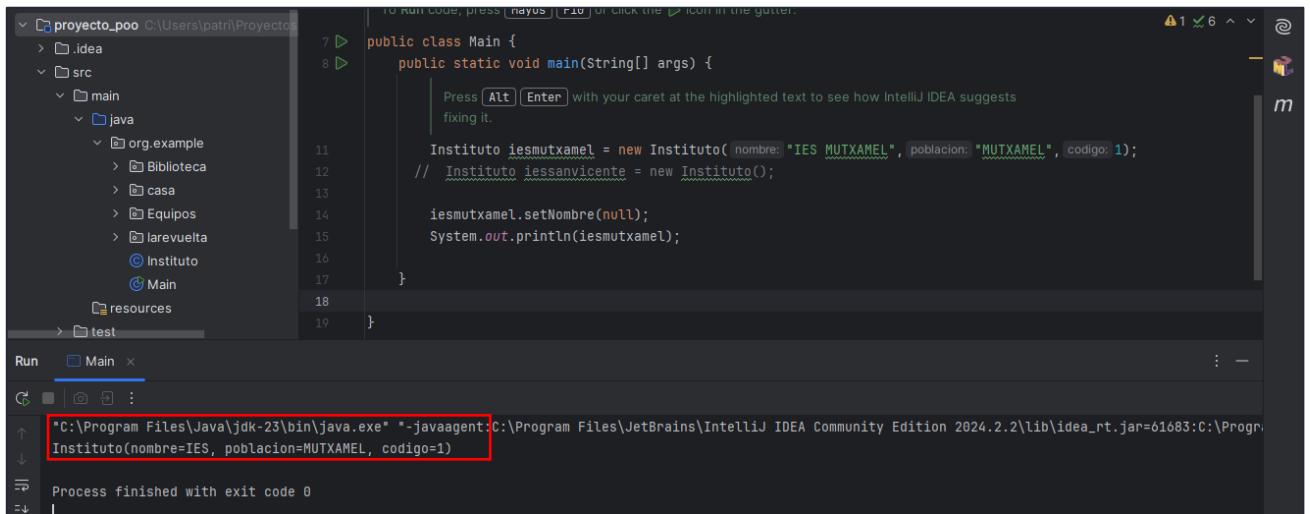
The Run tab shows the command: "C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\lib\idea\_rt.jar=61617:C:\Program...". The output window shows: "Exception in thread "main" java.lang.NullPointerException Create breakpoint : nombre is marked non-null but is null" and "at org.example.Instituto.setNombre(Instituto.java:19)".

Como normalmente no vamos a querer que nuestro programa aborte de forma abrupta, “arreglaremos” esto asignando un *nombre* por defecto, por ejemplo “IES”. La mejor manera de hacerlo es combinar `@NonNull` con `Objects.requireNonNullElse()`, que asignará el nuevo valor:

```
@AllArgsConstructor
@RequiredArgsConstructor
@ToString
@Getter @Setter
public class Instituto {

    @NonNull
    private String nombre;
    private String poblacion;
    private int codigo;

    public void setNombre(String nombre) {
        this.nombre = Objects.requireNonNullElse(nombre, "IES");
    }
}
```



Si no queremos asignar un nombre por defecto y simplemente queremos controlar el error, podemos agregar un bloque *try-catch* en el *main* para controlar excepciones que puedan producirse dentro del bloque de código susceptible de fallar (en nuestro caso el método *setNombre()* ):

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

        Instituto iesmutxamel = new Instituto("IES MUTXAMEL", "MUTXAMEL", 1);

        try {
            iesmutxamel.setNombre(null); //lanza NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Error: El nombre no puede ser nulo.");
        }

        System.out.println(iesmutxamel);
    }
}
```

```

    public class Main {
        public static void main(String[] args) {
            Instituto iesmutxamel = new Instituto( nombre: "IES MUTXAMEL", poblacion: "MUTXAMEL", codigo: 1);

            try {
                iesmutxamel.setNombre(null); //Lanza NullPointerException
            } catch (NullPointerException e) {
                System.out.println("Error: El nombre no puede ser nulo.");
            }

            System.out.println(iesmutxamel);
        }
    }

```

"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\lib\idea\_rt.jar=61900:C:\Program  
Error: El nombre no puede ser nulo.  
Instituto(nombre=IES MUTXAMEL, poblacion=MUTXAMEL, codigo=1)

Process finished with exit code 0

## → OTRAS ANOTACIONES INTERESANTES

### 1. *@RequiredArgsConstructor*

La anotación *@RequiredArgsConstructor* de Lombok genera un constructor que incluye únicamente los campos constantes (*final*) y aquellos marcados como *@NonNull*. Esto significa que los atributos que sean de tipo *final* o estén marcados como *@NonNull* serán obligatorios en el constructor generado.

Un ejemplo de uso:

```

@NoArgsConstructor
@AllArgsConstructor
@RequiredArgsConstructor
@ToString
@Getter @Setter
public class Instituto {
    private final String nombre; //Se incluirá en el constructor
    @NonNull
    private String poblacion; //Se incluirá en el constructor
    private int codigo; //No se incluirá en el constructor
}

```

El constructor “generado” con *@RequiredArgsConstructor* será equivalente a este:

```

public Instituto(String nombre, String poblacion) {
    this.nombre = nombre;
    this.poblacion = poblacion;
}

```

**⚠️** El atributo **codigo** no está en el constructor porque no es tipo *final* ni tiene la marca *@NonNull*.

m pom.xml ( proyecto\_poo)     Instituto.java x     Main.java

```
1 package org.example;
2
3 import lombok.*;
4
5 @NoArgsConstructor 4 usages
6 @AllArgsConstructor
7 @RequiredArgsConstructor
8 @ToString
9 @Getter @Setter
10 public class Instituto {
11     ?
12     private final String nombre;
13     private Str Variable 'nombre' might not have been initialized
14     private int Add constructor parameter Alt+Mayús+Intro More actions... Alt+Intro
15
16 }
17
18 }
```

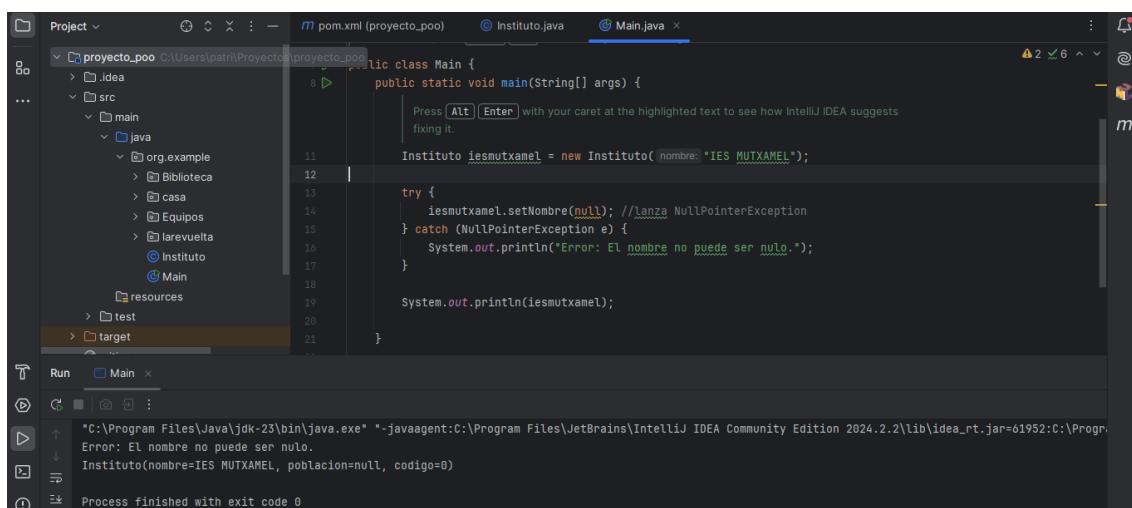
**⚠ Observa que al definir un atributo como *final* nos ha subrayado la marca del constructor por defecto (@*NoArgsConstructor*) como errónea, ya que una variable de este tipo siempre se debe inicializar y tener un valor (sí o sí).**

## 2. @Data

Como has podido ver en los apartados anteriores, la cantidad de etiquetas que hemos escrito hasta ahora ya es considerable... de momento 7 y subiendo.

Para facilitarnos todavía más la vida, *Lombok* incorpora una marca `@Data` que combina `@Getter`, `@Setter`, `@ToString`, `@EqualsAndHashCode` (no la usamos de momento) y `@RequiredArgsConstructor`.

```
@Data  
public class Instituto {  
  
    @NotNull  
    private String nombre;  
    private String poblacion;  
    private int codigo;  
  
}
```



## → REALIZACIÓN DE LA PRÁCTICA: Sistema de Gestión de un Instituto

Sigue los siguientes pasos para realizar la práctica. A partir de ahora, haz capturas de todo y ve pegándolas en un documento de texto. ¡Guarda el documento de vez en cuando para evitar que se borre el avance de tu trabajo si se cierra el editor de textos u ocurre cualquier problema en tu equipo!

### Implementar un sistema de gestión para un instituto donde hay estudiantes inscritos en cursos

1. Crea la clase **Estudiante** con los atributos:

→ *nombre* (obligatorio) (*String*)

→ *edad* (*int*)

→ *curso* (*Curso*).

2. Crea la clase **Curso** con los atributos:

→ *nombre* (obligatorio) (*String*)

→ *horas* (*int*)

3. Crea la clase **Instituto** con los atributos:

→ *nombre* (obligatorio) (*String*). Si el nombre es *null*, se debe asignar un nombre por defecto:

"*Instituto sin nombre*". Además, el nombre del instituto no debe poder modificarse después de la creación.

→ *población* (*String*)

→ Debe almacenar una *lista de estudiantes* y una *lista de cursos*. Se debe evitar la inserción de valores *null* en dichas listas. Tampoco se debe permitir agregar cursos duplicados (dos cursos con el mismo *nombre* y *horas*).

Condiciones:

- Usa las marcas de la librería *Lombok* para implementar todos los componentes de las clases que se piden de manera sencilla. **No uses @Data**.

Ejemplo de *main* para probar todas las clases:

```
public class AppCursos {  
  
    public static void main(String[] args) {  
        Instituto instituto = new Instituto("IES MUTXAMEL");  
  
        //crear cursos  
        Curso cursoJava = new Curso("Java", 100);  
        Curso cursoPython = new Curso("Python", 70);  
        instituto.agregarCurso(cursoJava);  
        instituto.agregarCurso(cursoPython);  
  
        //crear estudiantes
```

```

try {
    Estudiante estudiante1 = new Estudiante("Carlos", 20, cursoJava);
    Estudiante estudiante2 = new Estudiante("Ana", 22, cursoPython);
    instituto.agregarEstudiante(estudiante1);
    instituto.agregarEstudiante(estudiante2);
    //intento de crear un estudiante con nombre nulo (esto lanza
    NullPointerException)
    Estudiante estudianteErroneo = new Estudiante(null);

} catch (NullPointerException e) {
    System.out.println("Error: No se puede crear un estudiante con nombre
nulo.");
}

//intento de añadir un estudiante nulo en la lista de estudiantes
instituto.agregarEstudiante(null);
//intento de añadir un curso nulo en la lista de cursos
instituto.agregarCurso(null);

//mostrar cursos
System.out.println("Cursos disponibles:");
System.out.println(instituto.getListaCursos());

//mostrar estudiantes
System.out.println("Estudiantes registrados:");
System.out.println(instituto.getListaEstudiantes());
}
}

```

### Salida esperada:

```

"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\lib\idea_rt.jar=63448:C:
Error: No se puede crear un estudiante con nombre nulo.
No se puede agregar un estudiante nulo.
No se puede agregar un curso nulo.
Cursos disponibles:
[Curso(nombre=Java, horas=100), Curso(nombre=Python, horas=70)]
Estudiantes registrados:
[Estudiante(nombre=Carlos, edad=20, curso=Curso(nombre=Java, horas=100)), Estudiante(nombre=Ana, edad=22, curso=Curso(nombre=Python, horas=70))]

Process finished with exit code 0
|

```



### ENTREGA

**REALIZA UN INFORME EN PDF CON LA INFO GENERADA Y LOS PASOS SEGUIDOS PARA REALIZAR ESTA PRÁCTICA. EXPLICA TU CÓDIGO.**

**SÚBELO TODO A LA TAREA DE AULES DISPONIBLE.**

**ADEMÁS, PEGA LA URL DE TU PROYECTO EN GITHUB.**

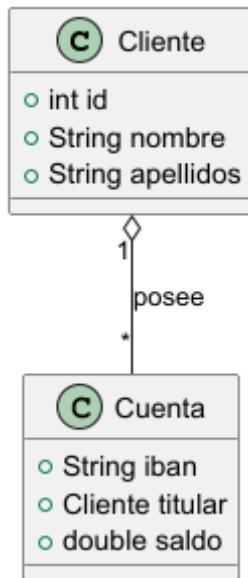
## EJERCICIOS PRE-EXAMEN PROGRAMACIÓN ORIENTADA A OBJETOS



### App Bancaria – BANC MUTXAMEL

Se desea realizar una aplicación para manejar los *Clientes* y las *Cuentas* de una sucursal bancaria de Mutxamel.

En la sucursal habrá:



donde:

- El atributo ***id*** de la clase *Cliente* será autogenerado conforme se vaya creando clientes.
- Para guardar los ***nombre*** y ***apellidos*** del *Cliente* se debe recoger el valor desde teclado, pidiéndoselo al usuario.
- El atributo ***iban*** de la clase *Cuenta* se asigna con el siguiente formato: “ESXX1”, “ESXX2”, etc., con un número autogenerado conforme se van creando cuentas.
- El ***saldo*** por defecto al crearse una *Cuenta* será 0€ y el ***titular*** nulo.
- Los *Clientes* y *Cuentas* que se vayan creando se almacenarán en un *ArrayList*. Ambos existen por sí mismos y no dependen del otro.

La app de **BancMutxamel** tendrá un menú principal:

**MENÚ PRINCIPAL:**

- [1] Mantenimiento de Clientes
- [2] Mantenimiento de Cuentas
- [X] Salir

1. Si el usuario selecciona la **opción 1 (mantenimiento de Clientes)**, se mostrará otro submenú con las siguientes opciones:

**CLIENTES:**

- [1] Altas
- [2] Bajas
- [3] Modificaciones
- [4] Listado de clientes
- [X] Volver al menú principal

- **Altas:** añadir clientes al *ArrayList*. El *nombre* y *apellidos* se pedirán por teclado.
- **Bajas:** eliminar clientes del *ArrayList* filtrando por nombre y apellidos. También se debe poner a *null* el *titular* todas sus cuentas asociadas.
- **Listado de clientes:** mostrar todos los clientes, con el siguiente formato,

**Listado de clientes (3):**

- 1. Carlos Sánchez Ruiz
- 2. Carla Rubio Moreno
- 3. Luis Aragón Núñez

(id). (nombre) (apellidos)

- **Modificaciones:** modificar alguno de los datos de los clientes (excepto *id* que no debemos permitirlo). Deberemos mostrar el listado de clientes con el formato anterior para que el usuario tenga a todos los clientes visibles y elija por *id* el que quiera modificar.

**Listado de clientes (3):**

- 1. Carlos Sánchez Ruiz
- 2. Carla Rubio Moreno
- 3. Luis Aragón Núñez

¿Qué cliente quieres modificar?

3

2. Si el usuario selecciona la **opción 2 (mantenimiento de Cuentas)**, mostraremos el siguiente submenú,

```

CUENTAS:
[1] Consultar cuentas cliente
[2] Ver todas las cuentas
[X] Volver al menú principal

```

- Ver todas las cuentas imprimirá el contenido del *ArrayList* que almacena todas las cuentas creadas en la sucursal (tengan titular o no).

```

Cuentas de la sucursal (3):
ESXX1 - saldo: 134€. Titular: Carla Rubio Moreno
ESXX2 - saldo: 23. Titular: Carla Rubio Moreno
ESXX3 - saldo: 0. Titular: sin titular
(isbn) - saldo: (saldo). Titular: titular.nombre + titular.apellidos

```

- Eligiendo Consultar cuentas cliente solicitaremos por pantalla que se introduzca el *nombre y apellidos* del cliente para buscar en el *ArrayList* de cuentas si existe alguna *Cuenta* asociada a ese *Cliente*.

- a) Si existe alguna cuenta con el titular asociado al cliente (o más de una), se mostrarán por pantalla para tenerlas identificadas. Además, mostraremos el submenú de acciones:

```

Cuentas asociadas a Carla Rubio Moreno:
ESXX1 - saldo: 134€
ESXX2 - saldo: 23€

MENÚ CUENTAS:
[1] Ingresar
[2] Pagar
[3] En rojos
[4] Consultar Saldo
[5] Ver datos del titular
[6] Crear nueva cuenta
[X] Volver al menú principal

```

- Para realizar cualquiera de las opciones del 1 al 4, siempre solicitaremos al usuario el *iban* de la cuenta con la que queremos operar.

- Ingresar: nos pedirá la cantidad a ingresar y la sumará al saldo total que tuviera la cuenta.
- Pagar: nos pedirá la cantidad a pagar y la restará al saldo total que tuviera la cuenta. Si la cantidad del pago es mayor al saldo disponible, lo haremos igualmente (nos quedaremos en números rojos).
- En rojos: comprobará y mostrará si la cuenta está con saldo negativo o no.
- Consultar saldo: imprimirá el saldo actual de la cuenta.
- Ver datos del titular: imprimirá toda la info del titular.
- Crear cuenta nueva: añadiremos una cuenta al *ArrayList* asignando como titular al cliente con el que estemos trabajando.

b) Si no existe una cuenta asociada al cliente consultado, se avisará al usuario de que su cliente no tiene ninguna *Cuenta* y le preguntaremos si quiere crear una nueva.

**Cuentas asociadas a Carla Rubio Moreno:**  
 El cliente no tiene cuentas activas.  
  
 [1] Crear nueva cuenta  
 [X] Volver al menú principal

## Programación

### EXAMEN TEMA 5 – ESTRUCTURAS DE DATOS

1. (2p) Dada la siguiente clase que usa anotaciones de la librería *Lombok*:

```
@RequiredArgsConstructor  
@AllArgsConstructor  
@ToString  
class Alumno {  
    @NonNull @Getter  
    private int nia;  
    @Getter @Setter  
    private String nombre;  
}
```

- a) Trascribe la estructura de la clase equivalente que se generaría automáticamente como *.class* al compilarla.
- b) Si intentamos añadir la anotación **@NoArgsConstructor**, ¿nos estaría permitido? Explica por qué.

2. (1p) Dada la siguiente clase:

```
@AllArgsConstructor  
@Getter  
class Alumno {  
  
    private int nia;  
    private String nombre;  
  
    public static void obtenerDescripcion(Alumno alumno) {  
        System.out.println("Alumno: " + alumno.getNombre() + ", NIA: " +  
alumno.getNia());  
    }  
}
```

Indica sobre el siguiente *main* la forma de llamar al método **obtenerDescripcion()**:

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

# Programación

## EXAMEN PRÁCTICO TEMA 5 – POO

(11/02/2025)



### **LEE ATENTAMENTE LAS SIGUIENTES INSTRUCCIONES ANTES DE EMPEZAR:**



- **Recopila en un documento de texto las evidencias de todo el examen. Guárdalo de vez en cuando para no perder el avance de tu trabajo.**
- Cuando termines, **pásalo a PDF y sube el documento creado a la entrega de AULES.**

### **PARTE 1: Configuración del entorno (0,5p)**

1. Crea un nuevo repositorio llamado “EXAMEN\_UD5\_[nombre]” desde *SourceTree*. El repositorio debe crearse en local y tener su espejo en remoto, por lo tanto, sincronízalo con *GitHub*.

**Pega a continuación la URL a tu nuevo repositorio de GitHub:**

2. Crea un nuevo proyecto Java (*Maven*) con *IntelliJ* -o el IDE que utilices- dentro del repositorio que acabas de crear. Llámalo “EXAMEN UD5”.
3. Crea en el proyecto un paquete nuevo llamado “concurso\_programamelo”.

**Sincroniza los cambios en tu repositorio remoto.**

### **PARTE 2: Resolución de problemas**

Programa en *Java* la solución a los siguientes ejercicios. Usa el proyecto que te acabas de crear en el apartado anterior. **Si no has conseguido crearlo correctamente, utiliza alguno de los proyectos que ya tenías para los ejercicios de clase y pega la URL de GitHub del repositorio al que vas a subir los cambios.**

#### **1. (6,5p) Consurso ProgramaMeLo con sede en el IES MUTXAMEL**

Dentro de unas semanas, el IES MUTXAMEL organizará un concurso de programación tipo olimpiada dirigido a los estudiantes de la asignatura de Programación y Robótica de 3º ESO.

El formato que ha pensado el Dpto. de Informática del centro es el siguiente:

- Cada estudiante que quiera participar como programador deberá buscar a más personas para formar un **grupo de hasta 4 personas**.
- Los equipos deberán resolver una serie de problemas planteados con Scratch. **Por cada problema resuelto, obtendrán 10 puntos que se sumará a su puntuación final.**
- La idea del concurso se ha propagado tanto que ya hemos recibido una proposición de un centro educativo de Barcelona para realizar allí una fase final nacional.



Para hacer posible todo esto, el centro necesita una aplicación para poder digitalizar la inscripción de los participantes junto a su equipo en el concurso y así tenerlo todo controlado.

### 1. (4,5p) En la app *ProgramaMeLo* habrá:

- Una clase **Concurso**, con los siguientes atributos:
  - *sede (String)* – debe ser siempre “MUTXAMEL”.
  - *fecha (LocalDate)* – se pide al usuario al crear un concurso nuevo.
  - *listaEquipos (ArrayList<Equipo>)* - por temas logísticos, se limita a 20 equipos como máximo.

Por defecto, se crea una lista sin equipos y se van agregando conforme van realizando su inscripción.

- Una clase **Equipo**, con los siguientes atributos:
  - *nombre\_equipo (String)*
  - *listaMiembros (ArrayList<String>)* – se debe permitir insertar varios elementos a la vez. Recuerda que el número máximo de miembros por equipo es 4.
  - *problemas\_resueltos (int)* – valor **0** por defecto hasta que se realice el concurso y se actualice.
  - *puntuación (int)* – valor **0** por defecto hasta que se realice el concurso y se actualice.

La lista de los miembros que componen al equipo se creará vacía en el momento de hacer la inscripción de dicho equipo, y se irá actualizando conforme se vaya fichando a compañer@s.

### A tener en cuenta...

- Dado que el instituto no quiere borrar a los equipos inscritos de su aplicación por si se clasifican para el concurso nacional en Barcelona, cualquier objeto de tipo *Concurso* y *Equipo* deben existir indistintamente dentro de la *app*.
- **No uses anotaciones de la librería Lombok.**

## 2. (2p) Funcionalidades extras a implementar en la clase *Concurso*.

- a) (1p) Implementa una funcionalidad en la app para que imprima el número total de equipos inscritos y sus datos con el siguiente formato

**- *nombre\_equipo. componentes*:**

```
Equipos inscritos: 2
- Los maquinitas. [Daniel, Lucas, Carla]
- Los triunfitos. [Juan, Patri, Alexia]
```

- b) (1p) Implementa una funcionalidad para que la app muestre al equipo ganador del concurso en función de la puntuación obtenida.

### Programa *main* de ejemplo

```
public class ProgramaMeLo {

    public static void main (String[] args) {

        Concurso iesmutxamel = new Concurso();
        System.out.println(iesmutxamel);

        Equipo maquinitas = new Equipo("Los maquinitas");
        System.out.println(maquinitas);
        iesmutxamel.inscribirEquipo(maquinitas);
        System.out.println(iesmutxamel);

        maquinitas.insertarMiembros("Daniel","Lucas");
        System.out.println(maquinitas);

        maquinitas.insertarMiembros("Carla","Eros","Lucía");

        Equipo triunfitos = new Equipo("Los triunfitos");
        iesmutxamel.inscribirEquipo(triunfitos);
        triunfitos.insertarMiembros("Juan","Patri","Alexia");
        System.out.println(triunfitos);

        maquinitas.setProblemas_resueltos(2);
        maquinitas.calcularPuntuacion();
        triunfitos.setProblemas_resueltos(3);
        triunfitos.calcularPuntuacion();

        //a)
        iesmutxamel.imprimirDatosEquipos();

        //b)
        iesmutxamel.equipoGanador();

    }
}
```

## Salida de ejemplo

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\lib\idea_rt.jar=56600:C:\Progr  
¿Cuándo se realizará el concurso?  
Dia:  
3  
Mes:  
3  
Año:  
2025  
Concurso{sede='MUTXAMEL', fecha=2025-03-03, listaEquipos=[]}  
Equipo{nombre_equipo='Los maquinitas', listaMiembros=[], problemas_resueltos=0, puntuacion=0}  
Nuevo equipo Los maquinitas inscrito.  
Concurso{sede='MUTXAMEL', fecha=2025-03-03, listaEquipos=[Equipo{nombre_equipo='Los maquinitas', listaMiembros=[], problemas_resueltos=0, puntuacion=0}]}  
Insertados nuevos miembros para el equipo Los maquinitas  
Equipo{nombre_equipo='Los maquinitas', listaMiembros=[Daniel, Lucas], problemas_resueltos=0, puntuacion=0}  
No se pueden insertar 3 miembros más en el equipo Los maquinitas. Ya tiene 2 miembros.  
Nuevo equipo Los triunfitos inscrito.  
Insertados nuevos miembros para el equipo Los triunfitos  
Equipo{nombre_equipo='Los triunfitos', listaMiembros=[Juan, Patri, Alexia], problemas_resueltos=0, puntuacion=0}  
Puntuación calculada para el equipo Los maquinitas: 20 puntos.  
Puntuación calculada para el equipo Los triunfitos: 30 puntos.  
Equipos inscritos: 2  
- Los maquinitas. [Daniel, Lucas]  
- Los triunfitos. [Juan, Patri, Alexia]  
El equipo ganador ha sido Los triunfitos con 30 puntos.
```

# Programación

## EXAMEN PRÁCTICO TEMA 5 – POO

(10/02/2025)



### LEE ATENTAMENTE LAS SIGUIENTES INSTRUCCIONES ANTES DE EMPEZAR:



- Recopila en un documento de texto las evidencias de todo el examen. Guárdalo de vez en cuando para no perder el avance de tu trabajo.
- Cuando termines, **pásalo a PDF y sube el documento creado a la entrega de AULES.**

### PARTE 1: Configuración del entorno (0,5p)

1. Crea un nuevo repositorio llamado “EXAMEN\_UD5\_[nombre]” desde SourceTree. El repositorio debe crearse en local y tener su espejo en remoto, por lo tanto, sincronízalo con GitHub.

**Pega a continuación la URL a tu nuevo repositorio de GitHub:**

2. Crea un nuevo proyecto Java (Maven) con IntelliJ -o el IDE que utilices- dentro del repositorio que acabas de crear. Llámalo “EXAMEN UD5”.
3. Crea en el proyecto un paquete nuevo llamado “carrera\_solidaria”.

**Sincroniza los cambios en tu repositorio remoto.**

### PARTE 2: Resolución de problemas

Programa en Java la solución a los siguientes ejercicios. Usa el proyecto que te acabas de crear en el apartado anterior. **Si no has conseguido crearlo correctamente, utiliza alguno de los proyectos que ya tenías para los ejercicios de clase y pega la URL de GitHub del repositorio al que vas a subir los cambios.**

#### **1. (6,5p) Carrera solidaria del IES MUTXAMEL por la DANA de Valencia**

Dentro de unas semanas, el IES MUTXAMEL organizará una carrera solidaria para sacar fondos y enviarlos a los afectados por la DANA ocurrida en Valencia a finales del mes de octubre.

El formato que ha pensado la dirección del centro es el siguiente:

- Cada estudiante que quiera participar como corredor deberá **buscar patrocinadores** (valen profesores, familiares, amigos,...) cualquiera que quiera ser solidario.
- **Cada corredor acuerda con sus patrocinadores una cantidad a pagar (€) por vuelta completada.** Es decir, si se acuerda una cantidad de 1€ y el corredor da 10 vueltas al circuito, el patrocinador deberá pagar 10€ al corredor, el cual deberá guardarse el dinero recaudado para entregar su donación al centro más tarde.



Para hacer posible todo esto, el centro necesita una aplicación para poder digitalizar la inscripción de los participantes en la carrera y así tenerlo todo controlado.

**1. (4,5p) En la app *KmSolidariosIESMutxamel* habrá:**

- Una clase **Organizador**, con los siguientes atributos:
  - *nombre (String)* – debe ser siempre “IES MUTXAMEL”.
  - *edicion (String)* – se empieza por la primera (*I*), si se crea otra la *II*,...
  - *listaCorredores (ArrayList<Corredor>)* - por temas logísticos, se limita a 700 participantes como máximo.

Por defecto, se crea sin corredores y se van agregando conforme van realizando su inscripción en conserjería.

- Una clase **Corredor**, con los siguientes atributos:
  - *nombre (String)*
  - *dorsal (int)* – autogenerado por orden de inscripción empezando por 1.
  - *listaPatrocinadores (ArrayList<String>)* – se debe permitir insertar varios elementos a la vez.
  - *importe\_vuelta (double)* – si no se indica, **0.50€** por defecto.
  - *vueltas\_completadas (int)* – valor **0** por defecto hasta que se realice la carrera y se actualice.

La lista de patrocinadores se creará vacía en el momento de hacer la inscripción del corredor, y se irá actualizando conforme se vayan consiguiendo acuerdos.

### A tener en cuenta...

- Dado que el instituto no quiere borrar a los corredores inscritos de su aplicación por si se realizan más ediciones de la carrera y quieren volver a participar, cualquier objeto de tipo *Organizador* y *Corredor* deben existir indistintamente dentro de la *app*.
- **No uses anotaciones de la librería Lombok.**

## 2. (2p) Funcionalidades extras a implementar en la clase *Organizador*.

- a) (1p) Implementa una funcionalidad en la app para que imprima el número total de corredores inscritos y sus nombres con el siguiente formato **[dorsal]. nombre**:

Corredores inscritos: 2  
[1]. Luis  
[2]. Carla

- b) (1p) Implementa una funcionalidad para que se pueda calcular el importe de la donación a realizar por un corredor en cuanto acabe la carrera y se hayan actualizado las vueltas dadas al circuito por cada uno de ellos.

### Programa *main* de ejemplo

```
public class AppCarrera {  
  
    public static void main (String[] args) {  
  
        Organizador iesmutxamel = new Organizador("I");  
        System.out.println(iesmutxamel);  
  
        Corredor corredor1 = new Corredor("Luis",1.5);  
        System.out.println(corredor1);  
        iesmutxamel.inscribir_corredor(corredor1);  
  
        corredor1.insertarPatrocinador("Madre","Profe_mates","Profe_ingles");  
        corredor1.insertarPatrocinador("Profe_cast");  
        System.out.println(corredor1);  
  
        Corredor corredor2 = new Corredor("Carla",0.0);  
        System.out.println(corredor2);  
        iesmutxamel.inscribir_corredor(corredor2);  
  
        System.out.println(iesmutxamel);  
  
        //a)  
        iesmutxamel.mostrarCorredores();  
  
        //b)  
        corredor1.setNumero_vueltas(15);  
        iesmutxamel.calcularDonacion(corredor1);  
  
    }  
}
```

## Salida de ejemplo

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\lib\idea_rt.jar=55728:C:\Program  
- Organizador [nombre='IES MUTXAMEL', edicion='I', listaCorredores=[]]  
- Corredor [nombre='Luis', dorsal=1, listaPatrocinadores=[], importe_vuelta=1.5, numero_vueltas=0]  
* Nueva inscripción realizada. Corredor : Luis con dorsal 1  
* Nuevos patrocinadores añadidos para el corredor Luis  
* Nuevos patrocinadores añadidos para el corredor Luis  
- Corredor [nombre='Luis', dorsal=1, listaPatrocinadores=[Madre, Profe_mates, Profe_ingles, Profe_cast], importe_vuelta=1.5, numero_vueltas=0]  
- Corredor [nombre='Carla', dorsal=2, listaPatrocinadores=[], importe_vuelta=0.5, numero_vueltas=0]  
* Nueva inscripción realizada. Corredor : Carla con dorsal 2  
- Organizador [nombre='IES MUTXAMEL', edicion='I', listaCorredores=[- Corredor [nombre='Luis', dorsal=1, listaPatrocinadores=[Madre, Profe_mates, Profe_in  
Corredores inscritos: 2  
[1]. Luis  
[2]. Carla  
El corredor Luis ha conseguido una donación de 22.5€  
Process finished with exit code 0
```

Profe\_cast], importe\_vuelta=1.5, numero\_vueltas=0], - Corredor [nombre='Carla', dorsal=2, listaPatrocinadores=[], importe\_vuelta=0.5, numero\_vueltas=0]]]



