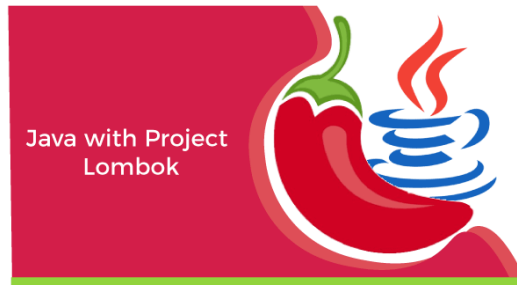


## PRÁCTICA 3. LIBRERÍAS DE AYUDA PARA LA IMPLEMENTACIÓN DE LA POO:

### LOMBOK



[Project Lombok](https://lombok.com/)

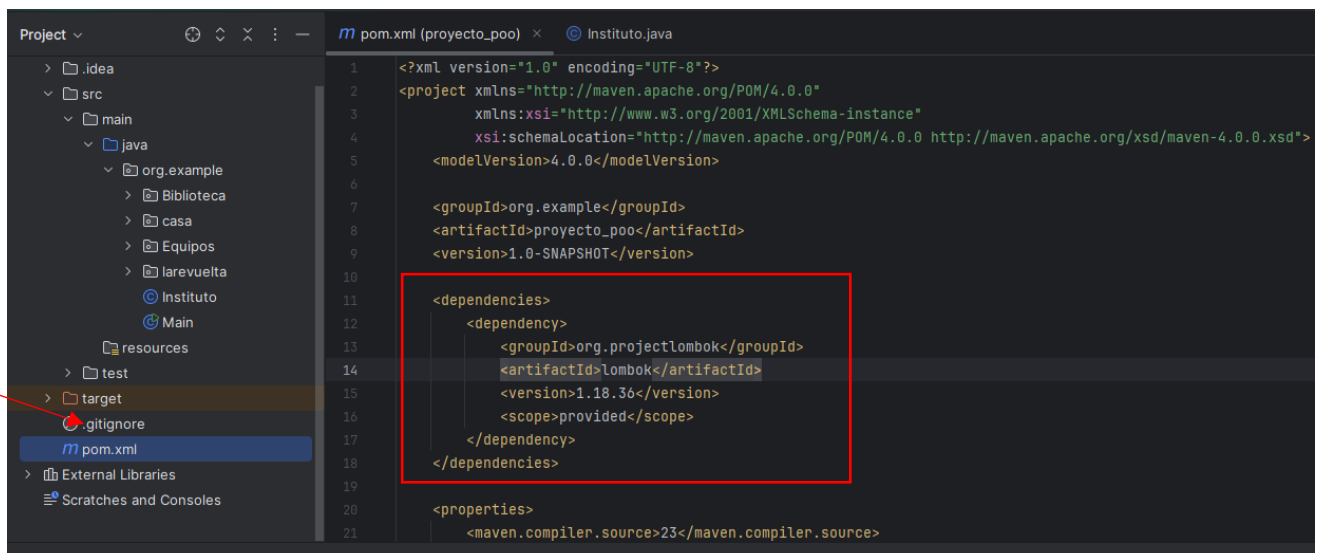
#### ➔ Introducción a Lombok en Java

**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.

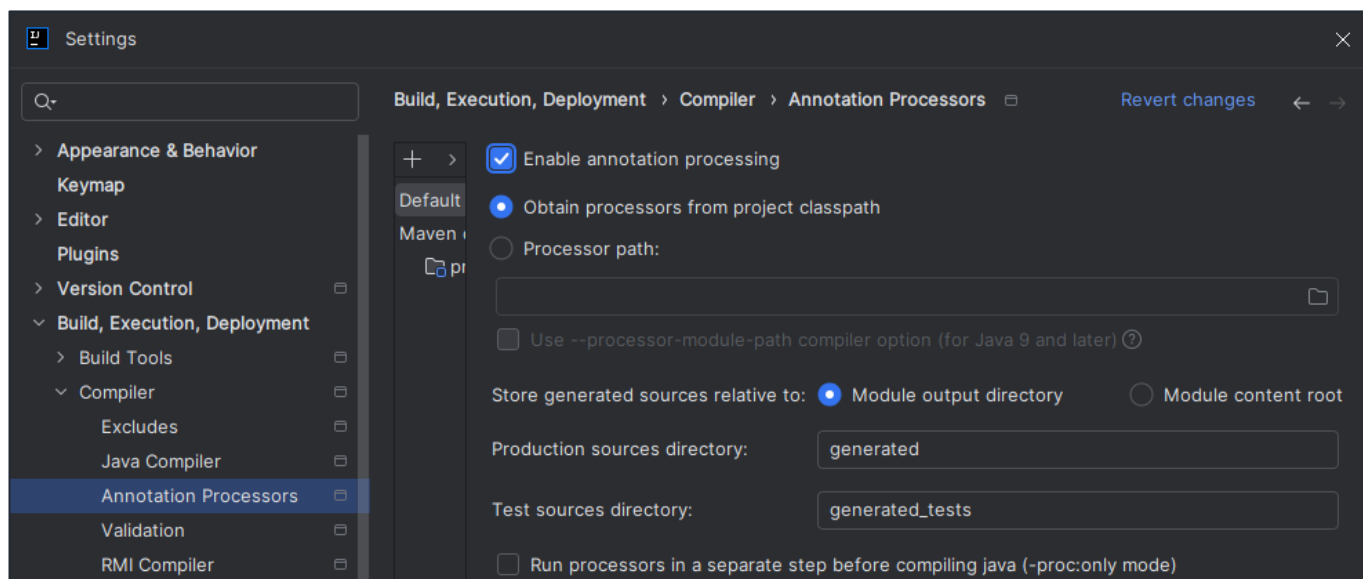
#### ➔ INSTALACIÓN

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

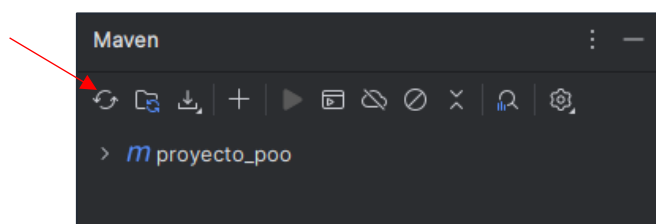
```
<dependencies>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.36</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```



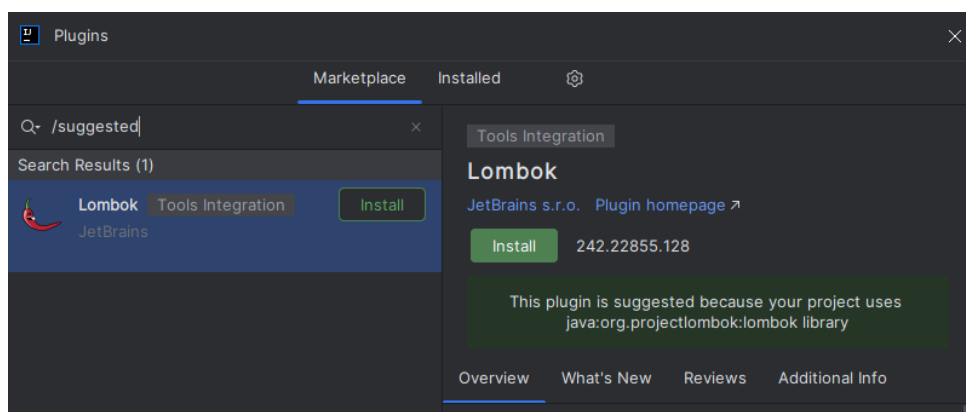
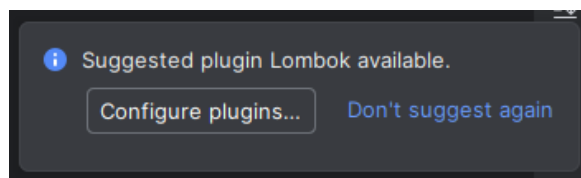
Vamos a activar el procesamiento de anotaciones. Lo habilitaremos en **Settings** → **Build, Execution, Deployment** → **Compiler** → **Annotation Processors**



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



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



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

## → ANOTACIONES PRINCIPALES DE LOMBOK

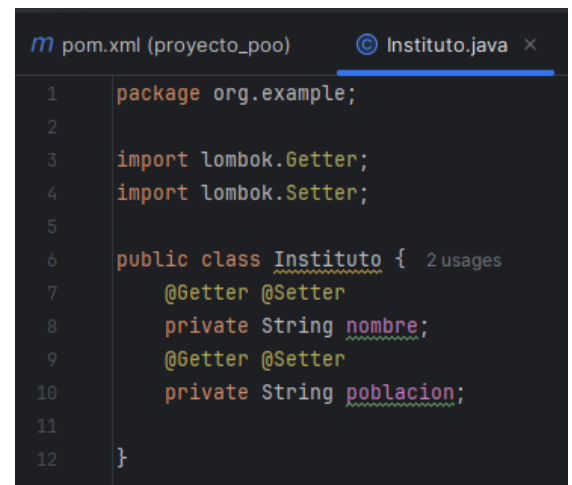
Una vez realizada toda la configuración, vamos a probar que *Lombok* funciona como se espera.

## 1. @Getter y @Setter

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

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

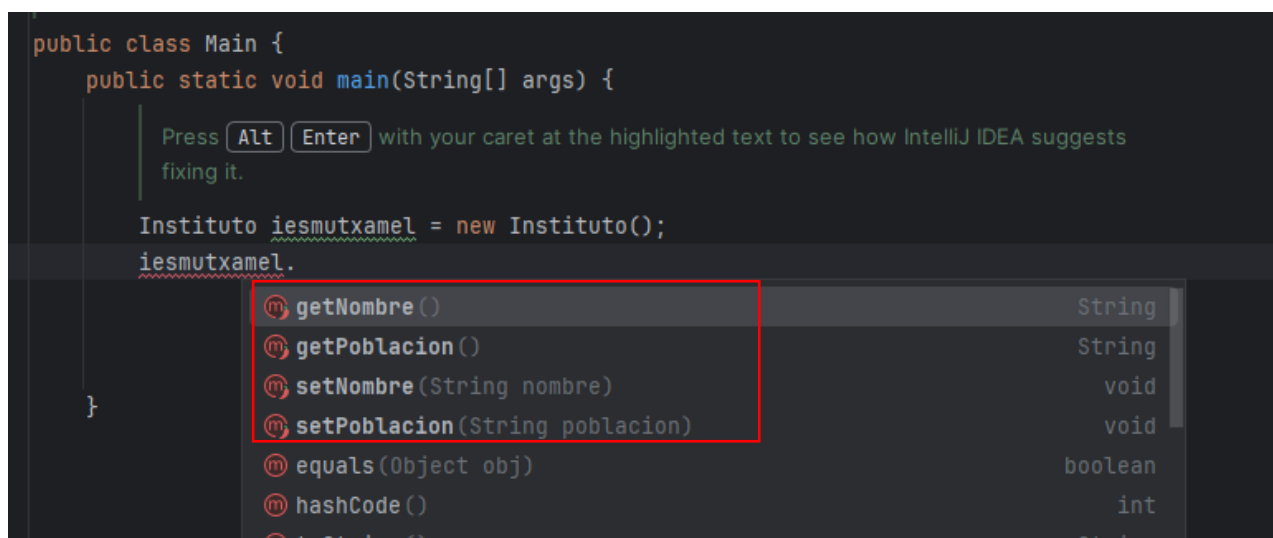
public class Instituto {
    @Getter @Setter
    private String nombre;
    @Getter @Setter
    private String poblacion;
}
```



```
m pom.xml (proyecto_poo) Instituto.java x
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”):



```
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.

Method	Return 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 habérselo ahorrado simplemente especificándolo encima de la clase:

```

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:

```

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.

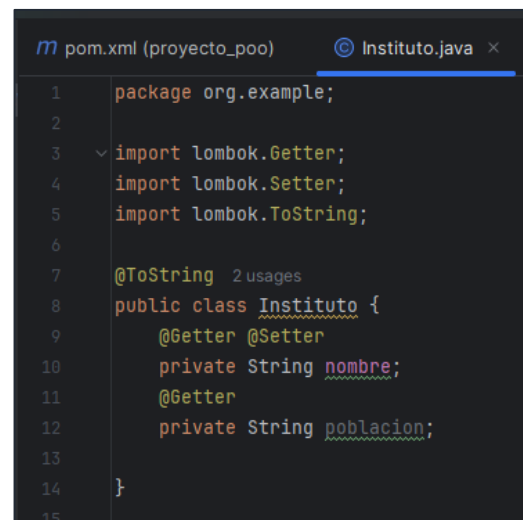
- `getNombre()` String
- `getPoblacion()` String
- `setNombre(String nombre)` void
- `equals(Object obj)` boolean

## 2. @ToString

Genera automáticamente el método `toString()`:

```
import lombok.ToString;

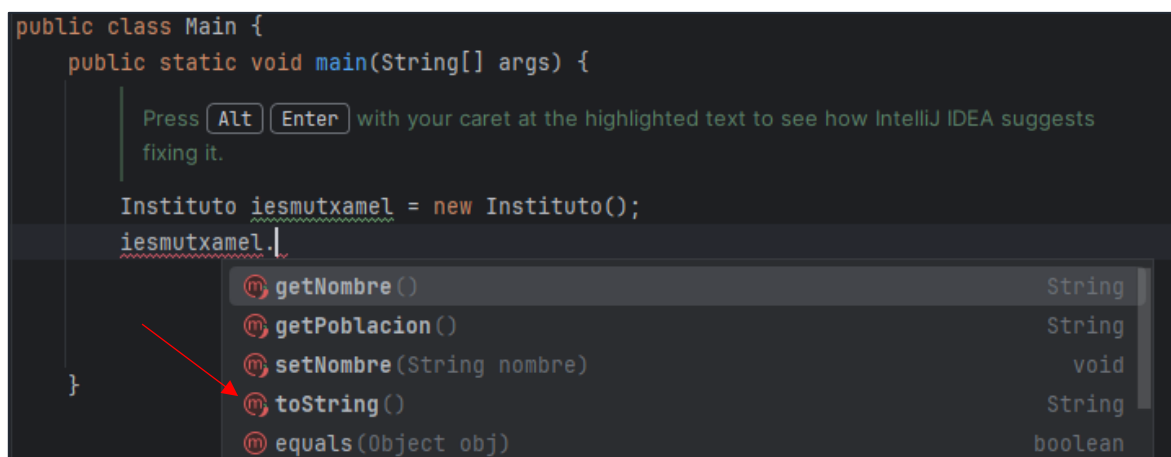
@ToString
public class Instituto {
    @Getter @Setter
    private String nombre;
    @Getter
    private String poblacion;
}
```



The screenshot shows the IntelliJ IDEA interface with two tabs: 'pom.xml (proyecto\_poo)' and 'Instituto.java'. The 'Instituto.java' tab is active, showing the following code:

```
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*:



The screenshot shows the IntelliJ IDEA interface with a code editor displaying the `Main` class. The `main` method is highlighted, and a tooltip提示 is shown: "Press [Alt] [Enter] with your caret at the highlighted text to see how IntelliJ IDEA suggests fixing it." Below the code, a code completion menu is open, showing the following methods:

- `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 atrib
@ToString
@Getter @Setter
public class Instituto {
    private String nombre;
    private String poblacion;
    private int codigo;
}
```



The screenshot shows the IntelliJ IDEA interface with two tabs: 'pom.xml (proyecto\_poo)' and 'Instituto.java'. The 'Instituto.java' tab is active, showing the following code:

```
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:

```

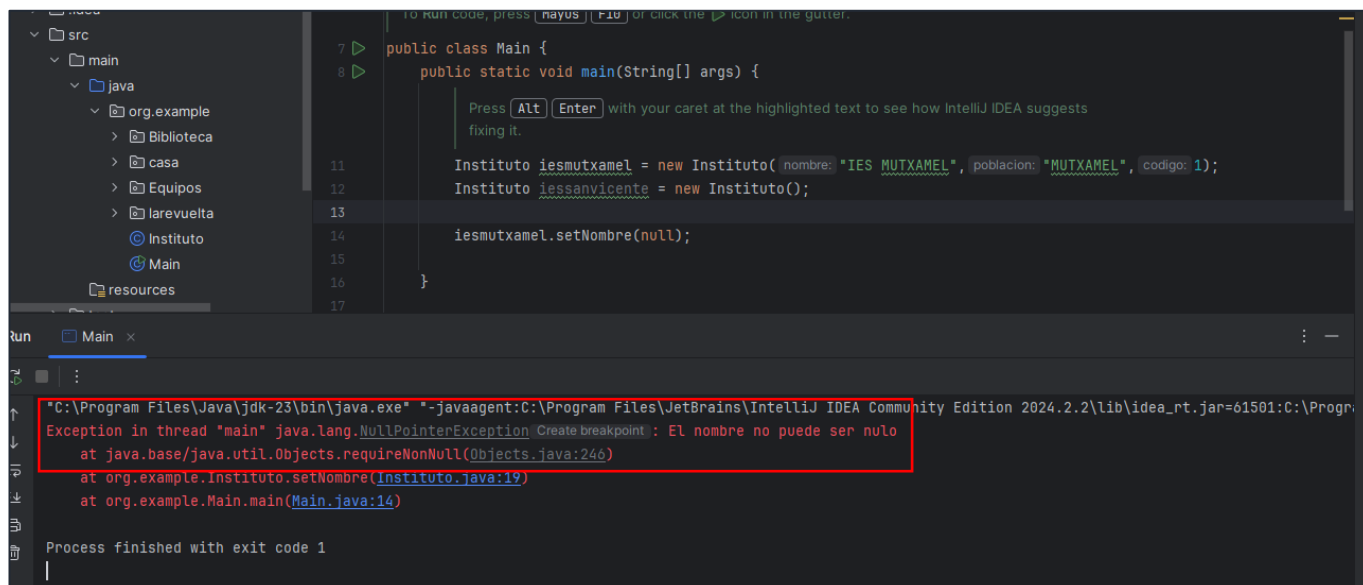
@AllArgsConstructor
@RequiredArgsConstructor
@ToString
@Getter @Setter
public class Instituto {

    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 intentáramos modificar el nombre de un instituto por un valor nulo:



Pruébalo tú...

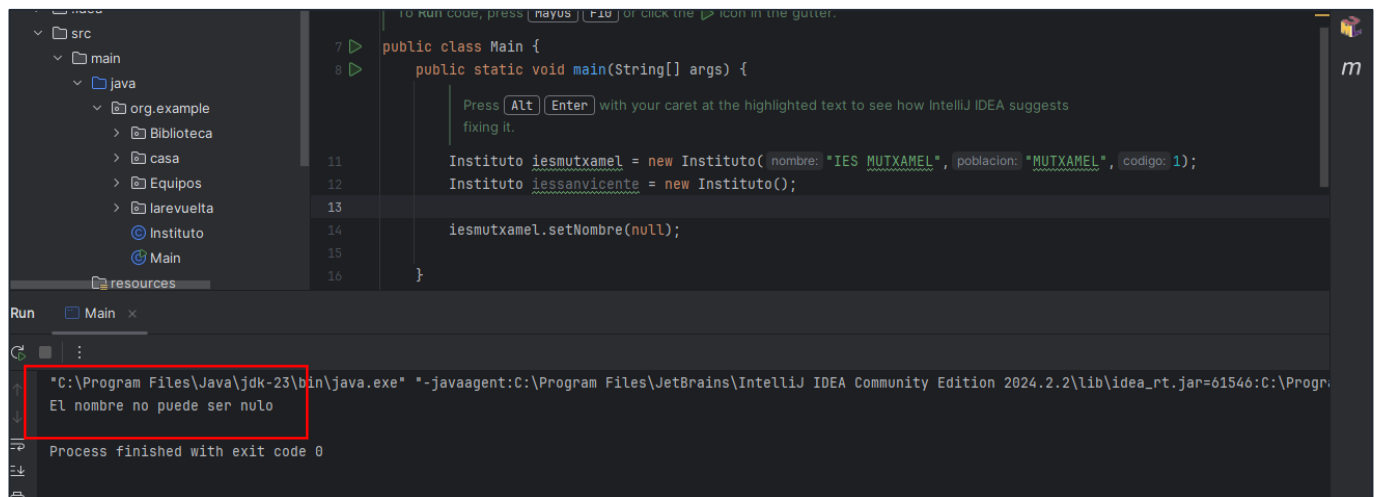
## 2. Uso de validaciones manuales

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;
        }
    }
}
```



**Pruébalo tú...**

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

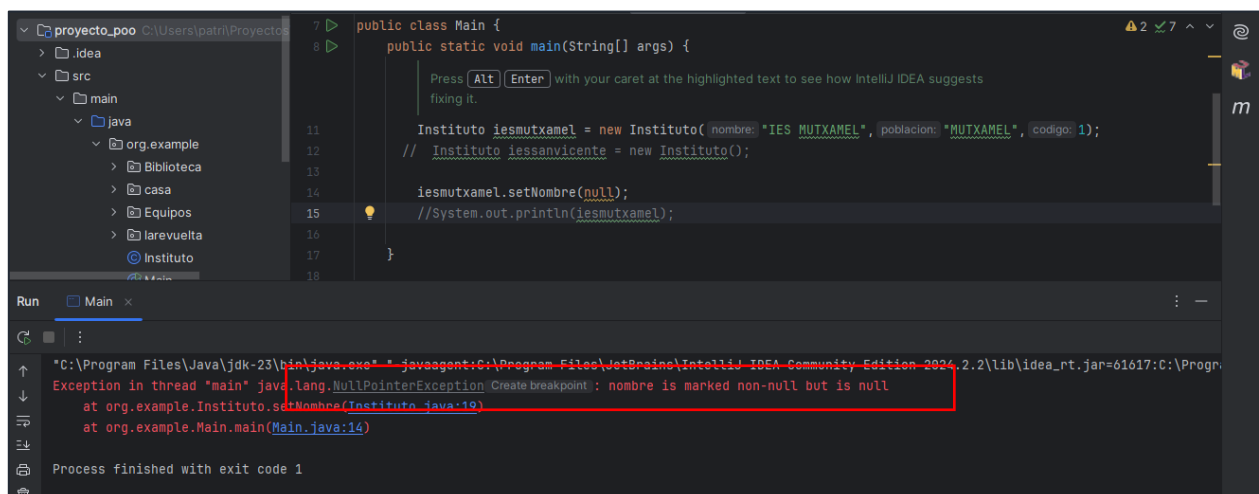
Pero la forma que más nos interesa, en este caso, es la que nos proporciona *Lombok* con la marca `@NonNull`:

```
@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`:



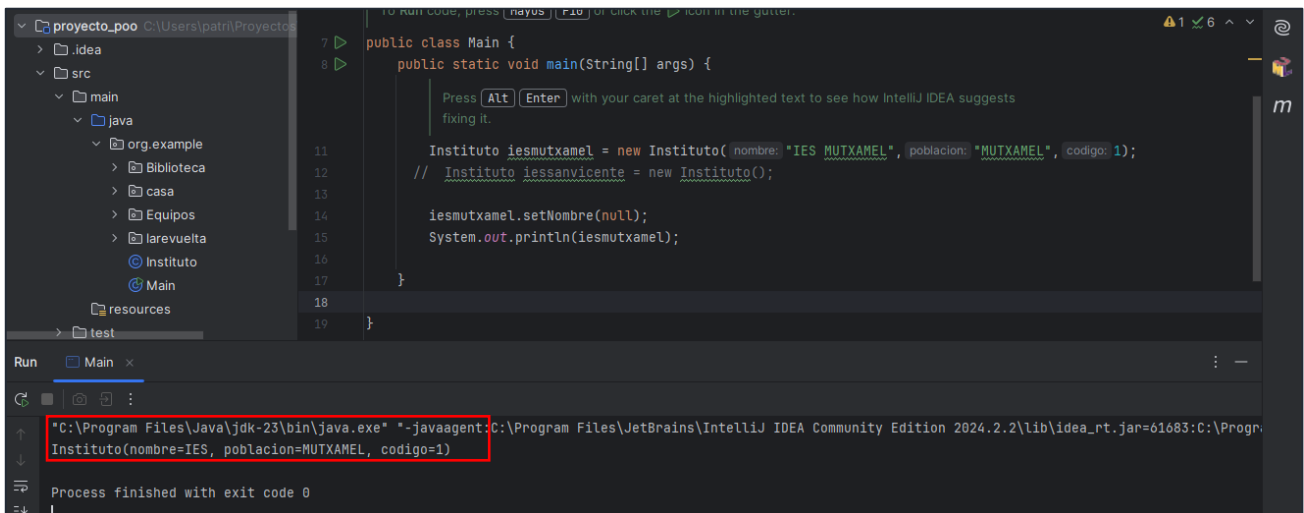


Como normalmente no vamos a querer que nuestro programa aborte de forma abrupta, “arreglaremos” esto asignando un *nombre por defecto*, por ejemplo “IES”. La mejor manera de hacerlo es combinar `@NonNull` con `Objects.requireNonNullElse()`, que asignará el nuevo valor:

```
@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");
    }
}
```



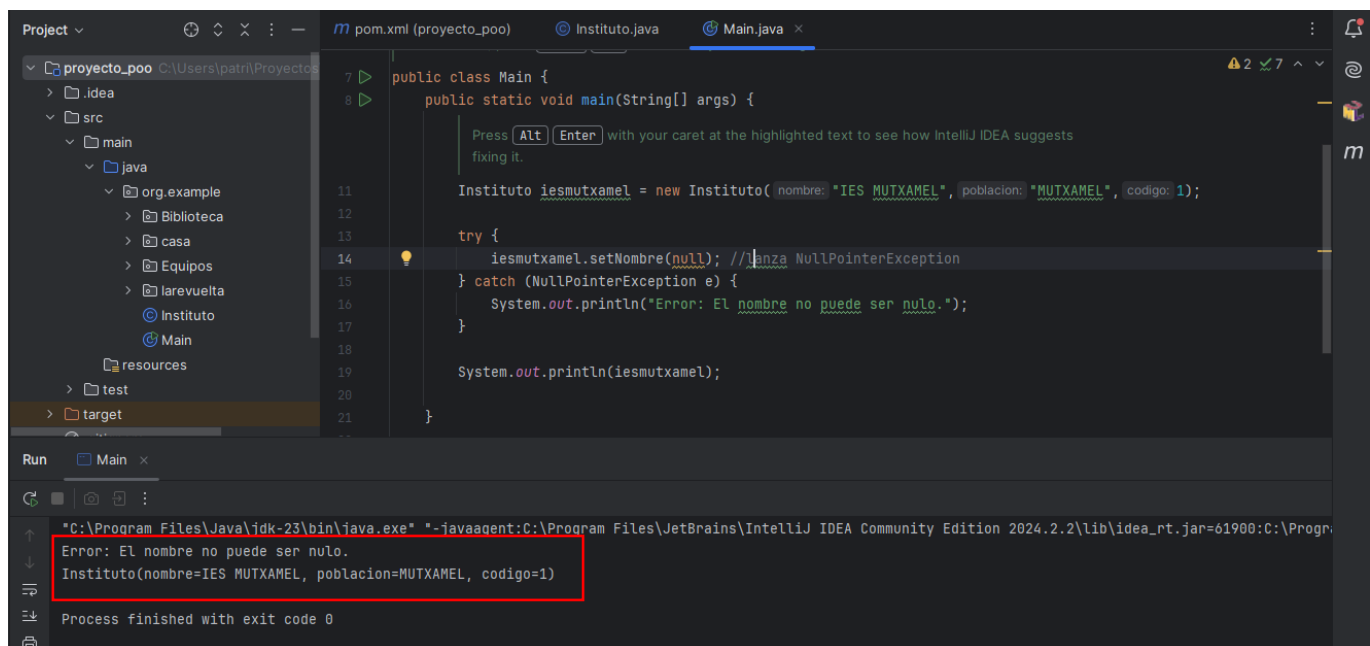
Si no queremos asignar un nombre por defecto y simplemente queremos controlar el error, podemos agregar un bloque *try-catch* en el *main* para controlar excepciones que puedan producirse dentro del bloque de código susceptible de fallar (en nuestro caso el método `setNombre()`):

```
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);
    }
}
```



## ➔ 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.

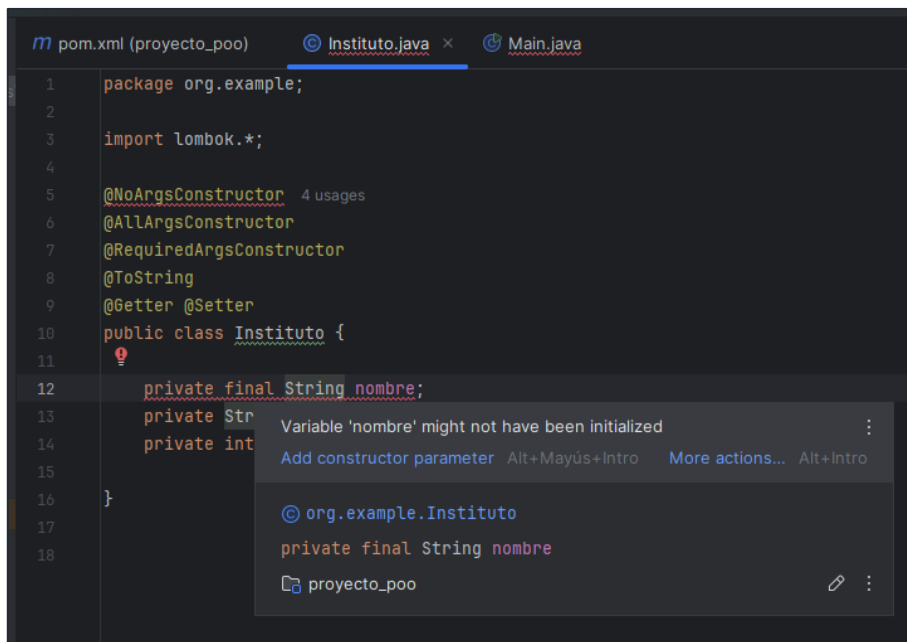
Un ejemplo de uso:

```
@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`.



⚠️ 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).

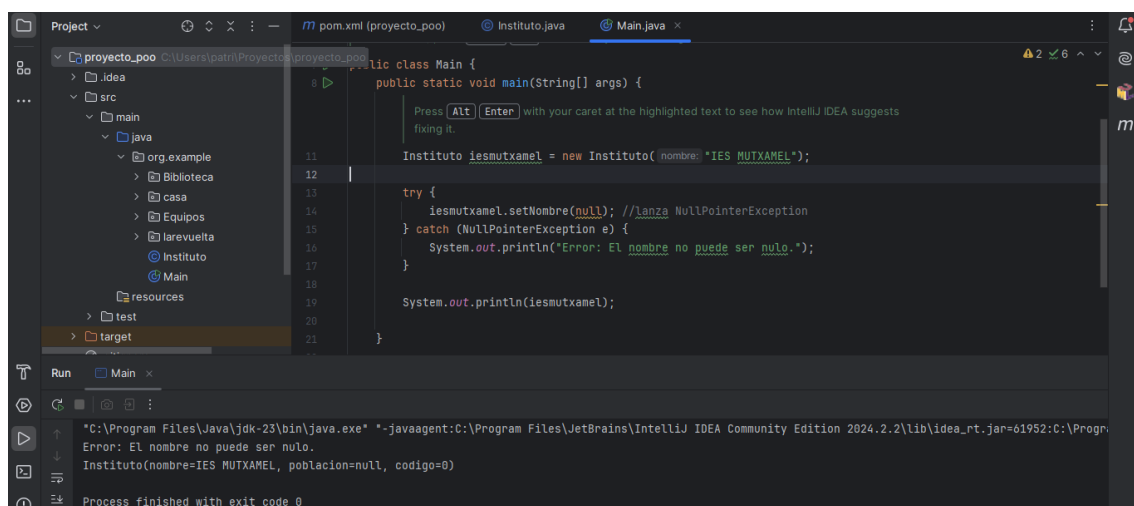
## 2. @Data

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 {

    @NonNull
    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.**