LOMBOCK

**Informe Completo sobre Lombok**

**Introducción**

Lombok es una biblioteca de Java que ayuda a reducir el código boilerplate mediante el uso de anotaciones. Al integrar Lombok en un proyecto, los desarrolladores pueden generar automáticamente métodos comunes como getters, setters, constructores, métodos toString, equals, hashCode, y más, simplificando significativamente el código y mejorando su legibilidad.

**Instalación**

**Gradle**

Para utilizar Lombok en un proyecto con Gradle, añade las siguientes líneas a tu archivo build.gradle:

dependencies {

compileOnly 'org.projectlombok:lombok:1.18.26'

annotationProcessor 'org.projectlombok:lombok:1.18.26'

}

**Maven**

Si utilizas Maven, añade la dependencia a tu archivo pom.xml:

xml

Copiar código

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.26</version>

<scope>provided</scope>

</dependency>

**Principales Anotaciones de Lombok**

**@Getter y @Setter**

Genera automáticamente los métodos getter y setter para los campos de la clase.

import lombok.Getter;

import lombok.Setter;

@Getter

@Setter

public class Persona {

private String nombre;

private int edad;

}

**@ToString**

Genera un método toString para la clase.

import lombok.ToString;

@ToString

public class Persona {

private String nombre;

private int edad;

}

**@EqualsAndHashCode**

Genera métodos equals y hashCode basados en los campos de la clase.

import lombok.EqualsAndHashCode;

@EqualsAndHashCode

public class Persona {

private String nombre;

private int edad;

}

**@NoArgsConstructor, @AllArgsConstructor y @RequiredArgsConstructor**

Genera constructores sin argumentos, con todos los argumentos y con los argumentos requeridos (campos final o con anotaciones específicas).

import lombok.NoArgsConstructor;

import lombok.AllArgsConstructor;

import lombok.RequiredArgsConstructor;

@NoArgsConstructor

@AllArgsConstructor

@RequiredArgsConstructor

public class Persona {

private final String nombre;

private int edad;

}

**@Data**

Combina varias anotaciones en una sola: @Getter, @Setter, @ToString, @EqualsAndHashCode, y @RequiredArgsConstructor.

import lombok.Data;

@Data

public class Persona {

private final String nombre;

private int edad;

}

**@Builder**

Permite implementar el patrón de diseño Builder.

import lombok.Builder;

@Builder

public class Persona {

private String nombre;

private int edad;

}

// Ejemplo de uso

Persona persona = Persona.builder()

.nombre("Juan")

.edad(30)

.build();

**Ejemplo Completo**

import lombok.Data;

import lombok.Builder;

import lombok.NoArgsConstructor;

import lombok.AllArgsConstructor;

@Data

@NoArgsConstructor

@AllArgsConstructor

@Builder

public class Persona {

private String nombre;

private int edad;

public static void main(String[] args) {

Persona persona = Persona.builder()

.nombre("Juan")

.edad(30)

.build();

System.out.println(persona);

}

}

**Otras Anotaciones de Lombok**

**@NonNull**

Genera verificaciones de nullidad para los parámetros de los métodos y constructores.

java

Copiar código

import lombok.NonNull;

public class Persona {

private String nombre;

public void setNombre(@NonNull String nombre) {

this.nombre = nombre;

}

}

**@Value**

Una alternativa a @Data para clases inmutables. Marca todos los campos como private final, genera getters, toString, equals, hashCode, y un constructor para todos los campos.

java

Copiar código

import lombok.Value;

@Value

public class Persona {

private String nombre;

private int edad;

}

**@SneakyThrows**

Permite lanzar excepciones sin declararlas en la firma del método.

**Informe Completo sobre Project Lombok**

**Introducción**

Project Lombok es una biblioteca de Java que busca reducir drásticamente el código repetitivo (*boilerplate code*) mediante el uso de anotaciones. En el desarrollo de aplicaciones Java, es común escribir código predecible y repetitivo para tareas como la creación de *getters*, *setters*, constructores, métodos equals(), hashCode() y toString(). Lombok automatiza la generación de este código en tiempo de compilación, lo que permite a los desarrolladores escribir un código más conciso, legible y fácil de mantener.

Al integrar Lombok en un proyecto, los desarrolladores pueden enfocarse más en la lógica de negocio y menos en la infraestructura del lenguaje, mejorando significativamente la productividad y la claridad del código.

**Instalación**

Para utilizar Lombok, debes añadirlo como una dependencia en tu sistema de construcción de proyectos.

Gradle

Añade las siguientes líneas a tu archivo

build.gradle:

Gradle

dependencies { //

compileOnly 'org.projectlombok:lombok:1.18.26' //

annotationProcessor 'org.projectlombok:lombok:1.18.26' //

}

Maven

Si utilizas Maven, añade la dependencia a tu archivo

pom.xml:

XML

<dependency> //

<groupId>org.projectlombok</groupId> //

<artifactId>lombok</artifactId> //

<version>1.18.26</version> //

<scope>provided</scope> //

</dependency>

Es crucial que tu IDE tenga soporte para Lombok (generalmente a través de un *plugin*) para que pueda reconocer y compilar correctamente el código generado.

**Principales Anotaciones de Lombok**

Lombok ofrece una variedad de anotaciones que cubren las necesidades más comunes en el desarrollo Java.

@Getter y @Setter

Estas anotaciones generan automáticamente los métodos *getter* y *setter* para los campos de la clase. Puedes aplicarlas a nivel de clase para todos los campos, o a nivel de campo para generar métodos específicos.

**Ejemplo:**

Java

import lombok.Getter; //

import lombok.Setter; //

@Getter //

@Setter //

public class Producto {

private String nombre; // Genera getNombre() y setNombre()

private double precio; // Genera getPrecio() y setPrecio()

private int stock; // Genera getStock() y setStock()

public static void main(String[] args) {

Producto producto = new Producto();

producto.setNombre("Laptop");

producto.setPrecio(1200.00);

producto.setStock(50);

System.out.println("Nombre: " + producto.getNombre());

System.out.println("Precio: " + producto.getPrecio());

System.out.println("Stock: " + producto.getStock());

}

}

**Salida:**

Nombre: Laptop

Precio: 1200.0

Stock: 50

@ToString

Genera un método toString() que devuelve una representación en cadena de los campos de la clase. Por defecto, incluye todos los campos no estáticos.

**Ejemplo:**

Java

import lombok.ToString; //

@ToString //

public class Coche {

private String marca;

private String modelo;

private int año;

public Coche(String marca, String modelo, int año) {

this.marca = marca;

this.modelo = modelo;

this.año = año;

}

public static void main(String[] args) {

Coche coche = new Coche("Ford", "Focus", 2020);

System.out.println(coche); // Llama automáticamente a coche.toString()

}

}

**Salida:**

Coche(marca=Ford, modelo=Focus, año=2020)

Para evitar ciclos infinitos en relaciones bidireccionales, puedes usar

@ToString(exclude = "campo"), como se explicará más adelante.

@EqualsAndHashCode

Genera implementaciones de los métodos equals() y hashCode() basadas en los campos de la clase. Es crucial para el correcto funcionamiento de colecciones como HashSet y HashMap, ya que garantiza que objetos lógicamente iguales sean tratados como tales. Si sobrescribes equals(), **siempre** debes sobrescribir hashCode() para cumplir el contrato de Object.

**Ejemplo:**

Java

import lombok.EqualsAndHashCode; //

@EqualsAndHashCode //

public class Punto {

private int x;

private int y;

public Punto(int x, int y) {

this.x = x;

this.y = y;

}

public static void main(String[] args) {

Punto p1 = new Punto(1, 2);

Punto p2 = new Punto(1, 2);

Punto p3 = new Punto(3, 4);

System.out.println("p1.equals(p2): " + p1.equals(p2)); // true, lógicamente iguales

System.out.println("p1.equals(p3): " + p1.equals(p3)); // false

System.out.println("p1.hashCode(): " + p1.hashCode());

System.out.println("p2.hashCode(): " + p2.hashCode()); // Mismo hash que p1

System.out.println("p3.hashCode(): " + p3.hashCode()); // Hash diferente

}

}

**Salida (los valores exactos de hashCode pueden variar):**

p1.equals(p2): true

p1.equals(p3): false

p1.hashCode(): 30972

p2.hashCode(): 30972

p3.hashCode(): 98316

@NoArgsConstructor, @AllArgsConstructor y @RequiredArgsConstructor

Estas anotaciones generan diferentes tipos de constructores:

* **@NoArgsConstructor**: Genera un constructor sin argumentos.
* **@AllArgsConstructor**: Genera un constructor con un argumento para cada campo de la clase.
* **@RequiredArgsConstructor**: Genera un constructor con argumentos para todos los campos final y para los campos anotados con @NonNull.

**Ejemplo:**

Java

import lombok.NoArgsConstructor; //

import lombok.AllArgsConstructor; //

import lombok.RequiredArgsConstructor; //

import lombok.Getter; // Necesario para acceder a los campos

@Getter // Para poder leer los valores después de la construcción

@NoArgsConstructor //

@AllArgsConstructor //

@RequiredArgsConstructor //

public class Libro {

private final String isbn; // Campo final, requerido por @RequiredArgsConstructor

private String titulo; // Campo no final, incluido por @AllArgsConstructor

private int paginas; // Campo no final, incluido por @AllArgsConstructor

public static void main(String[] args) {

// Usando @NoArgsConstructor (solo si no hay campos final sin inicializar)

// Libro libroSinArgs = new Libro(); // Error si isbn es final y no se inicializa

// Usando @RequiredArgsConstructor

Libro libroReq = new Libro("978-0321765723");

libroReq.setTitulo("Clean Code"); // Asumiendo que también tienes @Setter

libroReq.setPaginas(464);

System.out.println("Required Args: ISBN=" + libroReq.getIsbn() + ", Titulo=" + libroReq.getTitulo());

// Usando @AllArgsConstructor

Libro libroAll = new Libro("978-1234567890", "Design Patterns", 395);

System.out.println("All Args: ISBN=" + libroAll.getIsbn() + ", Titulo=" + libroAll.getTitulo() + ", Paginas=" + libroAll.getPaginas());

}

}

**Salida (ejemplo ilustrativo, depende de si se incluyen @Setter y @ToString):**

Required Args: ISBN=978-0321765723, Titulo=Clean Code

All Args: ISBN=978-1234567890, Titulo=Design Patterns, Paginas=395

@Data

Es una anotación de conveniencia que combina varias anotaciones comunes en una sola. Es equivalente a usar @Getter, @Setter, @ToString, @EqualsAndHashCode, y @RequiredArgsConstructor. Es muy útil para clases de datos simples (POJOs).

**Ejemplo:**

import lombok.Data; //

@Data //

public class Usuario {

private final String id; // Será requerido por @RequiredArgsConstructor

private String nombre;

private String email;

public static void main(String[] args) {

Usuario usuario = new Usuario("user123"); // Constructor de @RequiredArgsConstructor

usuario.setNombre("Carlos"); // Setter de @Setter

usuario.setEmail("carlos@example.com"); // Setter de @Setter

System.out.println(usuario); // toString() de @ToString

Usuario otroUsuario = new Usuario("user123");

otroUsuario.setNombre("Carlos");

otroUsuario.setEmail("carlos@example.com");

System.out.println("Usuarios son iguales: " + usuario.equals(otroUsuario)); // equals() de @EqualsAndHashCode

}

}

**Salida:**

Usuario(id=user123, nombre=Carlos, email=carlos@example.com)

Usuarios son iguales: true

@Builder

Permite implementar el patrón de diseño Builder, facilitando la creación de objetos complejos con muchos atributos opcionales de manera legible y segura. Lombok genera la clase Builder interna con métodos encadenables.

**Ejemplo:**

Java

import lombok.Builder; //

import lombok.ToString; // Para ver la salida

@Builder //

@ToString

public class Configuracion {

private String tema;

private int fontSize;

private boolean notificacionesActivas;

private String idioma;

public static void main(String[] args) {

// Crear una instancia de Configuracion usando el Builder

Configuracion config = Configuracion.builder() //

.tema("Dark") //

.fontSize(14) //

.notificacionesActivas(true)

.build(); // cite:

System.out.println(config);

// Otra configuración, sin especificar todos los campos

Configuracion config2 = Configuracion.builder()

.idioma("es")

.build();

System.out.println(config2);

}

}

**Salida:**

Configuracion(tema=Dark, fontSize=14, notificacionesActivas=true, idioma=null)

Configuracion(tema=null, fontSize=0, notificacionesActivas=false, idioma=es)

**Ejemplo Completo de Combinación de Anotaciones**

Este ejemplo muestra cómo combinar varias anotaciones de Lombok para una clase de entidad típica.

Java

import lombok.Data; //

import lombok.Builder; //

import lombok.NoArgsConstructor; //

import lombok.AllArgsConstructor; //

@Data // Combina Getter, Setter, ToString, EqualsAndHashCode, RequiredArgsConstructor

@NoArgsConstructor // Constructor sin argumentos

@AllArgsConstructor // Constructor con todos los argumentos

@Builder // Soporte para el patrón Builder

public class Persona { //

private String nombre; //

private int edad; //

private String ciudad; // Nuevo campo

public static void main(String[] args) {

// Uso del Builder

Persona personaBuilder = Persona.builder() // cite: 267

.nombre("Juan") // cite: 268

.edad(30) // cite: 269

.ciudad("Madrid")

.build(); // cite: 270

System.out.println("Persona (Builder): " + personaBuilder); //

// Uso del constructor con todos los argumentos

Persona personaAllArgs = new Persona("Maria", 25, "Barcelona");

System.out.println("Persona (AllArgsConstructor): " + personaAllArgs);

// Uso del constructor sin argumentos (requiere @NoArgsConstructor)

Persona personaNoArgs = new Persona();

personaNoArgs.setNombre("Pedro");

personaNoArgs.setEdad(40);

personaNoArgs.setCiudad("Sevilla");

System.out.println("Persona (NoArgsConstructor): " + personaNoArgs);

// Verificando equals() (generado por @Data)

Persona juan2 = Persona.builder().nombre("Juan").edad(30).ciudad("Madrid").build();

System.out.println("personaBuilder.equals(juan2): " + personaBuilder.equals(juan2));

}

}

**Salida:**

Persona (Builder): Persona(nombre=Juan, edad=30, ciudad=Madrid)

Persona (AllArgsConstructor): Persona(nombre=Maria, edad=25, ciudad=Barcelona)

Persona (NoArgsConstructor): Persona(nombre=Pedro, edad=40, ciudad=Sevilla)

personaBuilder.equals(juan2): true

**Otras Anotaciones Útiles de Lombok**

@NonNull

Genera verificaciones de nulidad para los parámetros de los métodos y constructores. Si un parámetro anotado con @NonNull es null, Lombok insertará un chequeo que lanzará un NullPointerException.

**Ejemplo:**

Java

import lombok.NonNull; //

public class Contacto {

private String nombre;

public void setNombre(@NonNull String nombre) { //

this.nombre = nombre;

}

public static void main(String[] args) {

Contacto contacto = new Contacto();

try {

contacto.setNombre(null); // Esto lanzará un NullPointerException

} catch (NullPointerException e) {

System.err.println("Error: " + e.getMessage());

}

contacto.setNombre("Ana");

System.out.println("Nombre establecido: " + contacto.nombre);

}

}

**Salida:**

Error: nombre is marked non-null but is null

Nombre establecido: Ana

@Value

Una alternativa a @Data para clases inmutables. A diferencia de @Data, @Value hace que todos los campos sean private final por defecto, genera solo *getters*, @ToString, @EqualsAndHashCode, y un constructor para todos los campos (@AllArgsConstructor). No genera *setters*.

**Ejemplo:**

Java

import lombok.Value; //

@Value //

public class Coordenada {

String latitud; // Se convierte en private final String latitud;

String longitud; // Se convierte en private final String longitud;

public static void main(String[] args) {

Coordenada c1 = new Coordenada("40.7128N", "74.0060W"); // Constructor generado

System.out.println(c1.getLatitud()); // Getter generado

System.out.println(c1); // toString() generado

// c1.setLatitud("..."); // Esto generaría un error de compilación, no hay setters

}

}

**Salida:**

40.7128N

Coordenada(latitud=40.7128N, longitud=74.0060W)

@SneakyThrows

Permite lanzar excepciones sin declararlas explícitamente en la firma del método (es decir, sin usar throws SomeException). Lombok inyecta el código para envolver la excepción en un RuntimeException o una excepción no chequeada, eliminando la necesidad de bloques try-catch o la declaración throws. **Debe usarse con precaución, ya que puede ocultar la propagación de excepciones y dificultar el manejo de errores.**

**Ejemplo:**

Java

import lombok.SneakyThrows; //

public class ServicioArchivo {

@SneakyThrows //

public void leerArchivo(String ruta) {

if (ruta == null || ruta.isEmpty()) {

throw new Exception("Ruta del archivo no puede ser nula o vacía."); //

}

System.out.println("Leyendo archivo de: " + ruta);

// Simular operación que puede lanzar una excepción chequeada

// Files.readAllBytes(Paths.get(ruta));

}

public static void main(String[] args) {

ServicioArchivo servicio = new ServicioArchivo();

try {

servicio.leerArchivo(null);

} catch (Exception e) {

System.err.println("Capturada la excepción: " + e.getMessage());

}

servicio.leerArchivo("documento.txt");

}

}

**Salida:**

Capturada la excepción: Ruta del archivo no puede ser nula o vacía.

Leyendo archivo de: documento.txt

Ejemplo de Uso de

@SuperBuilder

La anotación

@SuperBuilder en Lombok es una extensión de @Builder que proporciona soporte para la herencia de constructores en clases con relaciones de herencia. Permite crear

*builders* para clases base y derivadas que se pueden usar de manera fluida y segura en términos de tipo. Esto es particularmente útil cuando trabajas con jerarquías de clases y deseas mantener la funcionalidad de construcción de objetos de una manera sencilla y consistente.

A continuación, te muestro un ejemplo práctico de cómo usar

@SuperBuilder en una jerarquía de clases.

Clase Base

Persona

Java

import lombok.Getter; //

import lombok.experimental.SuperBuilder; //

@Getter //

@SuperBuilder //

public class Persona {

protected String nombre; //

protected int edad; //

}

Clase Derivada

Empleado

import lombok.Getter; //

import lombok.experimental.SuperBuilder; //

@Getter //

@SuperBuilder //

public class Empleado extends Persona { //

private String puesto; //

private double salario; //

}

Uso en el Código

Puedes usar los constructores generados por

@SuperBuilder para crear instancias de Persona y Empleado de manera fluida.

public class Main {

public static void main(String[] args) {

// Crear una instancia de Persona

Persona persona = Persona.builder() // cite: 344

.nombre("Juan") // cite: 345

.edad(30) // cite: 346

.build(); // cite: 347

System.out.println("Nombre Persona: " + persona.getNombre()); // cite: 348

System.out.println("Edad Persona: " + persona.getEdad()); // cite: 349

System.out.println("\n--- Creando instancia de Empleado ---");

// Crear una instancia de Empleado, incluyendo campos de Persona

Empleado empleado = Empleado.builder() //

.nombre("Ana") // Heredado de Persona

.edad(25) // Heredado de Persona

.puesto("Desarrolladora") // Campo de Empleado

.salario(50000.00) // Campo de Empleado

.build(); //

System.out.println("Nombre Empleado: " + empleado.getNombre()); //

System.out.println("Edad Empleado: " + empleado.getEdad()); //

System.out.println("Puesto Empleado: " + empleado.getPuesto()); //

System.out.println("Salario Empleado: " + empleado.getSalario()); //

}

}

**Salida:**

Nombre Persona: Juan

Edad Persona: 30

--- Creando instancia de Empleado ---

Nombre Empleado: Ana

Edad Empleado: 25

Puesto Empleado: Desarrolladora

Salario Empleado: 50000.0

Diferencias entre

@Builder y @SuperBuilder

* **@Builder**: Es adecuado para clases simples sin herencia o clases finales. No soporta la herencia de constructores.
* **@SuperBuilder**: Está diseñado específicamente para trabajar con jerarquías de clases. Soporta la herencia de constructores, permitiendo que las clases derivadas hereden los constructores de la clase base y añadan sus propios campos de manera fluida.

Ventajas de

@SuperBuilder

* **Facilita el manejo de jerarquías de clases**: Permite construir objetos de clases derivadas y base de manera fluida y consistente.
* **Reduce el código *boilerplate***: Automatiza la generación de constructores complejos en jerarquías de clases.
* **Mejora la legibilidad y mantenibilidad**: El uso de constructores fluidos y tipados hace que el código sea más fácil de leer y mantener.

Cómo Resolver el Problema de Ciclo Infinito con

@ToString(exclude = "campo")

Un problema común con @ToString ocurre en relaciones bidireccionales (por ejemplo, Empleado tiene un campo empresa, y Empresa tiene una Lista de Empleados). Si ambas clases tienen @ToString aplicado, al intentar imprimir una Empresa, esta intentaría imprimir sus Empleados, que a su vez intentarían imprimir la Empresa, creando un ciclo infinito y un StackOverflowError.

La instrucción

@ToString(exclude = "campo") de Lombok se usa para excluir uno o más campos de la representación de cadena generada automáticamente por la anotación @ToString.

**Ejemplo:**

Supongamos que tienes una clase Empleado con un campo empresa, y una clase Empresa que contendrá una lista de Empleados.

**Clase Empresa (con lista de empleados y @ToString)**

Java

import lombok.ToString;

import lombok.Getter;

import lombok.Setter;

import java.util.List;

@Getter

@Setter

@ToString

public class Empresa {

private String nombre;

private String rubro;

private List<EmpleadoConEmpresa> empleados; // Lista de empleados

public Empresa(String nombre, String rubro, List<EmpleadoConEmpresa> empleados) {

this.nombre = nombre;

this.rubro = rubro;

this.empleados = empleados;

}

}

**Clase EmpleadoConEmpresa (sin exclude)**

Java

import lombok.ToString;

import lombok.Getter;

import lombok.Setter;

@Getter

@Setter

@ToString // Si no excluyes 'empresa' aquí, habrá un ciclo.

public class EmpleadoConEmpresa {

private String nombre;

private int edad;

private String puesto;

private Empresa empresa; // Relación bidireccional

public EmpleadoConEmpresa(String nombre, int edad, String puesto, Empresa empresa) {

this.nombre = nombre;

this.edad = edad;

this.puesto = puesto;

this.empresa = empresa;

}

}

Si intentaras imprimir una instancia de Empresa que contiene una lista de EmpleadoConEmpresa, sin usar exclude, obtendrías un StackOverflowError.

Clase

Empleado con @ToString(exclude = "campo")

Para resolver este problema, excluimos el campo empresa de la representación toString() de Empleado.

Java

import lombok.ToString; //

import lombok.Getter;

import lombok.Setter;

import java.util.Arrays;

import java.util.List;

@Getter

@Setter

@ToString(exclude = "empresa") //

public class Empleado {

private String nombre; //

private int edad; //

private String puesto; //

private double salario; //

private Empresa empresa; //

public Empleado() { // Constructor sin argumentos para el main

}

public Empleado(String nombre, int edad, String puesto, double salario, Empresa empresa) {

this.nombre = nombre;

this.edad = edad;

this.puesto = puesto;

this.salario = salario;

this.empresa = empresa;

}

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

// Crear una instancia de Empresa (necesaria para el empleado)

Empresa techCorp = new Empresa("Tech Corp", "IT", null); // Inicialmente sin empleados

// Crear una instancia de Empleado

Empleado empleado = new Empleado(); //

empleado.setNombre("Juan"); //

empleado.setEdad(30); //

empleado.setPuesto("Desarrollador"); //

empleado.setSalario(50000.00); //

empleado.setEmpresa(techCorp); //

// Asignar el empleado a la empresa (para completar la bidireccionalidad si se necesitara)

techCorp.setEmpleados(Arrays.asList(empleado));

System.out.println(empleado.toString()); //

System.out.println(techCorp.toString()); // Imprimir la empresa también

}

}

// Clase Empresa simplificada para el ejemplo de ciclo infinito

@Getter

@Setter

@ToString(exclude = "empleados") // Excluir la lista de empleados para evitar ciclo

class EmpresaParaToString {

private String nombre;

private String rubro;

private List<Empleado> empleados;

public EmpresaParaToString(String nombre, String rubro, List<Empleado> empleados) {

this.nombre = nombre;

this.rubro = rubro;

this.empleados = empleados;

}

}

**Salida de empleado.toString():**

Empleado(nombre=Juan, edad=30, puesto=Desarrollador, salario=50000.0)

Como puedes ver, el campo

empresa ha sido excluido de la representación de cadena del Empleado.

**Salida de techCorp.toString() (usando EmpresaParaToString):**

EmpresaParaToString(nombre=Tech Corp, rubro=IT)

**Conclusión:** La instrucción @ToString(exclude = "campo") es útil cuando quieres personalizar la representación de cadena de tu clase generada por Lombok, excluyendo ciertos campos que no deseas incluir en la salida del método toString(). Esto puede ser útil para mejorar la legibilidad o para evitar exponer información sensible o irrelevante, y es crucial para prevenir

StackOverflowErrors en relaciones bidireccionales.

La Anotación

@Builder.Default

La anotación

@Builder.Default en Lombok se utiliza para especificar un valor predeterminado para un campo cuando se está utilizando el patrón de construcción (@Builder). Esto es necesario porque, al usar

@Builder, los valores de los campos se inicializan antes de que se aplique el constructor del *builder*, lo que significa que cualquier valor predeterminado que hayas asignado a los campos directamente en la declaración (ej. private int edad = 30;) no se aplicará a menos que uses @Builder.Default.

Ejemplo de Uso de

@Builder.Default

Imaginemos una clase Usuario donde queremos tener algunos valores predeterminados para ciertos campos.

**Clase Usuario sin @Builder.Default:**

Java

import lombok.Builder; //

import lombok.ToString; //

@Builder //

@ToString //

public class UsuarioSinDefault {

private String nombre; //

private int edad; // Por defecto 0 para int

private String email; // Por defecto null para String

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

UsuarioSinDefault usuario = UsuarioSinDefault.builder() //

.nombre("Carlos") //

.build(); //

System.out.println(usuario); //

}

}

**Salida:**

UsuarioSinDefault(nombre=Carlos, edad=0, email=null)

En este ejemplo, si no especificas el valor de

edad y email en el *builder*, sus valores serán 0 (para edad, ya que es un int) y null (para email, ya que es un String), porque los valores predeterminados asignados directamente al campo **no se aplican** cuando usas @Builder.

**Clase Usuario con @Builder.Default:**

Para asegurarte de que los valores predeterminados se apliquen incluso cuando usas

@Builder, puedes usar @Builder.Default:

Java

import lombok.Builder; //

import lombok.ToString; //

@Builder //

@ToString //

public class UsuarioConDefault {

private String nombre; //

@Builder.Default //

private int edad = 30; // Valor predeterminado para el builder

@Builder.Default //

private String email = "correo@ejemplo.com"; // Valor predeterminado para el builder

public static void main(String[] args) {

UsuarioConDefault usuario = UsuarioConDefault.builder() //

.nombre("Ana") //

.build(); //

System.out.println(usuario); //

}

}

**Salida:**

UsuarioConDefault(nombre=Ana, edad=30, email=correo@ejemplo.com)

Aquí, incluso si no especificas

edad y email en el *builder*, los valores predeterminados 30 y "correo@ejemplo.com" se aplican debido a @Builder.Default.

Cómo Funciona

@Builder.Default

* **Inicialización**: @Builder.Default asegura que el campo se inicialice con el valor predeterminado cuando se construye el objeto con el *builder*.
* **Construcción**: Si se proporciona un valor para ese campo en el *builder*, se utiliza ese valor; de lo contrario, se utiliza el valor predeterminado especificado.

Consideraciones

* **Inmutabilidad**: Funciona bien con patrones de construcción inmutables.
* **Legibilidad del Código**: Mejora la legibilidad al hacer explícitos los valores predeterminados dentro de la clase, en lugar de tener lógica adicional para establecer valores predeterminados.

**Conclusión:** La anotación @Builder.Default es una herramienta útil en Lombok para manejar valores predeterminados en campos cuando se utiliza el patrón de construcción (@Builder). Asegura que los valores predeterminados especificados se apliquen correctamente durante la construcción del objeto, lo que ayuda a mantener el código limpio y fácil de entender.

Ventajas de Usar Lombok

* **Reducción de Código *Boilerplate***: Lombok elimina la necesidad de escribir manualmente métodos comunes, reduciendo significativamente la cantidad de código.
* **Mejora de la Legibilidad**: Al eliminar el código repetitivo, el código restante es más fácil de leer y entender, permitiendo una visión más clara de la lógica de negocio.
* **Facilidad de Mantenimiento**: Los cambios en los métodos generados automáticamente están centralizados y se actualizan de manera consistente, lo que simplifica el mantenimiento.
* **Integración con IDEs**: Lombok es compatible con la mayoría de los IDEs populares, como IntelliJ IDEA, Eclipse y NetBeans, lo que facilita su uso en el flujo de trabajo de desarrollo.

Conclusión

Lombok es una herramienta poderosa para cualquier desarrollador Java que busque simplificar su código y mejorar su productividad. Al utilizar las anotaciones proporcionadas por Lombok, los desarrolladores pueden enfocarse más en la lógica de negocio y menos en el código

*boilerplate*. Su facilidad de uso y la compatibilidad con los principales IDEs lo hacen una opción atractiva para proyectos de todos los tamaños.