


entornos\_desarrollo\_2024-25 / UD6\_refactorizacion\_documentacion / apuntes  
/ ud6\_6\_javadoc.md 



**danielmartinan** Añadidas secciones UD6\_6 y UD6\_7

a898d93 · last month



658 lines (486 loc) · 21 KB

# UD6 - Optimización y Documentación

- [1. Documentación de código](#)
- [2. Introducción a Javadoc](#)
  - [2.1. Beneficios de usar Javadoc en proyectos Java](#)
  - [2.2. ¿Cuándo y cómo documentar con Javadoc?](#)
- [3. Sintaxis básica de Javadoc](#)
  - [3.1. Estructura de un comentario Javadoc](#)
  - [3.2. Etiquetas básicas en Javadoc](#)
    - [3.2.1. @param](#)
    - [3.2.2. @return](#)
    - [3.2.3. @throws o @exception](#)
- [4. Etiquetas avanzadas en Javadoc](#)
  - [4.1. @see – Referencias a otros elementos](#)
  - [4.2. @since – Indica la versión en la que se introdujo](#)
  - [4.3. @version – Especifica la versión actual](#)
  - [4.4. @author – Nombre del autor del código](#)
  - [4.5. @deprecated – Marcar elementos obsoletos](#)
  - [4.6. {@link } – Enlaces en línea dentro de una descripción](#)
  - [4.7. {@code } – Mostrar código formateado dentro de la documentación](#)
  - [4.8. {@literal } – Mostrar texto sin interpretar caracteres especiales](#)
- [5. Generación de documentación con Javadoc](#)
  - [5.1. Uso del comando javadoc](#)
  - [5.2. Opciones avanzadas de javadoc](#)

- [5.3. Estructura de la documentación generada](#)
- [5.4. Generación de Javadoc en entornos de desarrollo](#)
  - [5.4.1. En Eclipse](#)
  - [5.4.2. En IntelliJ IDEA](#)
  - [5.4.3. En VS Code](#)
- [6. Buenas prácticas en documentación con Javadoc](#)
  - [6.1. Escribir documentación clara y concisa](#)
  - [6.2. Documentar solo lo necesario](#)
  - [6.3. Evitar repetir información evidente](#)
  - [6.4. Usar formato y etiquetas correctamente](#)
  - [6.5. Mantener la documentación actualizada](#)
  - [6.6. Conclusión](#)

# 1. Documentación de código

[entornos\\_desarrollo\\_2024-25](#) / [UD6\\_refactorizacion\\_documentacion](#) / [apuntes](#)  
/ [ud6\\_6\\_javadoc.md](#)

↑ Top

Preview

Code

Blame

Raw



La forma más común de documentar código en Java es mediante comentarios, que son fragmentos de texto que se incluyen en el código fuente para explicar su funcionamiento. Estos fragmentos no son interpretados por el compilador y no afectan la ejecución del programa, pero son útiles para otros desarrolladores que necesitan entender el código.

En este contexto, Javadoc es una herramienta muy utilizada para generar documentación a partir de comentarios especiales en el código.

## 2. Introducción a Javadoc

Javadoc es una herramienta incluida en el **JDK de Java** que permite generar documentación en formato HTML a partir de comentarios especiales dentro del código fuente. Su principal objetivo es proporcionar una referencia clara y estructurada sobre clases, métodos y atributos de un programa, facilitando la comprensión y mantenimiento del código.

La documentación generada por Javadoc es ampliamente utilizada en proyectos profesionales, ya que permite describir la funcionalidad del código directamente en el archivo fuente, asegurando que la información esté siempre accesible y actualizada.

### 2.1. Beneficios de usar Javadoc en proyectos Java

El uso de Javadoc aporta diversas ventajas en el desarrollo de software:

- **Facilita el mantenimiento del código:** Permite comprender la funcionalidad de métodos y clases sin necesidad de analizar el código en detalle.
- **Mejora la colaboración en equipo:** Los desarrolladores pueden consultar la documentación sin depender de explicaciones externas.
- **Estandariza la documentación:** Sigue una estructura homogénea que facilita la generación automática de referencias.
- **Integración con entornos de desarrollo:** Herramientas como **Eclipse**, **IntelliJ IDEA** y **VS Code** pueden mostrar la documentación de Javadoc en tiempo real.

## 2.2. ¿Cuándo y cómo documentar con Javadoc?

El uso adecuado de Javadoc depende del contexto y la complejidad del código. Se recomienda documentar:

- **Clases y métodos públicos:** Especialmente en librerías o frameworks que serán utilizadas por otros desarrolladores.
- **Interfaces y abstracciones:** Para describir correctamente cómo deben ser implementadas.
- **Código con lógica compleja:** Cuando la implementación no es evidente con solo leer el código.

Ejemplo de comentario Javadoc en una clase:

```
/**
 * Representa una cuenta bancaria con operaciones básicas.
 * Permite depositar, retirar y consultar el saldo disponible.
 *
 * @author Juan Pérez
 * @version 1.0
 * @since 2024
 */
public class CuentaBancaria {
    private double saldo;

    /**
     * Deposita una cantidad en la cuenta.
     *
     * @param cantidad Monto a depositar, debe ser positivo.
     */
    public void depositar(double cantidad) {
        saldo += cantidad;
    }

    /**
     * Retira una cantidad de la cuenta si hay saldo suficiente.
```



```

*
* @param cantidad Monto a retirar.
* @throws IllegalArgumentException si el saldo es insuficiente.
*/
public void retirar(double cantidad) {
    if (cantidad > saldo) {
        throw new IllegalArgumentException("Saldo insuficiente");
    }
    saldo -= cantidad;
}

/**
 * Consulta el saldo actual de la cuenta.
 *
 * @return El saldo disponible.
 */
public double getSaldo() {
    return saldo;
}
}

```

En este ejemplo, Javadoc se usa para documentar la **clase**, sus **métodos** y los parámetros relevantes.

En la siguiente sección se abordará la sintaxis específica de Javadoc, incluyendo las etiquetas más utilizadas para mejorar la claridad de la documentación.

### 3. Sintaxis básica de Javadoc

Javadoc utiliza comentarios especiales dentro del código fuente para generar documentación estructurada. Estos comentarios deben colocarse antes de la declaración de una clase, método o atributo, y se distinguen de los comentarios regulares por su formato:

```

/**
 * Este es un comentario Javadoc.
 */

```



A diferencia de los comentarios estándar ( `//` y `/* */` ), los comentarios Javadoc comienzan con `/**` y terminan con `*/` . Dentro de ellos se pueden incluir descripciones, etiquetas especiales y formatos específicos.

#### 3.1. Estructura de un comentario Javadoc

Un comentario Javadoc suele seguir esta estructura:

1. **Descripción breve:** Explicación concisa del propósito de la clase o método.

2. **Etiquetas de metadatos:** Información adicional sobre parámetros, valores de retorno, excepciones y otros detalles.

Ejemplo básico en un método:

```
/**
 * Calcula el área de un rectángulo.
 *
 * @param base La base del rectángulo.
 * @param altura La altura del rectángulo.
 * @return El área calculada.
 */
public double calcularArea(double base, double altura) {
    return base * altura;
}
```



## 3.2. Etiquetas básicas en Javadoc

Las etiquetas Javadoc permiten estructurar mejor la documentación y añadir información específica sobre parámetros, valores de retorno y excepciones.

### 3.2.1. @param

Describe un parámetro del método. Se debe usar una línea por cada parámetro.

```
/**
 * Suma dos números enteros.
 *
 * @param a Primer número a sumar.
 * @param b Segundo número a sumar.
 * @return La suma de ambos números.
 */
public int sumar(int a, int b) {
    return a + b;
}
```



### 3.2.2. @return

Explica qué devuelve un método, si tiene un valor de retorno.

```
/**
 * Devuelve el nombre del usuario.
 *
 * @return Nombre del usuario.
 */
public String getNombre() {
```



```
    return nombre;
}
```

### 3.2.3. @throws o @exception

Indica que el método puede lanzar una excepción.

```
/**
 * Divide dos números.
 *
 * @param dividendo El número a dividir.
 * @param divisor El número por el que se divide.
 * @return El resultado de la división.
 * @throws ArithmeticException Si el divisor es cero.
 */
public double dividir(double dividendo, double divisor) throws ArithmeticEx
    if (divisor == 0) {
        throw new ArithmeticException("No se puede dividir por cero");
    }
    return dividendo / divisor;
}
```

Estos son los elementos esenciales para documentar correctamente métodos y clases en Javadoc. En la siguiente sección se profundizará en etiquetas más avanzadas como `@see`, `@since` y `@deprecated`.

## 4. Etiquetas avanzadas en Javadoc

Además de las etiquetas básicas como `@param`, `@return` y `@throws`, Javadoc permite el uso de etiquetas avanzadas para enriquecer la documentación con referencias, versiones, autores y otros metadatos.

### 4.1. @see – Referencias a otros elementos

Esta etiqueta permite enlazar a otras clases, métodos o documentación relacionada, facilitando la navegación dentro de la API.

```
/**
 * Calcula el área de un círculo.
 *
 * @param radio Radio del círculo.
 * @return Área del círculo.
 * @see Math#PI
 * @see <a href="https://es.wikipedia.org/wiki/C%C3%ADrculo">Círculo en Wik
 */
public double calcularArea(double radio) {
```

```
    return Math.PI * radio * radio;
}
```

También se puede utilizar para enlazar a documentación externa:

```
/**
 * Obtiene la fecha actual en formato ISO.
 *
 * @return Fecha en formato ISO 8601.
 * @see <a href="https://es.wikipedia.org/wiki/ISO_8601">Especificación ISO
 */
public String obtenerFechaISO() {
    return LocalDate.now().toString();
}
```



## 4.2. @since – Indica la versión en la que se introdujo

Es útil en proyectos que manejan versiones para indicar cuándo se agregó una clase o método.

```
/**
 * Clase que maneja operaciones de pago.
 *
 * @since 2.0
 */
public class ProcesadorPagos {
    // Código de la clase
}
```



La etiqueta `@since` permite a los desarrolladores saber cuándo se introdujo un elemento en el código, lo que facilita la migración y actualización de versiones.

## 4.3. @version – Especifica la versión actual

Se emplea para documentar la versión de una clase o interfaz.

```
/**
 * Representa un usuario en el sistema.
 *
 * @version 1.2
 */
public class Usuario {
    private String nombre;
}
```



## 4.4. @author – Nombre del autor del código

Permite indicar quién escribió la clase o método.

```
/**
 * Clase que representa un producto en un sistema de inventario.
 *
 * @author Juan Pérez, María Gómez
 */
public class Producto {
    private String nombre;
}
```



En caso de que el método o la clase sean modificados por varios autores, se pueden listar todos los nombres separados por comas:

## 4.5. @deprecated – Marcar elementos obsoletos

Indica que un método o clase ya no se recomienda y puede ser eliminado en futuras versiones. Se puede incluir una alternativa con @see .

```
/**
 * Obtiene el saldo de la cuenta.
 *
 * @deprecated Usar {@link #getSaldoActual()} en su lugar.
 * @return Saldo de la cuenta.
 */
@Deprecated
public double getSaldo() {
    return saldo;
}

/**
 * Obtiene el saldo actual de la cuenta.
 *
 * @return Saldo actualizado.
 */
public double getSaldoActual() {
    return saldo;
}
```



## 4.6. {@link } – Enlaces en línea dentro de una descripción

Permite hacer referencias a métodos, clases o documentación sin salir del contexto del párrafo.



```
/**
 * Calcula el interés de una cuenta.
 *
 * Usa la constante {@link Math#PI} en su cálculo.
 *
 * @param capital Capital inicial.
 * @return Interés calculado.
 */
public double calcularInteres(double capital) {
    return capital * Math.PI;
}
```



## 4.7. {@code } – Mostrar código formateado dentro de la documentación

Se usa para incluir fragmentos de código en línea sin que sean interpretados como texto normal.

```
/**
 * Devuelve el nombre del usuario en formato {@code String}.
 *
 * @return Nombre del usuario.
 */
public String getNombre() {
    return nombre;
}
```



## 4.8. {@literal } – Mostrar texto sin interpretar caracteres especiales

Si es necesario mostrar caracteres como `<`, `>`, `&`, `@`, sin que sean interpretados, se usa `{@literal }`.

```
/**
 * Representa una entidad HTML con la forma {@literal <tag>contenido</tag>}
 */
public class EtiquetaHTML {
}
```



Este conjunto de etiquetas avanzadas permite generar documentación más completa y organizada. En la siguiente sección se explicará cómo generar la documentación Javadoc en diferentes entornos y cómo personalizarla.

## 5. Generación de documentación con Javadoc

---

Una vez que el código ha sido correctamente documentado con comentarios Javadoc, se puede generar documentación en **HTML** utilizando la herramienta `javadoc`, incluida en el **JDK de Java**.

### 5.1. Uso del comando `javadoc`

Para generar la documentación, se utiliza el siguiente comando en la terminal o línea de comandos:

```
javadoc -d docs MiClase.java
```



**Explicación del comando:**

- `javadoc` → Ejecuta la herramienta.
- `-d docs` → Indica que la documentación se guardará en la carpeta `docs`.
- `MiClase.java` → Archivo fuente que se documentará.

Si se desea generar documentación para **varios archivos**, se pueden listar todos los archivos Java o usar `*.java`:

```
javadoc -d docs *.java
```



Esto generará una serie de archivos HTML dentro de la carpeta `docs`, incluyendo una página principal `index.html` con la documentación estructurada.

### 5.2. Opciones avanzadas de `javadoc`

Se pueden utilizar diversas opciones para personalizar la generación de la documentación.

```
javadoc -d docs -author -version -private *.java
```



**Parámetros usados:**

- `-author` → Muestra los autores indicados con `@author`.
- `-version` → Incluye la versión especificada con `@version`.
- `-private` → Documenta elementos privados además de los públicos y protegidos.

### 5.3. Estructura de la documentación generada

Después de ejecutar `javadoc` , se crea un conjunto de archivos HTML con la siguiente estructura:

```
docs/
|— index.html           # Página principal de la documentación
|— allclasses-index.html # Índice de todas las clases
|— package-summary.html # Resumen de cada paquete
|— clase1.html          # Documentación de la clase 1
|— clase2.html          # Documentación de la clase 2
|— ...
```



Para visualizar la documentación, basta con abrir `index.html` en un navegador.

## 5.4. Generación de Javadoc en entornos de desarrollo

### 5.4.1. En Eclipse

1. Ir a **Project** → **Generate Javadoc...**
2. Seleccionar los archivos o paquetes a documentar.
3. Configurar las opciones deseadas ( `@author` , `@version` , etc.).
4. Elegir una carpeta de destino y presionar **Finish**.

### 5.4.2. En IntelliJ IDEA

1. Ir a **Tools** → **Generate JavaDoc**.
2. Seleccionar las clases o paquetes.
3. Elegir la carpeta de destino.
4. Generar la documentación (Generate) y abrir `index.html` .

### 5.4.3. En VS Code

1. Instalar la extensión **Java Extension Pack**.
2. Usar el terminal integrado con el comando `javadoc -d docs *.java` .

## 6. Buenas prácticas en documentación con Javadoc

El uso adecuado de Javadoc no solo facilita la comprensión del código, sino que también mejora la mantenibilidad y colaboración dentro de un equipo de desarrollo. A continuación, se presentan una serie de buenas prácticas para documentar de manera efectiva.

### 6.1. Escribir documentación clara y concisa

Una documentación útil debe proporcionar información suficiente sin ser redundante.

## Ejemplo de una mala documentación:

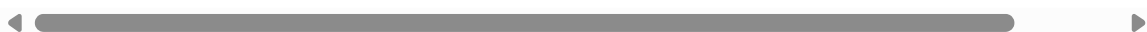
```
/**
 * Suma dos números.
 *
 * @param a El primer número.
 * @param b El segundo número.
 * @return La suma de a y b.
 */
public int sumar(int a, int b) {
    return a + b;
}
```



Esta documentación es innecesaria, ya que el método `sumar` es lo suficientemente descriptivo. Un comentario más útil podría incluir información adicional, como el comportamiento esperado o restricciones.

## Ejemplo mejorado:

```
/**
 * Suma dos números enteros. Si el resultado excede el valor máximo de un e
 * se producirá un desbordamiento sin generar una excepción.
 *
 * @param a Primer número a sumar.
 * @param b Segundo número a sumar.
 * @return Resultado de la suma.
 */
public int sumar(int a, int b) {
    return a + b;
}
```



## 6.2. Documentar solo lo necesario

No es necesario documentar métodos o clases que sean **triviales** o **autoexplicativos**. La regla general es documentar aquellos elementos que:

- Tienen una lógica compleja o no evidente.
- Forman parte de una API pública.
- Son utilizados por otros desarrolladores.

Por ejemplo, un simple método **getter** no necesita documentación:

```
// No es necesario documentar un simple getter
public String getNombre() {
```



```
    return nombre;
}
```

Sin embargo, si el getter tiene una funcionalidad especial, sí es recomendable documentarlo:

```
/**
 * Método para obtener el nombre en mayúsculas.
 *
 * @return Nombre en mayúsculas.
 */
public String getNombre() {
    return nombre.toUpperCase();
}
```



### 6.3. Evitar repetir información evidente

Si el nombre del método ya indica claramente su propósito, no es necesario repetirlo en la documentación.

Ejemplo innecesario:

```
/**
 * Este método cierra la conexión a la base de datos.
 */
public void cerrarConexion() {
    // Código para cerrar la conexión
}
```



En su lugar, se puede proporcionar información relevante, como qué sucede si se llama dos veces al método:

```
/**
 * Cierra la conexión a la base de datos si aún está abierta.
 * Si la conexión ya está cerrada, no realiza ninguna acción.
 */
public void cerrarConexion() {
    // Código para cerrar la conexión
}
```



### 6.4. Usar formato y etiquetas correctamente

Se recomienda usar etiquetas como `@param`, `@return` y `@throws` en el orden adecuado y con una descripción clara. El orden habitual es:

1. Descripción general del método o clase.

2. Etiquetas `@param` para los parámetros.
3. Etiqueta `@return` para el valor de retorno.
4. Etiqueta `@throws` para las excepciones lanzadas.
5. Otras etiquetas como `@see` , `@since` , `@version` , etc.

Ejemplo correcto:

```
/**
 * Busca un producto en el inventario.
 *
 * @param id Identificador único del producto.
 * @return El producto encontrado, o {@code null} si no existe.
 * @throws IllegalArgumentException Si el ID es negativo.
 * @see Producto
 * @since 1.0
 * @version 2.0
 */
public Producto buscarProducto(int id) {
    if (id < 0) {
        throw new IllegalArgumentException("El ID no puede ser negativo");
    }
    return inventario.get(id);
}
```

## 6.5. Mantener la documentación actualizada

Es fundamental que la documentación refleje fielmente el comportamiento actual del código. Un problema común es que el código cambia, pero la documentación queda obsoleta.

Ejemplo de documentación desactualizada:

```
/**
 * Divide dos números. Retorna -1 si el divisor es 0.
 *
 * @param a Dividendo.
 * @param b Divisor.
 * @return Resultado de la división o -1 si b es 0.
 */
public double dividir(double a, double b) {
    if (b == 0) {
        throw new ArithmeticException("División por cero no permitida");
    }
    return a / b;
}
```

El comentario indica que el método devuelve `-1` si el divisor es `0`, pero en realidad lanza una excepción.

Corrección:

```
/**
 * Divide dos números.
 *
 * @param a Dividendo.
 * @param b Divisor (no puede ser 0).
 * @return Resultado de la división.
 * @throws ArithmeticException Si el divisor es 0.
 */
public double dividir(double a, double b) {
    if (b == 0) {
        throw new ArithmeticException("División por cero no permitida");
    }
    return a / b;
}
```



## 6.6. Conclusión

Seguir estas buenas prácticas en Javadoc ayuda a mejorar la calidad del código y la colaboración entre desarrolladores. Una documentación clara, actualizada y bien estructurada facilita el mantenimiento del software y reduce la necesidad de explicaciones adicionales.