4.2. Documentación JavaDoc

1. Estilo consistente
2. Documentar ejemplos y casos de uso
3. No sobrecargar los método
4. Generar la documentación y revisarla

---

1. Estilo consistente

1.1. Analogía sobre ***estilo consistente***

"¿Qué pasaría si entramos en la Wikipedia y un artículo está escrito en primera persona ('Yo creo que Napoleón...'), otro solo tiene fotos, otro está en inglés y otro empieza por el final?"

- **Respuesta:** Sería inusable. Perdemos tiempo intentando entender *dónde* está la información.
- **Conclusión:** Un patrón consistente hace que el cerebro "ignore" el formato y se centre en el contenido. Queremos que leer el código de un compañero sea tan predecible como leer los ingredientes en una etiqueta de comida: siempre sabes dónde mirar.

1.2. Definición Técnica

Un patrón de documentación no es solo "usar JavaDoc", sino acordar un **contrato de estilo**. Se basa en tres pilares:

1. **Estructura Visual:** ¿En qué orden van las cosas? (Primero descripción, luego `@param`, luego `@return`).
2. **Tono y Tiempo Verbal:** ¿Hablamos al ordenador o al humano? ¿Usamos imperativo o infinitivo?
3. **Idioma:** ¿Español, Inglés o (el terror) Spanglish?

1.3. Ejemplo: "El Caos vs. El Patrón"

En el siguiente código, ejemplo clásico de trabajo en grupo donde cada uno ha documentado "a su manera".

❌ El escenario CAÓTICO (Sin patrón)

```
public class Calculadora {

    /**
     * Suma dos números.
     */
    public int sumar(int a, int b) { ... }

    /**
     * @param a: dividendo
     * @param b: divisor
     * Este método se encarga de dividir. Si b es 0 lanza error.
```

```
     * @return el resultado
     */
    public double dividir(int a, int b) { ... }

    /**
     * I am calculating the power of a number.
     * @return result
     */
    public double potencia(double base, double exp) { ... }
}
```

"¿Qué nos molesta de esto?"

- Uno es breve, otro mezcla descripción y tags.
- Uno está en inglés, otro en español.
- Uno describe la acción ("Suma"), otro describe al método ("Este método se encarga...").
- El orden de los tags es aleatorio.

☑ El escenario CONSISTENTE (Con patrón)

Así se vería cuando se aplica un patrón (por ejemplo: Tercera persona del presente, español, orden estricto).

```
public class Calculadora {

    /**
     * Calcula la suma de dos números enteros.
     *
     * @param a Primer sumando.
     * @param b Segundo sumando.
     * @return La suma de a y b.
     */
    public int sumar(int a, int b) { ... }

    /**
     * Calcula la división de dos números enteros.
     *
     * @param a El dividendo.
     * @param b El divisor.
     * @return El cociente de la división.
     * @throws ArithmeticException Si el divisor es 0.
     */
    public double dividir(int a, int b) { ... }

    /**
     * Calcula la potencia de un número.
     *
     * @param base La base de la potencia.
     * @param exp  El exponente al que se eleva la base.
     * @return El resultado de elevar la base al exponente.
     */
```

```
    public double potencia(double base, double exp) { ... }
}
```

1.4. Los estándares de documentación

En cada proyecto se dispondrá de una **"Hoja de Estilo"**. Ejemplo de reglas para un patrón consistente seria:

1. **Verbo inicial:** La primera frase siempre empieza con un verbo en 3ª persona del presente indicativo.
   - *Bien:* "Calcula...", "Obtiene...", "Valida..."
   - *Mal:* "Método para calcular...", "Voy a obtener...", "Calcular..."
2. **No a la obviedad redundante:**
   - *Mal:* `@param nombre El nombre.` (No aporta nada).
   - *Bien:* `@param nombre El nombre completo del usuario tal como aparece en el DNI.` (Aporta contexto).
3. **Orden Sagrado de Tags:**
   1. Descripción corta.
   2. Descripción larga (si hace falta).
   3. `@param`
   4. `@return`
   5. `@throws`
4. **Formato:** Siempre se deja una línea en blanco entre la descripción y los tags.

1.5. ¿Por qué es importante en "Entornos de Desarrollo"?

Tiene relevancia para las herramientas que usamos en el desarrollo:

- **El IDE (Visual Studio Code/Eclipse/IntelliJ/NetBeans):** Cuando pones el ratón encima de una función, el IDE muestra el pop-up de ayuda. Si no hay patrón, ese pop-up se ve horrible y desordenado.
- **Generación automática:** Al exportar el JavaDoc a HTML, si no hay consistencia, la web resultante parece rota y poco profesional.
- **Mantenimiento:** "Dentro de 6 meses, cuando tengamos que arreglar un bug en nuestro proyecto final, agradeceremos que todas las funciones os hablen en el mismo idioma".

Resumen

> "El código se compila para las máquinas, pero la documentación se escribe para las personas. Un patrón consistente es señal de respeto hacia el tiempo de vuestros compañeros (y el vuestro propio en el futuro)."

2. Documentar ejemplos y casos de uso

Documentar ejemplos es **vital** para que otros desarrolladores entiendan rápidamente cómo usar tu método sin tener que leer el código fuente.

Aunque Javadoc no tiene una etiqueta oficial llamada `@example` (como sí tienen otros lenguajes), la convención estándar es incluir los ejemplos dentro de la descripción principal utilizando etiquetas HTML y de

código.

Veremos las dos formas de hacerlo: la **clásica** (compatible con todas las versiones) y la **moderna** (Java 18+).

Opción 1: La forma clásica (Compatible con todo)

Se utilizan las etiquetas HTML `<pre>` (preformatted text) para mantener los saltos de línea y la indentación, junto con `{@code ...}` para que el código se vea con tipografía de código.

```
/**
 * Utilidades para manipulación de cadenas de texto.
 */
public class TextUtils {

    /**
     * Oculta todos los caracteres de una cadena excepto los últimos 4.
     * Es útil para mostrar números de tarjetas de crédito o identificadores
     * sensibles de forma segura en la interfaz de usuario.
     *
     * <h3>Ejemplos de uso:</h3>
     * <p>
     * A continuación se muestran algunos ejemplos de entrada y su salida
esperada:
     * </p>
     *
     * <pre>{@code
     * // Caso 1: Tarjeta de crédito estándar
     * TextUtils.maskSensitiveData("1234567812345678");
     * // Retorna: "************5678"
     *
     * // Caso 2: Cadena corta (menos de 4 caracteres)
     * TextUtils.maskSensitiveData("123");
     * // Retorna: "123"
     *
     * // Caso 3: Entrada nula
     * TextUtils.maskSensitiveData(null);
     * // Lanza: IllegalArgumentException
     * }</pre>
     *
     * @param input La cadena de texto que contiene información sensible.
     * @return La cadena enmascarada mostrando solo los últimos 4 caracteres.
     * @throws IllegalArgumentException Si el input es nulo.
     */
    public static String maskSensitiveData(String input) {
        if (input == null) {
            throw new IllegalArgumentException("El input no puede ser nulo");
        }
        if (input.length() <= 4) {
            return input;
        }
        String visiblePart = input.substring(input.length() - 4);
        String maskedPart = "*".repeat(input.length() - 4);
```

```
        return maskedPart + visiblePart;
    }
}
```

Puntos clave de este ejemplo:

1. `<h3>`**:** Se usa para crear un subtítulo visual en la documentación generada.
2. `<pre>{@code ... }`**:** Este bloque es el truco. `<pre>` respeta tus espacios y saltos de línea, y `{@code}` evita que tengas que escapar caracteres especiales (como `<` o `>`) y le da formato de fuente monoespaciada.
3. **Comentarios de salida:** Nota cómo uso `// Retorna: ...` para indicar explícitamente qué devuelve la función.

Opción 2: La forma moderna (Java 18 en adelante)

Este es nuestro caso, usamos Java 18 o superior, se introdujo la etiqueta `{@snippet ... }`. Es mucho más potente, permite resaltado de sintaxis (syntax highlighting) en el Javadoc generado y es más fácil de escribir que la combinación `<pre>{@code}`.

```
/**
 * Calcula el precio final aplicando un porcentaje de descuento.
 *
 * <p>Ejemplos de uso:</p>
 *
 * {@snippet :
 *   // Ejemplo 1: Descuento estándar del 20%
 *   double precio = PriceCalculator.aplicarDescuento(100.0, 20);
 *   System.out.println(precio); // Imprime: 80.0
 *
 *   // Ejemplo 2: Sin descuento (0%)
 *   double total = PriceCalculator.aplicarDescuento(50.0, 0);
 *   // Resultado: 50.0
 * }
 *
 * @param precio El precio original del producto.
 * @param porcentaje El descuento a aplicar (0 a 100).
 * @return El precio final calculado.
 */
public static double aplicarDescuento(double precio, int porcentaje) {
    // Lógica del método...
    return precio - (precio * porcentaje / 100.0);
}
```

Consejos para documentar ejemplos

1. **Muestra el "Happy Path":** El primer ejemplo debe ser el caso de uso más común y normal.
2. **Muestra casos borde (Edge Cases):** Es muy útil documentar qué pasa con entradas extrañas (null, listas vacías, números negativos).

3. **Usa comentarios de retorno:** Siempre coloca un comentario al final de la línea del ejemplo indicando el resultado (`// -> "resultado"` o `// returns true`).
4. **No compliques el código del ejemplo:** El ejemplo en el Javadoc no debe ser un algoritmo complejo, sino una llamada simple que ilustre cómo invocar al método.

3. No sobrecargar los métodos

Más comentarios no significan mejor código. De hecho, el exceso de documentación suele ser una señal de que el código no es lo suficientemente expresivo o de que el programador es inseguro.

Usaremos el concepto de **"Ruido vs. Señal"**.

- **Señal:** Información útil que no puedo deducir solo leyendo el nombre del método.
- **Ruido:** Información redundante que solo ocupa espacio y hay que mantener.

Aquí tienes tres enfoques o ejemplos comparativos para ilustrarlo:

3.1. El síndrome del "Capitán Obvio"

El error más común en estudiantes es: traducir el código a lenguaje natural línea por línea.

Atención: "*Si el comentario me dice exactamente lo mismo que el código, el comentario sobra. Es como ponerle un post-it a una puerta que diga 'Puerta'.*"

❌ MALA PRÁCTICA (Ruido)

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

¿Por qué está mal? Porque no aporta nada. Cualquier programador sabe qué hace `sumar(int a, int b)`.

☑ BUENA PRÁCTICA (Señal / Contexto)

En métodos tan triviales, a veces **no poner Javadoc** es la mejor opción (si el nombre es autoexplicativo). Pero si necesitamos documentar, debemos aportar contexto de negocio, no de sintaxis:

```
/**
 * Calcula el total acumulado de puntos de un usuario combinando
 * el saldo actual y los nuevos puntos ganados.
```

```
     *
     * @param saldoActual Puntos previos disponibles.
     * @param puntosGanados Puntos obtenidos en la última transacción.
     * @return El nuevo saldo total.
     */
    public int calcularTotalPuntos(int saldoActual, int puntosGanados) {
        return saldoActual + puntosGanados;
    }
```

3.2. Explicar el "Cómo" en lugar del "Qué"

El Javadoc es para el que **usa** el método, no para el que lo está programando. Al usuario del método no le importa si usaste un `for`, un `while` o una estructura de datos compleja interna.

Atención: "*Javadoc es el manual de instrucciones de la lavadora, no los planos del motor. Al usuario le interesa saber que el botón lava la ropa, no cómo giran los engranajes por dentro.*"

❌ MALA PRÁCTICA (Detalles de implementación)

```
/**
 * Crea un ArrayList vacío, recorre la lista de usuarios con un bucle for,
 * verifica con un if si la edad es mayor o igual a 18, y si es así,
 * lo añade a la lista auxiliar y finalmente devuelve esa lista.
 *
 * @param usuarios Lista de entrada
 * @return Lista filtrada
 */
public List<User> obtenerMayores(List<User> usuarios) {
   // ... código ...
}
```

¿Por qué está mal? Si mañana cambias el `ArrayList` por un `LinkedList` o usas `Streams` de Java 8, el comentario queda mentiroso y desactualizado.

☑ BUENA PRÁCTICA (Contrato y Comportamiento)

```
/**
 * Filtra y retorna únicamente los usuarios que son mayores de edad legal.
 * La lista devuelta es inmutable.
 *
 * @param usuarios Lista de usuarios a evaluar.
 * @return Una lista con los usuarios adultos, o una lista vacía si no hay
ninguno.
 */
public List<User> obtenerMayores(List<User> usuarios) {
   // ... código ...
}
```

Nota: Aquí explicamos cosas que el código no dice a simple vista (como que la lista devuelta es inmutable o qué pasa si no encuentra nadie).

3.3. Getters y Setters triviales

Este es un clásico relleno de líneas que ensucia la clase.

Diles: *"No documentéis Getters y Setters a menos que hagan algo raro. `getNombre` devuelve el nombre. Fin."*

❌ MALA PRÁCTICA (Sobrecarga visual)

```
/**
 * Obtiene el nombre.
 * @return el nombre.
 */
public String getNombre() { return nombre; }

/**
 * Establece el nombre.
 * @param nombre el nombre a establecer.
 */
public void setNombre(String nombre) { this.nombre = nombre; }
```

☑ BUENA PRÁCTICA (Solo lo necesario)

Solo se documenta si hay una regla de validación o un formato especial.

```
// Para getNombre() no hace falta nada, es obvio.

/**
 * Asigna el nombre del usuario.
 * Se eliminarán automáticamente los espacios en blanco al inicio y final.
 *
 * @param nombre El nombre (no puede ser nulo).
 * @throws IllegalArgumentException si el nombre está vacío.
 */
public void setNombre(String nombre) {
    if (nombre == null || nombre.trim().isEmpty()) throw new
IllegalArgumentException();
    this.nombre = nombre.trim();
}
```

Resumen

Antes de escribir un comentario Javadoc debemos hacernos estas dos preguntas:

1. **¿Estoy repitiendo el nombre del método?** (Si es sí -> Borra el comentario).
2. **¿Estoy explicando cómo funciona el código por dentro?** (Si es sí -> Mueve ese comentario dentro del método o bórralo).

El Javadoc debe responder a: **¿Para qué sirve esto? ¿Qué condiciones extrañas debo conocer? ¿Qué me va a devolver?**

4. Generar la documentación y revisarla

Vamos a ver como los comentarios se convierten en algo profesional.

Vamos a ver una guía paso a paso para VS Code y, lo más importante, una metodología para revisar esa documentación.

4.1. Generar Javadoc en Visual Studio Code

VS Code no trae un botón gigante de "Generar Javadoc" en la interfaz principal por defecto, pero gracias a la extensión **"Extension Pack for Java"** (sólo¿?), es muy fácil hacerlo.

Instrucciones:

1. **Abrir la Paleta de Comandos:** Presionar `Ctrl + Shift + P` (Windows/Linux) o `Cmd + Shift + P` (Mac).
2. **Buscar el comando:** Escribe `Java: Export Javadoc` y selecciona esa opción.
3. **Seleccionar qué documentar:** Se abrirá una pequeña interfaz. Asegúrate de marcar los archivos o paquetes que quieres documentar (normalmente todo `src`).
4. **Configurar la salida (Output):** Te pedirá dónde guardar la documentación.
    - *Recomendación:* Crea una carpeta nueva llamada `docs` o `javadoc` dentro de la carpeta del proyecto para no mezclarlo con el código.
5. **Clic en "Export" o "Finish":** VS Code ejecutará el comando por debajo. Observarás actividad en la terminal integrada.
6. **¡Ábrelo!** Iremos a la carpeta creada, buscaremos el archivo `index.html` y lo ábriremos con el navegador web (Chrome, Firefox, etc.). Esa es la página principal.

El comando Java: Export Javadoc pertenece a una extensión adicional muy popular llamada **"Javadoc Tools"**.

Podemos hacer dos cosas:

Opción A: La vía "Fácil y Visual" (Instalar la extensión)

Si quieres mantenerte dentro de la interfaz gráfica de VS Code.

1. Ir a la pestaña de **Extensiones** (el icono de los cubos a la izquierda).
2. Buscar **"Javadoc Tools"** (creada por *Madhav Dhingra*).

3. Darle a **Instalar**.
4. Ahora sí, al pulsar `Ctrl + Shift + P`, aparecerá el comando `Javadoc Tools: Export Javadoc`.

Opción B: La vía "Profesional" (Terminal) 👍 *Recomendada*

Depender de un botón mágico a veces es peligroso; saber invocar al compilador real es poder.

VS Code no es más que una carcasa bonita y que la herramienta real (`javadoc`) ya la tenemos instalada junto con Java.

Atencion: *No instalaremos el plugin, vamos a hacerlo como los profesionales de verdad: usando la terminal.*"

Instrucciones:

1. Abrid la terminal en VS Code (`Ctrl + ñ` o Menú *Terminal* > *New Terminal*).
2. Escribid el siguiente comando (ajustando los nombres de carpeta si es necesario):

```
javadoc -d docs -sourcepath src -subpackages miPaquete
```

(Si tienemos todo el código suelto en `src` sin paquetes, simplemente puedemos hacer):

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

Desglose del comando para entienderlo:

- `javadoc`: "Invoco a la herramienta de documentación de Java".
- `-d docs`: "Quiero que el **D**estino (destination) sea una carpeta llamada `docs`" (la creará sola).
- `src/*.java`: "Coge todos los archivos Java que estén dentro de la carpeta `src`".

Ventajas de hacerlo por terminal (Opción B)

1. **Es universal:** Funciona en VS Code, IntelliJ, Eclipse o en un servidor sin pantalla.
2. **Aprendemos qué pasa por debajo:** Entiendemos que `javadoc.exe` es un programa independiente.
3. **Menos problemas:** A veces la extensión falla si la configuración del proyecto es un poco rara; el comando de terminal suele ser más transparente con los errores.

Recomendación: Al usar este **Opción B** nos dará una sensación de control mucho mayor ("¡He escrito un comando y ha creado una web!") que simplemente instalar otro plugin.

4.2. Cómo revisar la documentación (Auditoría de Calidad)

Generar la documentación no es el final, es el principio de la revisión. Muchos alumnos generan el HTML y lo entregan sin mirarlo. ERROR.

Vamos a hacer una **"Auditoría de Usabilidad"** siguiendo estos 3 pasos o "Tests":

4.2.1. El Test Visual (Formato)

El objetivo es detectar si rompieron el HTML o el formato.

- **¿Se ven bien los bloques de código?**
  - *Mal:* El código de ejemplo sale todo en una sola línea. (Solución: Faltó la etiqueta `<pre>`).
  - *Bien:* El código tiene indentación y saltos de línea.
- **¿Se ven bien las listas?**
  - *Mal:* Los pasos 1, 2 y 3 salen seguidos en un párrafo texto plano.
  - *Bien:* Aparecen como una lista ordenada (usaron `<ul>` o `<ol>`).
- **Caracteres extraños:** ¿Aparecen símbolos raros en las tildes o ñ? (Problema de encoding/charset, aunque hoy día es menos común, pasa).

4.2.2. El Test del "Desconocido" (Contenido)

Intercambia tu documentación (solo el HTML generado) con un compañero, o que ellos mismos intenten leerla imaginando que **no tienen acceso al código fuente**.

Deben responder a estas preguntas mirando **solo** la web generada:

- "*¿Sé qué devuelve este método si le paso un `null`?*" (Si la respuesta es "No sé", falta documentar la excepción o el comportamiento).
- "*¿Entiendo para qué sirve este método sin tener que adivinarlo por el nombre?*"
- "*¿El ejemplo de uso que pusieron se entiende o es confuso?*"

4.2.3. El Test del "Capitán Obvio" (Limpieza)

Revisad vuestra propia documentación buscando redundancias.

- Si veis: `getEdad()` -> "*Obtiene la edad.*" -> **¡Error!** Eso ensucia la documentación.
- Si veis parámetros sin descripción (`@param a`) -> **¡Error!** Si no vas a explicar qué es `a`, mejor cambia el nombre de la variable a `edadUsuario` y a veces ni hace falta el tag `@param`.

Un truco "Pro" para VS Code: La vista previa rápida

Antes de generar todo el sitio web, podemos revisar los Javadocs en tiempo real mientras codificamos.

- **Acción:** Poner el ratón encima (hover) del nombre del método que acabáis de documentar.
- **Resultado:** VS Code muestra una ventana emergente renderizando el Javadoc.
- **Utilidad:** Es perfecto para ver si se han equivocado con las etiquetas HTML (`<b>`, `<pre>`, etc.) sin tener que exportar todo el proyecto cada vez.

Resumen:

> **Checklist de entrega de Javadoc:**
>
> 1. ☑ **Generar:** Usar el comando `javadoc -d docs src/*.java` en una terminal.
> 2. ☑ **Abrir:** Ejecutar `index.html` en el navegador.
> 3. ☑ **Verificar formato:** ¿Los ejemplos de código se leen bien o están desalineados?
> 4. ☑ **Verificar utilidad:** ¿Si oculto el código, la documentación me explica cómo usar la clase?
> 5. ☑ **Limpieza:** ¿He borrado los comentarios redundantes ("Obtiene X")?