**Documento explicativo**

1. **Introducción**
   * ¿Qué es Java y por qué es importante conocer sus novedades?
   * Breve mención de la evolución de Java en los últimos años.
2. **Novedades de Java 13**
   * **Text Blocks (JEP 355)** → Bloques de texto multilínea.
   * **Switch Expressions (JEP 354)** → Mejoras en la sintaxis de switch.
   * **Reimplementación de Socket API (JEP 353)** → Mejoras en el rendimiento.
3. **Novedades de Java 14**
   * **Records (JEP 359)** → Clases inmutables con menos código.
   * **Pattern Matching para instanceof (JEP 305)** → Reducción de código repetitivo.
   * **Nuevas excepciones NullPointerException con detalles (JEP 358)**.
   * **Switch Expressions mejoradas (JEP 361)**.
4. **Novedades de Java 15**
   * **Text Blocks (finalizado) (JEP 378)**.
   * **Sellado de clases (Sealed Classes, JEP 360)** → Restricción de herencia.
   * **Foreign-Memory Access API (JEP 370)** → Acceso seguro a memoria externa.
   * **Deprecación de Nashorn JavaScript Engine (JEP 372)**.
5. **Comparación con versiones anteriores**
   * ¿Qué mejoran estas novedades?
   * ¿Qué limitaciones solucionan?
   * ¿Siguen siendo compatibles con código anterior?
6. **Conclusión**
   * Impacto de estas mejoras en el desarrollo actual.
   * Recomendaciones para usarlas.

**Novedades en Java 13, 14 y 15**

**Introducción**

En este documento, voy a analizar las novedades más importantes introducidas en Java 13, 14 y 15, explicando sus ventajas, posibles desventajas y diferencias con versiones anteriores.

**Novedades en Java 13**

**1. Bloques de texto (Text Blocks - JEP 355)**

Antes de Java 13, escribir cadenas multilínea en Java era un dolor de cabeza. Había que concatenar varias líneas con \n o usar +, lo que hacía el código difícil de leer.

🔹 **¿Qué soluciona?**  
Ahora se pueden definir bloques de texto con triple comillas """, haciendo que el código sea más limpio y fácil de leer.

**Ejemplo en Java 12 o anterior:**

String json = "{\n" +

" \"nombre\": \"Roberto\",\n" +

" \"edad\": 20\n" +

"}";

**Ejemplo con Text Blocks en Java 13:**

String json = """

{

"nombre": "Roberto",

"edad": 20

}

""";

🔹 **Ventajas:**

* Código más limpio y legible.
* No hace falta usar \n para saltos de línea.

🔹 **Posibles desventajas:**

* Solo disponible desde Java 13.

**2. Mejoras en switch (Switch Expressions - JEP 354)**

El switch tradicional en Java era verboso y propenso a errores por los break. Java 13 introduce una sintaxis más corta y clara.

**Antes de Java 13:**

int dia = 3;

String nombreDia;

switch (dia) {

case 1: nombreDia = "Lunes"; break;

case 2: nombreDia = "Martes"; break;

case 3: nombreDia = "Miércoles"; break;

default: nombreDia = "Desconocido";

}

**En Java 13:**

String nombreDia = switch (dia) {

case 1 -> "Lunes";

case 2 -> "Martes";

case 3 -> "Miércoles";

default -> "Desconocido";

};

🔹 **Ventajas:**

* Más corto y fácil de leer.
* Se evitan errores por olvidar break.

**Novedades en Java 14**

**3. Registros (Records - JEP 359)**

En Java, cuando queremos crear una clase solo para almacenar datos, hay que escribir mucho código con getters, setters y toString(). Los **Records** solucionan esto.

**Antes de Java 14:**

class Persona {

private final String nombre;

private final int edad;

public Persona(String nombre, int edad) {

this.nombre = nombre;

this.edad = edad;

}

public String getNombre() { return nombre; }

public int getEdad() { return edad; }

}

**Con Records en Java 14:**

record Persona(String nombre, int edad) {}

🔹 **Ventajas:**

* Menos código para definir clases de datos.
* equals(), hashCode() y toString() generados automáticamente.

🔹 **Desventajas:**

* Son inmutables, no se pueden modificar sus valores.

**4. Pattern Matching para instanceof (JEP 305)**

Antes, cada vez que usábamos instanceof, había que hacer un casteo manual. Ahora, esto es más simple.

**Antes de Java 14:**

if (obj instanceof String) {

String str = (String) obj;

System.out.println(str.length());

}

**Con Pattern Matching en Java 14:**

if (obj instanceof String str) {

System.out.println(str.length());

}

🔹 **Ventajas:**

* Menos código y más seguridad.

**5. Excepciones NullPointerException mejoradas (JEP 358)**

En versiones anteriores de Java, cuando ocurría un **NullPointerException (NPE)**, el mensaje de error solo indicaba que se había producido una excepción debido a un intento de acceder a un objeto null, sin especificar qué variable causó el problema. Esto dificultaba la depuración, ya que los desarrolladores debían inspeccionar manualmente el código para encontrar el origen del error.

Por ejemplo, antes de Java 14, si se ejecutaba el siguiente código:

String nombre = null;

int longitud = nombre.length(); // Esto lanza un NullPointerException

El mensaje de error simplemente sería algo como:

Exception in thread "main" java.lang.NullPointerException

Este mensaje no proporcionaba detalles sobre la variable que causó la excepción.

**Mejora en Java 14**

Con la introducción de la JEP 358, los mensajes de NullPointerException ahora incluyen información específica sobre qué variable o expresión causó el error.

Por ejemplo, con el mismo código anterior, el nuevo mensaje de error sería:

Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()" because "nombre" is null.

**Ventajas:**

* Proporciona más contexto en el mensaje de error, lo que facilita la identificación de la causa del problema.
* Mejora la depuración, ahorrando tiempo y esfuerzo en la resolución de errores.

**¿Cómo funciona?**

Ahora, cuando ocurre un **NullPointerException**, Java analiza detalladamente la expresión que provocó la excepción e identifica la variable que tenía el valor null. Este análisis detallado ayuda a los desarrolladores a localizar rápidamente el error sin necesidad de revisar el código línea por línea.

**Casos de uso**

🔹 Identificación rápida de errores relacionados con objetos null.  
🔹 Reducción del tiempo de depuración en aplicaciones grandes o complejas.

**Novedades en Java 15**

**6. Clases selladas (Sealed Classes - JEP 360)**

Con las **Clases Selladas**, podemos restringir qué clases pueden heredar de una clase base.

**Ejemplo:**

sealed class Figura permits Circulo, Rectangulo {}

final class Circulo extends Figura {}

final class Rectangulo extends Figura {}

🔹 **Ventajas:**

* Más control sobre la herencia.

🔹 **Desventajas:**

* Solo útil en escenarios específicos.

**7. Acceso a memoria externa (Foreign-Memory API - JEP 370)**

Antes de Java 15, acceder a memoria fuera del heap (la memoria gestionada por el recolector de basura de Java) era complicado y poco seguro, generalmente usando la clase Unsafe. Ahora, Java introduce una API experimental para acceder a memoria externa de manera eficiente y segura.

**🔹 ¿Qué soluciona?**

Facilita el acceso a memoria fuera del heap sin depender de Unsafe, permitiendo mejorar el rendimiento en ciertas aplicaciones.

**🔹 Ejemplo de uso**

Antes de Java 15 (con Unsafe):

Unsafe unsafe = getUnsafe();

long address = unsafe.allocateMemory(100);

unsafe.putInt(address, 42);

int value = unsafe.getInt(address);

unsafe.freeMemory(address);

Con Foreign-Memory Access API en Java 15:

try (MemorySegment segment = MemorySegment.allocateNative(100)) {

MemoryAccess.setIntAtOffset(segment, 0, 42);

int value = MemoryAccess.getIntAtOffset(segment, 0);

}

**🔹 Ventajas**

- Más seguro que Unsafe.

- Mayor eficiencia en aplicaciones que manejan grandes volúmenes de datos en memoria.

**🔹 Uso típico**

Bases de datos en memoria.

Interacción con hardware o sistemas nativos.

**Conclusión**

Después de ver las novedades de Java 13, 14 y 15, está claro que el lenguaje sigue evolucionando para hacer el código más limpio, seguro y eficiente. Con cosas como **Text Blocks** y **Pattern Matching**, escribir en Java es mucho más cómodo, evitando código repetitivo y haciéndolo más legible. Otras mejoras, como los **Records**, ahorran tiempo al definir clases de datos, mientras que las **Sealed Classes** y la **Foreign-Memory Access API** abren nuevas posibilidades en cuanto a control y rendimiento.

Estas actualizaciones no solo facilitan el desarrollo, sino que también mejoran la depuración y el rendimiento de las aplicaciones. Aunque algunas de estas funciones todavía son experimentales, merece la pena conocerlas y empezar a usarlas en proyectos para aprovechar sus ventajas.

Java sigue mejorando, y estar al día con sus cambios es clave para desarrollar mejor y más rápido.