**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, vamos 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:**

java

String json = "{\n" +

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

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

"}";

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

java

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:**

java

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:**

java

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:**

java

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:**

java

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:**

java

if (obj instanceof String) {

String str = (String) obj;

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

}

**Con Pattern Matching en Java 14:**

java

if (obj instanceof String str) {

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

}

🔹 **Ventajas:**

* Menos código y más seguridad.

**2. 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. Esto dejaba a los desarrolladores sin información sobre qué variable específica causó el problema, lo que dificultaba el proceso de depuración.

Por ejemplo, antes de Java 14, si tenías 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

No proporcionaba detalles sobre qué variable causó el error. Esto era frustrante, especialmente en aplicaciones grandes con muchas variables.

Mejora en Java 14: Con la introducción de la mejora en JEP 358, ahora los mensajes de NullPointerException incluyen detalles más específicos sobre qué variable o expresión causó el problema. Esto hace que sea mucho más fácil rastrear y solucionar los errores.

Ahora, con el mismo código, el mensaje de error podría ser algo como:

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: Cuando se produce un NullPointerException, Java ahora captura la variable o expresión exacta que causó el error y lo incluye en el mensaje de la excepción. Esto se realiza mediante un mecanismo de análisis más detallado que verifica qué objeto era null en el momento de la excepción.

Uso típico:

* Identificación rápida de errores relacionados con objetos null, sin tener que inspeccionar el código línea por línea.

**Novedades en Java 15**

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

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

**Ejemplo:**

java

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.

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

Esta API experimental introducida en Java 15 (y en versiones posteriores seguirá siendo mejorada) permite a los desarrolladores interactuar directamente con memoria fuera del *heap* (memoria gestionada por el recolector de basura de Java) de manera segura y eficiente. Para entenderlo mejor, es importante conocer algunos conceptos clave:

* **Memoria fuera del heap**: A veces, las aplicaciones necesitan acceder a regiones de memoria que no son administradas por el recolector de basura (como memoria de dispositivos o buffers nativos). Java, tradicionalmente, no permite el acceso directo a esta memoria desde el código Java, ya que las interacciones con la memoria fuera del heap requieren un manejo cuidadoso para evitar errores de seguridad y corrupción de datos.
* **Clase Unsafe**: Para interactuar con memoria fuera del heap, Java permitía usar la clase Unsafe, que proporcionaba métodos de bajo nivel para acceder a memoria directa. Sin embargo, esta clase tenía varios inconvenientes: no estaba documentada, no era segura de usar y tenía limitaciones de compatibilidad entre plataformas.
* **Foreign-Memory Access API**: Con la introducción de esta API experimental, Java ahora ofrece una manera más segura de interactuar con memoria externa mediante clases específicas que gestionan este tipo de operaciones. La API permite a los desarrolladores acceder y manipular directamente bloques de memoria fuera del heap, sin tener que recurrir a la clase Unsafe. Esto mejora la seguridad del código y facilita la interoperabilidad con aplicaciones de bajo nivel (como aquellas que interactúan con C, bibliotecas nativas, o bases de datos en memoria).

**Ventajas**:

* + Mejora el rendimiento y la seguridad al permitir el acceso a memoria externa sin los peligros asociados con Unsafe.
  + Aumenta la eficiencia en aplicaciones que requieren manejar grandes cantidades de datos en memoria (por ejemplo, aplicaciones de procesamiento de datos en tiempo real).

**Uso típico**:

* + Trabajar con bases de datos en memoria.
  + Implementar bibliotecas de interacción con hardware o sistemas operativos de bajo nivel.

**Conclusión**

Las versiones 13, 14 y 15 de Java han traído mejoras importantes en la sintaxis, optimización del código y nuevas características que hacen el lenguaje más moderno y eficiente. Destacan los **Text Blocks**, los **Records**, el **Pattern Matching** y las **Sealed Classes**, que ayudan a escribir código más limpio y seguro. Aunque algunas de estas novedades solo estarán disponibles en versiones recientes, merece la pena conocerlas para aprovechar las mejoras en el desarrollo de aplicaciones.