Hoy en día Java es un lenguaje mucho más conciso y atractivo de lo que era hace un tiempo atrás

Java

EL CAMBIO ES LA ÚNICA CONSTANTE

JORDY ANDRÉS RODRÍGUEZ ARANGO

"No se trata de escribir código, se trata de mejorar procesos"

4 SOFTWARE DEVELOPERS

Índice

Java, el cambio es la única constante

Con lenguajes que cada vez toman más popularidad, como es el caso de Python cuya sintaxis es más concisa y simple, muchos desarrolladores asumen que Java es un lenguaje que se ha quedado un poco en el pasado. Por eso, la invitación es a que descubras a través de este material, la transformación que ha tenido Java desde hace algunos años.

Jordy A. Rodríguez Arango

Acerca del autor

Hola soy Jordy Rodríguez CEO y fundador de 4SoftwareDevelopers, CTO y Co-Fundador de HUSO Group SAS.

Soy un desarrollador de software con 10 años de experiencia en todo ese tiempo he tenido la oportunidad de trabajar en muchísimos proyectos en muchos de ellos he utilizado lenguajes y herramientas como Java, TypeScript, Spring Framework, Hibernate, NestJS, Angular, React Native, Python y NodeJS.

También he liderado múltiples proyectos en distintas organizaciones. Me considero a mi mismo como un apasionado por la academia, trato de vivir día a día, en busca de nuevos retos y tecnologías informáticas.

Contacto

Retroalimentación:

Creo firmemente en la mejora continua, la cual obedece al principio de la constante retroalimentación, por eso, de antemano agradezco si usted desea apoyar mi proceso de mejora continua con cualquier comentario acerca de este título, solo debe enviar un correo a ebooks@4softwaredevelopers.com, no olvide mencionar en el asunto el título del libro.

Piratería:

Si encuentras copias ilegales de esta obra no dudes en denunciarlas para ello puedes enviar un correo a jordy.rodriguez@4softwaredevelopers.com, no olvides incluir en el cuerpo del correo un enlace al sitio donde se encuentra la copia ilegal.

Redes Sociales

Puede tener un contacto más directo conmigo en Twitter.

Introducción

Java es un lenguaje de propósito general, concurrente y orientado a objetos que apareció en el año 1996 y que de acuerdo con Oracle, es utilizado hoy en día por más de 3 billones de dispositivos alrededor del mundo.

Resulta curioso pensar que para la época que apareció Java, se mostró como un lenguaje mucho más conciso y simple en comparación a lenguajes como C o C++. Su creador James Gosling y todo su equipo, tenían como propósito crear un lenguaje que pudiera ser ejecutado en cualquier entorno escribiendo el código solo una vez, este principio se conoce como WORA (**W**rite **O**nce **R**un **A**nywhere) por sus siglas en inglés, y ciertamente lo lograron.

Java con el pasar del tiempo se convirtió en el lenguaje más popular del mundo, tomando ventaja sobre lenguajes como C, C++ y PHP. Sin embargo, llegó un momento en el cual los cambios en el lenguaje se vieron un poco estancados. Esto generó que otras plataformas como Python y NodeJS tomarán mucha más fuerza (además de que también son excelentes).

La realidad que he podido ver de primera mano, es que muchos desarrolladores Java no conocen el gran cambio que ha tenido el lenguaje desde hace apróximadamente cinco años, al punto que la creencia general es que Java es un lenguaje en el que realizar una tarea sencilla requiere de mucho código, sobre todo si lo comparamos con sus más grandes rivales hoy en día. Mi intención con este material es mostrarles que esto ha cambiado y que Java

ahora es un lenguaje mucho más conciso y atractivo si se quiere decir que su versión de hace algunos años atrás.

Este libro es el lugar correcto, ya sea que conoces o no el lenguaje. Con este material espero que te sientas atraído por Java, te animes a probarlo y si ya lo conoces, que puedas actualizar tus conocimientos.

A lo largo de este libro, responderemos incógnitas que tu como informático, independiente que conozcas o no el lenguaje, puedas hacerte en el transcurso de tu carrera y veremos la evolución de Java como lenguaje desde un punto de vista técnico con algunos ejemplos que podrás recrear en cualquier IDE (Integrated Development Environment). Recomendamos Eclipse, pero eres libre de escoger el IDE de tu preferencia.

¿Qué puedo hacer con Java?

Puede que no lo creas pero esta es una pregunta que me hacen mucho, no solo personas que recién inician a aprender sino también programadores experimentados. Básicamente porque existe una creencia un tanto extraña de que Java es un lenguaje para la creación de aplicaciones de escritorio o cliente-servidor. Lo cierto es que Java es un lenguaje muy versátil con el que puedes hacer lo que quieras, aplicaciones de escritorio, herramientas de líneas de comando, aplicaciones web, servicios web, tareas de ejecución automática, etc.

Java es un lenguaje cuyas posibilidades son infinitas. Incluso se puede usar para IoT o Machine Learning, aunque en este último, Python es probablemente la mejor opción. Depende de ti lo que quieras hacer con este poderoso lenguaje, aunque en el mundo de hoy su uso está más ligado a aplicaciones y servicios web.

Como te habrás dado cuenta, Java es un lenguaje muy poderoso que puedes usar casi que en cualquier contexto (más adelante analizaremos si vale la pena usar un framework que complemente el desarrollo sobre este lenguaje).

¿Java es un lenguaje anticuado?

Si bien Java apareció por primera vez en 1996, no podemos decir que es un lenguaje anticuado. Realmente en la actualidad, el equipo de desarrollo de Java está enfocado en hacer de Java un lenguaje mucho más conciso y moderno. Ya quedó en el pasado aquel ciclo de actualizaciones algo lento al que estábamos acostumbrados.

Todo cambió con la llegada de Java 8 y para demostrarlo, solo basta con mirar un pequeño ejemplo de código en el cual se filtra una lista y se obtiene un resultado determinado

Filtro de una lista antes de Java 8

```
public class AntesDeJava8{

 public static void main(String[] args) {
   List<String> lines =
   Arrays.asList("spring", "node", "4sd");

   List<String> result = filter(lines, "4sd");
   for (String temp : result) {
     System.out.println(temp);
     //output : spring, node
   }
 }


 private static List<String> filter(List<String> lines,
                                    String filter) {
    List<String> result = new ArrayList<>();
    for (String line : lines) {
       if (!"4sd".equals(line)) {
         result.add(line);
       }
       return result;
 }

}
```

A partir de Java 8

```
public class Java8{

 public static void main(String[] args) {
   List<String> lines =
   Arrays.asList("spring", "node", "4sd");

   List<String> result = lines.stream()
               .filter(line -> !"4sd".equals(line))
               .collect(Collectors.toList());

   result.forEach(System.out::println);
   //output : spring, node
 }

}
```

Si analizamos las imágenes anteriores, nos damos cuenta que a pesar de ser el mismo lenguaje de programación del que estamos hablando y de haber solo una versión de diferencia entre una imagen y la otra, el cambio es bastante significativo.

Java 8 representa para mí una nueva forma de programar en Java, además del comienzo de una nueva etapa en el lenguaje, una etapa en la que tendremos un lenguaje más conciso. Es de resaltar el tremendo esfuerzo del equipo de desarrollo de Java por mejorar el lenguaje, cabe aclarar que la forma de programar antes de Java 8 no es errónea en lo más mínimo, simplemente es más extensa de lo que conocemos a partir de Java 8, de ahí en adelante todo mejora y no solo para el ciclo de las actualizaciones que es mucho más rápido sino que cada versión llega con cambios que nos ayudan a ser mucho más productivos. Explicaré esto un poco más adelante.

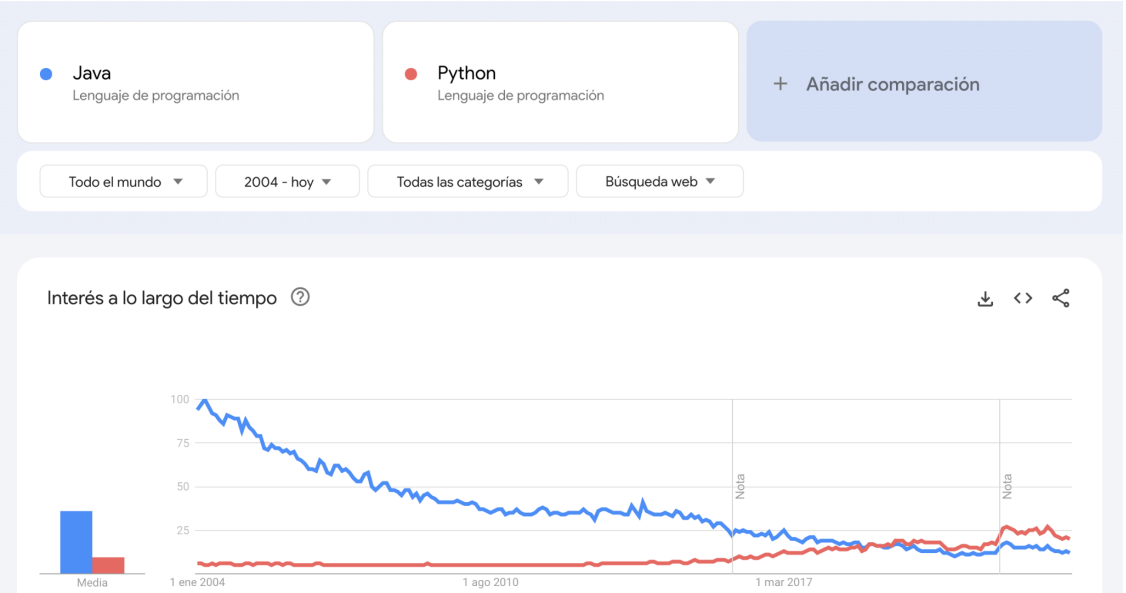
¿Qué tan popular es Java actualmente?

Tendencias de Google

Para dar una respuesta a esta pregunta vamos a analizar varias fuentes.

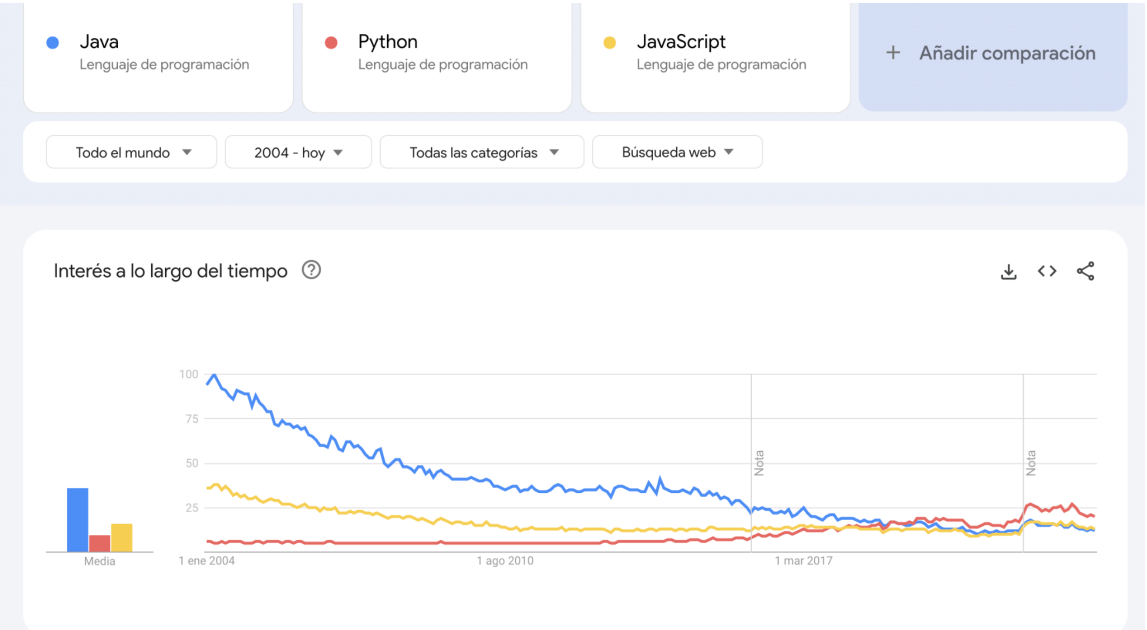
Primero vamos a hacer un análisis de las búsquedas de Google desde el lanzamiento de Java 8 hasta el día de hoy.

Si observamos de cerca la gráfica, vemos que el volumen de búsquedas se ha reducido un poco con el pasar de los años.



Si realizamos la comparación con Python, como era de esperarse y se evidencia, la popularidad de Python desde aquel entonces ha aumentado a tal punto que empieza a superar a Java y no es para menos, Python también es un excelente lenguaje.

Ahora agregamos JavaScript a la gráfica anterior.



Puede que después de ver esto, estés un poco sorprendido. Por el momento, el volumen de búsquedas en Google de JavaScript no

es tan grande como el de Python o Java, algo a aclarar es que no estoy asociando términos de búsqueda relacionados como pueden ser Spring Framework en el caso de Java o Django en el caso de Python, ni hablar de JavaScript con la cantidad de tecnologías tan populares que tiene, solo estoy comparando directamente el término de búsqueda.

Te invito a que tú mismo realices la comparación ingresando al sitio de Google Trends. Cuéntame en Twitter los resultados que hayas obtenido.

Stack OverFlow

Anualmente, Stack OverFlow diseña una encuesta para los usuarios del sitio, en la cual se busca clasificar temas como tecnologías más populares, condiciones de trabajo de los usuarios, género, entre muchos otros temas más.

De acuerdo con el resultado de la encuesta de 2022, Java ocupa el sexto lugar en las tecnologías más populares, solo superado por JavaScript, HTML/CSS, SQL, Python y TypeScript, es decir que solo existen de momento tres lenguajes de programación más populares que Java en Stack OverFlow y son Python, JavaScript y SQL.

Sin dudas, esto ratifica que Java sigue siendo un lenguaje bastante sólido en cuanto a la popularidad. Algo que debo aclarar es que la encuesta de Stack OverFlow fue respondida por cerca de 70,000 usuarios, aunque son 10,000 usuarios menos que el año anterior.

Tú mismo puedes consultar los resultados haciendo clic aquí e incluso puedes comparar los resultados con los de años anteriores cambiando el año que aparece en la URL.

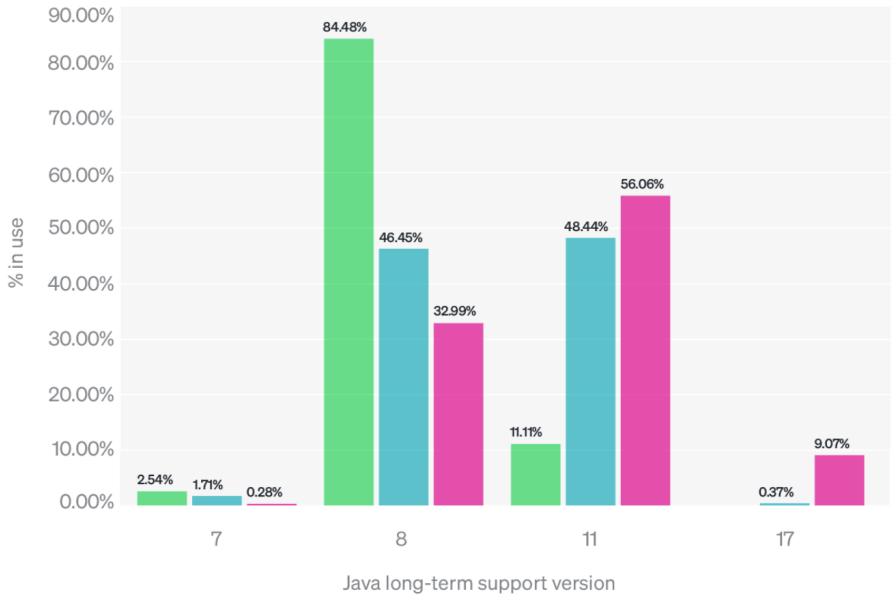
GitHub

De acuerdo con el Octoverse de GitHub del año pasado, Java se encuentra situado en tercer lugar, solo superado por JavaScript y Python.

Te invito a que veas los resultados del Octoverse tu mismo haciendo click aquí.

Java 11 - logró ser la más usada

De acuerdo con los resultados de una encuesta realizada en el año 2022 por New Relic, la mayoría de los desarrolladores utilizan Java 11 para sus aplicaciones en producción, veamos el siguiente gráfico.



Vemos claramente que más del 56% de los encuestados utilizan esta versión de Java en producción, algo que cambió significativamente desde que lanzamos este libro es que Java 8

poco a poco va dejando de ser la más usada mientras que Java 7 está en último lugar solo con el 0.28%, una muestra clara de que en forma progresiva Java 11 tomó el control y además podemos observar como van cambiando las cosas con Java 17 teniendo poco a poco un crecimiento.

Te invito a que veas por ti mismo el detalle de la encuesta, solo tienes que hacer click aquí.

Conclusión

Lo cierto es que la popularidad de Java no está en duda, es uno de los lenguajes más populares del mundo y creo que esto no cambiará dentro de poco.

Puede que no sea el más popular y eso está bien, JavaScript y Python son también excelentes, pero considero que Java también lo es y su aporte al mundo es igualmente gigantesco, no por nada grandes empresas como Spotify, Google, Amazon y muchas más lo utilizan.

El nuevo ciclo de actualizaciones, el enfoque en hacer un lenguaje mucho más conciso por parte del equipo de Java me llena de esperanza y sé que en el futuro muchas personas seguirán utilizando el lenguaje.

La polémica del licenciamiento de Oracle y el nuevo ciclo de actualizaciones

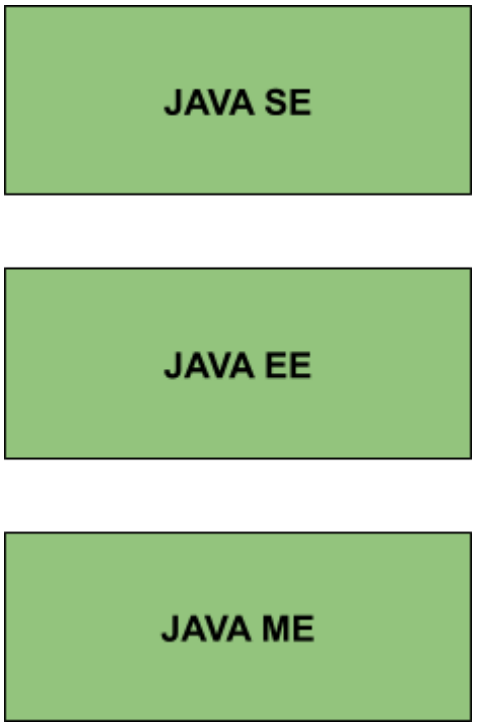
Antes de iniciar con la exploración de cada una de las nuevas versiones del lenguaje, hay un tema que no podemos pasar por alto.

Todo empezó hace algún tiempo cuando Oracle anunció que a partir de Enero del año 2019, las actualizaciones sobre el JDK 8 y JDK posteriores de Oracle ya no serían públicas. Para muchos, esto significó que Java a partir de 2019 dejaría de ser gratis y en muchos lugares de internet se afirmó sin dudarlo que Java ya no estaría disponible para uso personal y que era necesario pagar por una licencia para desarrollar sobre esta plataforma.

Lo cierto es que no hay nada más alejado de la realidad y puedo decirte sin temor a equivocarme, que **Java seguirá siendo gratis**, pero no hay duda en que debemos hablar sobre un par de cambios que Oracle realizó sobre Java y por la naturaleza de los mismos es imprescindible que los conozcas.

Los tres componentes principales

Actualmente Java cuenta con tres componentes principales los cuales veremos a continuación:



Java SE (Standard Edition)

Conjunto de librerías y componentes base del lenguaje incluidos por defecto en el JDK.

Java EE (Enterprise Edition)

Conjunto de librerías y componentes generalmente utilizados para el desarrollo web.

Java ME (Micro Edition)

Subconjunto de librerías de la versión SE, generalmente utilizado para el desarrollo sobre dispositivos móviles o para IoT.

Como ya mencioné anteriormente, Oracle anunció que las actualizaciones públicas para sus JDK ya no estarían disponibles a partir de Enero de 2019. Esto sin duda alguna significa que para usar el JDK de Oracle en un futuro no muy lejano probablemente debemos pagar alguna licencia, Lo que muchas personas desconocen es que Oracle no es la única empresa que ofrece un JDK. El JDK de Oracle, ni siquiera es la base para la creación de otros JDK, la base de creación de cualquier JDK es el OpenJDK, este es el conjunto de librerías y componentes de código abierto básicas que utiliza todo JDK.

Después de saber esto, es oportuno conocer que organizaciones como Eclipse Foundation e IBM, crean sus propias versiones de JDK basadas en el OpenJDK.

A pesar de que el OpenJDK es un proyecto mantenido por Oracle, es totalmente independiente del JDK de Oracle y puede ser usado sin la necesidad de ninguna licencia para uso personal o comercial. Es decir, a partir de ahora, lo mejor probablemente sea utilizar el OpenJDK o algún JDK distinto al de Oracle cuyas actualizaciones sean públicas.

Mitos del OpenJDK

Mito #1: “El OpenJDK es inestable y presenta fallas en ambientes de desarrollo, pruebas y producción”

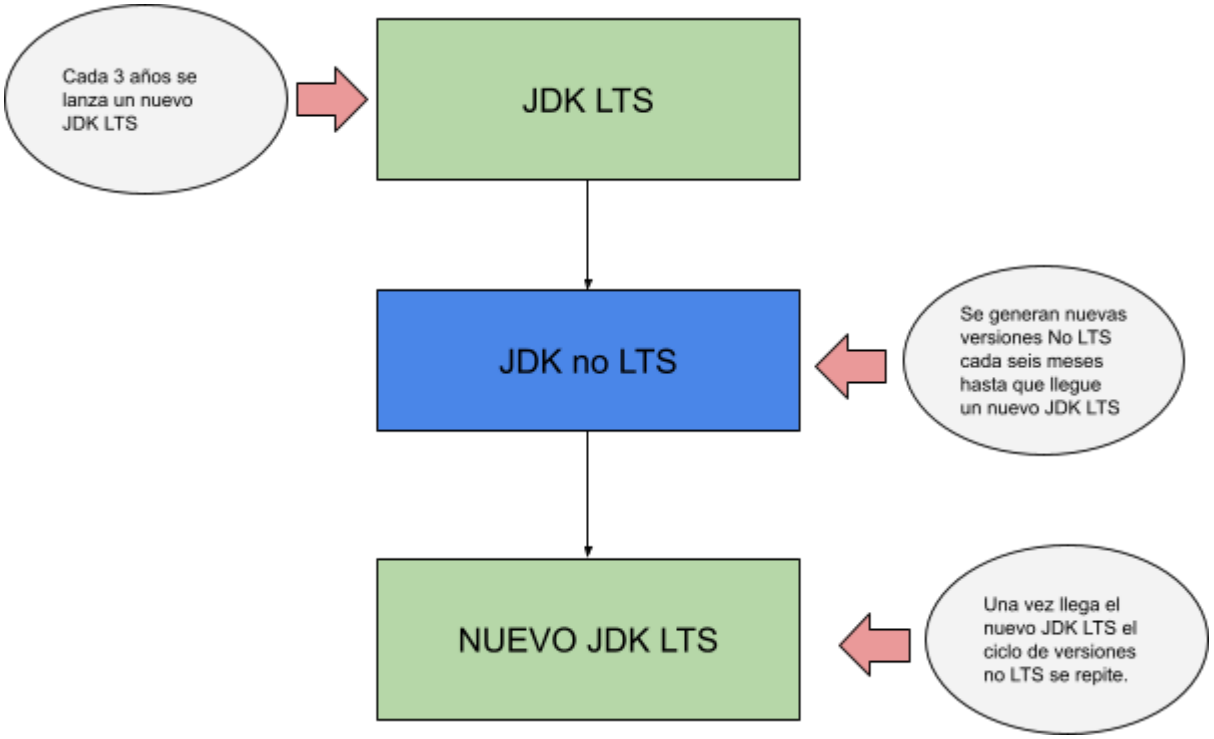
Si bien es cierto que en el pasado el OpenJDK presentaba algunos problemas, fue Oracle mismo quien se encargó de asegurar que estos fueran parte del pasado. Ahora podemos utilizar el OpenJDK de Oracle sin ningún problema, puedo asegurarte que el proceso de lanzamiento de cada una de las versiones del OpenJDK es muy riguroso y tiene una comunidad de expertos bastante amplia que lo respalda.

Mito #2: “El OpenJDK es incompleto”

El OpenJDK cuenta con el conjunto mínimo de librerías que se necesitan para el desarrollo sobre Java, probablemente algunos componentes que se incluyen en el JDK de Oracle ahora vienen por separado como es el caso de JavaFx, el cual viene por separado del OpenJDK y es llamado OpenJFX. Lo importante es que tendrás los mismos resultados o puedes buscar un JDK que lo incluya por defecto.

Estos y otros mitos son los que rodean al OpenJDK, pero como ya vimos anteriormente, no son más que eso y están muy alejados de la realidad. Hasta este punto podemos concluir que a pesar de lo anunciado por Oracle, no hay motivo para preocuparnos y podemos seguir disfrutando de Java sin ningún inconveniente, ya sea haciendo uso del OpenJDK o de cualquier otro JDK creado por las organizaciones autorizadas para hacerlo.

El nuevo ciclo de versiones y la inclusión de LTS y no LTS



Después de ver el gráfico anterior puede que ya sea más claro para muchos como es el nuevo ciclo de versiones del JDK, de igual manera pasaré a explicarlo a continuación.

A partir de la versión 8 del JDK, Oracle anunció que cada tres años se designará una versión como LTS (Long Time Support), es decir, versiones con soporte extendido. A diferencia del manejo de versiones anterior, Oracle no actualizará el lenguaje cada tres años sino que cada seis meses se lanzará una nueva versión no LTS cuya vigencia de soporte y actualizaciones será solo de seis meses hasta que una nueva versión no LTS o LTS (si se cumple el ciclo de tres años) aparezca en el camino.

Con las versiones no LTS, se prepara el camino para la nueva versión LTS. Siendo sincero, esta decisión de Oracle me parece

excelente ya que gracias a este anuncio podremos visualizar los nuevos cambios sobre el lenguaje poco a poco a lo largo del tiempo y no un enorme conjunto de cambios de una versión del JDK a otra.

Mapa de versiones del JDK

Es curioso pensar que cuando lanzamos este ebook la última versión fue la 12 ahora ya llegamos a la versión 22 y ya tuvimos otros LTS como fue Java 17 y muy próximamente Java 21, clic aquí para consultar el histórico de todas las versiones de Java.

Puede llegar a ser confuso el hecho de que la versión 9 salió tres años después que la versión 8 y esto es por que el anuncio de Oracle con respecto a las nuevas versiones recién se realizó en el año 2018.

Podríamos decir que la versión 8 es una mezcla del manejo de versiones anteriores y el manejo de versiones actual del JDK. La versión 8 representa un punto de inflexión en el lenguaje desde el punto de vista técnico debido a la cantidad de cambios significativos que llegaron a partir de esa versión y también por el cambio que significó con respecto al manejo de versiones del lenguaje.

¿Qué ha cambiado desde la llegada de Java 8?

Antes ya mencioné que a partir de Java 8, la cantidad de cambios en la forma de programar en Java es bastante significativa así que de aquí en adelante nos enfocaremos en los principales cambios que llegaron de la mano de cada versión del lenguaje.

Antes de empezar, aclaro que vamos a explorar desde Java 8 hasta la última versión a la fecha y para hacer un poco más liviano este material, mencionaré los cambios más importantes con respecto a la forma de programar en el lenguaje. Probablemente en un próximo título podremos explorar cada nueva versión mucho más a fondo.

Recuerda que cada vez que hagamos una actualización sobre este contenido lo recibirás en tu correo electrónico, este es uno de los dos beneficios de por vida que tienes por haber adquirido este material.

Java 8, el punto de inflexión

Métodos default y static en las interfaces

```
public interface Ejemplo{
  static int sumar(int a, int b) {
    return a + b;
  }

  default void imprimir(String str){
    System.out.println(str);
  }
}
```

Antes de Java 8, las interfaces solo podían tener métodos abstractos públicos. No era posible agregar una nueva funcionalidad a la interfaz existente sin que posteriormente todas las clases que implementan dicha interfaz, agregaran una implementación de los nuevos métodos, ni tampoco era posible crear métodos con cuerpo dentro de una interfaz.

A partir de Java 8, las interfaces pueden tener métodos estáticos y predeterminados, los cuales a pesar de estar declarados dentro de una interfaz, tienen un comportamiento definido, a continuación veremos algunas diferencias entre la forma de usar cada uno de estos métodos.

Implementación del método estático definido anteriormente.

```
int resultado = Ejemplo.sumar(1,2);
```

Como podemos ver en el código anterior, este método se encuentra disponible solo a través de la interfaz y no a través de la implementación de la misma, teniendo eso en cuenta el siguiente código no compila.

```
Ejemplo ej = new EjemploImpl();
int resultado = ej.sumar(1,2);
```

A diferencia del método definido como `default` el cual solo está disponible a través de la implementación de la interfaz como lo veremos en el siguiente código.

```
Ejemplo ej = new EjemploImpl();
int resultado =
ej.imprimir("4SoftwareDevelopers");
```

Una vez más teniendo en cuenta lo anterior, el siguiente código no compila, debido a que los métodos default no pueden ser accedidos directamente desde la interfaz.

```
Ejemplo.imprimir("4SoftwareDevelopers");
```

Interfaces Funcionales

Una interfaz con exactamente un método abstracto se llama Interfaz funcional. Se agrega la anotación `@FunctionalInterface` para indicar que la interfaz es una interfaz funcional.

No es necesario agregar esta anotación a dicha interfaz; basta con cumplir la regla de que contenga un método abstracto, pero es una buena práctica usarlo con interfaces funcionales para evitar la adición accidental de métodos adicionales. Si la interfaz

es marcada con la anotación `@FunctionalInterface` y tratamos de tener más de un método abstracto, produce un error de compilación.

El mayor beneficio de las interfaces funcionales es que podemos usar **expresiones lambda** para crear instancias y evitar el uso de una implementación de clase anónima de gran tamaño.

Expresiones Lambda

Sin duda alguna, uno de los cambios más significativos que llegó de la mano de Java 8 son las expresiones lambda. Con estas expresiones es posible utilizar la programación funcional en el mundo orientado a objetos de Java. Los objetos son la base del lenguaje y nunca podemos tener una función sin un Objeto, es por eso que el lenguaje Java brinda soporte para usar expresiones lambda solo con interfaces funcionales.

Debido a que solo hay una función abstracta en las interfaces funcionales, no hay confusión al aplicar la expresión lambda al método. La sintaxis de las expresiones lambda es `(argumento) -> (cuerpo)`.

Ahora veamos un ejemplo de implementación de las expresiones lambda utilizando la interfaz `java.lang.Runnable`

```
    Runnable r1 = () -> System.out.println("Hola
    desde Runnable");
```

Vamos a explicar el código que acabamos de crear anteriormente:

- Runnable es una interfaz funcional, es por eso que podemos usar la expresión lambda para crear su instancia.
- Como el método run () no toma ningún argumento, nuestra expresión lambda tampoco tiene ningún argumento.

- Al igual que los bloques if-else, podemos evitar las llaves ({}) ya que tenemos una sola declaración en el cuerpo del método. Para declaraciones múltiples, tendríamos que usar llaves como cualquier otro método.

Antes de analizar los beneficios de las expresiones lambda veamos cómo se realizaba este mismo proceso antes de la versión 8 del lenguaje.

```
Runnable r1 = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hola desde
Runnable");
    }
};
```

¿Lo ves?, espero que con este empieces a ver lo mucho que cambió Java desde la llegada de Java 8.

¿Por qué usar las expresiones lambda?

- **Reducción de líneas de código**
  Una de las ventajas claras de usar expresiones lambda es que la cantidad de código se reduce, ya hemos visto con qué facilidad podemos crear una instancia de una interfaz funcional utilizando la expresión lambda en lugar de usar una clase anónima.

- **Fácil integración con API Stream**
  Otro beneficio de usar la expresión lambda es que podemos beneficiarnos de la compatibilidad de las operaciones de la API Stream, la cual veremos a continuación.

Stream API

Junto con Java 8 llegó un nuevo paquete llamado `java.util.stream`. El enfoque principal de esta nueva API es realizar operaciones de filtro, mapeo, agrupamiento y entre muchas otras.

Vamos a realizar un pequeño ejemplo tomando una lista de números enteros y sumando sus valores, sólo cuando el valor de este sea mayor a cinco.

```
    private static int sumarContenido(List<Integer>
    list) {
        return list.stream().filter(i -> i >
    5).mapToInt(i    -> i).sum();
    }
```

De nuevo vamos a intentar entender el código que acabamos de escribir

- Invocamos el método `stream` sobre la lista que recibimos como argumento la cual recibe el nombre de `list` el cual convierte nuestra lista en un objeto del tipo `Stream.`

- Posteriormente invocamos el método filter el cual pertenece a Stream API. Dentro de este método se agrega una expresión lambda donde `i` es el argumento de nuestra expresión el cual tomaría el valor de cada uno de los números de la lista, posteriormente en el cuerpo de la expresión se valida que el número sea mayor a 5, dando esto como resultado un nuevo objeto solo con los números mayores a cinco.

- Una vez se ha filtrado la lista, necesitamos invocar el método `mapToInt` el cual también recibe como argumento

una expresión lambda, posteriormente invocamos el método `sum` el cual nos devuelve un entero con la suma de todas las posiciones de la lista.

Ahora veamos cómo se realizaba este proceso antes de Java 8

```
private static int sumarContenido(List<Integer>
list) {
     Iterator<Integer> it = list.iterator();
     int sum = 0;
     while (it.hasNext()) {
          int num = it.next();
          if (num > 5) {
               sum += num;
          }
     }
     return sum;
}
```

Nuevamente podemos observar cómo a partir de Java 8 la reducción del código es bastante significativa.

Optional<T>

Una de las quejas más habituales que he escuchado acerca de Java es el cuidadoso control que requiere el evitar la famosa excepción `NullPointerException` y es una queja bien fundamentada, yo mismo he tenido que crear cantidades de código muy repetitivo para evitar esta excepción y ciertamente lenguajes como Kotlin han solventado este problema de una forma muy interesante.

A partir de Java 8 llegó una nueva funcionalidad que nos apoya en este proceso ya que esta clase funciona como un contenedor para

el objeto de tipo T (El valor de T es genérico y puede ser ocupado por cualquier clase). Puede devolver un valor de este objeto si este valor no es nulo. Cuando el valor dentro de este contenedor es nulo, permite realizar algunas acciones predefinidas en lugar de lanzar un `NullPointerException`.

Ciertamente veremos algunas formas de ir creando un objeto del tipo Optional a continuación.

```
Optional<String> optional = Optional.empty();
```

El código anterior, crea una instancia de la clase Optional vacío del tipo String, si deseamos crear un objeto opcional a partir de un valor existente podemos realizar el siguiente código:

```
String str = "valor";
Optional<String> optional = Optional.of(str);
```

Por último si el valor del que enviaremos al Optional puede ser nulo podemos aplicar el siguiente código.

```
Optional<String> optional =
Optional.ofNullable(getString());
```

En el código anterior se invoca el método `getString()` el cual no está declarado, simplemente lo mencioné para hacer referencia a un método que retorna un objeto String cuyo valor puede ser nulo. En caso de que el valor sea nulo se retornará un objeto Optional vacío.

Vamos a ver un pequeño ejemplo, imaginemos que necesitamos obtener un objeto del tipo `List<Integer>` y que en el caso de ser nulo, deseamos sustituirlo por una nueva instancia de `ArrayList<Integer>`.

En Java 8 esto se hace de la siguiente manera.

```
List<Integer> listOpt = getList().orElseGet(() -> new
ArrayList<>());
```

En este caso, el método `getList()` funciona de la misma manera que el método `getString()` del ejemplo anterior. Este método retorna un objeto del tipo `Optional<List<Integer>>` y gracias a Optional podemos invocar el método `orElseGet()` en el cual, haciendo uso de las expresiones lambda, podemos definir el objeto a obtener en caso de que el valor retornado por el método `getList()` sea nulo.

Conclusión

Como pudimos observar hasta ahora, la forma de programar en Java cambió significativamente con la llegada de Java 8. Podemos notar que el principal objetivo del equipo de desarrollo de Java es que seamos más productivos y esto se logra creando un lenguaje más conciso. Ningún tema lo abordé hasta el fondo para no desviar el propósito de este material.

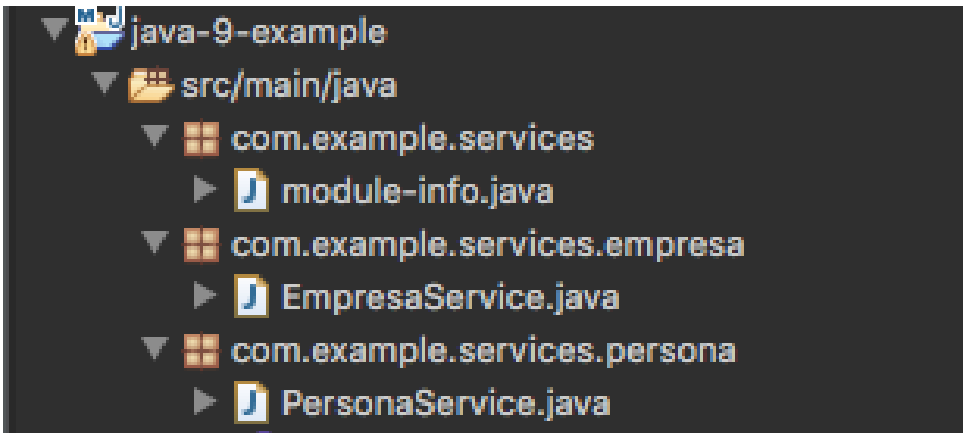
Espero que con esto, te animes a programar en Java si aún no lo haces y si ya lo estás haciendo, tal vez estés trabajando con una versión antigua y es hora de migrar. Esto recién comienza, a lo largo de las otras versiones veremos otros cambios importantes.

Java 9, el arribo del Proyecto Jigsaw

Modularidad

Sin duda alguna, la característica más importante que llegó con Java 9 fue el proyecto Jigsaw el cual introduce el concepto de modularidad al sistema monolítico de Java. El objetivo principal de este proyecto es hacer que Java SE y el JDK sean fácilmente escalables en dispositivos móviles.

Para hacer uso del proyecto Jigsaw, solo debemos incluir el archivo `module-info.java` en el paquete que queremos convertir en módulo, veamos un ejemplo.



Cada módulo contiene un archivo **`module.info.java`** que describe las dependencias del módulo y la distribución del mismo.

De acuerdo con la imagen anterior, tenemos el módulo `com.example.services` el cual tiene dos paquetes `empresa` y `persona`. A continuación veremos cuál debe ser el contenido del archivo **`module-info.java`** si queremos exponer solo el paquete `empresa` y mantener el paquete `persona` privado.

```
module com.example.services {
    exports com.example.services.empresa;
}
```

Ahora imaginemos que tenemos el módulo `com.example.controller` dentro de la misma aplicación el cual contiene la capa de presentación de la aplicación. Si queremos utilizar el módulo que definimos anteriormente, el archivo module-info.java de este módulo debe ser definido de la siguiente manera:

```
module com.example.controller {
    requires com.example.services;
}
```

De esta manera podemos acceder al paquete `empresa` que se encuentra definido en `com.example.services`.

Sin duda alguna, la introducción de la modularidad (la cual por cierto tiene un mejor encapsulamiento que los **jar**), nos ayudará a tener sistemas de información mejor distribuidos y las labores de mantenimiento serán mucho más fáciles de llevar.

JShell

Después de una larga espera para muchos, por fin Java incluye una consola RELP (**R**ead-**E**val-**P**rint **L**oop), esta herramienta básicamente nos permite interactuar con el lenguaje sin necesidad de crear un proyecto o archivo Java; es algo similar a la consola con la que cuenta Python. Podemos encontrar el ejecutable de esta herramienta en la carpeta bin de nuestro directorio de instalación del JDK.

En la JShell podemos hacer muchas cosas como crear variables,

métodos e incluso ejecutarlos. A continuación, veremos un ejemplo sencillo de lo que podemos hacer gracias a esta herramienta.

Definiendo un objeto del tipo Stream:

```
jshell> Stream<String> nombres = Stream.of("Jordy",
"Carlos","Camilo");
nombres
==>java.util.stream.ReferencePipeline$Head@6e1567f1
```

Creando un método que imprima el contenido del Stream:

```
jshell> public void imprimirStream(Stream<String>
stream){
System.out.println(stream.collect(Collectors.toList()));
}
created method imprimirStream(Stream<String>)
```

Invocando el método creado:

```
jshell> imprimirStream(nombres);
[Jordy, Carlos, Camilo]
```

JShell también cuenta con el autocompletado de comandos, solo debemos empezar a escribir nuestro comando y presionar la tecla **Tab**, de esta manera podremos ver todas las opciones disponibles en relación con el comando que tratamos de ejecutar. Adicional a esto, JShell cuenta con algunos comando propios adicionales a expresiones Java, veamos algunos ejemplos.

El comando `jshell> /imports` muestra todos los imports que estas utilizando, a continuación se muestra un ejemplo, ten en cuenta que los import que se verán a continuación son los que incluye la JShell por defecto.

```
jshell> /imports
import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*
```

**** Si necesitas agregar algún import, puedes hacerlo usando la sentencia `import`, como se hace dentro de una clase Java convencional.***

Si en algún momento usted olvidó las variables que había declarado no hay ningún problema, puede usar el comando `jshell> /vars` y obtendrá una lista con las variables definidas, tal y como se muestra a continuación:

```
jshell> /vars
|    Stream<String> nombres =
java.util.stream.ReferencePipeline$Head@6e1567f
1
```

De igual forma puede hacer el mismo proceso con los métodos definidos:

```
jshell> /methods
|    void imprimirStream(Stream<String>)
```

Si necesita cerrar el JShell sólo debe ejecutar el comando

```
jshell> /exit
```

```
jshell> /exit
|  Goodbye
```

Por último, si usted desea explorar la JShell, puede apoyarse en el comando `jshell> /help`.

Conclusión

Java 9 trajo consigo cambios importantes en el lenguaje, sin duda la inclusión de la modularidad y de la JShell viene siendo pedida por la comunidad desde hace algún tiempo.

Nuevamente es de resaltar la labor del equipo de desarrollo de Java, porque hasta el momento estos cambios nos ayudan a ser más productivos.

Por supuesto estos no son los únicos cambios que llegaron en esta versión del JDK. Aunque sin dudas considero que los mencionados aquí son los más importantes y los que más cambiaron la forma de trabajo sobre el lenguaje.

Java 10, var es una realidad

Java 10 no llegó con muchos cambios innovadores. No obstante, considero pertinente resaltar que el equipo de desarrollo de Java entregó la versión según lo prometido, solo 6 meses después del lanzamiento de Java 9.

Me identifico un poco con esta situación, creo que muchos de los que trabajamos en esta industria hemos tenido la presión de entregar un producto de acuerdo a una fecha determinada, y lo curioso es que muchas veces esa fecha no es escogida por nosotros.

Lo importante es que se logró cumplir con el objetivo y se entregó un producto completo, estable y que corresponde al altísimo nivel que requiere el desarrollo del JDK.

Inferencia de tipos sobre variables locales

Probablemente este es el cambio más importante en esta versión del JDK. La palabra reservada `var` puede reemplazar la inferencia de tipos estática con la que venimos trabajando, de esta manera la declaración de las variables locales es más simple y podemos dejar que el compilador haga todo el trabajo. Hay que tener en cuenta que por el momento solo podemos usar `var` para variables locales, no podemos utilizarla en miembros de clase ni tampoco como argumento de métodos.

Veamos algunos ejemplos del uso de `var` para entenderlo mejor:

```
    var nombre = "4SoftwareDevelopers"; // infiere
    el tipo String
```

```
var charArr = nombre.toCharArray(); // infiere
el tipo char[]
```

Otro ejemplo de un uso correcto de var es:

```
var letras = List.of("z", "y", "x");
for (var lt : letras) {
   System.out.println(nr);
}
```

Ahora veamos ejemplos con usos incorrectos de var, todos los que veremos a continuación generan un error de compilación:

```
var prueba;
var letras = {"a", "b", "c"};
var lambda = a -> a + " ";

private var getNombre() {
   return "4SoftwareDevelopers";
}
```

Adicional a lo anterior, debemos tener en cuenta que una vez se infiere el tipo de la variable, este no se puede cambiar. Veamos un ejemplo donde el siguiente código no compila:

```
var personas = new ArrayList<Persona>(); //tipo
inferido: ArrayList<Persona>
personas = new LinkedList<Persona>(); // error,
en la línea anterior se infirió el tipo
ArrayList<Persona>
```

Conclusión

Como mencioné antes, esta versión no llegó con muchos cambios relevantes desde el punto de vista del desarrollador. Sin embargo, la inclusión de la inferencia de tipos fue algo que me encantó de esta versión ya que hace solo unos años atrás, algo como esto era impensable por la forma en la que solíamos trabajar en el lenguaje.

Son este tipo de cosas las que me convencen del buen camino que ha tomado el lenguaje, ya es costumbre que nos entreguen versiones que se enfocan totalmente en mejorar nuestra productividad.

Java 11, El nuevo LTS

Adiós a la ceremonia del primer programa

Desde Java 9 contamos con un JShell que nos evita el tener que crear una clase, compilarla y después ejecutarla para poder trabajar sobre el lenguaje, por otro lado de seguro muchas personas van a preferir no usar el JShell por distintas razones, por ende se verán forzados a un “largo” proceso para crear una simple clase en Java, por ejemplo, imaginemos que vamos a crear una clase con el típico ejemplo de Hola Mundo en Java, anteriormente lo que debíamos hacer era crear un archivo, después con el comando `javac` compilar dicho archivo y posteriormente con el comando `java` ejecutarlo; la buena noticia es que esto quedó en el pasado, ahora solo debemos hacer uso del comando `java` para ejecutar una clase sin necesidad de usar el comando `javac`. Veamos un ejemplo.

Creación de clase:

```
public class Main{
   public static void main(String[] args){
     System.out.println("Hola
4softwareDevelopers");
   }
}
```

Ejecución de la clase:

```
java Main.java
Hola 4SoftwareDevelopers
```

Como vemos, nuevamente el equipo de desarrollo del JDK nos

muestra que están totalmente enfocados a que seamos más productivos.

La característica principal de esta versión

Si bien en todas las versiones del JDK se agregan nuevas API que mejoran el lenguaje, Java 11 es una versión que se caracteriza por tener un gran número de APIs sobre las distintas clases del JDK. Una que me llamó mucho la atención fue la clase String sobre la cual se agregaron algunos métodos como `IsBlank`, `repeat`, `lines` entre otros, te recomiendo que las pruebes por ti mismo y me cuentes en Twitter tu opinión al respecto.

Conclusión

Si vemos en retrospectiva poco a poco estamos viendo ante nuestros ojos como Java se hace un lenguaje mucho más conciso y con algunos componentes que para mi lo convierten en un lenguaje muy atractivo.

Java 12, una versión no tan enfocada en cambiar el lenguaje

Probablemente esta versión sea la menos sorprendente de todas en cuanto a cambios significativos en el lenguaje. Ten en que desde la versión 10 hay otros aspectos igualmente importantes en los cuales el equipo de Java ha trabajado arduamente,como por ejemplo el rendimiento del lenguaje, la inclusión de un nuevo recolector de basura entre muchos otros.

Básicamente, es por eso que desde dicha versión no hemos visto cambios muy significativos en la forma de programar.

El nuevo switch

A partir de esta versión hay una nueva forma de implementar el switch. Una forma más concisa, veamos un ejemplo para entenderlo mejor.

Antes de Java 12:

```
    int nivel = 0;

    switch(cargo){
     case CEO:
     case CTO:
      nivel = 6;
     break;
     case ARQUITECTO:
      nivel = 5;
     break;
     case DESARROLLADOR:
     case LIDER_TECNICO:
```

```
        nivel = 4;
       break;
      }
```

A partir de Java 12:

```
    int nivel = switch (cargo) {
      case CEO, CTO -> 6;
      case ARQUITECTO -> 5;
      case DESAROLLADOR, LIDER_TECNICO -> 4;
      default -> 1
    };
```

Como podemos ver en el ejemplo anterior, ahora es mucho más fácil implementar la sentencia switch. Ciertamente la versatilidad de esta sentencia me llamó mucho la atención ya que ahora es mucho más concisa.

Conclusión

Esta versión como ya mencioné antes, no incluyó una gran cantidad de cambios en el lenguaje. Sin dudas, el cambio del switch es el más significativo sobre la estructura básica del lenguaje.

Java 13, los bloques de texto no son una realidad del todo

Al igual que Java 12, esta versión no es LTS y tampoco cuenta con una gran cantidad de cambios significativos sobre la forma de programar en el lenguaje. Algo interesante es que esta versión introdujo un concepto muy interesante llamado “preview features” de los cuales hablaremos más adelante en este capítulo.

Text blocks

Java siempre ha sufrido un poco por la forma en que se definen las cadenas. Una cadena comienza con una comilla doble y termina con una comilla doble, lo cual está bien, excepto que la cadena no puede abarcar más de una línea. Esto lleva a soluciones alternativas, como incluir `\n` donde se requieren saltos de línea o concatenar cadenas y líneas nuevas, pero esto evita que nuestro código sea limpio.

Fue así como el equipo de desarrollo del JDK decidió incluir los bloques de texto como una solución a la situación planteada anteriormente, para indicar que una cadena es un bloque de texto debemos utilizar tres comillas dobles al principio y al final del bloque de texto. Algo a tener en cuenta es que, estos bloques de texto deben contar con más de una línea. Veamos un ejemplo de cómo usar los bloques de texto.

Lo primero que debes saber es que los bloques de texto no pueden ser usados en una sola línea como ya mencioné anteriormente, veamos:

```
String bloque = """Bloque de una linea""";
```

El código anterior lanzará un error similar al siguiente:

```
BloqueDeTexto.java:3: error: illegal text block open
delimiter sequence, missing line terminator
```

Ahora veamos un ejemplo en el cual el código funciona correctamente:

```
String bloque = """
                    <html>
                      <body>
                        <p>My web page</p>
                      </body>
                    <html>
                   """;
```

El código anterior funciona correctamente y si lo analizamos un momento me parece que es realmente interesante esta nueva forma de crear bloques de texto en Java.

Puedes conocer más a detalle cómo usar los bloques de texto aquí, cuéntame en Twitter tu experiencia con el uso de los bloques de texto.

Preview features

Si bien los bloques de texto son ahora una realidad, estos fueron agregados a esta nueva versión como una funcionalidad de vista previa o preview feature en inglés, todas las funcionalidades de vista previa no son incluidas como una parte relevante de la especificación del lenguaje. El beneficio que tiene esto es que nosotros como usuarios podemos probar las funcionalidades y en caso de ser necesario dar feedback al equipo de desarrollo de Java, lo cual significa que en un futuro cercano, los bloques de

texto o cualquier otra funcionalidad marcada como funcionalidad de vista previa pueden cambiar significativamente.

Algo que es necesario aclarar es que las funcionalidades de vista previa **no son beta**, todas estas funcionalidades son estables y se encuentran completamente implementadas.

Nota: Debido a que las funcionalidades de vista previa no son parte de la especificación del lenguaje, es necesario que habilitemos su uso al momento de compilar una clase o ejecutarla.

Vamos a suponer que el ejemplo bloque de texto que mostramos anteriormente se encuentra dentro de una clase llamada **BloqueDeTexto.java**, veamos como sería la compilación y ejecución de esta clase.

Compilación:

```
javac --enable-preview --release 13 BloqueDeTexto.java
```

Algo interesante del comando anterior es que el flag `-release` es nuevo, podemos también usar el flag `-source` en su lugar.

Ejecución:

```
java --enable-preview BloqueDeTexto
```

Las funcionalidades de vista previa también pueden tener impacto sobre los métodos de las clases del JDK, por ejemplo, en la clase **String** fueron agregados estos tres nuevos métodos:

- `formatted()`: Permite formatear un String de acuerdo a los parámetros enviados, es prácticamente igual al método `format(this, args)`.

- `stripIndent()`: Este método elimina espacios incidentales de la cadena. Esto es útil si está leyendo cadenas de varias líneas y desea aplicar la misma eliminación de espacios en blanco incidentales que ocurre con una declaración explícita.

- `translateEscapes()`: Este método retorna la cadena con secuencias de escape, (por ejemplo, \r, \n) traducidas al valor Unicode apropiado.

Lo interesante de estos métodos más allá de sus funcionalidades es que a pesar de ser nuevos ya fueron marcados como deprecados y es probable que en próximas versiones del lenguaje ya no existan, la razón es que estos métodos están directamente asociados a los bloques de texto y debido a que los bloques de texto son una funcionalidad de vista previa que está sujeta a cambios en un futuro cercano, el futuro de estos métodos es un tanto incierto.

Dentro de la comunidad de Java, existe una propuesta realmente interesante de marcar métodos de este tipo con la anotación `@PreviewFeature` lo cual nos permitiría manejar todo esto de una forma mucho más sencilla, lo cierto es que de incluirse esta anotación sería en una próxima versión del JDK.

Para cerrar esta sección, es necesario precisar que todas las nuevas funcionalidades no serán funcionalidades de vista previa. De acuerdo con lo dicho por Mark Reinhold (Arquitecto en jefe de Java Platform Group) este proceso tiene como objetivo recolectar información de ciertas funcionalidades antes de que sean incorporadas de forma definitiva al lenguaje, algo así como una fase de pruebas de aceptación para algunas funcionalidades específicas.

Una nueva forma de implementar switch

La llegada del JDK 12 trajo consigo una nueva forma de implementar switch mucho más concis, en esa versión también es posible asignar una variable de retorno a un switch usando la estructura de switch que se utilizaba en el pasado. Veamos un ejemplo:

```
int nivel = switch(cargo){
 case CEO:
 case CTO:
   break 6;
 case ARQUITECTO:
   break 5;
 case DESARROLLADOR:
 case LIDER_TECNICO:
   break 4;
}
```

Esta nueva funcionalidad a pesar de ser útil, generó cierta controversia en la la comunidad de Java, se mencionó que usar la palabra reservada `break` para indicar la asignación de un valor podía llegar a ser confuso, fue por eso que se introdujo una nueva palabra reservada, que permite que en el caso de usar este switch nuestro código luzca limpio. Veamos un ejemplo:

```
int nivel = switch(cargo){
 case CEO:
 case CTO:
   yield 6;
 case ARQUITECTO:
```

```
        yield 5;
      case DESARROLLADOR:
      case LIDER_TECNICO:
        yield 4;
    }
```

Mi recomendación es que utilices la nueva forma de implementar `switch`. Puede ver esto a detalle en el capítulo anterior, donde hablé de la nueva forma de usar switch en Java 12. Ciertamente esa forma de implementar es mucho más concisa y limpia.

Conclusión

Java 13 llega como una versión discreta en cuanto a cambios en el lenguaje aunque también nos trae una nueva forma de explorar las funcionalidades del JDK. Hablamos de las funcionalidades de vista previa y es que situaciones como la ocurrida con `switch` la cual impulsó la llegada de `yield` fueron las que llevaron al equipo del JDK a realizar la implementación de las funcionalidades de vista previa, es ahí donde está su real beneficio.

Esta forma de trabajo, permite que funcionalidades nuevas sean utilizadas por el usuario final. Es decir, por nosotros los desarrolladores y hagamos un análisis real de los potenciales puntos de mejora que pueden llegar a tener, esto es un proceso de comunidad en toda la extensión de la palabra, nosotros como desarrolladores podemos aportar a la construcción de este lenguaje y hacerlo mucho mejor.

Java 14, novedades que todos estábamos esperando

Esta nueva versión de Java, llega con varias funcionalidades interesantes que vamos a explorar en esta sección, realmente todas estas nuevas funcionalidades a nivel programático creo que aportan mucho a nuestra productividad y debo confesar que muchas de ellas las he estado esperando.

La eficiencia del nuevo instanceof

Estoy seguro que muchos de nosotros en determinado momento hemos necesitado hacer uso de la palabra reservada instanceof, si no conoces el funcionamiento de esta palabra reservada te lo explico, instanceof te permite identificar cual el es tipo de una variable determinada, veamos un ejemplo:

```
if(obj instanceof String) {
  //Entra aquí si obj es o hereda de String
} else if(obj instanceof Long) {
  //Entra aquí si obj es o hereda de Long
}
```

En el código anterior vemos cómo se realiza la validación sobre una variable llamada **`obj`**, generalmente esto se hace cuando la variable obj es de tipo Object o es hija de una clase específica, claramente esto está relacionado con la lógica de negocio de cada aplicación, ahora que ya conoces el uso que se le da a **`instanceof`**, es necesario que conozcas un problema que tenía el uso de esta palabra reservada, para eso voy a mostrar como se usaba **`instanceof`** antes de Java 14:

```
if(obj instanceof String) {
 String str = (String) obj;
 System.out.println(str);
} else if(obj instanceof Long) {
 Long number = (Long) obj;
 System.out.println(number);
}
```

El problema que tiene el código anterior es simple, posterior a la validación del **`instanceof`** se hacía necesario convertir la variable a una nueva variable del tipo que estamos identificando es decir si la variable **`obj`** era una instancia de **`String`**, debíamos crear una nueva variable de tipo **`String`** y convertir la variable **`obj`** a **`String`**.

Esto no es un problema debido a que Java sea fuertemente tipado, esto es un problema ligado a que ese código es innecesario, ya que si estoy validando el tipo de dato de una determinada variable en la gran mayoría de casos necesitaré usarla para una determinada lógica. Lo importante es que en Java 14 se soluciona este problema veamos:

```
if(obj instanceof String str) {
  System.out.println(str);
} else if(obj instanceof Long number) {
  System.out.println(number);
}
```

Como vemos a partir de esta nueva versión del lenguaje podemos en una sola línea de código generar la conversión automática de la variable que estamos comparando y generar una nueva variable. Son estas pequeñas cosas que hacen que

Java siga siendo un lenguaje moderno y atractivo.

NullPointerException es ahora un problema del pasado

El equipo de desarrollo de Java nos da un regalo que realmente nos va a ayudar a identificar errores ligado al **`NullPointerException`**, tal vez la excepción más popular del lenguaje veamos de que se trata.

En siguiente fragmento de código al momento de ser ejecutado genera una excepción **`NullPointerException`**, veamos:

```
String nombre = null;
nombre.toUpperCase();
```

Si ejecutamos este código antes de Java 14 el mensaje que veríamos de excepción es el siguiente:

```
Exception in thread "main"
java.lang.NullPointerException
at MiClase.main(MiClase.java:5)
```

Realmente si analizamos un momento el mensaje de error es bastante descriptivo y es relativamente fácil encontrar el error con un código tan simple, pero veamos un código en el que no resulta tan simple encontrar el determinado error:

```
var persona = new Persona();
```

```
persona.getCiudad().getDepartamento().getPais();
```

Si obtenemos un mensaje de error igual al anteriormente mencionado va a ser una tarea complicada identificar en dónde está el problema, pero si estamos usando Java 14 el mensaje de error sería el siguiente:

```
Exception in thread "main"
java.lang.NullPointerException:
        Cannot read field 'pais' because
'ciudad.departamento' is null.
    at MiClase.main(MiClase.java:5)
```

De esta forma es muy fácil identificar dónde está el error en el código, esto nos ayuda a aumentar la productividad que tenemos al momento de desarrollar.

Debo confesar que este es uno de los cambios que más me han gustado de todos lo que he hablado hasta aquí, muchas personas que inician tendrán una herramienta más efectiva a la hora de aprender el lenguaje y esto es algo que beneficiará a la comunidad sin duda alguna.

Records una funcionalidad preview que genera mucho valor

En la sección anterior hablamos un poco acerca de los Preview Features que encontraremos en las nuevas versiones del JDK, en Java 14 hay una funcionalidad preview que sin dudas genera un impacto significativo e interesante sobre el lenguaje.

Algo que siempre me llamó mucho la atención de Java es que necesitaba escribir mucho código para crear una clase que tuviera

constructores, modificadores de acceso, equals(), hashCode(), toString(), etc.

Desde esta nueva versión nos damos cuenta que el equipo de desarrollo de Java, está planeando que esto quede en el pasado, veamos el código que era necesario realizar antes de Java 14, para crear una clase con las características antes mencionadas:

```
public class Persona {
    private String nombre;
    private String apellido;

    public Point(String nombre, STring apellido) {
        this.nombre = nombre;
        this.apellido = apellido;
    }

   public String getNombre(){
      return this.nombre;
   }

   public void setNombre(String nombre) {
      this.nombre = nombre;
   }

   public String getApellido() {
     return this.apellido;
   }

   public void setApellido(String apellido) {
     this.apellido = apellido;
   }

   //Métodos equals, hashCode, toString.
```

```
}
```

Ahora veamos el código que debemos hacer a partir de Java 14 para crear una clase con una estructura muy similar a la anterior:

```
public record Persona(String nombre, String apellido){}
```

Esa pequeña porción de código nos genera un código muy similar al que creamos anteriormente, si necesitamos compilar el código anterior podemos hacer ejecutando el siguiente comando:

```
javac --enable-preview -source 14 Persona.java
```

Es necesario aclarar que para la ejecución exitosa de ese comando se debe tener instalado el JDK 14.

Si queremos ver el código compilado por el comando anterior podemos ejecutar el siguiente comando:

```
javap Persona
```

Este comando nos mostrará un resultado similar al siguiente:

```
public final class Person extends java.lang.Record {
  public Person(java.lang.String, java.lang.String);
```

```
  public java.lang.String toString();
  public final int hashCode();
  public final boolean equals(java.lang.Object);
  public java.lang.String nombre();
  public java.lang.String apellido();
}
```

Podemos observar que se genera una clase que extiende de **`java.lang.Record`**, ahora bien, ya conoces esta poderosa herramienta es momento de que hablemos acerca de las limitaciones que tienen los records en Java:

- Los records no pueden ser hijos de ninguna otra clase y no pueden declarar campos de instancia que no sean finales y privados.

- Los records no pueden ser abstractos y son implícitamente finales, esto es básicamente porque el API de récords de Java está definida por su estado y no puede ser modificado por otra clase o por otro récord.

- Los miembros de clase de un récord son implícitamente finales.

Los TextBlocks son cada vez mejores (segunda preview)

En la sección dedicada a Java 13 hablamos acerca de los TextBlocks y aclaramos que es una funcionalidad de vista previa, lo interesante aquí es que en Java 14 continúa siendo una funcionalidad de vista previa, por otro lado he de resaltar que se agregó una funcionalidad que vale la pena detallar.

Separador de nuevas líneas

Ahora es posible agregar un separador para cada línea del bloque de texto, veamos:

```
String bloque = """
                    Este es un texto \
                    de prueba en el que se muestra \
                    este nuevo separador de \
                    línea.\
                  """;
```

Con esto se espera que podamos lograr delimitar la cantidad de caracteres entre otras funcionalidades que estaremos detallando en próximas versiones del JDK.

Conclusión

Después de leer las funcionalidades más importantes que fueron introducidas a nivel de código por el equipo de Java, me siento bastante contento, nuevamente me agrada el camino que está tomando Java, buscando convertirse en un lenguaje más conciso y atractivo para los desarrolladores de software.

Si bien es un lenguaje con una popularidad y reputación sólida, sabemos que para conservarse así, es necesario mantenerse en un constante cambio.

Valoro mucho la incluso de los nuevos mensajes sobre la excepción **`NullPointerException`**, se que este punto será vital para quienes recién inician y también para decrecer el tiempo que dedicamos a solucionar bugs de nuestros sistemas.

Confío en que el equipo de Java nos seguirá sorprendiendo en las nuevas versiones del JDK y espero que pruebes tu mismo cada una de estas nuevas funcionalidades.

Java 15, pocos cambios pero mucho valor

Sealed Clases (Preview feature)

A partir de esta versión podremos utilizar las llamadas sealed clases con las cuales podremos restringir o limitar que otras clases o interfaces pueden heredar de nuestra clase, veamos un ejemplo:

```
package com.dev.sealed;

public sealed class Vehiculo
permits Barco, Avion, Motocicleta {}
```

En el código estamos restringiendo la clase **Vehiculo** para que solo pueda ser heredada por las clases Barco, Avion y Motocicleta, En este caso dichas clases deben encontrarse en el mismo paquete o en el mismo módulo.

También podemos hacerlo para clases que se encuentra dentro de otros contextos veamos un ejemplo:

```
package com.dev.sealed;

public sealed class Vehiculo
permits com.dev.vehiculos.aquaticos.Barco,
        com.dev.vehiculos.aereos.Avion,
        com.dev.vehiculos.terrestres.Motocicleta {}
```

Como podemos apreciar en este caso las clases a las que se les permite la herencia se encuentran en paquetes diferentes a la clase **Vehiculo**.

Conclusión

Esta versión trajo una cantidad baja de cambios a nivel de programación, sin embargo la inclusión de las clases **sealed** es algo muy novedoso y que nos otorga más control sobre nuestro código.

Adicional a esto, cabe resaltar que el nuevo **instanceof** del cual hablamos en el capítulo anterior ya se encuentra en una segunda preview, y además de esto, la funcionalidad de **TextBlocks** ya no se encuentra en preview así que desde está versión formará parte de las próximas versiones del JDK.

Java 16, Vector API y mucho más

Invocación de métodos default usando reflection

Antes de empezar a hablar de este cambio quisiera hablarte un poco de reflection.

Reflection es una librería nativa de Java que nos permite acceder a los metadatos de nuestros archivos java, entiéndase metadatos como los métodos, variables, valores de los mismos, básicamente hablamos de poder a lo que compone una clase.

Reflection es muy utilizado en la creación de librerías genéricas y llamado dinámico de métodos. A grandes rasgos de eso se trata reflection, estos y otros temas los podemos tratar a fondo con los miembros de nuestro canal. Si te interesa hacerte miembro puedes hacer click aquí.

Ahora bien, los métodos default fueron introducidos en Java 8 y se pueden definir en las interfaces, sin embargo no existía la posibilidad de invocar dichos métodos a través de reflection, pero ahora con la llegada de Java 16 podemos hacer esto posible, veamos un ejemplo:

Así definimos un método default en una interfaz:

```
interface HolaMundo {
    default String hola() {
        return "mundo";
    }
}
```

Ahora veamos como podemos invocar este método usando reflection:

```
Object proxy = Proxy.newProxyInstance(getSystemClassLoader(), new
Class<?>[] { HolaMundo.class },
    (prox, method, args) -> {
        if (method.isDefault()) {
            return InvocationHandler.invokeDefault(prox, method, args);
        }
        // ...
    }
);
Method method = proxy.getClass().getMethod("hola");
assertThat(method.invoke(proxy)).isEqualTo("mundo");
```

El código anterior puede resultar un tanto complejo pero si te fijas lo que hacemos es cargar la interfaz `HolaMundo`, después verificar si existe un método default con la instrucción `if (method.isDefault())` y posteriormente con la clase `InvocationHandler` a través del método `invokeDefault` invocamos el método en cuestión, retornando su resultado.

Accede a momentos del día con DateTimeFormatter

Ahora puedes formatear fechas accediendo a momentos del día, veamos un ejemplo:

```
LocalTime date = LocalTime.parse("15:25:08.690791");
DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("h B");
```

El código anterior tiene una particularidad y es la letra “B” esa letra es la que nos permite acceder a momentos del día, podemos usar hasta cinco “B”, ahora veamos el resultado del código anterior:

```
3 in the afternoon
```

Imagino que con esto tu curiosidad con respecto a este tema se ha disparado, por eso te dejo un link en el que encontrarás más información, solo debes hacer click aquí.

Vector API, el cambio más grande de esta versión

La idea de esta API es proporcionar una forma de realizar cálculos vectoriales que pueda funcionar de manera más óptima (en arquitecturas de CPU compatibles) que el método tradicional de cálculos.

Para poder comprender un poco mejor veamos un ejemplo en el que multiplicamos las posiciones de dos arreglos:

```
int[] a = {1, 2, 3, 4};
int[] b = {5, 6, 7, 8};

var c = new int[a.length];

for (int i = 0; i < a.length; i++) {
    c[i] = a[i] * b[i];
}
```

Ahora veamos este mismo ejercicio pero usando VectorAPI:

```
int[] a = {1, 2, 3, 4};
int[] b = {5, 6, 7, 8};

var vectorA = IntVector.fromArray(IntVector.SPECIES_128, a, 0);
var vectorB = IntVector.fromArray(IntVector.SPECIES_128, b, 0);
var vectorC = vectorA.mul(vectorB);
vectorC.intoArray(c, 0);
```

Si analizamos el código anterior utilizamos la clase IntVectors a partir de los dos arreglos creados en las dos primeras líneas de código posteriormente usamos el método estático fromArray. El primer parámetro es el tamaño del vector, seguido por el arreglo y el desplazamiento (aquí establecido en 0). Lo más importante aquí es el tamaño del vector que estamos obteniendo a 128 bits. En Java, cada int requiere 4 byte.

Dado que tenemos un arreglo de cuatro posiciones, se necesitan 128 bits para almacenarla. Nuestro único Vector puede almacenar toda la matriz.

En ciertas arquitecturas, el compilador podrá optimizar el código de bytes para reducir el cálculo de 4 ciclos a solo 1. Estas optimizaciones benefician áreas como el aprendizaje automático y la criptografía.

Conclusión

Esta versión de Java llegó con una serie de cambios bastante interesantes que vale la pena investigar un poco más a fondo, sobre todo si tu idea es dedicarte a temas relacionados con criptos e IA, por otro lado algo que no mencioné es que los Records ahora pueden ser creados dentro de otras clases, ya que es muy probable que esta funcionalidad continúe teniendo más y más cambios.

Java 17, Un nuevo LTS con muchos cambios generales

El adiós a las Applets

Después de muchos años y después que casi todos los navegadores dejarán de brindar soporte a esta API, Java 17 anuncia que el API de Applets será eliminada por completo en un próximo release.

Sealed Clases es una realidad

Las sealed clases entraron como un preview feature en la versión 15 del JDK y también durante la versión 16, ahora son formalmente parte del JDK, puedes ir a la sección del JDK 15 para refrescar un poco de qué va esta nueva funcionalidad.

Conclusión

Esta versión tiene muchos cambios generales, sin embargo pocos son en la forma en la que programamos, eso sucede porque ya hay muchas versiones en las que se viene preparando el camino esta versión LTS solo trata de retocar algunas cosas, remover otras y nada más.

Java 18, la comunidad fue escuchada.

UTF-8 por defecto.

Antes de Java 18, la codificación de carácteres predeterminado dependía del entorno, lo que significa que JVM lo asignaba, según el entorno de ejecución, como el sistema operativo, la configuración regional del usuario y otros factores. Por ejemplo, en macOS, el juego de caracteres predeterminado es UTF-8; en Windows, el juego de caracteres predeterminado es "windows-1252" (si está en inglés).

Dado que la codificación de carácteres predeterminado no es el mismo de una máquina a otra, las API que utilizan la codificación de carácteres predeterminado pueden provocar errores o comportamientos no deseados, especialmente las API de IO como java.io.FileReader y java.io.FileWriter ambas usadas comúnmente para la manipulación de archivos.

Simple Web Server

Oracle lleva ya varios años haciendo esfuerzos para modernizar el entorno de desarrollo de Java y desde esta versión contamos con un servidor web sencillo, similar al que podemos crear con otros lenguajes como NodeJS o Go, se trata de una herramienta de comandos bastante sencilla.

Se trata del comando `./jwebserver` el cual nos permite desplegar un servidor web por defecto en el puerto 8000, dicho servidor tiene alguna limitaciones como que solo puede recibir peticiones GET o HEAD, sin embargo espero que en el futuro se haga más robusto y podamos contar con una herramienta un poco más compleja.

Aunque no dudo que resultará bastante útil para muchas cosas, te invito a que consultes la documentación del comando haciendo click aquí.

Conclusión

La versión 18 para mí es una de las más interesantes hasta este momento, sobre todo por la inclusión del servidor web, si bien son cambios pequeños en forma son bastantes grandes en fondo, también el cambio de carácteres nos pueden ahorrar muchos dolores de cabeza.

Java 19, previews emocionantes

Record Patterns

Los records ya se vienen tratando de hace algún tiempo en los nuevos releases pero ahora pueden representar un cambio aún mayor para nosotros, si este cambio se implementa con éxito en una versión posterior ya no tendremos que hacer clases con get y set, simplemente las podremos obtener con pocas líneas de código te mostraré un ejemplo:

```
public class Test {

  record Persona(String nombre, int edad) {
  }

  static void imprimirDatos(Object o) {
      if (o instanceof Persona p) {
          String nombre = p.nombre();  // get nombre()
          int edad = p.edad();  // get edad()
          System.out.println(nombre);
          System.out.println(edad);
      }
  }

  public static void main(String[] args) {
      imprimirDatos(new Persona("4SD", 6));
  }
}
```

Pattern Matching para Switch

Además de Record, parece que vamos a poder contar con una sentencia Switch más extensa y flexible, en este nuevo Switch vamos a poder agregar la verificación de tipos que realizamos con el nuevo `instanceof`, del que ya hablé en el capítulo de Java 14, explicaré más a fondo sobre esto cuando ya sea una realidad de momento es genial saber que seguimos avanzando hacía un mejor Java.

Conclusión

Además de los preview feature mencionados hay otros varios que considero bastante interesantes como son: Virtual Threads, Structured Concurrency, Vector API, entre otros.

No cabe duda que el avance es notable y el proceso de “modernización” de Java va por buen camino.

Java 20, un paso más

¿Nada nuevo?

Esta es quizás una de las versiones de Java que puede llegar a pasar un poco desapercibidas ya que no representa una gran cantidad de cambios en la forma de programar, sin embargo es una versión que le da mucha continuidad a lo que el lenguaje viene haciendo.

Por ejemplo hay varias nuevas funcionalidades que continúan en Preview Feature o están en un estado de incubación (Incubators) como lo son los Scoped Values (Incubator), Records Patterns (Second Preview) y Pattern Matching for Swtich (Fourth Preview).

Nuevos deprecados

Hay un nuevo deprecado que me pareció interesante resaltar, se trata de los constructores de la clase java.net.URL, ahora vamos a tener que crear nuevas instancias de esta clase de la siguiente manera:

Antes de Java 20:

```
URL url = new URL("https://www.4softwaredevelopers.com");
```

Desde Java 20:

```
URL url = URI.create("https://www.happycoders.eu").toURL();
```

Conclusion

Evidentemente cada versión del JDK viene con un montón de cambios ya sea en el backend de Java, en la JVM o en diferentes cuestiones como el performance, el proceso de compilación, etc.

Por ende ninguna versión del lenguaje debe ser subestimada, aunque no tengamos grandes cambios en la forma de programar muchas cosas suceden sin que nosotros nos demos cuenta de ello.

Java 21, un nuevo LTS lleno de vida

Virtual Threads

Después de varios “preview” los Virtual Threads son ahora una realidad, sin duda alguna este es uno de los cambios más importantes de Java en los últimos años.

¿Qué son los Virtual Threads?

Cuando estamos programando en Java es muy común hacer uso de Hilos (Threads) estos nos permiten crear de una forma simple y sencilla diferentes subprocesos a nivel del sistema operativo, estos son muy comunes cuando hablamos de aplicaciones web que procesan grandes cantidad de información, aplicaciones que se ejecutan en segundo plano, entre otros.

Sin embargo los hilos generalmente tienden a ser bastante costosos a nivel del sistema operativo, esto significa que por cada nueva instancia de un hilo que realizamos anteriormente un nuevo subproceso era creado en el sistema operativo, si bien esto en aplicaciones pequeñas o medianas podía no significar un gran golpe a su rendimiento, cuando hablamos de aplicaciones que deben procesos millones de peticiones, si hay un enorme problema.

Es aquí donde los Virtual Threads entran a ser una herramienta increíble, gracias a ellos podemos crear millones de subprocesos virtual sin la necesidad de que cada uno de ellos represente millones de subprocesos en el sistema operativo.

¿Cómo funcionan?

Básicamente ahora contaremos con un sofisticado sistema de “pool” que se encargará por nosotros de administrar los subprocesos que creamos, permitiendo así que no sea necesario que cada subproceso o hilo simbolice un nuevo proceso a nivel del servidor.

Este sistema es una especie de “pool” sobre el cual cada subproceso ya iniciado se encarga de verificar si hay algún otro proceso que pueda ser ejecutado sin la necesidad de abrir uno nuevo a nivel del sistema operativo, anteriormente con la clase Thread se iniciaba un nuevo subproceso en el sistema operativo, ahora antes de realizar este proceso la JVM se encarga de realizar un proceso de validaciones para quien es el encargado de ejecutar el subproceso que iniciamos.

Virtual Threads en acción

Veamos ahora un ejemplo de un Virtual Thread, voy a simular que iniciamos 1000 subprocesos, cada una de las cuales espera un segundo (para simular el acceso a una API externa) y luego devuelve un resultado (un número aleatorio en el ejemplo).

```
public class Task implements Callable<Integer> {

  private final int number;

  public Task(int number) {
    this.number = number;
  }

  @Override
  public Integer call() {
    System.out.printf(
        "Thread %s - Task %d waiting...%n",
        Thread.currentThread().getName(), number);

    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.printf(
          "Thread %s - Task %d canceled.%n",
           Thread.currentThread().getName(), number);
      return -1;
    }

    System.out.printf(
        "Thread %s - Task %d finished.%n",
         Thread.currentThread().getName(), number);
    return ThreadLocalRandom.current().nextInt(100);
  }
}
```

Ahora medimos cuánto tiempo le toma a un grupo de 100 subprocesos que no utilizan Virtual Threads.

```
try (ExecutorService executor =
     Executors.newFixedThreadPool(100)) {
  List<Task> tasks = new ArrayList<>();
  for (int i = 0; i < 1_000; i++) {
    tasks.add(new Task(i));
  }

  long time = System.currentTimeMillis();

  List<Future<Integer>> futures =
  executor.invokeAll(tasks);

  long sum = 0;
  for (Future<Integer> future : futures) {
    sum += future.get();
  }

  time = System.currentTimeMillis() - time;

  System.out.println("sum = " + sum + "; time = " +
  time + " ms");
}
```

ExecutorService es autoclosable desde Java 9 por eso podemos instanciarlo en un try-with-resources, el try-with-resources existe desde Java 7 y es bastante útil para cerrar automáticamente recursos que procesamos dentro de un try-catch.

El código que acabamos de crear tarda alrededor de 10 segundos en ejecutarse, ya que cada hilo tenía que procesar diez tareas secuencialmente, cada una de las cuales duraba aproximadamente un segundo.

Ahora para probar el poder de los Virtual Threads, vamos a cambiar la siguiente sentencia:

```
Executors.newFixedThreadPool(100);
```

Por la sentencia:

```
Executors.newVirtualThreadPerTaskExecutor();
```

Con ese sencillo cambio hemos sido capaces de optimizar muchísimo el código que creamos pasando de tomar cerca de 10 segundos en ejecutarse a tardar solo un segundo, esto es gracias al proceso que hay detrás de los Virtual Threads, solo queda imaginar las posibilidades.

Colecciones secuenciadas

Desde Java 21 vamos a poder acceder a elementos de nuestras colecciones de una forma mucho más simple de lo que lo hacíamos en el pasado, veamos un ejemplo.

Antes de Java 21:

```
var last = list.get(list.size() - 1);
```

Desde Java 21:

```
var last = list.getLast();
```

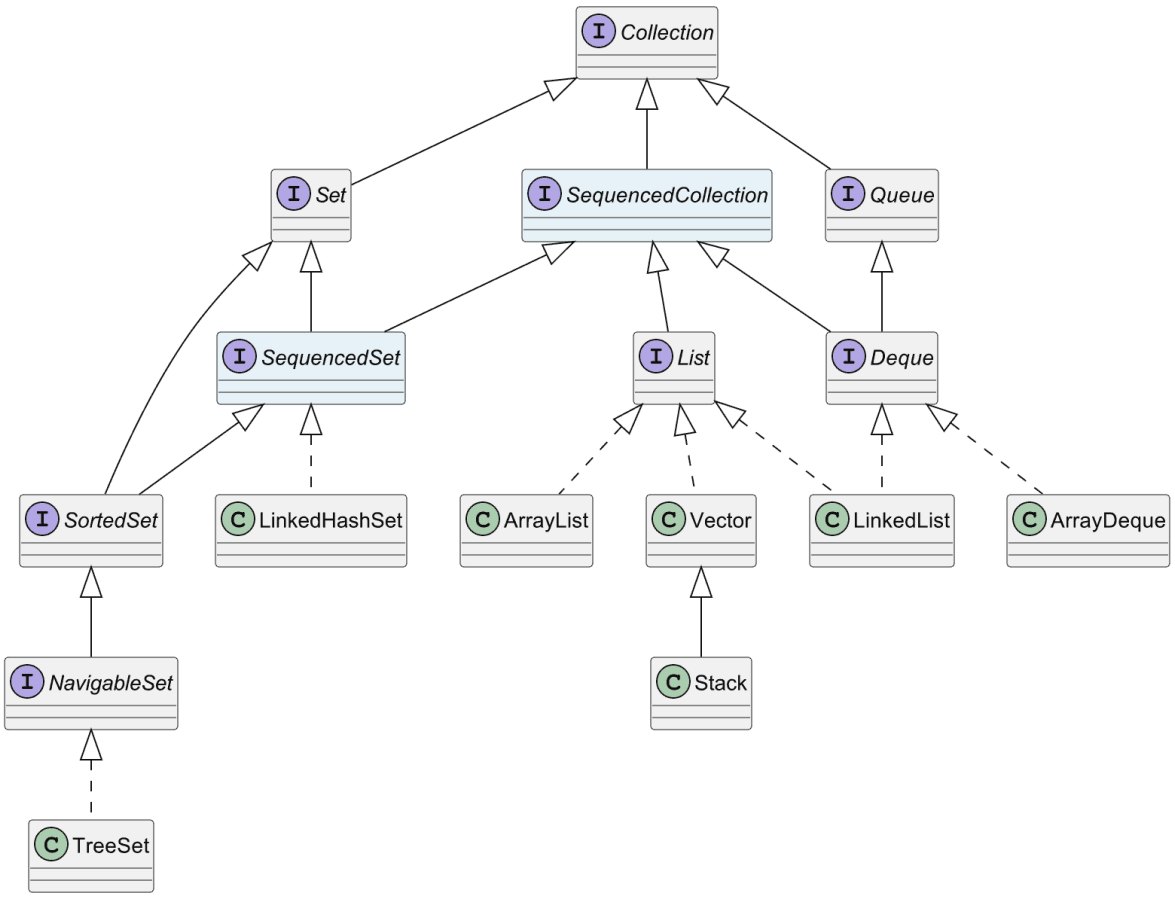
Además de la adición del método `getLast()`, también se agregó el método `getFirst().`
Todo esto es parte de una nueva interfaz llamada `SequencedCollection` esta interfaz también afectó interfaces como, `List`, `SortedSet`, y `NavigableSet`, por consecuencia también todas las clases que implementan estas interfaces ahora cuentan con estos métodos clases como, `ArrayList`, `LinkedList`, `LinkedHasSet`, etc.

Adicionalmente se agregaron otros métodos que pueden resultar útiles

```
void addFirst(E)
void addLast(E)
E removeFirst()
E removeLast()
SequencedCollection reversed();
```

También se agregó una nueva interfaz que hereda de `SequencedCollection` llamada `SequenceSet` esta interfaz básicamente sobreescribe el método reversed y aplica una lógica específica.

Después de haber agregado esta nueva interfaz la jerarquía de las colecciones ahora se representa de la siguiente manera:



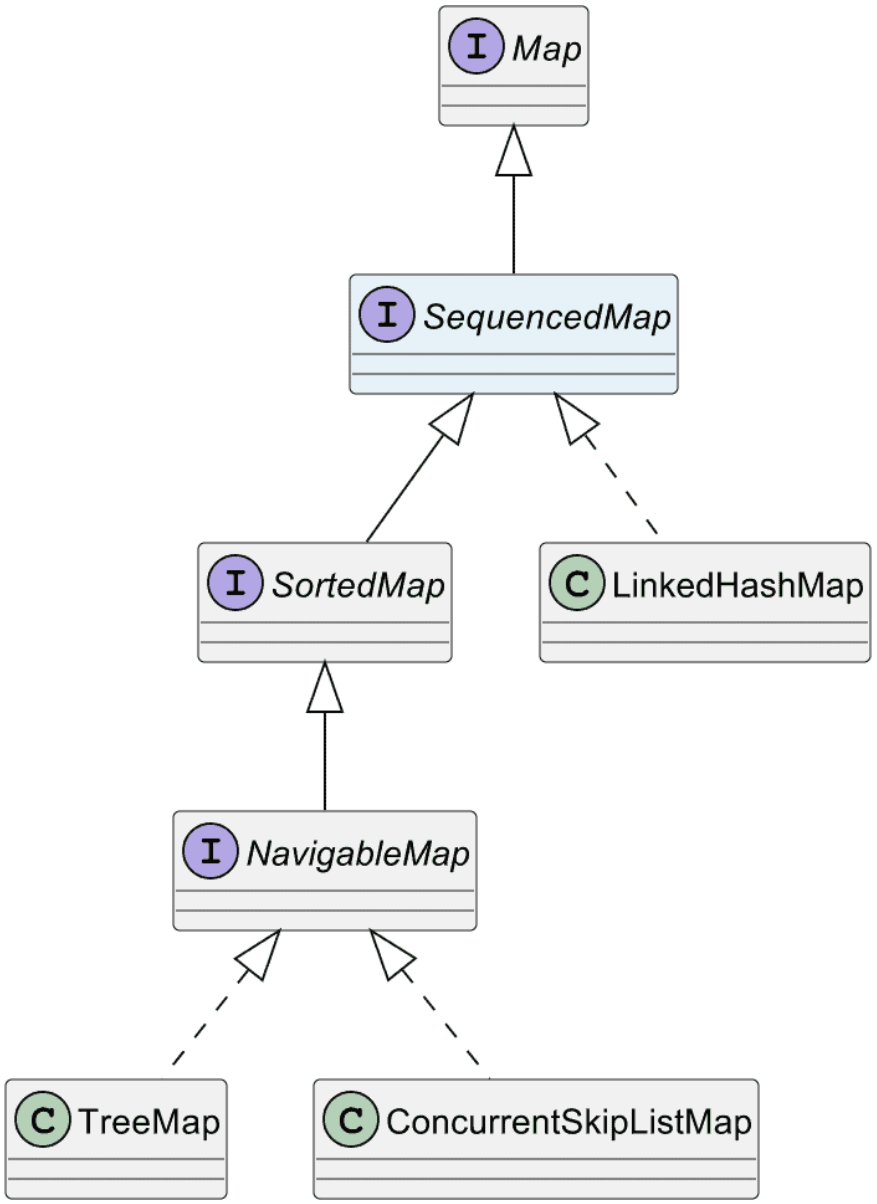
Mapas secuenciados

En Java, los mapas y listas son dos jerarquías de clases distintas, sin embargo también se agregaron métodos similares para los mapas y sus implementaciones, veamos:

```
Entry<K, V> firstEntry();
Entry<K, V> lastEntry();
Entry<K, V> pollFirstEntry();
Entry<K, V> pollLastEntry();
V putFirst(K, V)
```

```
V putLast(K, V)
SequencedMap<K, V> reversed()
SequencedSet sequencedKeySet()
SequencedCollection<V> sequencedValues()
SequencedSet<Entry<K,V>> sequencedEntrySet()
```

Evidentemente esto también impacta la jerarquía de clases de los mapas, veamos:



Record Patterns

Desde su introducción en Java 19 los Record son de esas cosas que muchos hemos pedido a gritos, básicamente porque simplifican mucho la sintaxis del lenguaje, ya podemos decir adiós al exceso de Pojos y podemos dar la bienvenida a una nueva forma de crear clases simples en Java.

Veamos algunos ejemplos.

Así podemos definir un Record:

```
public record Position(int x, int y) {}
```

Ahora veamos cómo combinar un Record con `instanceof`:

```
public void print(Object o) {
  if (o instanceof Position p) {
    System.out.printf("o es una posicion: %d/%d%n",
    p.x(), p.y());
  } else if (object instanceof String s) {
    System.out.printf("o es un string: %s%n", s);
  } else {
    System.out.printf("o es otra cosa: %s%n",
    o);
```

```
  }
}
```

Incluso podríamos definir el Record con sus propiedad al inicio de la comparación, veamos:

```
public void print(Object o) {
  if (o instanceof Position(int x, int y)) {
    System.out.printf("o is a position: %d/%d%n",
    x, y);
  } else if (o instanceof String s) {
    System.out.printf("o is a string: %s%n", s);
  } else {
    System.out.printf("o is something else: %s%n",
    o);
  }
}
```

Switch Pattern Matching

En Java 21 también se hizo oficial la llegada del Pattern Matching para Switch esto es algo que llega a Switch en combinación con los Record y que nos va a permitir extender las funcionalidades de un Switch, veamos un ejemplo:

```
Object obj = getObject();

switch (obj) {
  case String s when s.length() > 5 ->
  System.out.println(s.toUpperCase());
  case String s -> System.out.println(s.toLowerCase());
  case Integer i -> System.out.println(i * i);
  case Position(int x, int y) ->
```

```
  System.out.println(x + "/" + y);
  default -> {}
}
```

Como vemos ahora el switch ha extendido significativamente sus funcionalidades incluyendo palabras clave como when, sirviendo también para la comparación de tipos de objetos, algo que antes solo era posible con `instanceof`, veamos este mismo ejemplo pero sin switch pattern matching:

```
Object obj = getObject();

if (obj instanceof String s && s.length() > 5)
    System.out.println(s.toUpperCase());
else if (obj instanceof String s)
    System.out.println(s.toLowerCase());
else if (obj instanceof Integer i)
    System.out.println(i * i);
else if (obj instanceof Position(int x, int y))
    System.out.println(x + "/" + y);
```

Qualified Enum Constants

Antes de Java 21 los Enum solo podían ser usados con el patrón tradicional de Switch, pero ahora podemos potenciarlos significativamente. Veamos un ejemplo, vamos a combinar una “sealed class” junto con enums y switch.

```
public sealed interface Direction permits
CompassDirection, VerticalDirection {}
```

```
public enum CompassDirection implements Direction {
```

```
NORTH, SOUTH, EAST, WEST }
```

```
public enum VerticalDirection implements Direction {
UP, DOWN }
```

Hasta Java 20 nuestro código se vería así:

```
void codigoJava20(Direction direction) {
  switch (direction) {
    case CompassDirection  d when d ==
         CompassDirection.NORTH ->
         System.out.println("Norte");
    case CompassDirection  d when d ==
         CompassDirection.SOUTH ->
         System.out.println("Sur");
    case CompassDirection  d when d ==
         CompassDirection.EAST  ->
         System.out.println("Este");
    case CompassDirection  d when d ==
         CompassDirection.WEST  ->
         System.out.println("Oeste");
    case VerticalDirection d when d ==
         VerticalDirection.UP   ->
         System.out.println("Arriba");
    case VerticalDirection d when d ==
         VerticalDirection.DOWN ->
         System.out.println("Abajo");
    default -> throw new
         IllegalArgumentException("Desconocido " +
         direction);
  }
}
```

Desde Java 21 podemos hacer algo mucho mejor:

```
void codigoJava21(Direction direction) {
  switch (direction) {
    case CompassDirection.NORTH ->
      System.out.println("Norte");
    case CompassDirection.SOUTH ->
      System.out.println("Sur");
    case CompassDirection.EAST  ->
      System.out.println("Este");
    case CompassDirection.WEST  ->
      System.out.println("Oeste");
    case VerticalDirection.UP   ->
      System.out.println("Arriba");
    case VerticalDirection.DOWN ->
      System.out.println("Abajo");
  }
}
```

Conclusión

Java 21 es el nuevo LTS de Java que incorpora un cambio que representa un antes y un después para este lenguaje, me refiero a los virtual threads, sin duda alguna desde que empecé este eBook hasta la fecha he podido ver cómo este lenguaje continúa su evolución y se mantiene firme en el camino que emprendió hace ya varios años.

No hay un final

A lo largo de este libro compartí contigo información relevante sobre Java, analizamos su popularidad, sus cambios más importantes a lo largo de las versiones, su historia y algunas cosas más.

Cuando empecé a escribir este material quería apegarme a los principios de la construcción de software ágil creando un mínimo producto viable, el cual pudiera ir ajustando y mejorando con el pasar del tiempo.

Bajo esa premisa se me ocurrió que lo mejor era tratar de publicar actualizaciones de forma constante para mantenerte atraído al lenguaje y al libro, en un periodo de seis meses vas a recibir en tu correo electrónico una nueva versión del libro que contendrá mejoras en aspectos como la redacción, ortografía, etc.

Además de una nueva sección con la versión de Java que se haya lanzado o con cualquier hecho relevante sobre el lenguaje o sobre la comunidad que sienta que debes saber.

Personalmente no quiero que te vayas sin primero recomendarte todo nuestro trabajo puedes buscarnos en Youtube como 4SoftwareDevelopers (escribelo junto). Ahí verás todo el contenido gratuito que tenemos a tu disposición.

Nunca pensé que escribiría un libro, uno con muchos aspectos por mejorar pero que me llena de orgullo y no precisamente un orgullo que alimenta mi ego, sino un orgullo que me inspira a continuar con esta labor.

Sin temor a equivocarme puedo decirte que NADIE en el mercado te daría 2 beneficios de por vida por ese precio, simplemente porque no es rentable, el precio es solo una estrategia que creí conveniente para lograr vender el libro y dar a conocer nuestra comunidad de casi 15k a la fecha.

Por favor toma este libro y compartelo con alguien, elige a alguien importante para ti que le interese el tema y envíalo o prestaselo (si es que tienes una copia física), dale a conocer nuestro canal de Youtube y redes sociales. Ese es el primer paso para lograr crear una comunidad sólida.

Queremos dedicarnos a esto al 100%, hablar de otros lenguajes, explorar más tecnologías, hacer mejores, vídeos, comprar equipo, entre otras cosas. Pero solos no lo lograremos, te necesitamos en esto y tu apoyo no solo garantiza tu educación, si no que también garantiza la educación de miles de personas a corto y largo plazo.

Gracias por haber llegado hasta el final, puedes enviarme un email a jordy.rodriguez@4softwaredevelopers.com con cualquier apreciación que tengas sobre este material, será un placer para mí leer sus opiniones y seguir creciendo paso a paso.

Nos vemos en la siguiente actualización.