

**TRABAJO 3**

TEMA: Expresiones Lambda y Stream en Java 8

CURSO: Herramientas para el Desarrollo de Software.

PROFESOR: Gustavo Coronel

ALUMNO: Gabriel Gonzalo Mercati Rojas

2017

INTRODUCCION

En lenguajes orientados a objetos, la clave de todo es el objeto, pero sucede lo contrario en los lenguajes de programación funcionales donde son éstas donde recae toda la importancia, en ellas se pueden pasar alrededor como si fueran variables esto se ve en Haskell, javascript y otros lenguajes funcionales. Los lenguajes funcionales tienen otras construcciones que no están fácilmente disponibles en los lenguajes orientados a objetos como son las expresiones lambda.

Las características más importantes de Java SE 8 son la adición de Expresiones Lambda y la API Stream. Con la adición de expresiones lambda podemos crear código más conciso y significativo, además de abrir la puerta hacia la programación funcional en Java, en donde las funciones juegan un papel fundamental. Por otro lado, la API Stream nos permite realizar operaciones de tipo filtro/mapeo/reducción sobre colecciones de datos de forma secuencial o paralela y que su implementación sea transparente para el desarrollador. Lambdas y Stream son una combinación muy poderosa que requiere un cambio de paradigma en la forma en la que hemos escrito código Java hasta el momento.

Una expresión Lambda es una función anónima, básicamente es un método abstracto es decir un método que sólo está definido en una interfaz, pero no implementado, y esa es la clave de las funciones lambda, al no estar implementado, el programador lo puede implementar dónde el crea conveniente sin haber heredado de la interfaz.

Programación Orientada a Objetos

La programación orientada a objetos (POO, u OOP según sus siglas en inglés) es un paradigma de programación que viene a innovar la forma de obtener resultados. Los objetos manipulan los datos de entrada para la obtención de datos de salida específicos, donde cada objeto ofrece una funcionalidad especial.

Muchos de los objetos pre-diseñados de los lenguajes de programación actuales permiten la agrupación en bibliotecas o librerías, sin embargo, muchos de estos lenguajes permiten al usuario la creación de sus propias bibliotecas.

Está basada en varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento.

Los objetos son entidades que tienen un determinado "estado", "comportamiento (método)" e "identidad":

La identidad es una propiedad de un objeto que lo diferencia del resto; dicho con otras palabras, es su identificador (concepto análogo al de identificador de una variable o una constante).

Un objeto contiene toda la información que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases e incluso frente a objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos. A su vez, los objetos disponen de mecanismos de interacción llamados métodos, que favorecen la comunicación entre ellos. Esta comunicación favorece a su vez el cambio de estado en los propios objetos. Esta característica lleva a tratarlos como unidades indivisibles, en las que no se separa el estado y el comportamiento.

Los métodos (comportamiento) y atributos (estado) están estrechamente relacionados por la propiedad de conjunto. Esta propiedad destaca que una clase requiere de métodos para poder tratar los atributos con los que cuenta. El programador debe pensar indistintamente en ambos conceptos, sin separar ni darle mayor importancia a alguno de ellos. Hacerlo podría producir el hábito erróneo de crear clases contenedoras de información por un lado y clases con métodos que manejen a las primeras por el otro. De esta manera se estaría realizando una "programación estructurada camuflada" en un lenguaje de POO.

La programación orientada a objetos difiere de la programación estructurada tradicional, en la que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida. La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones, y en segundo lugar en las estructuras de datos que esos procedimientos manejan. En la programación estructurada solo se escriben funciones que procesan datos. Los programadores que emplean POO, en cambio, primero definen objetos para luego enviarles mensajes solicitándoles que realicen sus métodos por sí mismos.

Programación Funcional

En ciencias de la computación, la programación funcional es un paradigma de programación declarativa basado en el uso de funciones matemáticas, en contraste con la programación imperativa, que enfatiza los cambios de estado mediante la mutación de variables. La programación funcional tiene sus raíces en el cálculo lambda, un sistema formal desarrollado en los años 1930 para investigar la definición de función, la aplicación de las funciones y la recursión. Muchos lenguajes de programación funcionales pueden ser vistos como elaboraciones del cálculo lambda.

En la práctica, la diferencia entre una función matemática y la noción de una "función" utilizada en la programación imperativa, es que las funciones imperativas pueden tener efectos secundarios, como cambiar el valor de cálculos realizados previamente. Por esta razón carecen de transparencia referencial, es decir, la misma expresión sintáctica puede resultar en valores diferentes en varios momentos de la ejecución del programa. Con código funcional, en contraste, el valor generado por una función depende exclusivamente de los argumentos alimentados a la función. Al eliminar los efectos secundarios se puede entender y predecir el comportamiento de un programa mucho más fácilmente. Ésta es una de las principales motivaciones para utilizar la programación funcional.

Introducción de la Programación Funcional en Java 8

La programación funcional puede ayudarnos a crear software más robusto, mantenible y fácil de testear. Java en su versión 8 permite usar la potencia de la programación funcional sin abandonar su orientación a objetos.

Pero, ¿qué hace que lenguajes como C, C++ o Java adopten la programación funcional?

Como dijo C.A.R. Hoare:

**“Hay dos formas de diseñar software: una forma es hacerlo tan simple que obviamente no haya deficiencias, y la otra es hacerlo tan complicado que no haya deficiencias obvias. El primer método es mucho más difícil”.**

Desde que empiezas a programar con Java te enseñan qué es un lenguaje de programación orientado a objetos y que Java es uno de ellos. Te enseñan a programar de manera imperativa y es como has programado desde entonces. Con estas herramientas tienes que crear la mejor solución posible y hasta hoy no ha ido mal. Pero, ¿y si te digo que hay una forma mejor de llegar a esas soluciones?

En la programación declarativa no definimos cómo queremos resolver un problema, sino que definimos cuál es el problema.

**¿Cuál es la importancia de las Expresiones Lambda?**

Con la introducción de las expresiones lambda en java se puedan aprovechar los beneficios del lenguaje funcional dentro de un lenguaje orientado a objetos. Las expresiones lambda proporcionan un medio para escribir códigos claros y concisos con el uso de expresiones.

LA SINTAXIS DE UN EXPRESIÓN SUELE DARSE DE LA SIGUIENTE FORMA:

(argumentos)->{cuerpo}

Por ejemplo:

(arg1, arg2…) -> { cuerpo}

Esta sintaxis puede cambiar tanto como para los argumentos o el cuerpo de la expresión Lambda de acuerdo a lo siguiente.

**PARA LOS ARGUMENTOS:**

Los argumentos de una función Lambda pueden ser declarados explícitamente o a su vez pueden ser inferidos por el compilador de acuerdo al contexto, cuando digo de acuerdo al contexto, me refiero a que si el método que estamos implementado recibe una cadena el compilador asumirá que el argumento es una cadena y así con el tipo de dato que estuviéramos recibiendo en el método.

Entonces un argumento se puede declarar explícitamente, esto se refiere al tipo de dato, por ejemplo: (int x)-> {cuerpo} y si tiene más parámetros sería (int x, int y….)-> {cuerpo}.

También se lo puede hacer de forma implícita, por ejemplo: (x)-> {cuerpo}, de echo si existe un sólo argumento y se lo declara de forma implícita puede ir incluso sin paréntesis por ejemplo x -> {cuerpo}, si se declara más de un parámetro obligatoriamente deben ir seguido de comas y entre paréntesis.

Por último, puede haber expresiones Lambda en las que no haya argumentos y se expresan de la siguiente forma: ()-> {cuerpo}.

**PARA EL CUERPO DE LA EXPRESIÓN:**

El cuerpo de la expresión puede ir o no dentro de llaves esto puede variar como se ve en los siguientes ejemplos.

Es obligatorio que el cuerpo de una expresión vaya entre llaves en los siguientes casos:

Cuando devuelve más de un valor o la sentencia tiene más de una instrucción, por ejemplo:

(int a, int b) -> { return a + b; }.

() -> { return 3.1415 }.

Mientras que cuando devuelve un sólo valor no es obligatorio, de todas maneras el compilador no muestra error si se pone entre llaves por ejemplo, (aunque hay algunas excepciones):

() -> 10

n -> System.out.print(n + ” “)

Pero se la puede encerrar dentro de llaves, para esto es necesario añadir al final de la sentencia un punto y coma:

n -> {System.out.print(n + ” “);}

() -> new ArrayList<Integer>().

Uso de Streams en Java 8

¿Qué haríamos sin las colecciones? Casi todas las aplicaciones de Java crean y procesan colecciones. Son esenciales para muchas tareas de programación: permiten agrupar y procesar datos. Por ejemplo, el desarrollador podría querer crear una colección de transacciones bancarias para representar el extracto de un cliente. Luego, tal vez quiera procesar toda la colección para averiguar cuánto dinero gastó el cliente. A pesar de su importancia, el procesamiento de colecciones en Java dista de ser perfecto.

En primer lugar, los patrones de procesamiento de colecciones típicos son similares a las operaciones del estilo de las que se usan en SQL para "buscar" (por ejemplo, buscar la transacción de mayor valor) o "agrupar" (por ejemplo, agrupar todas las transacciones relacionadas con compras de almacén). La mayoría de las bases de datos permiten establecer operaciones como esas de manera declarativa. Por ejemplo, la siguiente consulta de SQL permite buscar la identificación de la transacción de mayor valor: "SELECT id, MAX(value) from transactions".

Como puede verse, no es necesario programar cómo calcular el valor máximo (por ejemplo, mediante bucles y una variable para hacer el seguimiento del mayor valor). Solo se expresa qué resultado se espera. Así, debemos preocuparnos menos acerca de cómo codificar explícitamente las consultas; el lenguaje lo hace por nosotros. ¿Por qué no se puede hacer algo parecido con las colecciones? ¿Quién no se ha encontrado codificando esas operaciones con bucles una y otra vez?

En segundo lugar, ¿cómo podemos hacer para procesar colecciones realmente grandes con eficiencia? Idealmente, para acelerar el procesamiento conviene trabajar con arquitecturas de núcleos mútiples. No obstante, programar código paralelo es una tarea ardua y en la que es fácil cometer errores.

Los diseñadores de la interfaz API de Java han incorporado en su actualización una nueva abstracción denominada Stream, que permite procesar datos de modo declarativo. Más aún, los streams permiten aprovechar las arquitecturas de núcleos múltiples sin necesidad de programar líneas de código multiproceso.

**Programación con streams: primeros pasos**  
Empecemos por ver algo de teoría. ¿Cuál es la definición de stream? En pocas palabras, podría decirse que es una "secuencia de elementos de un origen que admite operaciones concatenadas". Ahora desglosemos la definición:

* **Secuencia de elementos:** Un stream brinda una interfaz para un conjunto de valores secuenciales de un tipo de elemento particular. No obstante, los streams no almacenan elementos; estos se calculan cuando se recibe la solicitud correspondiente.
* **Origen:** Los streams toman su insumo de un origen de datos, como colecciones, matrices o recursos de E/S.
* **Operaciones concatenadas:** Los streams admiten operaciones estilo SQL y operaciones comunes a la mayoría de los lenguajes de programación funcionales, como filter, map, reduce, find, match y sorted, entre otras.

Más aún, las operaciones de los streams tienen dos características fundamentales que las distinguen de las operaciones con colecciones:

* **Estructura de proceso:** Muchas operaciones de stream devuelven otro stream. Así, es posible encadenar operaciones para formar un proceso más abarcador. Esto, a su vez, permite lograr ciertas optimizaciones, por ejemplo, mediante las nociones de "pereza"*(*laziness*)* y "corte de circuitos"*(*short-circuiting*)*, que analizaremos más adelante.
* **Iteración interna:** A diferencia del trabajo con colecciones, en que la iteración es explícita (iteración externa), las operaciones del stream llevan a cabo la iteración tras bambalinas.

