Expresiones Lambda y Stream API

Equipo 4

Abad Mendizabal, Alessandra Angela

Benavente Valdez, Percy Justo

Cuizano Cautivo, Silvia Yulisa

Huaylinos Suárez, Bruno Antonio

Junio 2022

EXPRESIONES LAMBDA

Historia y Definición

Alonzo Church

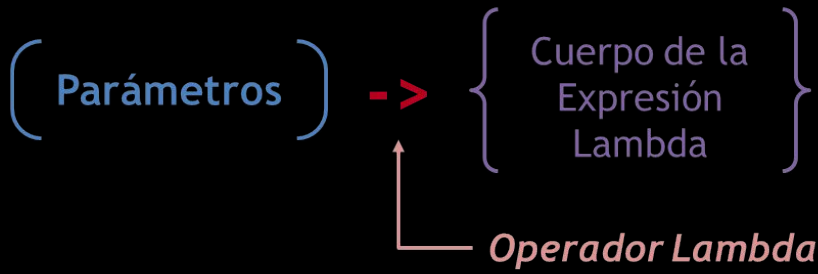
“Cálculo lambda” un sistema formal en lógica matemática.

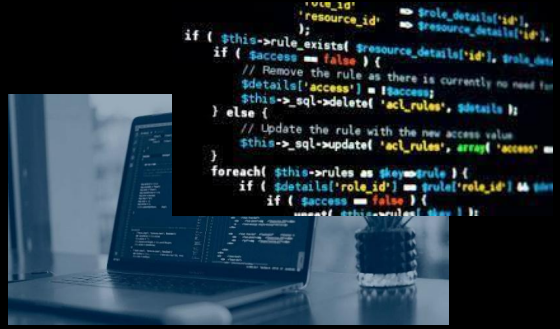
Una expresión lambda representa una función anónima.

$$\lambda x \rightarrow x+x$$

Función anónima que toma un número x y devuelve el resultado x + x.

Expresión Lambda





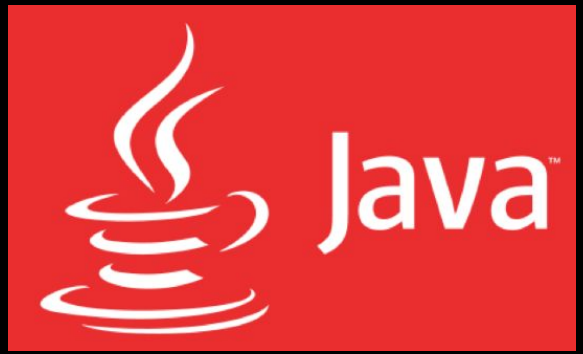
Una expresión lambda se compone de un conjunto de parámetros, un operador lambda (->) y un cuerpo de la función.

Sintaxis

```
() -> System.out.println("Hello Lambda")

x -> x + 10

(int x, int y) -> { return x + y; }

(String x, String y) -> x.length() - y.length()

(int a, int b) -->  a + b

(int a) --> a +  1

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

() --> new ArrayList();
```

¿Por qué Java necesita Expresiones Lambda?

- Java necesita cambios para simplificar la **codificación paralela**.
- Es de **utilidad** para evitar tener que escribir métodos que solo utilizamos una vez.
- **Simplifica** cómo pasar comportamiento como un parámetro (podemos pasar expresiones lambda a métodos como argumentos).



Interfaces Funcionales

```java
@FunctionalInterface
public interface Runnable {
      public abstract void run();
}

Runnable r = () -> System.out.println("Hello Lambda");
```

Una interfaz funcional es una interfaz con un único método abstracto.

```java
public interface MiInterfaz {
    default void saluda() {
        System.out.println("Un saludo!");
    }
    public abstract int calcula(int dato1, int dato2);
}
```

Tipos de Expresiones Lambda

Consumer <T> *Consumidores*	Aceptan un solo valor y no devuelven valor alguno.
Supplier <T> *Proveedores*	Expresiones que no tienen parámetros pero devuelven un resultado.
Predicate <T> *Predicados*	Recibe un argumento tipo T y retorna un valor lógico.
Function <T,R> *Funciones*	Aceptan un argumento y devuelven un valor como resultado, cuyos tipos no tienen porqué ser iguales.
UnaryOperator <T> *Operadores Unarios*	Recibe un argumento T y retorna un valor del mismo tipo T.
BinaryOperator <T> *Operadores Binarios*	Recibe dos argumentos y retorna un resultado, todos del mismo tipo.

Consumidores

```
String message -> System.out.println(message);
```

BiConsumidores

```
(String key, String value) -> System.out.println("Key: %s, value: %s%n", key, value);
```

Proveedores

```
() -> return createRandomInteger()
```

Predicados

```
String message -> message.length > 50
```

BiPredicados

```
(path, attr) -> String.valueOf(path).endsWith(".js") && attr.size() > 1024
```

Funciones

```
Order persistedOrder -> persistedOrder.getIdientifier();
```

BiFunciones

```
(Address address, String name) -> new Person(name, address);
```

Operadores Unarios

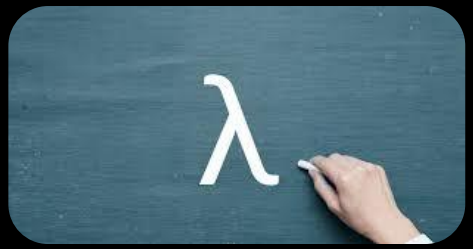
```
String message -> message.toLowerCase()
```

Operadores Binarios

```
(String message, String anotherMesssage) -> message.concat(anotherMessage);
```

Referencia a Métodos

Las referencias a los métodos nos permiten reutilizar un método como expresión lambda.



referenciaObjetivo::nombreDelMetodo

Con las referencias a los métodos se ofrece una anotación más rápida para expresiones lambda simples y existen 3 tipos diferentes:

- Métodos estáticos.
- Métodos de instancia de un tipo.
- Métodos de instancia de un objeto existente.

```
File f -> f.canRead();
```

```
File::canRead
```

Referencia a Métodos

Método Estático

```
(String info) -> System.out.println(info) // Expresión lambda sin referencias.
System.out::println // Expresión lambda con referencia a método estático.
```

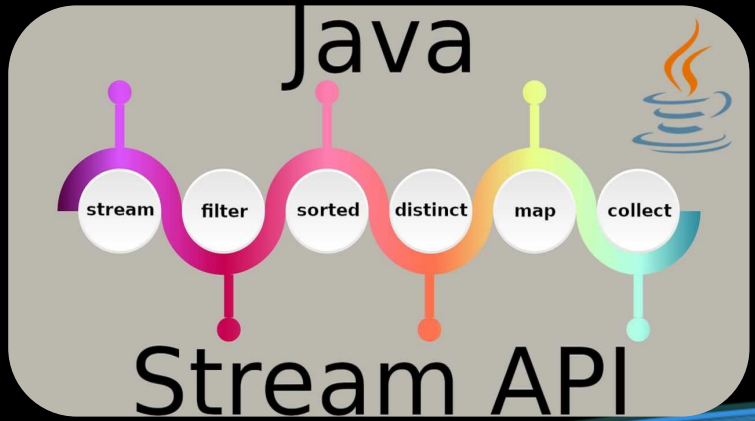
Método de un Tipo

```
(Student student, int registryIndex) -> student.getRegistry(registryIndex) // Expresión lambda sin referencias.
Student::getRegistry // Expresión lambda con referencia a método de un tipo.
```

Método de un Objeto Existente

```
Student student -> getMarks(student) // Expresión lambda sin referencias.
this::getMarks // Expresión lambda con referencia a método de un objeto existente.
```

STREAM API



EXPRESION STREAM

Stream es una librería que facilita mucho el trabajo con collections, como List y Sets, y consiste en solo llamar un método. names.stream()

Luego de esto se llama al resto de las funciones.

Filter

Esta función es muy útil para filtrar acorde a una condición, la cual tiene que ser de tipo boolean. Siguiendo con el ejemplo de los nombres, digamos que necesitamos solo los nombres que contengan la letra “o”:

```
names.stream().filter(name-> name.contains("o")).forEach(System.out::println);
```

dando como resultado:

John

John

Mohammado

Mohammado

Map

La map() operación toma una expresión lambda como su único argumento y usa esta expresión para transformar el valor o el tipo de cada elemento en la secuencia. Por ejemplo, lo siguiente nos da una nueva secuencia, donde cada String se ha convertido a mayúsculas:

```
names.stream().map(String::toUpperCase).forEach(System.out::println);
```

```
ALEX
ALEX
ALEX
JOHN
JOHN
MARIAM
MOHAMMADO
MOHAMMADO
VINCENT
```

O digamos que, de la lista de nombres del ejemplo anterior, quisieras tener un arreglo (array) del largo (length) de cada nombre:

```
names.stream() .map(name -> name.length()).collect(Collectors.toList());
```

Aquí sin afectar al array de names utilizado en el ejemplo anterior, se puede obtener otro arreglo (array) con el respectivo largo (lenght) de cada nombre.

Sorted

Esta función cumple el rol de ordenar el arreglo (array) acorde a una regla de comparación.

```
names.stream().sorted(Comparator.naturalOrder()) .forEach(System.out::println);
```

En este caso usamos Comparator, que tiene varios métodos que nos ayudan a comparar dos objetos, por ejemplo, naturalOrder, que en este caso ordenaría los nombres en orden alfabético. También podemos usar reverseOrder que ordenará los nombres en sentido contrario. El output final sería:

```
Alex
Alex
Alex
John
John
Mariam
Mohammado
Mohammado
Vincent
```

Pero digamos que también quisiéramos hacer otro tipo de comparación, por ejemplo, con el largo del nombre. En este caso podríamos llamar a la función comparing que toma como parámetro una lambda.

```
names.stream() .sorted(Comparator.comparing(String::length)).forEach(System.out::println);
```

```
John
John
Alex
Alex
Alex
Mariam
Vincent
Mohammado
Mohammado
```

Por último, en caso de que deseara agregar n filtros más, por ejemplo, quisiera ordenar los nombres por su largo y luego ordenarlos por orden alfabético, entonces, primero compararía el largo del String y luego, en el caso de que los String fuesen de igual largo, compararía en orden alfabético. Para hacer esto, se tendría que hacer un llamado a thenComparing:

```
names.stream().sorted( Comparator.comparing(String::length).thenComparing(Comparator.naturalOrder())
).forEach(System.out::println);
```

Alex

Alex

Alex

John

John

Mariam

Vincent

Mohammado

Mohammado

thenComparing funciona para agregar más filtros y se puede usar n cantidad de veces(comparing().thenComparing().thenComparing()…thenComparing()), dejando prioridad a los filtros anteriores, como en el caso anterior se le dio prioridad al largo del String, y luego se compararon según el orden natural, es importante que se la primera comparación sea solo comparing y luego llamar a thenComparing, sino no compilaría.

Distinct

Distinct ayuda a evitar datos duplicados y su uso es muy sencillo. Digamos que en el ejemplo anterior queremos ver los nombres sin que estos se repitan:

```
names.stream().distinct().forEach(System.out::println);
```

El resultado se mostraría así:

John

Mariam

Alex

Mohammado

Vincent

Limit

La función Limit ayuda a reducir el tamaño del arreglo (array) al número de registros que necesites:

```
names.stream().limit(5).forEach(System.out::println);
```

El resultado se mostraría así:

John

John

Mariam

Alex

Mohammado

Conclusiones

- Nos acerca a la programación funcional.
- Hace nuestro código más preciso y legible, mejorando, en consecuencia, su mantenibilidad.
- Su utilización junto con la API Stream hace más fácil la ejecución concurrente de tareas.
- Los Streams facilitan mucho trabajar con grandes cantidades de datos de forma muy prolija a la hora de escribirlo en el código.
- También tiene beneficios en cuanto a rendimiento.

BIBLIOGRAFÍA

Java Lambda Expressions: Consumer, Supplier and Function. (2022). Thedeveloperblog.com. https://thedeveloperblog.com/lambda-java

Luis, J. (2015, December 4). *Expresiones Lambda con Java 8 - Adictos al trabajo*. Adictos al Trabajo. https://www.adictosaltrabajo.com/2015/12/04/expresiones-lambda-con-java-8/

Jessica, T. (2017, November 14). Java 8 for Android Development: Stream API and Date & Time Libraries https://code.tutsplus.com/tutorials/java-8-for-android-development-stream-api-and-date-time-libraries--cms-29904

Luis, B. Comenzando con Lambdas y Streams en Java https://www.cleveritgroup.com/blog/comenzando-con-lambdas-y-streams-en-java