Expresiones Lambda y Streams

Expresiones lambda y sus tipos

Se integran desde la JDK 1.8, para incorporar programación funcional, se conocen también como funciones anónimas, y básicamente implementan métodos sin necesidad de una clase. Puede recibir varios parámetros y retorna uno.

Sintaxis

```
Sin parámetros:    ()->sentencia
Con un parámetro:  Parámetro -> sentencia
Con más de un parámetro: (parametro1,parametro2) ->sentencia
Con más de una sentencia: (Parámetros) ->{sentencia 1;
                                          sentencia 2;}
```

Su uso mayormente es para implementar métodos de interfaces funcionales, entendiendo una interfaz funcional como aquella que solo posee un método abstracto, @FunctionalIterface.

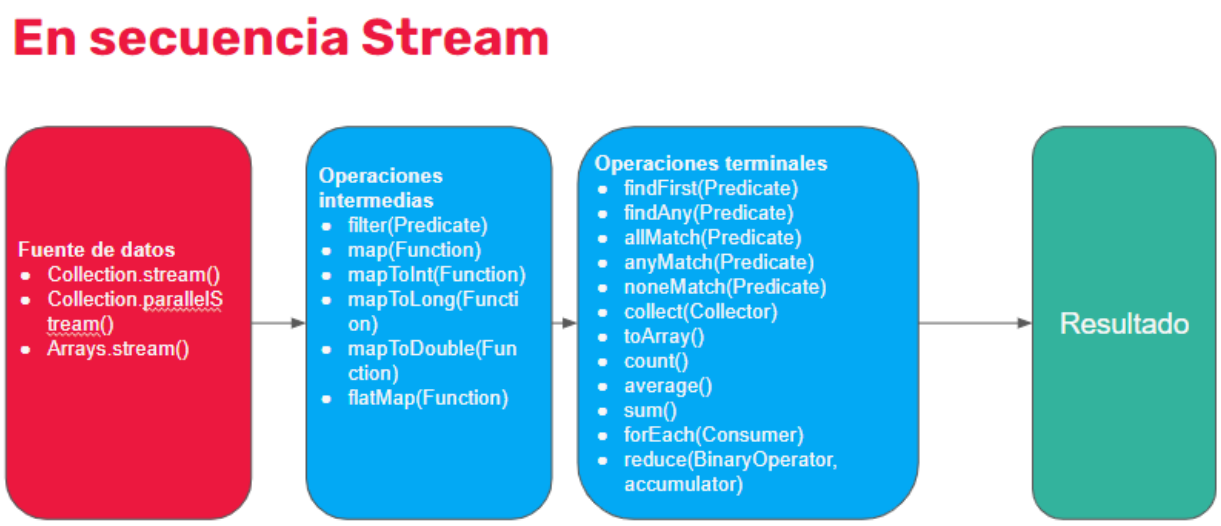
Tipos de expresiones Lambda

Consumidores (Consumer<T>)	Aceptan un parámetro pero no devuelven ninguno.	`p-> System.out.print(p)`
BiConsumidores (BiConsumer<T,U>)	Aceptan dos parámetros pero no devuelven ninguno	`(p,q)->System.out.print(p+q)`
Proveedores (Supplier<T>)	No pòseen parámetros pero si devuelven un resultado	`()-> 10`
Funciones (Function<T,U>)	Acepta un parámetro y devuelven un resultado.	`p-> p+10`
BiFunciones (BiFunction<T,U,R>)	Aceptan 2 parámetros y devuelven un resultado.	`(p,q)-> p+q`
Predicado (Predicate<T>)	Acepta un parámetro y devuelve un booleano	`p-> p==10`
BiPredicado (BiPredicate<T,U>)	Acepta dos parámetros y devuelve un booleano	`(p,q)-> p==q`

Api Stream

Se definen como un flujo de elementos que ejecutan un conjunto de funciones en forma secuencial y paralela, permitiendo el procesamiento de forma declarativa (programación funcional). Los streams son una forma de tratar los elementos de una colección de una **forma simple.**

- Permiten crear flujos de datos continuos para procesar con operadores
- Hacen uso de expresiones lambda para reducir las tareas de transformación.
- Algunos operadores para transformar o componer son el filter, map, flatmap, sorted, foreach, reduce.
- Son inmutables.
- Pueden crearse desde colecciones: Set, List, Map.



Implementación de Streams

Supongamos una lista que esté repleta de nombres y lo que queremos hacer es recorrer toda la lista e imprimir cada uno de ellos.

El *código clásico* que podemos utilizar es el siguiente:

```
public static void printForeach(){
   List<String> names = getStringArray(); //método para traer los strings
   for (String name : names) {
      System.out.println(name);
   }
}
```

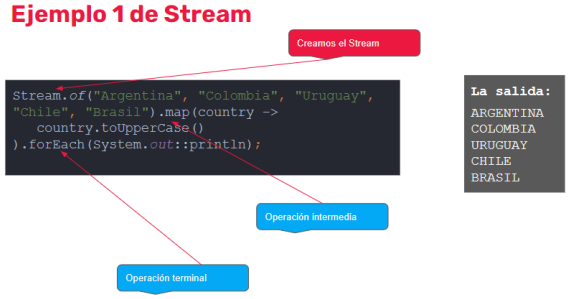
¿Cómo sería aplicando Streams?

Si aplicamos streams, podríamos hacer esta porción de código en una sola línea… ¿Cómo? mediante el siguiente código:

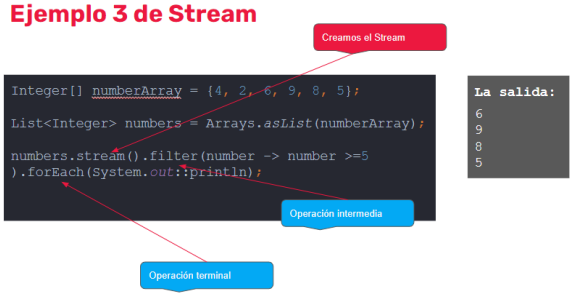
```
public static void printStream(){
    List<String> names = getStringArray();
    names.stream().forEach(System.out::println);
}
```

Estamos utilizando el método **stream()** para obtener el Stream, seguido, solo utilizamos el método **forEach()** en el cual definimos lo que vamos a hacer para cada elemento de la colección.

Ejemplo 1 de Stream



Ejemplo 3 de Stream



Ejemplo 2 de Stream