

Bootcamp Java Developer

Fase 1 - Java Analyst
Módulo 11



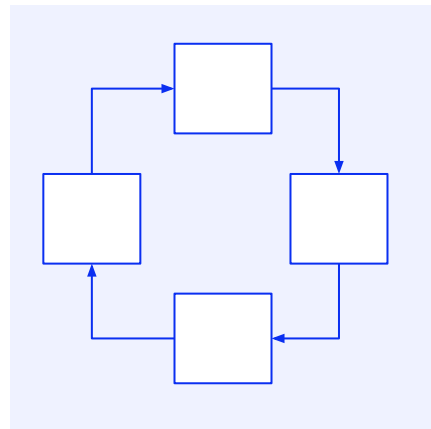
Stream

Introducción a Stream

Es una interfaz que **provee clases para procesar datos de manera funcional como flujo de datos**. No es otra colección.

También, provee una serie de métodos de orden superior (funciones que reciben una o más funciones y retornan otra función o un objeto).

Estas funciones ayudan a transformar, filtrar y reducir una colección dada.



Excepciones

La excepción **RuntimeException** es la súper clase encargada de lanzar un error en tiempo de ejecución de una aplicación.

La excepción **IllegalStateException** se encargará de lanzar un error cuando se intente acceder a un objeto en un momento inapropiado, puede ser por uso o por otras causas.

Un Stream debe operarse (mediante la invocación de una operación de secuencia intermedia o terminal) **solo una vez**.



Stream

La interfaz **BaseStream** provee un flujo de objetos que permite el procesamiento paralelo y secuencial de dicho flujo.

La interfaz **Stream** provee los métodos necesarios para hacer operaciones, aunque suele ser confundida con una colección sabemos que tienen objetivos muy diferentes. Las colecciones proporcionan una manera eficaz de acceso y modificación de datos y los flujos de operaciones.

BaseStream<T, Stream<T>>

Stream<T>

Métodos Stream

Tipo	Método	Descripción
boolean	<code>allMatch(Predicate<? super T> predicate)</code>	Devuelve true si todos los elementos coinciden con el predicado proporcionado.
boolean	<code>anyMatch(Predicate<? super T> predicate)</code>	Devuelve true si algún elemento coincide con el predicado proporcionado.
<code><R,A> R</code>	<code>collect(Collector<? super T,A,R> collector)</code>	Realiza una operación de reducción en los elementos usando un Collector.
long	<code>count()</code>	Devuelve el recuento de elementos.
<code>Stream<T></code>	<code>distinct()</code>	Devuelve un flujo que consta de los distintos elementos (<code>Object.equals(Object)</code>)
<code>static <T> Stream<T></code>	<code>empty()</code>	Devuelve true si está vacío el flujo.

Métodos Stream (continuación):

Tipo	Método	Descripción
<code>Stream<T></code>	<code>filter(Predicate<? super T> predicate)</code>	Devuelve un flujo que consta de los elementos que coinciden con el predicado dado.
<code>void</code>	<code>forEach(Consumer<? super T> action)</code>	Hace una acción para cada elemento de este flujo.
<code><R> Stream<R></code>	<code>map(Function<? super T,? extends R> mapper)</code>	Devuelve un flujo que consta de los resultados de aplicar la función dada a los elementos.
<code>static <T> Stream<T></code>	<code>of(T... values)</code>	Devuelve un flujo ordenado secuencial, cuyos elementos son los valores especificados.
<code>T</code>	<code>reduce(T identity, BinaryOperator<T> accumulator)</code>	Realiza una reducción en los elementos, al utilizar el valor de identidad proporcionado y una función de acumulación asociativa, y devuelve el valor reducido.

Function

Es una función que **toma los parámetros, los procesa o los opera.**

Ejemplo tradicional: a cada elemento de la colección se le multiplica por dos y se agrega a una nueva colección.

```
List<Integer> numerosOriginal = new ArrayList<>();
numerosOriginal.add(2);
numerosOriginal.add(3);
numerosOriginal.add(4);
List<Integer> numerosNuevos = new ArrayList<>();
System.out.println(numerosOriginal);
for (Integer numero : numerosOriginal) {
    numerosNuevos.add(numero * 2);
}
System.out.println(numerosNuevos);
```

Numero Original: [2, 3, 4]
Numero Nuevos: [4, 6, 8]

Predicate

Es una función que **toma los parámetros** y **evalúa si cumple con la condición indicada**.

Ejemplo tradicional: se evalúa cada elemento de la colección y si es par, se agrega a una nueva colección.

```
List<Integer> numerosOriginal = new ArrayList<>();
numerosOriginal.add(2);
numerosOriginal.add(3);
numerosOriginal.add(4);
List<Integer> numerosNuevos = new ArrayList<>();
System.out.println("Numeros Original: " + numerosOriginal);
for (Integer numero : numerosOriginal) {
    if (numero%2==0) {
        numerosNuevos.add(numero);
    }
}
System.out.println("Numeros Nuevos: " + numerosNuevos);
```

Numeros Original: [2, 3, 4]
Numeros Nuevos: [4, 6, 8]

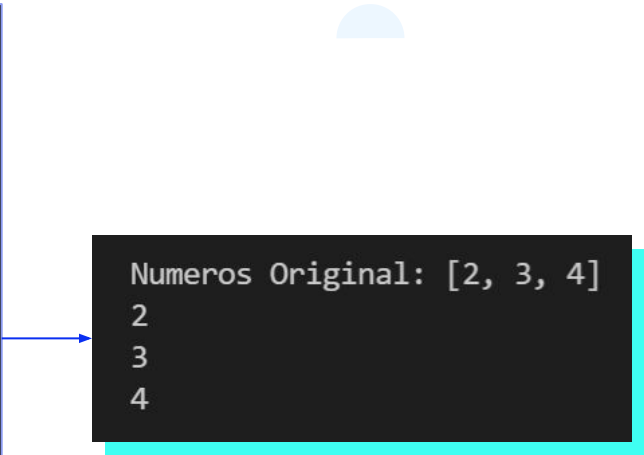
Consumer

Es una función que **toma los argumentos y no retorna nada**, simplemente **procesa**.

Ejemplo tradicional: se envía cada elemento como argumento al método `System.out.println()`.

```
List<Integer> numerosOriginal = new ArrayList<>();
numerosOriginal.add(2);
numerosOriginal.add(3);
numerosOriginal.add(4);

System.out.println("Numeros Original: " + numerosOriginal);
for (Integer numero : numerosOriginal) {
    System.out.println(numero);
}
```



```
Numeros Original: [2, 3, 4]
2
3
4
```

Conceptos

- Las colecciones poseen el método **stream()** que retorna un flujo con el contenido de la colección.
- En Java 8 las **Function**, **Predicate** y **Consumer** se realizan con expresiones Lambdas:
 - `((Parametros) -> Function).`
 - `((Parametros) -> Predicate).`
 - `((Parametros) -> Consumer).`
- Adicionalmente, se debe tener en cuenta que no se puede realizar más de una operación a la vez, ya que lanzaría una excepción de tipo **IllegalStateException**.
- Los métodos **filter()**, **map()** son operaciones intermedias.
- Los métodos **count()** y **sum()** son operaciones de terminal.

Métodos

forEach

Es un método que se usa para **recorrer cada uno de los elementos** y espera como argumento un **Consumer**. Los Stream y Colecciones poseen este método de recorrido.



```
List<String> nombres = Arrays.asList("Octavio", "Sabrina", "Sebastian", "Ariel", "Nahuel");

System.out.println("Iteracion Tradicional:");

for (String nombre : nombres) {
    System.out.println(nombre);
}

System.out.println("Iteracion Funcional:");
nombres.forEach((e) -> System.out.println(e));
```

Map

Método que se usa para **transformar un objeto en otro** vía la aplicación de una función y retornará un nuevo flujo.

```
List<String> nombres = Arrays.asList("Octavio", "Sabrina", "Sebastian", "Ariel", "Nahuel");

System.out.println("Nombres en Mayuscula Funcional:");
List<String> nombresMayusculasFuncional = nombres.stream().map((e) -> e.toUpperCase())
    .collect(Collectors.toList());

System.out.println("Iteracion Funcional:");
nombresMayusculasFuncional.forEach((e) -> System.out.println(e));
```

Filter

Este método **filtra elementos en función de un predicado** que se haya enviado.



```
List<String> nombres = Arrays.asList("Octavio", "Sabrina", "Sebastian", "Ariel", "Nahuel");

System.out.println("Nombres que comiencen con la letra S Funcional:");
List<String> nombresComienzanConSFuncional = nombres.stream().filter((e) -> e.startsWith("S"))
    .collect(Collectors.toList());

System.out.println("Iteracion Funcional:");
nombresComienzanConSFuncional.forEach((e) -> System.out.println(e));
```

Reduce

Este método recibe una función de acumulación y retorna un solo valor.



```
List<String> nombres = Arrays.asList("Octavio", "Sabrina", "Sebastian", "Ariel", "Nahuel");

System.out.println("Convertir lista a cadena Funcional:");
String cadenaNombresFuncional = nombres.stream().reduce("", (a, b) -> a + " " + b);

System.out.println("Cadena de Nombres Funcional: " + cadenaNombresFuncional);
```


**¡Sigamos
trabajando!**

