Java Streams

UTN – Facultad Regional Paraná

Titular: Ernesto Zapata Icart

Aux: Mariano Carpio

Java 8 Stream - Definición

Stream representa una secuencia de objetos desde un origen de datos, y que soporta operaciones agregadas.

- **Secuencia de elementos**: un Stream provee un conjunto de elementos de un determinado tipo accedidos de manera secuencial. Un Stream retorna elementos computados a demanda. No se almacenan los elementos.
- **Origen**: provienen de Collections, Arrays o de recursos I/O como ImputSource.
- **Operaciones agregadas**: soporta operaciones agregadas como filter, map, limit, reduce, find, match, etc.
- **Pipelining**: significa que el resultado de una operación agregada devuelve como resultado otro Stream, de modo tal que se pueden encadenar las operaciones.
- **Iteraciones automáticas**: las operaciones de Stream hacen las iteraciones internamente sobre el origen de datos; en contraste con Collections donde se requiere la iteración explícita

Java 8 Stream - API

- El nuevo paquete `java.util.stream` provee utilidades que soportan el estilo funcional de programación sobre un **stream** de valores.
- Los **streams** pueden ser **secuenciales** o **paralelos**.
- **Strams** son útiles cuando se necesita filtrar valores y para llevar a cabo operaciones sucesivas sobre resultados.
- Los **streams** fueron diseñados para las expresiones Lambda.
- Los **streams** son más poderosos, rápidos y más eficientes en términos de consumo de memoria que los **List**.
- Los **streams** pueden ser convertidos fácilmente en arreglos o listas.
- Los **streams** pueden se obtenidos “al vuelo”.

Java 8 Stream – Creación

- Desde una secuencia de valores
  - `Stream.of(val1, val2, …)`
- Desde un arreglo como parámetro
  - `Stream.of(someArray)`
  - `Arrays.stream(someArray)`
- Desde un List (y otras colecciones)
  - `someList.stream()`
  - `someOtherCollection.stream()`

Java 8 Stream – Operaciones

- **Operaciones Intermedias**: las operaciones intermedias mantienen el stream abierto para futuras sucesivas operaciones. Las operaciones intermedias son de ejecución ***lazy***.
- **Operaciones Terminales**: Las operaciones terminales determinan la operación final sobre el stream. El final de ejecución de esta operación implica que el stream se cierra y deja de ser usable.

Java 8 Stream – Operaciones: ejemplo

- **List**<String> strings =

  Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
- **List**<String> filtered = strings.**stream**()

  .**filter**(string -> !string.isEmpty())

  .**collect**(Collectors.toList());

Java 8 Stream – Operaciones: forEach

- ***forEach*** itera sobre cada elemento del stream. El siguiente código muestra como imprimir 10 números aleatorios usando **forEach**.

```
List<String> items = new ArrayList<>();
        items.add("A");   items.add("B");
        items.add("C");   items.add("D");
        items.add("E");

        //lambda
        //Output : A,B,C,D,E
        items.forEach(item->System.out.println(item));
```

Java 8 Stream – Operaciones: map

- El método ***map*** es usado para mapear cada elemento con su correspondiente resultado. El siguiente código devuelve una lista con los cuadrados no repetidos de cada número del stream.

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
List<Integer> squaresList = numbers.stream()
        .map( i -> i*i)
        .distinct()
        .collect(Collectors.toList());
```

Java 8 Stream – Operaciones: filter

- El método filter es usado para eliminar elementos basados en un criterio. El siguiente código imprime la cuenta de cadenas vacías en el stream.

```
List<String>strings =
        Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
int count = strings.stream()
        .filter(string -> string.isEmpty())
        .count();
```

Java 8 Stream – Operaciones: limit

- El método limit es usado para reducir el tamaño del stream. El siguiente código limita el stream a 10 elementos.

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

Java 8 Stream – Operaciones: sorted

- El método sorted es usado para ordenar el stream. El siguiente código muestra como ordenar 10 enteros en un stream.

```
Random random = new Random();
random.ints().limit(10).sorted().forEach(System.out::println);
```

Java 8 Stream – Operaciones: parallelStream

- El método parallelStream es la alternativa para el procesamiento paralelo. El siguiente código imprime la cuenta de cadenas vacias usando el parallelStream.

```
// stream().parallel()


List<String> strings =
        Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
int count = strings.parallelStream()
        .filter(string -> string.isEmpty())
        .count();
```

Java 8 Stream – Operaciones: collectors

- Los Collectors son usados para combinar el resultado del procesamiento de elementos en el stream.

```
List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
List<String> filtered = strings.stream()
        .filter(string -> !string.isEmpty())
        .collect(Collectors.toList());
System.out.println("Filtered List: " + filtered);
String mergedString = strings.stream().filter(string ->
!string.isEmpty()).collect(Collectors.joining(", "));
System.out.println("Merged String: " + mergedString);
```