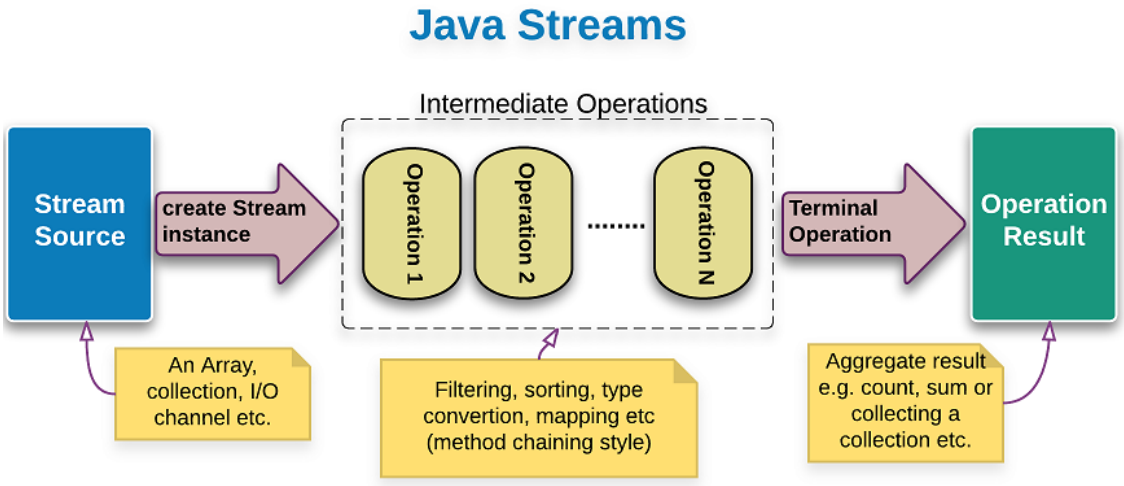
Java Streams

Nombre: Juan Martin Barraza

Dni: 41.135.676

¿Qué son los Streams?

Podríamos decir que los Streams son “envoltorios” de colecciones de datos que nos permiten operar con estas colecciones y hacer que el procesamiento masivo de datos sea eficiente y fácil de leer. Con eficiente, se refiere a que se paraleliza el uso de las CPUs, sin problemas de concurrencias porque los Streams **NO** albergar datos, solo realizan operaciones sobre ellos.



Generalmente cuando trabajamos con colecciones filtramos, calculamos y realizamos muchos tipos de operaciones sobre los datos. Los Streams nos permiten a través del paradigma funcional abstraernos del cómo programar esas operaciones y sólo centrarnos en que resultado se espera y escribirlo de una manera muy declarativa.

¿Cómo se crean los Streams?

Todas las colecciones (Collection<E>) tienen métodos stream() y parallelStream() a partir de los cuales se puede construir un Stream (Stream<E>).

```
Collection<String> stringList = new ArrayList<>();
Stream<String> stringStream = stringList.parallelStream();
```

Se puede crear un Stream<E>.partir de una matriz usando uno de dos métodos:

```
String[] values = { "aaa", "bbbb", "ddd", "cccc" };
Stream<String> stringStream = Arrays.stream(values);
Stream<String> stringStreamAlternative = Stream.of(values);
```

La diferencia entre Arrays.Stream<E>). y Stream.of() es que Stream.of()tiene un parámetro varargs, por lo que se puede usar como:

```
Stream<Integer> integerStream = Stream.of(1, 2, 3);
```

También hay Stream primitivas que puedes usar. Por ejemplo:

```
IntStream intStream = IntStream.of(1, 2, 3);
DoubleStream doubleStream = DoubleStream.of(1.0, 2.0, 3.0);
```

Operaciones Intermedias de Streams

Una operación intermedia especifica las tareas a realizar sobre los elementos del Stream y siempre produce un nuevo Stream. Las operaciones intermedias son "perezosas", es decir, no se ejecutan hasta que se invoque a una operación terminal. Esto permite a los desarrolladores optimizar el rendimiento del procesamiento de los Streams.

Como operaciones intermedias tenemos:

- ***Filter:*** Produce un Stream que contiene solo los elementos que satisfacen una condición.

```
import java.util.*;

class GFG {

   // Driver code
   public static void main(String[] args)
   {

      // Creating a list of Integers
      List<Integer> list = Arrays.asList(3, 4, 6, 12, 20);

      // Using Stream filter(Predicate predicate)
      // to get a stream consisting of the
      // elements that are divisible by 5
      list.stream().filter(num -> num % 5 == 0).forEach(System.out::println);
   }
}
```

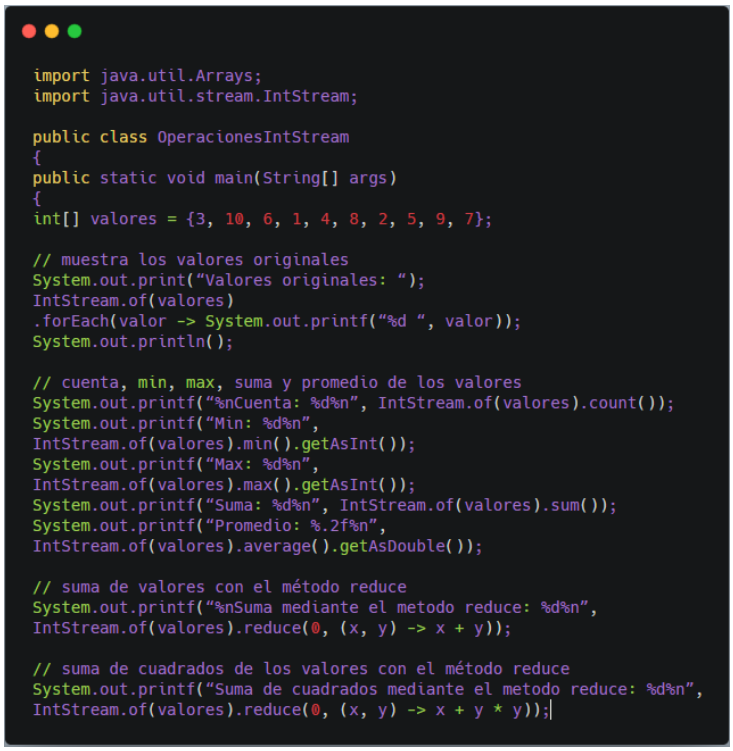
- ***Distinct:*** Produce un Stream que solo contiene los elementos únicos.
- ***Limit:*** Produce un Stream con el numero especificado de elementos a partir del inicio del Stream original.
- ***Map:*** Produce un Stream en el que cada elemento del Stream original está asociado a un nuevo valor (posiblemente de un tipo distinto). El nuevo Stream tiene la misma cantidad de elementos que el Stream original.
- ***Sorted:*** Produce un Stream en el que los elementos están ordenados. El nuevo Stream tiene el mismo número de elementos que el Stream original.

Operaciones terminales de los Streams

Una operación terminal inicia el procesamiento de las operaciones intermedias de una canalización de Stream y produce un resultado. Las operaciones terminales, en contra parte con las operaciones intermedias, son “ansiosas”, ya que realizan la operación solicitada cuando se las invoca.

Como operaciones terminales tenemos:

- ***forEach:*** Realiza un procesamiento sobre cada elemento en un Stream.
- ***Average:*** Calcula el promedio de los elementos en un Stream numérico.
- ***Count:*** Devuelve el número de elementos de un Stream.
- ***Max:*** Localiza el valor más grande de un Stream.
- ***Min:*** Localiza el valor más pequeño de un Stream.
- ***Reduce:*** Reduce los elementos de una colección a un solo valor mediante el uso de una función de acumulación asociativa.
- ***Collect:*** Crea una nueva colección de elementos que contienen los resultados de las operaciones anteriores del Stream.
- ***toArray:*** Crea un arreglo que contiene los resultados de las operaciones anteriores del Stream.
- ***findFirst:*** Encuentra el primer elemento del Stream con base en las operaciones intermedias, termina inmediatamente el procesamiento de invocación del Stream una vez que se encontró dicho elemento.
- ***findAny:*** Encuentra cualquier elemento del Stream con base en las operaciones intermedias, termina inmediatamente el procesamiento de invocación del Stream una vez que se encuentra dicho elemento.
- ***anyMatch:*** Determina si alguno de los elementos del Stream coincide con una condición especificada, cuando un elemento coincide, esta termina de inmediato la invocación del Stream.
- ***allMatch:*** Determina si todos los elementos del Stream satisfacen la condición especificada.



```java
import java.util.Arrays;
import java.util.stream.IntStream;

public class OperacionesIntStream
{
public static void main(String[] args)
{
int[] valores = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};

// muestra los valores originales
System.out.print("Valores originales: ");
IntStream.of(valores)
.forEach(valor -> System.out.printf("%d ", valor));
System.out.println();

// cuenta, min, max, suma y promedio de los valores
System.out.printf("%nCuenta: %d%n", IntStream.of(valores).count());
System.out.printf("Min: %d%n",
IntStream.of(valores).min().getAsInt());
System.out.printf("Max: %d%n",
IntStream.of(valores).max().getAsInt());
System.out.printf("Suma: %d%n", IntStream.of(valores).sum());
System.out.printf("Promedio: %.2f%n",
IntStream.of(valores).average().getAsDouble());

// suma de valores con el método reduce
System.out.printf("%nSuma mediante el metodo reduce: %d%n",
IntStream.of(valores).reduce(0, (x, y) -> x + y));

// suma de cuadrados de los valores con el método reduce
System.out.printf("Suma de cuadrados mediante el metodo reduce: %d%n",
IntStream.of(valores).reduce(0, (x, y) -> x + y * y));
```

Stream funciona con algo llamado *Lazy Evaluation*, que consiste en que Java revisa si la función que estás haciendo luego será utilizada para algo o simplemente será ignorada. En el caso de que sea ignorada el código de Stream no se ejecutará. Por ejemplo, en los casos anteriores, de no haber agregado **forEach(),** no se habría ejecutado el Sorted, el Limit ni el resto, porque forEach trabaja como función terminal.

Orden de Ejecución

El procesamiento de un objeto Stream puede ser secuencial o paralelo .

En un modo **secuencial** , los elementos se procesan en el orden de la fuente de la Stream . Si se ordena el Stream (como una implementación de SortedMap o una List), se garantiza que el procesamiento coincidirá con el orden de la fuente. En otros casos, sin embargo, se debe tener cuidado de no depender de la ordenación.

Ejemplo:

```
List<Integer> integerList = Arrays.asList(0, 1, 2, 3, 42);

// Secuencial
long howManyOddNumbers = integerList.stream()
                        .filter(e -> (e % 2) == 1)
                        .count();

System.out.println(howManyOddNumbers); // Output: 2
```

El modo **paralelo** permite el uso de múltiples hilos en múltiples núcleos, pero no hay garantía del orden en que se procesan los elementos.

Si se invocan varios métodos en una secuencia Stream , no es necesario invocar todos los métodos. Por ejemplo, si un Stream se filtra y el número de elementos se reduce a uno, no se producirá una llamada posterior a un método como la sort . Esto puede aumentar el rendimiento de un Stream secuencial, una optimización que no es posible con un Stream paralelo.

Ejemplo:

```
// Paralelo
long howManyOddNumbersParallel = integerList.parallelStream()
                                 .filter(e -> (e % 2) == 1)
                                 .count();

System.out.println(howManyOddNumbersParallel); // Output: 2
```

Conclusión

A modo de conclusión, podemos decir que los Streams facilitan mucho trabajar con grandes cantidades de datos de forma muy prolija a la hora de escribirlo en el código. Más allá de eso, también tiene beneficios y contras en cuanto a rendimiento.