Trabajo Práctico 3 Ejercicio 13

Introducción a la Programación Orientada a Objetos

Índice

Clic para ir al título

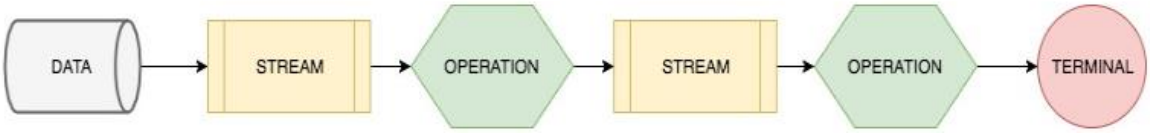
Brucezzi Thiago - 42418523

1. Realizar una monografía sobre Streams y su uso. La misma debe contener al menos la siguiente información:

¿Qué son?

Podemos definir Streams como una secuencia de funciones que se ejecutan una detrás de otra, de forma anidada.

Un Stream luce **similar** a una collection permitiendo además realizar operaciones directamente sobre el Stream. Cada operación devuelve un nuevo stream sobre el cual podemos seguir encadenando otras operaciones.



Cómo se crean / Creación de los mismos.

Todos los javas Collection<E> tienen métodos stream() y parallelStream() partir de los cuales se puede construir un 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() 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);

Estas corrientes primitivas también se pueden construir utilizando el método Arrays.stream() :

- IntStream intStream = Arrays.stream(new int[]{ 1, 2, 3 });

Es posible crear un Stream desde una matriz con un rango específico.

- int[] values= new int[]{1, 2, 3, 4, 5};
- IntStream intStram = Arrays.stream(values, 1, 3);

Tenga en cuenta que cualquier flujo primitivo se puede convertir a flujo de tipo en caja utilizando el método en boxed :

- Stream<Integer> integerStream = intStream.boxed();

Esto puede ser útil en algunos casos si desea recopilar datos, ya que la secuencia primitiva no tiene ningún método de collect que tome un Collector como argumento.

Operaciones intermedias

Las operaciones sobre Stream pueden ser intermedias o finales. En el caso de ser intermedias (filter, sorted, map) stream devolverá nuevamente otro stream permitiendo la continuidad de pasos o funciones sobre ella misma. Esto es llamado 'pipelining'.

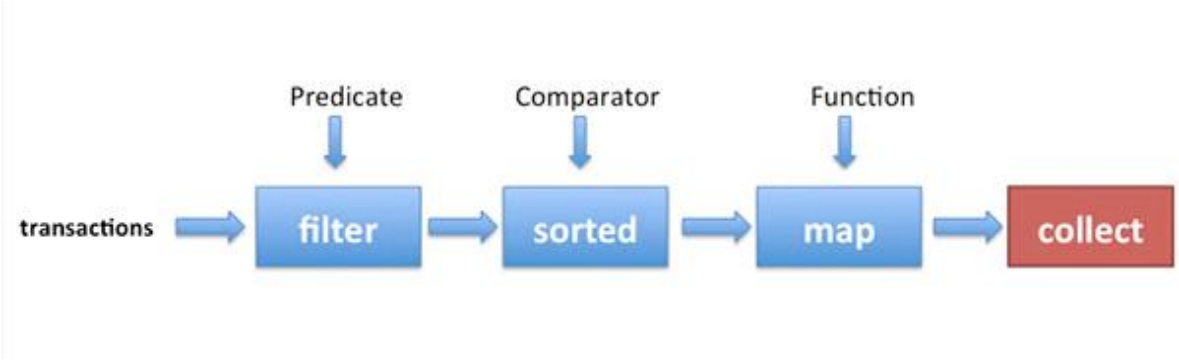
Filter (para filtrar elementos según un predicado particular)

Sorted (para ordenar los elementos según un comparador)

Map (para extraer información)

Todas las operaciones, a excepción de collect, devuelven un Stream, por lo que es posible encadenarlas y formar un proceso, que puede verse como consulta respecto de los datos del origen.

En computación, una **pipeline,** también conocida como **data pipeline** de datos, es un conjunto de elementos de procesamiento de datos conectados en serie, donde la salida de un elemento es la entrada del siguiente. Los elementos de una canalización a menudo se ejecutan en paralelo o en tiempo cortado. A menudo se inserta cierta cantidad de almacenamiento de búfer entre los elementos.



Finalizadores u Operaciones terminales

Una operación terminal inicia el procesamiento de las operaciones intermedias de una canalización de flujo y produce un resultado. Las operaciones terminales son ansiosas, ya que realizan la operación solicitada cuando se les invoca. Hablaremos más sobre las operaciones perezosas y ansiosas a media que las veamos en el capítulo; el lector verá cómo es que las operaciones perezosas pueden mejorar el rendimiento.

La tubería es evaluada cuando se invoca la operación terminal

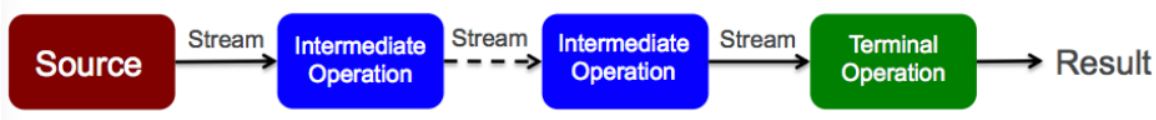
◦ Todas las operaciones pueden ser ejecutadas secuencialmente o en paralelo

◦ las operaciones intermedias pueden ser unidas

◦ Evitando pases redundantes sobre los datos

◦ Operaciones de corto-circuito (ejemplo findFirst)

◦ Evaluación perezosa (Lazy)



¿A qué se refiere con que los Streams son Lazy?

Los Streams son Lazys porque las operaciones intermedias no se evalúan hasta que se invoca la operación de terminal.

Cada operación intermedia crea una nueva secuencia, almacena la operación / función proporcionada y devuelve la nueva secuencia.

La tubería acumula estos flujos recién creados.

El momento en que se llama a la operación del terminal, comienza el recorrido de los streams y la función asociada se realiza una por una.

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

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

// sequential
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.

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

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

Imperativo

◦ Los valores asociados con nombres puedes ser modificados

◦ El orden de ejecución de comandos establece un contrato

◦ Si es modificado, el comportamiento de la aplicación podría cambiar

Funcional

◦ Los valores asociados con nombres no pueden ser cambiados

◦ El orden de ejecución no tiene impacto en el resultado

◦ No existe un orden de ejecución preestablecido

Buenas prácticas de uso

Supongamos que quieres procesar una lista de productos y realizar diferentes procesos como:

- Recorrer la lista
- Filtrar durante el recorrido los elementos según alguna condición
- Ordenar el resultado anterior
- Mostar el resultado

Con el uso de Java SE8 y los Streams esto se hace mucho mas simple que recorrer y mostrar uno a uno mediante un for, por ejemplo.

- **De forma tradicional se haría de la siguiente manera:**

```
1   public class Product {
2
3       private long id;
4       private String name;
5       private String type;
6       private Double price;
7
8       // get and set...
9   }
```

```java
List<Product> products = getProducts();
        List<Product> resultFilter = new ArrayList<>();

        // foreach filter
        for(Product p : products) {
            if("B".equals(p.getType())){
                resultFilter.add(p);
            }
        }
        // sort
        Collections.sort(resultFilter, new Comparator<Product>() {
            @Override
            public int compare(Product p1, Product p2) {
                return p1.getPrice().compareTo(p2.getPrice());
            }
        });
        // print
        for(Product p: resultFilter) {
            System.out.println(p);
        }
```

- **Pero con el uso de Streams y Lambdas:**

```java
List<Product> products = getProducts();

        // filter + sort + print
        products.stream().filter(p ->"B".equals(p.getType())).
                sorted(comparing(Product::getPrice)).
                forEach(System.out::println);
```

2. En cada punto anterior brinde ejemplos.

- Ejemplo de diferencia entre Collection y Stream:

```java
String einstein = "La imaginación es más importante que el conocimiento";
        String[] splited = einstein.split("\\s+");

        // Collection to Iterate
        List<String> words = Arrays.asList(splited);

        int count = 0;
        for(String w : words) {
            if(w.startsWith("i")) {
                count++;
            }
        }

        // JAVA 8
        words.stream().filter(w -> w.startsWith("i")).count();
```

- Ejemplo de creación de Streams:

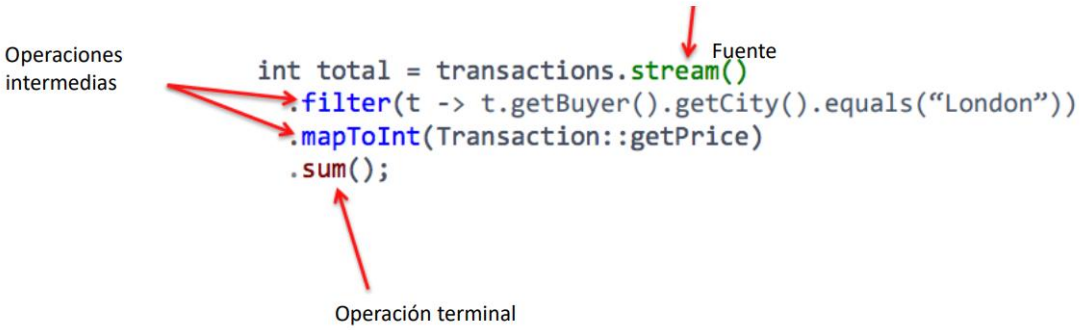
Brindados todos los ejemplos arriba.

- Ejemplo de Operaciones Intermedias:

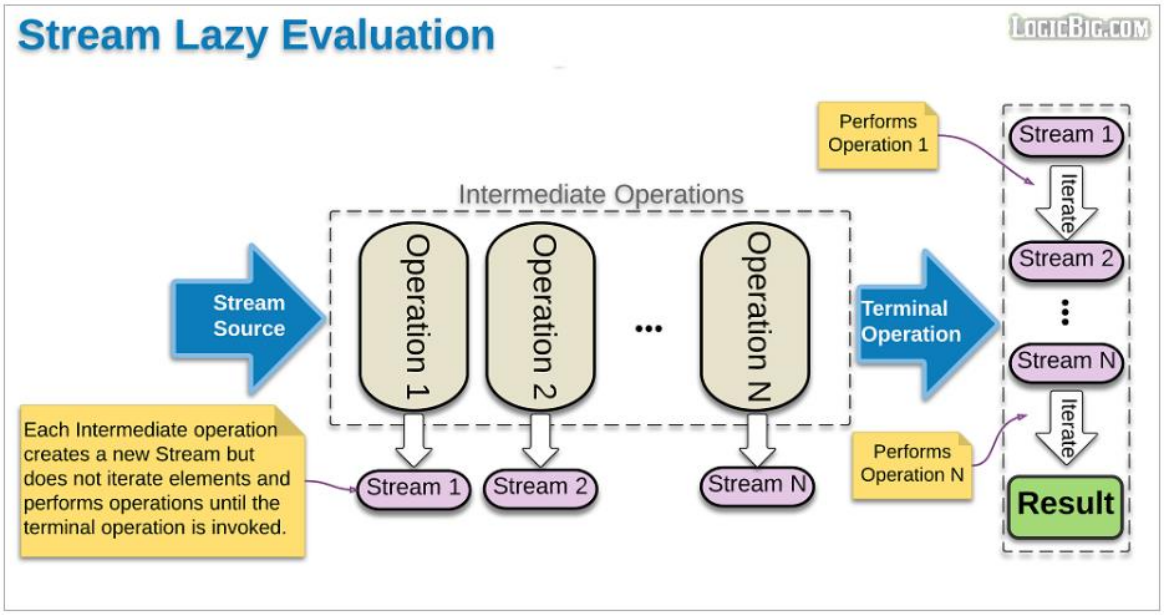
```java
// Intermediate operations
        Stream<Product> stream = products.stream();
        Stream<Product> filterStream = stream.filter(p -> "B".equals(p.getType()));
        Stream<Product> sorted = filterStream.sorted(comparing(Product::getPrice));

        // Final operation
        sorted.forEach(System.out::println);
```

- Ejemplo de Finalizadores u Operaciones terminales:



- Ejemplo de Streams Lazys:



- Ejemplo de Orden de Ejecución:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
List<Integer> twoEvenSquares = numbers.stream().filter(n -> {
    System.out.println("filtering " + n);
    return n % 2 == 0;
}).map(n -> {
    System.out.println("mapping " + n);
    return n * n;
}).limit(2).collect(Collectors.toList());


for(Integer i : twoEvenSquares)
{
    System.out.println(i);
}
```

cuando se ejecutó la lógica por debajo de la salida llegó

```
filtering 1
filtering 2
mapping 2
filtering 3
filtering 4
mapping 4
4
16
```

si la secuencia sigue el concepto de cortocircuito (donde usamos la operación de flujo de límite), entonces la salida debe ser como abajo:

```
filtering 1
filtering 2
filtering 3
filtering 4
mapping 2
mapping 4
4
16
```

- Ejemplo de Buenas Practicas de Uso:

Brindado los ejemplos arriba.

FUENTES:

https://experto.dev/java-8-uso-stream-basico/

https://riptutorial.com/es/java/example/3014/creando-un-stream

https://www.oracle.com/lad/technical-resources/articles/java/processing-streams-java-se8.html

https://lemus.webs.upv.es/wordpress/wp-content/uploads/2018/02/lambda02.pdf

http://www.ingeuni.com/?page_id=51

https://www.logicbig.com/tutorials/core-java-tutorial/java-util-stream/lazy-evaluation.html#:~:text=Streams%20are%20lazy%20because%20intermediate,and%20return%20the%20new%20stream.

https://stackoverflow.com/questions/29915591/java-8-stream-operations-execution-order

https://riptutorial.com/es/java/example/383/usando-streams