A Complete Tutorial on Java Streams

Java is a strongly Object Oriented Programming Language had its biggest leap towards functional programming by the addition of lambda expression and Stream API in Java 8 onwards.

Let's try to find out what streams are and how it can be useful in writing effective Java code.

Table of Contents:

Streams

Streams are included in java.util.stream package; stream package contains various utilities that can be used as operations to perform the bulk operation on some Collection or any data source. A stream is not to be confused with some data structure like an array or some Collection used for storing data.

It is created from sources like an array, ArrayList, List or any Collection, a file, or some I/O channel, stream pipeline. Stream elements can be used only once its life as they are fed to some operation if an attempt is made to access it later an IllegalStateException is thrown.

Sample Usage:

```
String[] fruits = new String[]{"apple", "oranges", "banana"
Stream<String>  fruitsStream = Arrays.stream(marks)
```

or

```
ArrayList<String> fruits = newArrayList<>();
fruits.add(“apple”);
..
Stream<String> fruitStream = fruits.stream();
```

or

```
Stream<String> fruitStream = Stream.of([]"apple", "oranges"
```

Let's write a code to find all fruits whose name ends with character "e" using for loop:

```
List  newlist  = new ArrayList<String>();
for(String fruit : fruits)
{
If (fruit.endsWith("e")
    {
    newlist.add(fruit);
}
}
```

Now let's see how it can be done using streams in a single line.

```
List newList = fruits.stream().filter( x-
>x.endsWith("e")).collect(Collectors.toList());
```

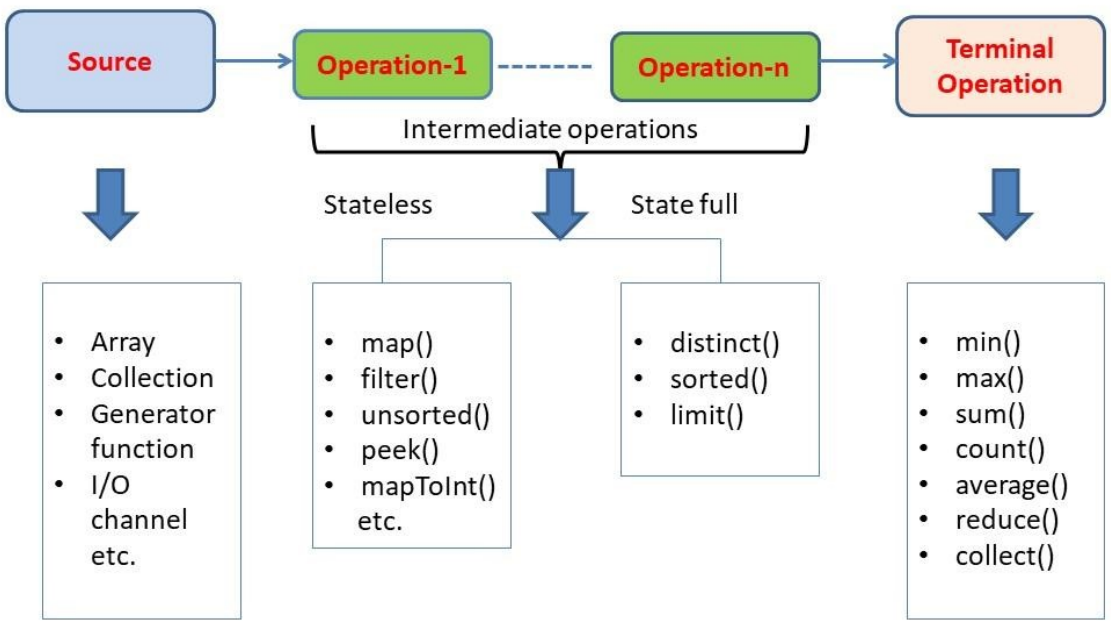
In the above code, the stream is from List object and this stream is fed to another intermediate operation using filter(), this function iterates over each element of the input stream and filters all the element according to lambda expression which produces a new stream of filtered elements as input to the terminal operation collect which produces a new List.

Advantages of using streams:

1. Efficient and shortcode.
2. Streams provide a very easy way to do parallel computation without having to worry about the multi-threading implementations.
3. Streams provide a large set of operations that can be utilized in many scenarios.
4. It provides a more memory efficient way as the stream is closed, once it's consumed and there are no extra objects and variables created which lingers on, waiting to be garbage collected.
5. With the use of lambda expressions, a wide range of functionalities can be implemented.

Stream pipeline

A Stream pipeline consists of a stream source zero or more intermediate operations and a terminal operation. Figure 1 shows the components of a stream pipeline.



Components of a stream pipeline

Source

The stream can be created from the array, Set, List, Map or any Collection, any generator function like a random function generator, from a file or any IO channel, some computational pipeline.

Intermediate operation

From a source, a stream of elements is generated on which some operations are applied to achieve the desired result. In a stream pipeline, there can be one or more intermediate operations. Intermediate operations can be of two types stateless and stateful. Each intermediate operation consumes a stream and produces another stream as per the implemented logic.

Stateless operations

Operation's which doesn't require to maintain the state of the stream and has nothing to do with the other elements of the stream, Operations like map (), filter () mapToInt (), mapToDouble(), peek(), unsorted(), etc.

Stateful operations

Operations in which elements can't be processed individually and they are required to do some comparison with other elements, like distinct (), sorted (), limit (), etc.

Terminal Operations

In a stream pipeline there can be only one terminal operation that produces some desired single result or side effect, it consumes a stream but doesn't produce a stream. Example terminal operations min() , max(), sum(), average(), collect(), for Each(), reduce() etc.

Parallel stream

Traditionally when we wanted to perform any bulk operation on a collection by using for loop it's done sequentially. Streams provide support for parallel computation to exploit multiple cores on a processing unit, it can be achieved very easily by creating a stream ().parallel () or any Collection.parallelStream().

Example: Find fruits, whose names end with "e" using a parallel stream.

```
List newList = fruits.stream().parallel().filter( x-
x.endsWith("e")).collect(Collectors.toList());
```

Or

```
Stream fruitStream = fruits.parallelStream();
```

Map filter reduce

In functional programming Map filter reduce operations are very popular and come handy in a wide range of situations. All these operations make use of lambda expressions.

Stream.map() :

The map is an intermediate operation that consumes a stream and produces a stream. It applies a given lambda expression or method reference to each element of the stream and converts it to a new stream.

Example: Convert each element of the list to uppercase.

```
List newList = fruits.stream().map( x-
>x.toUpperCase()).collect(Collectors.toList());
```

OR using method reference

```
List newList = fruits.stream().map(
String::toUpperCase).collect(Collectors.toList());
```

Stream.filter() :

The filter is an intermediate operation that consumes a stream and produces a stream. It applies a given lambda expression to each element of the stream and filters the input stream to the new stream.

Example: filter all elements that start with “A”.

```
List newList = fruits.stream().map( x->x.toUpperCase()).filter(x-
>x.startsWith(“A”)).collect(Collectors.toList());
```

Stream.reduce() :

Reduce is a terminal operation that consumes a stream, applies the lambda expression to each element and produces a single result and not a stream.

Example: Concatenate all fruits that start with “A”.

```
List newList = fruits.stream().map( x->x.toUpperCase()).filter(x-
>x.startsWith(“A”)).reduce(“”,(x,y)-> x+y);
```

Collectors

Java.util.stream.Collectors is a final utility class that contains various methods which are used with terminal operation Stream.collect() . It is generally used to convert the output stream after various operations are applied to some Collection using Collectors.toList() , Collectors.toSet() etc.

Collectors also provide some very useful utility methods like Collectors.groupingBy () and Collectors. Counting ().

Example: Let’s count no fruits of each type.

```
Map<String, Long> map = fruits.stream().map(x-
>x.toUpperCase()).collect(Collectors.groupingBy(Function.identity(),C
ollectors.counting()));
```

This gives the output as:

```
{APPLE =2 , PINEAPPLE=1, GUAVA=1,ORANGES =2,MANGO=1}
```

Summary

In a nutshell, we can say java streams can be very helpful when we are dealing with any Collections data structure and need to perform various bulk operations. In day to day programming, we always use some Collections like ArrayList, Set, Hashmap, etc., with such a large set of utilities packaged with java streams, it makes programmers’ task easier with these readymade operations available. And with parallel streams, one can easily exploit parallel processing without having to deal with multithreading.

Related Posts:

Automation Testing in CI/CD Pipeline

CI/CD Pipeline, also known as DevOps pipeline, is used to continuously integrate and deploy software products The [...] →

How to Develop Lightweight Components Using Ruby on Rails Helpers and Stimulus

Ruby on Rails Rails is an open-source web development framework initially introduced in 2004 and written in Ruby [...] →