

[Download as PDF](#)

STREAMS in Java8

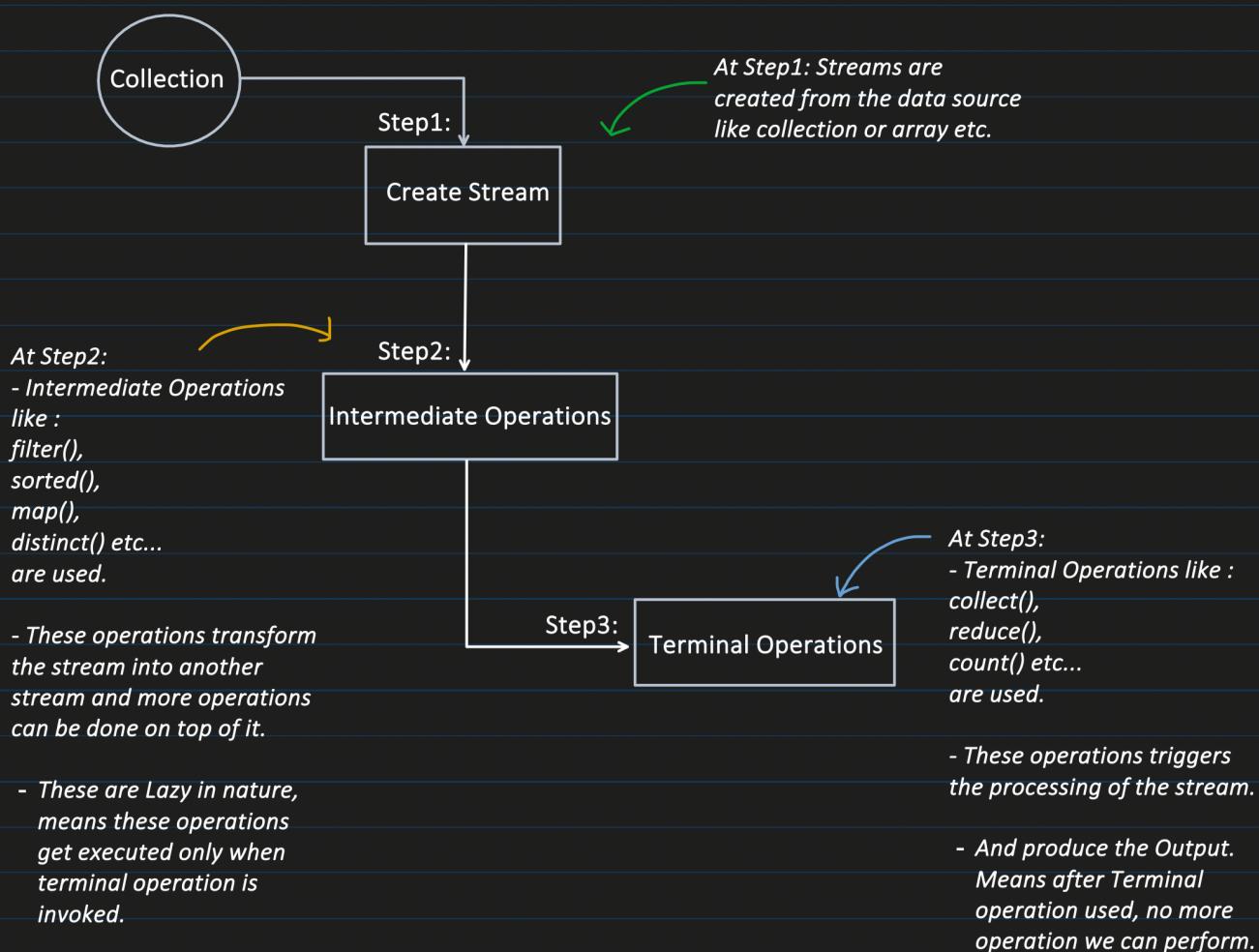
"Concept && Coding" YT Video Notes

Stream (Java8)

Sunday, 1 October 2023 12:33 PM

What is Stream?

- We can consider Stream as a pipeline, through which our collection elements passes through.
- While elements passes through pipelines, it perform various operations like sorting, filtering etc.
- Useful when deals with bulk processing. (can do parallel processing)



For Example:

```
public class StreamExample {  
  
    public static void main(String args[]){  
  
        List<Integer> salaryList = new ArrayList<>();  
        salaryList.add(3000);  
        salaryList.add(4100);  
        salaryList.add(9000);  
        salaryList.add(1000);  
        salaryList.add(3500);  
  
        int count = 0;  
        for(Integer salary : salaryList){  
            if(salary > 3000){  
                count++;  
            }  
        }  
  
        System.out.println("Total Employee with salary > 3000: " + count);  
    }  
}  
  
public class StreamExample {  
  
    public static void main(String args[]){  
  
        List<Integer> salaryList = new ArrayList<>();  
        salaryList.add(3000);  
        salaryList.add(4100);  
        salaryList.add(9000);  
        salaryList.add(1000);  
        salaryList.add(3500);  
  
        long output = salaryList.stream().filter((Integer sal) -> sal>3000).count();  
        System.out.println("Total Employee with salary > 3000: " + output);  
    }  
}
```

Output:

```
Total Employee with salary > 3000: 3
```

Different ways to create a stream:

1. From Collection:

```
List<Integer> salaryList = Arrays.asList(3000, 4100, 9000, 1000, 3500);  
Stream<Integer> streamFromIntegerList = salaryList.stream();
```

2. From Array:

```
Integer[] salaryArray = {3000, 4100, 9000, 1000, 3500};  
Stream<Integer> streamFromIntegerArray = Arrays.stream(salaryArray);
```

3. From Static Method:

```
Stream<Integer> streamFromStaticMethod= Stream.of(...values: 1000, 3500, 4000, 9000);
```

4. From Stream Builder:

```
Stream.Builder<Integer> streamBuilder = Stream.builder();
streamBuilder.add(1000).add(9000).add(3500);

Stream<Integer> streamFromStreamBuilder = streamBuilder.build();
```

5. From Stream Iterate:

```
Stream<Integer> streamFromIterate = Stream.iterate(seed: 1000, (Integer n) -> n + 5000).limit(maxSize: 5);
```

Different Intermediate Operations:

We can chain multiple Intermediate operations together to perform more complex processing before applying terminal operation to produce the result.

No.	Intermediate Operation	Description
1.	<pre>filter(Predicate<T> predicate)</pre> <pre>Stream<String> nameStream = Stream.of(...values: "HELLO", "EVERYBODY", "HOW", "ARE", "YOU", "DOING"); Stream<String> filteredStream = nameStream.filter((String name) -> name.length() <= 3); List<String> filteredNameList = filteredStream.collect(Collectors.toList());</pre> <pre>//OUTPUT: HOW, ARE, YOU</pre>	Filters the element.
2.	<pre>map(Function<T, R> mapper)</pre> <pre>Stream<String> nameStream = Stream.of(...values: "HELLO", "EVERYBODY", "HOW", "ARE", "YOU", "DOING"); Stream<String> filteredNames = nameStream.map((String name) -> name.toLowerCase()); //OUTPUT: hello, everybody, how, are, you, doing</pre>	Used to transform each element

flatMap(Function<T, Stream<R>> mapper)

```
3. List<List<String>> sentenceList = Arrays.asList(  
    Arrays.asList("I", "LOVE", "JAVA"),  
    Arrays.asList("CONCEPTS", "ARE", "CLEAR"),  
    Arrays.asList("ITS", "VERY", "EASY")  
);  
  
Stream<String> wordsStream1 = sentenceList.stream()  
    .flatMap((List<String> sentence) -> sentence.stream());  
//Output: I, LOVE, JAVA, CONCEPTS, ARE, CLEAR, ITS, VERY, EASY  
  
Stream<String> wordsStream2 = sentenceList.stream()  
    .flatMap((List<String> sentence) -> sentence.stream().map((String value) -> value.toLowerCase()));  
//Output: i, love, java, concepts, are, clear, its, very, easy
```

Used to iterate over each element of the complex collection, and helps to flatten it.

distinct()

```
4. Integer[] arr = {1,5,2,7,4,4,2,0,9};  
  
Stream<Integer> arrStream = Arrays.stream(arr).distinct();  
//Output: 1, 5, 2, 7, 4, 0, 9
```

Removes duplicate from the stream.

sorted()

```
5. Integer[] arr = {1,5,2,7,4,4,2,0,9};  
  
Stream<Integer> arrStream = Arrays.stream(arr).sorted();  
//Output: 0, 1, 2, 2, 4, 4, 5, 7, 9
```

Sorts the elements.

```
Integer[] arr = {1,5,2,7,4,4,2,0,9};  
  
Stream<Integer> arrStream = Arrays.stream(arr).sorted((Integer val1, Integer val2) -> val2-val1);  
//Output: 9, 7, 5, 4, 4, 2, 2, 1, 0
```

peek(Consumer<T> action)

```
6.     List<Integer> numbers = Arrays.asList(2,1,3,4,6);
      Stream<Integer> numberStream = numbers.stream()
          .filter((Integer val)-> val>2)
          .peek((Integer val) -> System.out.println(val)) //it will print 3, 4, 6
          .map((Integer val) -> -1*val);
      List<Integer> numberList = numberStream.collect(Collectors.toList());
```

Helps you to see the intermediate result of the stream which is getting processed.

limit(long maxSize)

```
7.     List<Integer> numbers = Arrays.asList(2,1,3,4,6);
      Stream<Integer> numberStream = numbers.stream().limit(maxSize: 3);

      List<Integer> numberList = numberStream.collect(Collectors.toList());
      //Output: 2, 1, 3
```

Truncate the stream, to have no longer than given maxSize

skip(long n)

```
8.     List<Integer> numbers = Arrays.asList(2,1,3,4,6);
      Stream<Integer> numberStream = numbers.stream().skip(n: 3);

      List<Integer> numberList = numberStream.collect(Collectors.toList());
      //Output: 4, 6
```

Skip the first n elements of the stream.

mapToIntToIntFunction<T> mapper)

```
9.     List<String> numbers = Arrays.asList("2", "1", "4", "7");
      IntStream numberStream = numbers.stream().mapToInt((String val) -> Integer.parseInt(val));

      int[] numberArray = numberStream.toArray();
      //Output: 2, 1, 4, 7

      int[] numbersArray = {2, 1, 4, 7};
      IntStream numbersStream = Arrays.stream(numbersArray);
      numbersStream.filter((int val) -> val > 2);
      int[] filteredArray = numbersStream.toArray();
      //Output: 4, 7
```

helps to work with primitive "int" data types

mapToLong(ToLongFunction<T> mapper)
(pls try it out....)

helps to work with primitive "long" data types

map.ToDouble(ToDoubleFunction<T> mapper)
(pls try it out....)

helps to work with primitive "double" data types

Why we call Intermediate operation "Lazy":

```
public class StreamExample {  
  
    public static void main(String args[]){  
  
        List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);  
        Stream<Integer> numbersStream = numbers.stream().filter((Integer val) -> val >=3).peek((Integer val) -> System.out.println(val));  
    }  
}
```

Output:
Nothing would be printed in the Output

```
public class StreamExample {  
  
    public static void main(String args[]){  
  
        List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);  
        Stream<Integer> numbersStream = numbers.stream().filter((Integer val) -> val >=3).peek((Integer val) -> System.out.println(val));  
  
        numbersStream.count(); //count is one of the terminal operation  
    }  
}
```

Output:

4
7
10

Sequence of Stream Operations:

```

public class StreamExample {

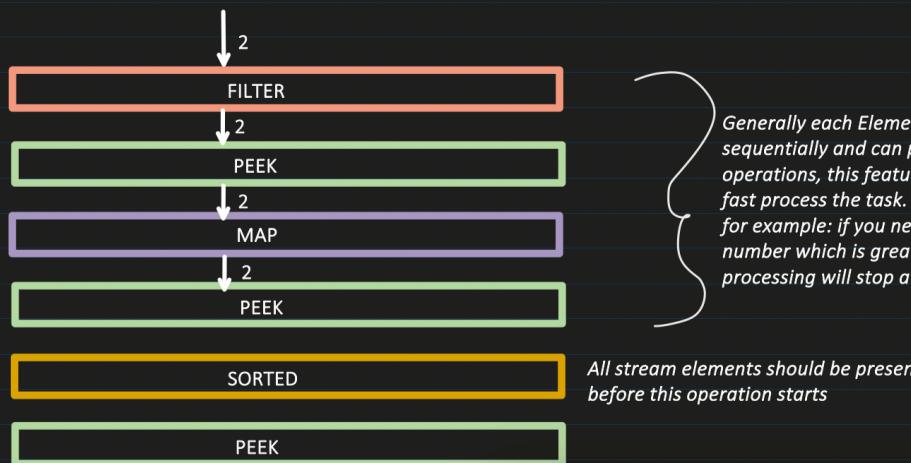
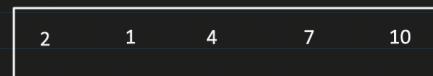
    public static void main(String args[]){

        List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);
        Stream<Integer> numbersStream = numbers.stream()
            .filter((Integer val) -> val >=3)
            .peek((Integer val)-> System.out.println("after filter:" + val))
            .map((Integer val) -> (val * -1))
            .peek((Integer val)-> System.out.println("after negating:" + val))
            .sorted()
            .peek((Integer val)-> System.out.println("after Sorted:" + val));

        List<Integer> filteredNumberStream = numbersStream.collect(Collectors.toList());
    }
}

```

Expected Output:	Actual Output:
after filter: 4	after filter: 4
after filter: 7	after negating: -4
after filter: 10	after filter: 7
after negating: -4	after negating: -7
after negating: -7	after filter: 10
after negating: -10	after negating: -10
after Sorted: -10	after Sorted: -10
after Sorted: -7	after Sorted: -7
after Sorted: -4	after Sorted: -4



Different Terminal Operations:

Terminal operations are the ones that produces the result. It triggers the processing of the stream.

No.	Terminal Operations	Description
1.	<pre> forEach(Consumer<T> action) List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10); numbers.stream() .filter((Integer val) -> val >=3) .forEach((Integer val)-> System.out.println(val)); //OUTPUT: 4, 7, 10 </pre>	<p>Perform action on each element of the Stream. DO NOT Returns any value.</p>
2.	<pre> List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10); Object[] filteredNumberArrType1 = numbers.stream() .filter((Integer val) -> val >=3) .toArray(); Integer[] filteredNumberArrType2 = numbers.stream() .filter((Integer val) -> val >=3) .toArray((int size) -> new Integer[size]); </pre>	<p>Collects the elements of the stream into an Array.</p>

3.	<pre> reduce(BinaryOperator<T> accumulator) List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10); Optional<Integer> reducedValue = numbers.stream() .reduce((Integer val1, Integer val2) -> val1+val2); System.out.println(reducedValue.get()); //output: 24 </pre>	<p>does reduction on the elements of the stream. Perform associative aggregation function.</p>
4.	<pre> collect(Collector<T, A, R> collector) List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10); List<Integer> filteredNumber = numbers.stream() .filter((Integer val) -> val >=3) .collect(Collectors.toList()); </pre>	<p>can be used to collects the elements of the stream into an List.</p>

min(Comparator<T> comparator) and
max(Comparator<T> comparator)

5.

```
List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);

Optional<Integer> minimumValueType1 = numbers.stream()
    .filter((Integer val) -> val >= 3)
    .min((Integer val1, Integer val2) -> val1-val2);

System.out.println(minimumValueType1.get());
//output: 4

Optional<Integer> minimumValueType2 = numbers.stream()
    .filter((Integer val) -> val >= 3)
    .min((Integer val1, Integer val2) -> val2-val1);

System.out.println(minimumValueType2.get());
//output: 10
```

Finds the minimum or maximum element from the stream based on the comparator provided.

Can you pls comment the output for max for both the types

count()

6.

```
List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);

long noOfValuesPresent = numbers.stream()
    .filter((Integer val1) -> val1 >= 3)
    .count();

System.out.println(noOfValuesPresent);
//output: 3
```

returns the count of element present in the stream

anyMatch(Predicate<T> predicate)

7.

```
List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);

boolean hasValueGreaterThanThree = numbers.stream()
    .anyMatch((Integer val) -> val>3);

System.out.println(hasValueGreaterThanThree);
//output: true
```

Checks if any value in the stream match the given predicate and return the boolean.

allMatch(Predicate<T> predicate)

Can you pls try it out...

9.

```
noneMatch(Predicate<T> predicate)
Can you pls try it out...
```

Checks if all value in the stream match the given predicate and return the boolean.

findFirst()

10.

```
List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);

Optional<Integer> firstValue = numbers.stream()
    .filter((Integer val) -> val >= 3)
    .findFirst();

System.out.println(firstValue.get());
//output: 4
```

finds the first element of the stream.

findAny()

Can you pls try it out...

finds any random element of the stream.

How many times we can use a single stream:

- One Terminal Operation is used on a Stream, it is closed/consumed and can not be used again for another terminal operation.

```
List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);

Stream<Integer> filteredNumbers = numbers.stream()
    .filter((Integer val) -> val >= 3);

filteredNumbers.forEach((Integer val) -> System.out.println(val)); // consumed the filteredNumbers stream

//trying to use the closed stream again
List<Integer> listFromStream = filteredNumbers.collect(Collectors.toList());
```

Output:

```
4
7
10
Exception in thread "main" java.lang.IllegalStateException Create breakpoint : stream has already been operated upon or closed
at StreamExample.main(StreamExample.java:19)
```

Parallel Stream:

Helps to perform operation on stream concurrently, taking advantage of multi core CPU.

ParallelStream() method is used instead of regular stream() method.

Internally it does:

- Task splitting : it uses "spliterator" function to split the data into multiple chunks.
- Task submission and parallel processing : Uses Fork-Join pool technique.

Note: i will cover Fork-Join pool implementation in Multithreading topic

```
public class StreamExample {  
  
    public static void main(String args[]) {  
  
        List<Integer> numbers = Arrays.asList(11, 22, 33, 44, 55, 66, 77, 88, 99, 110);  
  
        // Sequential processing  
        long sequentialProcessingStartTime = System.currentTimeMillis();  
        numbers.stream()  
            .map((Integer val) -> val * val)  
            .forEach((Integer val) -> System.out.println(val));  
        System.out.println("Sequential processing Time Taken: " + (System.currentTimeMillis() - sequentialProcessingStartTime) + " millisecond");  
  
        // Parallel processing  
        long parallelProcessingStartTime = System.currentTimeMillis();  
        numbers.parallelStream()  
            .map((Integer val) -> val * val)  
            .forEach((Integer val) -> System.out.println(val));  
        System.out.println("Parallel processing Time Taken: " + (System.currentTimeMillis() - parallelProcessingStartTime) + " millisecond");  
    }  
}
```

Output:

```
121  
484  
1089  
1936  
3025  
4356  
5929  
7744  
9801  
12100  
Sequential processing Time Taken: 64 millisecond  
7744  
121  
5929  
4356  
1936  
484  
12100  
1089  
9801  
3025  
Parallel processing Time Taken: 5 millisecond
```

