Streams in Java 8

By TecheStop

Properties of Streams

- Streams are not any other data structure. It does not store any value. Rather it sources the elements from a datastrutcure which can be operated upon through a pipeline of operations.
- Any change done through the stream pipeline does not affect the source of stream.
- Intermediate operations like filtering are performed lazily on the source stream which leads to an optimized processing.
- Elements of the streams are visited only once during the life of a stream.

Stream creation

Stream can be created in following ways.

- ```
  Stream<String> stringStream = Stream.empty();
  ```
- ```
  List<String> stringList = Arrays.asList("str1", "str2", "str3");

  Stream<String> stringStream1  = stringList.stream();
  ```
- ```
  String[] strings = new String[]{"a","b","c"};

  Stream<String> stringStream2 = Arrays.stream(strings);
  ```

- ```
  Stream<String> strStream = Stream.generate(() -> "item").limit(10);
  ```
- ```
  Stream<Integer> intStream = Stream.iterate(20, n -> n + 1).limit(20);
  ```
  ```

Referencing a stream

```
List<String> stringList = Arrays.asList("str1", "Str2",
"str3");

Stream<String> stream = stringList.stream().filter((s) ->
s.contains("str"));

Optional<String> result = stream.findFirst();


//Using the stream again will cause
IllegalStateException

Optional<String> result2 = stream.findAny();
```

Stream pipeline

Stream pipeline consists of three parts :

- Source
- Intermediate operation
- Terminate operation

```
List<String> stringList = Arrays.asList("str1", "Str2",
"str3");

Stream<String> stream = stringList.stream().filter((s) ->
s.contains("str"));

System.out.println(stream.skip(1).map((s) -> s.concat("
suffix") ).sorted().count());
```

Intermediate operations

- Map
- Filter
- Skip
- Sorted
- Peek
- Distinct

Terminal operations

- Collect
- Count
- Sum
- Max
- FindFirst
- FindAny
- ForEach
- Reduce

Lazy invocation

Intermediate operations are lazy i.e. they will be called only if required for the terminal operation.

```
List<String> stringList = Arrays.asList("str1", "Str2", "str3");

Optional<String> stream = stringList.stream().filter(element
-> {

   System.out.println("filter() was called");

   return element.contains("2");

}).map(element -> {

   System.out.println("map() was called");

   return element.toUpperCase();

}).findFirst();
```

Order of execution

Intermediate operation which reduces the size of the stream should be executed first.

```
long size = list.stream().skip(2).map(element -> element.substring(0, 3)).count();
```

Reduce and collect method

```
List<Integer> intList = Arrays.asList(3,5,6);

Stream<Integer> integerStream = intList.stream();

Optional<Integer> result = integerStream.reduce((a,b) ->
a+b);

result.ifPresent((i) -> System.out.println(i.intValue()));
```

Reduce and collect method

```
List<Integer> intList = Arrays.asList(3,5,6);

Stream<Integer> integerStream = intList.stream();

System.out.println(integerStream.collect(Collectors.averagingInt(Integer::intValue)));
```

```
List<String> stringList = Arrays.asList("Hello","This","is","Prateek");

Stream<String> stringStream = stringList.stream();

System.out.println(stringStream.collect(Collectors.joining(" ","*** "," ***")));
```

Parallel Streams

```
Stream<Product> streamOfCollection =
productList.parallelStream();

boolean isParallel = streamOfCollection.isParallel();

boolean bigPrice = streamOfCollection

  .map(product -> product.getPrice() * 12)

  .anyMatch(price -> price > 200);
```