Mapping Operations in Stream

In this lesson, we will look at the mapping operations and the different ways to transform a stream.

We'll cover the following

Mapping operations are those operations that transform the elements of a stream and return a new stream with transformed elements.

We can use a variety of methods to transform a stream into another stream object. The two most common methods used are `map()` and `flatMap()`.

Understanding map()

The `map()` method takes a lambda expression as its only argument and uses it to change every individual element in the stream. Its return value is a new stream object containing the changed elements.

Below is the method definition:

> **<R> Stream<R> map(Function<? super T, ? extends R> mapper)**

Input Parameter -> A function to apply to each element.

Return Type -> Returns a stream consisting of the results of applying the given function to the elements of the stream.

Let's look at a basic example of `map()`. In the below example, we have a list of names. We need to print all the names on the list in the upper case.

```
import java.util.ArrayList;
import java.util.List;

public class StreamDemo {


    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Dave");
        list.add("Joe");
        list.add("Ryan");
        list.add("Iyan");
        list.add("Ray");
        // map() is used to convert each name to upper case.
        // Note: The map() method does not modify the original list.
        list.stream()
                .map(name -> name.toUpperCase()) //map() takes an input of Function<T, R> type.
                .forEach(System.out::println);   // forEach() takes an input of Consumer type.

    }
}
```

Understanding mapToInt()

Let’s look at one more example.

Given a list of words, we need to print the length of each word.

To solve this problem, we can use a `map()`, which takes **s -> s.length()** lambda expression as input. However, have you noticed anything here?

The input is a string and output is an integer. If we use **map(s -> s.length())** , then it will return a stream of integers.

However, in the first lesson, we discussed that if we are dealing with primitives then we should use primitive flavors of stream.

The `mapToInt()` method comes into the picture here. If we use the `mapToInt()` method instead of `map()`, it will return `IntStream` instead of `Stream`.

So, if we are sure that our function is going to return a primitive, instead of using `map()` use `mapToInt()`, `mapToLong()` or `mapToDouble()`.

```
import java.util.ArrayList;
import java.util.List;

public class StreamDemo {

    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Dave");
        list.add("Joe");
        list.add("Ryan");
        list.add("Iyan");
        list.add("Ray");

        list.stream()
                .mapToInt(name -> name.length())
                .forEach(System.out::println);

    }
}
```

Understanding flatMap()

Stream `flatMap()` method is used to flatten a stream of collections to a stream of elements combined from all collections.

Basically, `flatMap()` is used to do following operation:

- `Stream<String[]> -> flatMap -> Stream<String>`
- `Stream<Set<String>> -> flatMap -> Stream<String>`
- `Stream<List<String>> -> flatMap -> Stream<String>`

Now, the question is why do we need to flatten our stream? The reason is that intermediate methods such as `filter()` and `distinct()` do not work on streams of `Collections`.

These methods only work on streams of primitives or objects. So, we need to flatten our stream before using these intermediate functions.

Let's see an example of `flatMap()`. In the below code we have a `List<List<String>>`.

We need to filter the strings and then print the filtered strings. The below code, will not print anything because we are not flattening our stream.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class StreamDemo {

    public static void main(String[] args) {
        List<List<String>> list = new ArrayList<>();
        list.add(Arrays.asList("a","b","c"));
        list.add(Arrays.asList("d","e","f"));
        list.add(Arrays.asList("g","h","i"));
        list.add(Arrays.asList("j","k","l"));

        Stream<List<String>> stream1 = list.stream();
        // filter() method do not work on stream of collections
        Stream<List<String>> stream2 = stream1.filter(x -> "a".equals(x.toString()));
        //This will not print anything
        stream2.forEach(System.out::println);
    }
}
```

Now, we will use `flatMap()` to flatten our stream.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class StreamDemo {

    public static void main(String[] args) {
        List<List<String>> list = new ArrayList<>();
        list.add(Arrays.asList("a","b","c"));
        list.add(Arrays.asList("d","e","f"));
        list.add(Arrays.asList("g","h","i"));
        list.add(Arrays.asList("j","k","l"));
        //Created a stream from the list.
        Stream<List<String>> stream1 = list.stream();
        // Flattened the stream.
        Stream<String> stream2 = stream1.flatMap(s -> s.stream());
        //Applied filter on flattened stream.
        Stream<String> stream3 = stream2.filter(x -> "a".equals(x));

        stream3.forEach(System.out::println);
    }
}
```

The above code can be written in a concise format as shown below. It was first written as an individual operation just for explanation.

```
list.stream()
                .flatMap(s -> s.stream())
                .filter(x -> "a".equals(x))
                .forEach(System.out::println);
```

Similar to the `map()` method, `flatMap()` also has a primitive variation. These are:

- `flatMapToInt`

- [illegible]
- `flatMapToLong`
- `flatMapToDouble`

In the next lesson, we will discuss the method references.