4. Core stream operations

Stream abstraction have a long list of useful functions for you. I am not going to cover them all, but I plan here to list down all most important ones, which you must know first hand.

Before moving ahead, lets build a collection of String beforehand. We will build out example on this list, so that it is easy to relate and understand.

```
List<String> memberNames = new ArrayList<>();
memberNames.add("Amitabh");
memberNames.add("Shekhar");
memberNames.add("Aman");
memberNames.add("Rahul");
memberNames.add("Shahrukh");
memberNames.add("Salman");
memberNames.add("Yana");
memberNames.add("Lokesh");
```

These core methods have been divided into 2 parts given below:

4.1. Intermediate operations

Intermediate operations return the stream itself so you can chain multiple method calls in a row. Let's learn important ones.

4.1.1. Stream.filter()

Filter accepts a predicate to filter all elements of the stream. This operation is intermediate which enables us to call another stream operation (e.g. forEach) on the result.

```
memberNames.stream().filter((s) -> s.startsWith("A"))
                            .forEach(System.out::println);


Output:

Amitabh
Aman
```

4.1.2. Stream.map()

The intermediate operation map converts each element into another object via the given function. The following example converts each string into an upper-cased string. But you can also use map to transform each object into another type.

```
memberNames.stream().filter((s) -> s.startsWith("A"))
                            .map(String::toUpperCase)
                            .forEach(System.out::println);


Output:

AMITABH
AMAN
```

4.1.2. Stream.sorted()

Sorted is an intermediate operation which returns a sorted view of the stream. The elements are sorted in natural order unless you pass a custom Comparator.

```
memberNames.stream().sorted()
                           .map(String::toUpperCase)
                           .forEach(System.out::println);
Output:

AMAN
AMITABH
LOKESH
RAHUL
SALMAN
SHAHRUKH
SHEKHAR
YANA
```

Keep in mind that sorted does only create a sorted view of the stream without manipulating the ordering of the backed collection. The ordering of memberNames is untouched.

4.2. Terminal operations

Terminal operations return a result of a certain type instead of again a Stream.

4.2.1. Stream.forEach()

This method helps in iterating over all elements of a stream and perform some operation on each of them. The operation is passed as lambda expression parameter.

```
memberNames.forEach(System.out::println);
```

4.2.2. Stream.collect()

`collect()` method used to receive elements from a steam and store them in a collection and mentioned in parameter function.

```
List<String> memNamesInUppercase = memberNames.stream().sorted()
                                        .map(String::toUpperCase)
                                        .collect(Collectors.toList());


System.out.print(memNamesInUppercase);

Outpout: [AMAN, AMITABH, LOKESH, RAHUL, SALMAN, SHAHRUKH, SHEKHAR,
YANA]
```

4.2.3. Stream.match()

Various matching operations can be used to check whether a certain predicate matches the stream. All of those operations are terminal and return a boolean result.

```
boolean matchedResult = memberNames.stream()
                            .anyMatch((s) -> s.startsWith("A"));


System.out.println(matchedResult);

matchedResult = memberNames.stream()
                            .allMatch((s) -> s.startsWith("A"));
```

```
System.out.println(matchedResult);

matchedResult = memberNames.stream()
                           .noneMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult);

Output:

true
false
false
```

4.2.4. Stream.count()

Count is a terminal operation returning the number of elements in the stream as a long.

```
long totalMatched = memberNames.stream()
                           .filter((s) -> s.startsWith("A"))
                           .count();

System.out.println(totalMatched);

Output: 2
```

4.2.5. Stream.reduce()

This terminal operation performs a reduction on the elements of the stream with the given function. The result is an Optional holding the reduced value.

```
Optional<String> reduced = memberNames.stream()
                           .reduce((s1,s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);

Output:
Amitabh#Shekhar#Aman#Rahul#Shahrukh#Salman#Yana#Lokesh
```

5. Stream short-circuit operations

Though, stream operations are performed on all elements inside a collection satisfying a predicate, It is often desired to break the operation whenever a matching element is encountered during iteration. In external iteration, you will do with if-else block. In internal iteration, there are certain methods you can use for this purpose. Let's see example of two such methods:

5.1. Stream.anyMatch()

This will return true once a condition passed as predicate satisfy. It will not process any more elements.

```
boolean matched = memberNames.stream()
                             .anyMatch((s) -> s.startsWith("A"));

System.out.println(matched);

Output: true
```

5.2. Stream.findFirst()

It will return first element from stream and then will not process any more element.

```
String firstMatchedName = memberNames.stream()
                        .filter((s) -> s.startsWith("L"))
                        .findFirst().get();

System.out.println(firstMatchedName);

Output: Lokesh
```

Stream Operations

Intermediate Operations

- filter()
- map()
- flatMap()
- distinct()
- sorted()
- peek()
- limit()
- skip()

Terminal Operations

- forEach()
- forEachOrdered()
- toArray()
- reduce()
- collect()
- min()
- max()

- count()
- anyMatch()
- allMatch()
- noneMatch()
- findFirst()
- findAny()