

Theory: Stream pipelines

🕒 31 minutes 0 / 5 problems solved

Skip this topic

Start practicing

489 users solved this topic. Latest completion was 1 day ago.

By this topic, you have already learned different types of stream operations such as `filter`, `map` and `reduce`. Now is the time to start using these operations together and get into more details on the real stream pipelines. In a sense, the idea of this topic is to consolidate your knowledge about streams and to guide you through more complex practice exercises.

§1. More about operations in stream pipelines

As a rule, production-ready streams contain multiple operations at once. It is possible to distinguish the following kinds of operations:

- filtering: using `filter` or other methods to skip some of the elements like `skip`, `limit`, `takeWhile` and so on;
- mapping or modifying stream elements: for example, sorting or removing duplicates;
- reducing or combining: `reduce`, `max`, `min`, `collect`, `count`, `findAny`, and so on.

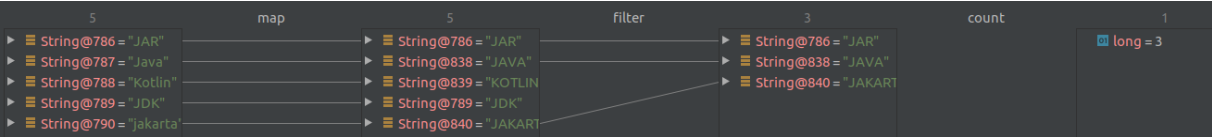
This is not a coincidence. These groups of operations compose a standard data pipeline in many information systems and streams are well suited to simulate them.

Let's consider an example of a stream with several operations. Suppose, there is a list of strings named `words`. We would like to count the total number of words that start with `"JA"`. The case is not important: `"ja"`, `"jA"`, and `"Ja"` are suitable as well.

Here is our solution with the `map`, `filter` and `count` operations.

```
1 List<String> words = List.of("JAR", "Java", "Kotlin", "JDK", "jakarta");
2
3 long numberOfWords = words.stream()
4
5     .map(String::toUpperCase)           // convert all words to upper case
6     .filter(s -> s.startsWith("JA"))    // filter words using a prefix
7     .count();                           // count the suitable words
8
9 System.out.println(numberOfWords); // 3
```

Here is a picture that explains how this stream works:



If you use IntelliJ IDEA as your primary IDE and would like to visualize stream pipelines, you can [read this article](#) and install [the Java Stream Debugger plugin](#).

It is obvious, that the result is 3 because the list contains only three suitable words (`"JAR"`, `"Java"`, `"jakarta"`).

§2. The order of execution

But there is also one less obvious thing in the case of the previous example: the order of operations in this stream. It seems that the `filter` operation is only called after the `map` operation has converted all the elements to the upper case. But that is not always true. We can see it by ourselves by adding the `peek` operation to print the intermediate elements of the stream.

Current topic:

[Stream pipelines](#) ...

Topic depends on:

- ✗ [Function composition](#) ...
- ✗ [Stream filtering](#) ...
- ✗ [Map and flatMap](#) ...
- ✗ [Reduction methods](#) ...
- ✗ [Streams of primitives](#) ...

Topic is required for:

[Parallel streams](#) ...

Table of contents:

- [1 Stream pipelines](#)
- [§1. More about operations in stream pipelines](#)
- [§2. The order of execution](#)
- [§3. Streams with custom classes](#)
- [§4. Mapping and reducing functions](#)
- [§5. Conclusion](#)
- [Feedback & Comments](#)

[As Javadoc says](#), the `peek` method exists mainly to support debugging, where you want to see the elements as they flow past a certain point in a pipeline. Do not forget to remove it after debugging is completed.

After adding the `peek` operation before and after `filter`, the stream will look like this:

```
1 long numberOfWords = words.stream()
2     .map(String::toUpperCase)
3     .peek(System.out::println)
4     .filter(s -> s.startsWith("JA"))
5     .peek(System.out::println)
6     .count();
```

And here is its output:

```
1 JAR
2 JAR
3 JAVA
4 JAVA
5 KOTLIN
6 JDK
7 JAKARTA
8 JAKARTA
```

This output actually means, that the `filter` operation is applied to an element right after the element was mapped.

Do not try to predict the order of operations in a complex stream. Depending on the operations, the actual execution order may slightly differ from the expected one because of the internal stream optimization. The main point is that a stream will produce the result regardless of the execution order.

§3. Streams with custom classes

In real situations, streams often process custom classes designed specifically for the program.

Let's assume that we have the `Event` class that represents a public event, such as a conference, a film premiere, or a concert. It has two fields:

- `beginning` (`LocalDate`) is a date when the event happens;
- `name` (`String`) that is the name of the event (for instance, `"JavaOne – 2017"`).

Also, the class has getters and setters for each field with the corresponding names.

We also have a list of instances named `events`.

Let's find all names of events that will occur from December 30 to December 31, 2017 (inclusively).

```
1 LocalDate after = LocalDate.of(2017, 12, 29);
2 LocalDate before = LocalDate.of(2018, 1, 1);
3
4 List<String> suitableEvents = events.stream()
5     .filter(e -
> e.getBeginning().isAfter(after) && e.getBeginning().isBefore(before))
6     .map(Event::getName)
7     .collect(Collectors.toList());
```

The code above finds names of all suitable events and collects them to a new list of strings. The map methods allow us to make the transition from `Event` objects to the `String` objects.

§4. Mapping and reducing functions

Since functions are presented as objects of certain classes, we can `map` and `reduce` them similar to regular stream elements.

For example, we have a collection of integer predicates. Let's negate each predicate by using a map operator and then conjunct all predicates into one by using a reduce operator.

```
1 |
public static IntPredicate negateEachAndConjunctAll(Collection<IntPredicate> predi
cates) {
2 |     return predicates.stream()
3 |         .map(IntPredicate::negate)
4 |         .reduce(n -> true, IntPredicate::and);
5 | }
```


In this example, `map` negates each predicate in a stream and then `reduce` conjuncts all predicates into one. The **initial value** (seed) of reducing is a predicate that is always `true`, because it's the neutral value for conjunction.

So, the input predicates `P1(x), P2(x), ..., Pn(x)` will be reduced into one predicate `Q(x) = not P1(x) and not P2(x) and ... and not Pn(x)`. Of course, this is not the most frequent way to apply streams, but it's worth knowing that such use is also possible.

\$5. Conclusion

As you have seen, stream pipelines allow writing short and readable code to perform various evaluations. It is possible to combine many different operations in a single powerful stream. Keep in mind, that the order of performing the operation is determined by the stream itself. Do not try to influence it in any way.

 Report a typo

59 users liked this theory.  didn't like it. What about you?



Start practicing

[Comments \(1\)](#)[Hints \(0\)](#)[Useful links \(0\)](#)[Show discussion](#)