


Theory: Taking elements

 12 minutes

0 / 4 problems solved

Skip this topic

Start practicing

465 users solved this topic. Latest completion was about 12 hours ago.

If you'd like to take only a certain number of elements from a stream or skip some of them, you can invoke the `limit(n)` or `skip(m)` methods. But what if you need to take or skip some elements until some condition is true? Starting from Java 9, streams have two convenient methods to do this. They can *take* or *drop* the longest contiguous subsequence of elements from the stream based on the given predicate.

§1. The takeWhile method

The `takeWhile` method takes elements from the stream until the first inappropriate element is encountered. This element and all the rest are discarded. Which element is inappropriate is determined by the predicate passed to this method. If all elements will match the predicate, the result will contain the same elements as the initial stream.

In the following example, we create a stream of numbers and take elements while they are greater than zero.

```
1 List<Integer> numbers =
2     Stream.of(3, 5, 1, 2, 0, 4, 5)
3         .takeWhile(n -> n > 0)
4         .collect(Collectors.toList());
5
6 System.out.println(numbers); // [3, 5, 1, 2]
```

The `takeWhile` method stops after taking the element `0`, because the condition becomes `false`.

We believe that the method is clear enough not to dwell on it for a long time.

§2. The dropWhile method

There is also an opposite method called `dropWhile`. It drops the elements which match the given predicate until the first element does not match it. This and all the remaining elements are included in the result. If all elements will match the predicate, the result will be an empty stream.

Here is the same example as before, but with `dropWhile` instead of `takeWhile`.

```
1 List<Integer> numbers =
2     Stream.of(3, 5, 1, 2, 0, 4, 5)
3         .dropWhile(n -> n > 0)
4         .collect(Collectors.toList());
5
6 System.out.println(numbers); // [0, 4, 5]
```

The `dropWhile` method stops dropping right after taking the element `0`, because the condition becomes `false`. The elements `0`, `4`, and `5` remain in the stream.

§3. The case of unordered streams

Both methods `takeWhile` and `dropWhile` work well in case of ordered streams. Such streams are created from ordered collections (e.g. lists) or obtained during operations (like sorting). But what if we are dealing with an unordered collection such as a set? It turns out that in this case the behavior of both operations is nondeterministic.

As an example, suppose we have a set of Java conferences with one Kotlin conference. We can try to keep taking conferences names as long as they start with `"J"` and stop at the first inappropriate conference.

Current topic:

[Taking elements](#) ...

Topic depends on:

✗ [Functional data processing with streams](#) ...

Table of contents:

[1 Taking elements](#)

[§1. The takeWhile method](#)

[§2. The dropWhile method](#)

[§3. The case of unordered streams](#)

[Feedback & Comments](#)

```
1 Set<String> conferences = Set.of(
2     "JokerConf", "JavaZone",
3     "KotlinConf", "JFokus"
4 );
5
6 conferences.stream()
7     .takeWhile(word -> word.startsWith("J"))
8     .forEach(System.out::println);
```

Since we don't know the order of the names in a set, the result of this code may always be different, containing from 0 to 3 conference names. This is definitely not what we wanted here.

Using `takeWhile` and `dropWhile` with unordered streams leads to nondeterministic behavior, and as a result, to bugs.

How to achieve repeatable results in this case?

- to use `List` instead of `Set`;
- to add the `sorted()` method before `takeWhile()` to keep always the same order of elements within the stream.

This concludes our consideration of taking / dropping elements from a stream.

 Report a typo

48 users liked this theory. 1 didn't like it. What about you?



Start practicing

[Comments \(0\)](#)

[Hints \(0\)](#)

[Useful links \(0\)](#)

[Show discussion](#)