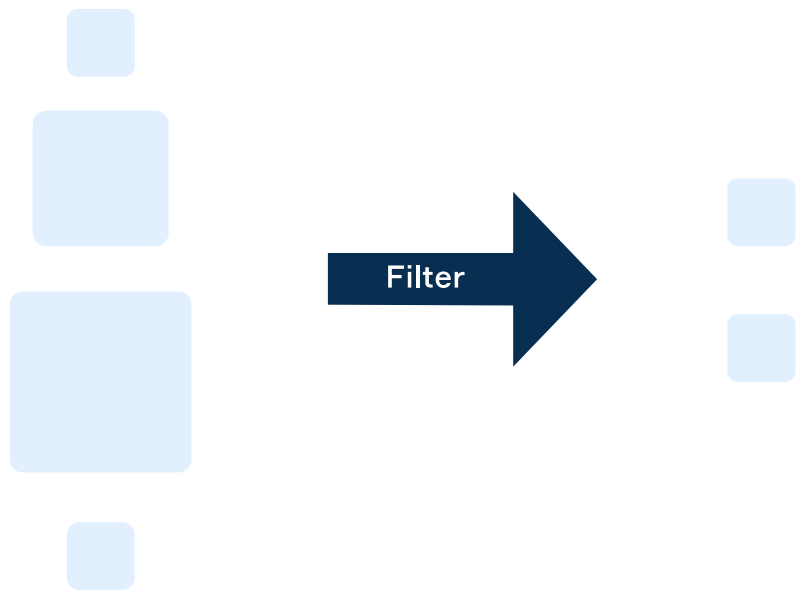
Theory: Stream filtering

16 minutes 0 / 5 problems solved

Filtering is an important operation that allows us to obtain only those elements of the collection that meet a specified condition. For example, to get songs over seven minutes long among all your music library, we can filter songs by their duration. In this topic, we'll figure out how to use Java Stream API intermediate filter operation to cope with such challenges.

§1. The filter method

To filter elements, streams provide the `filter` method. It returns a new stream consisting only of those elements that match the given predicate.



As an example, here is a list of prime numbers (a prime number is a whole number greater than 1 whose only factors are 1 and itself):

```
List<Integer> primeNumbers = Arrays.asList(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31)
;
```

We'd like to create a new list consisting of prime numbers that belong to the range from 11 to 23 (inclusively).

```
List<Integer> filteredPrimeNumbers = primeNumbers.stream() // create a stream from
 the list
          .filter(n -> n >= 11 && n <= 23) // filter elements
          .collect(Collectors.toList());   // collect elements in a new list
```

So, the `filteredPrimeNumbers` list is:

```
[11, 13, 17, 19, 23]
```

Since the `filter` method takes a predicate, it is possible to instantiate an object directly and pass it to the method.

```
 Predicate<Integer> between11and23 = n -
> n >= 11 && n <= 23; // instantiate the predicate


List<Integer> filteredPrimeNumbers = primeNumbers.stream() // create a stream from
 the list

      .filter(between11and23)        // pass the predicate to the filter method
          .collect(Collectors.toList()); // collect elements in a new list
```

Of course, the result is the same as before.

§2. Using multiple filters

Sometimes, two or more filters are used together. For example:

Current topic:

Stream filtering ...

Topic depends on:

× Functional data processing with streams ...

Topic is required for:

Stream pipelines ...

Table of contents:

- to separate a complex logic from a single filter;
- to filter the stream, then process it by other methods and then filter again.

Let's consider an example. Given a list of programming languages with empty strings.

```
List<String> programmingLanguages = Arrays.asList("Java", "", "scala", "Kotlin", "
", "clojure");
```

We'd like to count how many programming languages start with an upper letter ignoring all the empty strings.

```
long count = programmingLanguages.stream()
        .filter(lang -> lang.length() > 0) // consider only non-
empty strings
        .filter(lang -> Character.isUpperCase(lang.charAt(0)))
        .count(); // count suitable languages
```

The count is 2 (`"Java"` , `"Kotlin"`).

These two filter operations can be replaced with a single operation that takes a complex predicate:

```
filter(lang -> lang.length() > 0 && Character.isUpperCase(lang.charAt(0)))
```

But this code is a little less readable.

Report a typo

72 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

Comments (1) Hints (0) Useful links (0) Show discussion