718 users solved this topic. Latest completion was about 1 hour ago.

Theory: Functional data processing with streams

29 minutes 0 / 5 problems solved

Skip this topic Start practicing

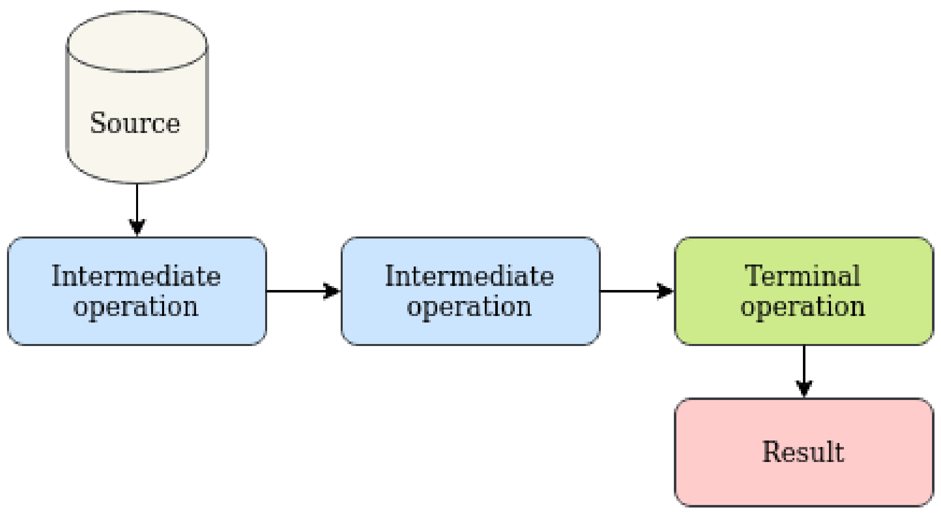
You already know how to process sequences of objects using loops and collections. However, this approach is quite error-prone due to mutable variables and complex looping logic. It can also make your code less readable which results in complications with its further development.

Java 8 gave us a new tool called **Stream API** that provides a functional approach to processing collections of objects. By using Stream API, a programmer doesn't need to write explicit loops since each stream has an internal optimized loop. Streams allow us to focus on the question "**what should the code do?**" instead of "**how should the code do it?**". In addition, such an approach makes parallelizing easy.

In this topic, you will learn the basics of this new data processing concept, how to create streams, and which operations can be performed on them.

§1. The basic concept of streams

In a sense, a **stream** reminds a collection. But it does not actually store elements. Instead, it conveys elements from a **source** such as a collection, a generator function, a file, an I/O channel, another stream, or something else, and then processes the elements by using a sequence of predefined operations combined into a single pipeline.



There are three stages of working with a stream:

1. Obtaining the stream from a source.
2. Performing intermediate operations with the stream to process data.
3. Performing a terminal operation to produce a result.

It is important that a stream always has only a single terminal operation and an arbitrary number of intermediate operations. We will learn the main differences between them further within this topic.

Now you got the basic concept of streams and it is time to take a look at some code!

§2. A loop vs a stream example

All classes associated with streams are located in the `java.util.stream` package. There are several common stream classes: `Stream<T>`, `IntStream`, `LongStream` and `DoubleStream`. While the generic stream works with reference types, others work with the corresponding primitive types. In this topic, we will only consider the generic stream.

Let's consider a simple example. Suppose we have a list of numbers and we'd like to count the numbers that are greater than `5`:

Current topic:

Functional data processing with streams ···

Topic depends on:

× List ···

× Set ···

× Standard functional interfaces ···

× Optional ···

Topic is required for:

Stream filtering ···

Map and flatMap ···

Reduction methods ···

Streams of primitives ···

Taking elements ···

Infinite streams ···

Table of contents:

```java
List<Integer> numbers = List.of(1, 4, 7, 6, 2, 9, 7, 8);
```

A "traditional" way to do it is to write a loop like the following:

```java
long count = 0;
for (int number : numbers) {
    if (number > 5) {
        count++;
    }
}
System.out.println(count); // 5
```

This code prints "5" because the initial list contains only five numbers that are greater than 5 (7, 6, 9, 7, 8).

A loop with a filtering condition is a commonly used construct in programming. It is possible to simplify this code by rewriting it using a stream:

```java
long count = numbers.stream()
        .filter(number -> number > 5)
        .count(); // 5
```

Here we get a stream from the `numbers` list, then filter its elements by using a predicate lambda expression and then count the numbers that satisfy the condition. Although this code produces the same result, it is easier to read and modify. For example, we can easily change it to skip the first four numbers from the list.

```java
long count = numbers.stream()
        .skip(4)  // skip 1, 4, 7, 6
        .filter(number -> number > 5)
        .count();  // 3
```

See how easy it is! We just invoke another operation on the stream to make it work. Performing the same modification when using the loop will be harder.

The processing of a stream is performed as a chain of method calls separated by dots with a single terminal operation. To improve readability it is recommended to put each call into a new line if the stream contains more than one operation.

Now you have an idea of how streams look in code. Let's take a look at their methods in detail.

§3. Creating streams

There are a lot of ways to create a stream including using a list, a set, a string, an array, and so on as a source.

1) The most common way to create a stream is to take it from a collection. Any collection has the `stream()` method for this purpose.

```java
List<Integer> famousNumbers = List.of(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55);
Stream<Integer> numbersStream = famousNumbers.stream();

Set<String> usefulConcepts = Set.of("functions", "lazy", "immutability");
Stream<String> conceptsStream = usefulConcepts.stream();
```

2) It is also possible to obtain a stream from an array:

```java
Stream<Double> doubleStream = Arrays.stream(new Double[]
{ 1.01, 1d, 0.99, 1.02, 1d, 0.99 });
```

3) or directly from some values:

```java
Stream<String> persons = Stream.of("John", "Demetra", "Cleopatra");
```

4) or concatenate other streams together:

```java
Stream<String> stream1 = Stream.of(/* some values */);
Stream<String> stream2 = Stream.of(/* some values */);
Stream<String> resultStream = Stream.concat(stream1, stream2);
```

5) There are some possibilities to create empty streams (that can be used as return values from methods):

```java
Stream<Integer> empty1 = Stream.of();
Stream<Integer> empty2 = Stream.empty();
```

There are also other methods to create streams from different sources: from a file, from I/O stream, and so on. We will consider some of them in the relevant topics. The purpose of this topic is to give you an introduction, not to list all existing methods.

§4. Groups of stream operations

All stream operations are divided into two groups: **intermediate** and **terminal** operations.

- **Intermediate operations** are not evaluated immediately when invoking. They simply return new streams to call next operations on them. Such operations are known as **lazy** because they do not actually do anything useful.
- **Terminal operations** begin all evaluations with the stream to produce a result or to make a side-effect. As we mentioned before, a stream always has only one terminal operation.

Once a terminal operation has been evaluated, it is impossible to reuse the stream again. If you try doing that the program will throw `IllegalStateException`.

Streams provide a huge number of operations to be performed on the elements. The only way to learn them all is good practice. That is why we give you only a small list of operations for each group. Other operations are considered in the following topics.

Intermediate operations

- `filter` returns a new stream that includes the elements that match a **predicate**;
- `limit` returns a new stream that consists of the first `n` elements of this stream;
- `skip` returns a new stream without the first `n` elements of this stream;
- `distinct` returns a new stream consisting of only unique elements according to results of `equals`;
- `sorted` returns a new stream that includes elements sorted according to the natural order or a given **comparator**;
- `peek` returns the same stream of elements but allows observing the current elements of the stream for debugging;
- `map` returns a new stream that consists of the elements that were obtained by applying a function (i.e. transforming each element).

Terminal operations

- `count` returns the number of elements in the stream as a `long` value;
- `max` / `min` returns `Optional` maximum / minimum element of the stream according to the given comparator;
- `reduce` combines values from the stream into a single value (an aggregate value);
- `findFirst` / `findAny` returns the first / any element of the stream as an `Optional`;
- `anyMatch` returns `true` if at least one element matches a predicate (see also: `allMatch`, `noneMatch`);
- `forEach` takes a **consumer** and applies it to each element of the stream (for example, printing it);
- `collect` returns a collection of the values in the stream;

- `toArray` returns an array of the values in a stream.

Such operations (methods) as `filter`, `map`, `reduce`, `forEach`, `anyMatch` and some others are called **higher-order functions** because they accept other functions as the arguments.

Some terminal operations return `Optional` because the stream can be empty and you need to specify a default value or an action if it is empty.

§5. An example

As an example, let's use stream operations to print all names of companies without duplicates in the upper case.

```
List<String> companies = List.of(
        "Google", "Amazon", "Samsung",
        "GOOGLE", "amazon", "Oracle"
);

companies.stream()
        .map(String::toUpperCase) // transform each name to the upper case
        .distinct() // intermediate operation: keep only unique words
        .forEach(System.out::println); // print every company
```

Here we use two intermediate operations (`map` and `distinct`) and one terminal operation `forEach`.

The code prints only unique company names as we expected:

```
GOOGLE
AMAZON
SAMSUNG
ORACLE
```

Using methods references (like `String::toUpperCase` or `System.out::println`) make your stream-based code even more readable than using lambda expressions. It is recommended to use this way or small single-line lambda expressions rather than complex long body lambda expressions.

§6. Conclusion

Stream API makes data processing easier by separating a complex logic into a sequence of well-defined operations ("stages"). It is much easier to read and modify such code than when we use classic loops and mutable states.

There are a few points you should keep in mind at the end of this topic:

- a stream can be created from any collection by invoking the `stream()` method;
- there are two types of operations: intermediate and terminal;
- an intermediate operation just returns a new stream;
- a terminal operation starts the evaluation process;
- it is impossible to reuse a stream that has been evaluated once;
- there are many methods for processing streams, some of them taking functions as arguments.

All other information can always be found in the documentation or by using tips in your IDE when solving problems.

Report a typo

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

Start practicing

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