Java → Functional programming → Functional streams → Streams of primitives

Theory: Streams of primitives

13 minutes 0 / 4 problems solved

Skip this topic Start practicing

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

This topic duplicates some parts of previous ones, since they are still improving.

The generic `Stream<T>` class is used to process objects which are always represented by reference types. For example, to work with integers, it's possible to create `Stream<Integer>` that wraps primitive `int`'s into the `Integer` class. But this is not an efficient way to work with integers, since it needs additional wrapper objects. Fortunately, there are three primitive specialized types called `IntStream`, `LongStream`, and `DoubleStream` which can effectively process primitive values without extra boxing.

There is no `CharStream`, `ByteStream`, `ShortStream`, `FloatStream` and `BooleanStream`.

§1. Creating primitive type streams

There are lots of ways to create primitive type streams. Some of the ways are suitable for all streams while others are not.

- Passing elements in the `of` method:

```
IntStream ints = IntStream.of(1, 2, 3);
LongStream longs = LongStream.of(1, 2, 3);
DoubleStream doubles = DoubleStream.of(12.2, 18.1);
```

This looks quite similar to collections. It is also possible to create an empty stream invoking `IntStream.of()` or `IntStream.empty()`.

- From an array of primitives:

```
IntStream numbers = Arrays.stream(new int[]{1, 2, 3});
```

This way works for all types of primitive specialized streams. It is also possible to specify `start` (inclusive) and `end` (exclusive) positions to create a stream only from a subarray.

- For `IntStream` and `LongStream` it's possible to invoke `range()` and `rangeClosed()` to create streams from ranges.

```
IntStream numbers = IntStream.range(10, 15); // from 10 (incl) to 15 (excl)
LongStream longs = LongStream.rangeClosed(1_000_000, 2_000_000); // it includes both borders
```

The difference is the method `rangeClosed` includes its upper bound while `range` does not.

- Getting `IntStream` from a string:

```
IntStream stream = "aibohphobia".chars(); // It returns IntStream!
```

This only works for `IntStream` since characters can be represented as `int`'s.

Other ways to create primitive type streams are the same as for generic streams: `generate`, `iterate`, `concat` and so on. Here is an example of generating `DoubleStream` with ten random numbers and printing them:

```
DoubleStream.generate(Math::random)
        .limit(10)
        .forEach(System.out::println);
```

Current topic:

Streams of primitives ···

Topic depends on:

× Functional data processing with streams ···

Topic is required for:

Stream pipelines ···

Table of contents:

Now let's look at some operations with primitive streams.

§2. Additional operations

The primitive streams have all the same methods as generic streams, but their methods accept primitive specialized functions as arguments. For example the `forEach` method of `IntStream` takes `IntConsumer`, but not `Consumer<Integer>`. Fortunately, this does not affect the possibilities of the streams.

There are a few additional aggregating operations such as `min`, `max`, `average` and `sum`. The first three return an optional object which represents a result or nothing since the initial stream can be empty.

> Note, actually `Stream<T>` also provides `min` and `max` but its methods need a comparator as the argument.

The following code demonstrates the methods:

```
int[] numbers = { 10, 11, 25, 14, 22, 21, 18 };

int max = IntStream.of(numbers).max().getAsInt();
System.out.println(max); // 25

int min = IntStream.of(numbers).min().getAsInt();
System.out.println(min); // 10

double avg = IntStream.of(numbers).average().orElse(0.0);

System.out.println(avg); // 17.2857...



int sum = IntStream.of(numbers).sum();

System.out.println(sum); // 121
```

It is also possible to calculate these aggregates at once using a single invocation of the `summaryStatistics` method.

```
IntSummaryStatistics stat = IntStream.rangeClosed(1, 55_555).summaryStatistics();

System.out.println(String.format("Count: %d, Min: %d, Max: %d, Avg: %.1f",

       stat.getCount(), stat.getMin(), stat.getMax(), stat.getAverage()));
```

Here are the results:

```
Count: 55555, Min: 1, Max: 55555, Avg: 27778.0
```

Wow! And no loops here.

§3. Transforming streams

You can perform various transformations of primitive type streams.

- Transforming a primitive type stream to another one using `asDoubleStream()` for `IntStream` and `LongStream`, or `asLongStream` for `IntStream` only.

Here is an example which converts a stream of integers into a stream of doubles:

```
IntStream.of(1, 2, 3, 4)
        .asDoubleStream()
        .forEach(System.out::println); // it prints doubles 1.0, 2.0, ...
```

- Transforming a primitive type stream into the generalized stream using the `boxed()` method (i.e., `IntStream` → `Stream<Integer>`). All the primitive type streams have this.

```
Stream<Integer> streamOfNumbers = IntStream.range(1, 10).boxed();
```

- **Transforming a generalized stream into a stream of primitives** can be done invoking one of `mapToInt()`, `mapToLong()` or `mapToDouble()` methods with the `i -> i` lambda expression as the argument:

```
List<Integer> numbers = List.of(1, 5, 9);
int sum = numbers.stream().mapToInt(i -> i).sum(); // 15
```

This can be especially useful when you want to invoke one of the specific primitive stream's methods.

§4. Summary

We've considered three primitive specialized streams which are similar to the generalized stream but have some features:

- their methods take primitive specialized functions as arguments;
- they have additional methods such as `range`, `sum`, `average`, `summaryStatistics` and some others;
- they have the performance benefits since there is no need to perform boxing/unboxing operations;
- they can be converted in the generalized stream and vice versa.

Report a typo

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

😍 🙂 😐 🙁 😡

Start practicing

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