DZone

An Introduction to Functional Programming in Java 8 (Part 3): Streams

by Niklas Wuensche MVB · Feb. 27, 17 · Java Zone

Download Microservices for Java Developers: A hands-on introduction to frameworks and containers. Brought to you in partnership with Red Hat.

In the last part, we learned about the Optional type and how to use it correctly.

Today, we will learn about `Streams`, which you use as a functional alternative of working with Collections. Some method were already seen when we used Optionals, so be sure to check out the part about Optionals.

Where Do We Use Streams?

You might ask what's wrong with the current way to store multiple objects? Why shouldn't you use Lists, Sets, and so on anymore?

I want to point out: Nothing is wrong with them. But when you want to work functional (what you hopefully want after the last parts of this blog), you should consider using them. The standard workflow is to convert your data structure into a Stream. Then, you want to work on them in a functional manner and, in the end, you transform them back into the data structure of your choice.

And that's the reason we will learn to transform the most common data structures into streams.

Why Do We Use Streams?

Streams are a wonderful new way to work with data collections. They were introduced in Java 8. One of the many reasons you should use them is the *Cascade* pattern that Streams use. This basically means that almost every Stream method returns the Stream again, so you can continue to work with it. In the next sections, you will see how this works and that it makes the code nicer.

Streams are also immutable. So every time you manipulate it, you create a new Stream.

Another nice thing about them is that they respect the properties of *fP*. If you convert a Data Structure into a Stream and work on it, the original data structure won't be changed. So no side effects here!

How to Convert Data Structures Into Streams

Convert Multiple Objects Into a Stream

Convert Multiple Objects into a Stream

If you want to make a Stream out of some objects, you can use the method `Stream.of()`

```
public void convertObjects() {
    Stream<String> objectStream = Stream.of("Hello", "World");
}
```

Converting Collections (Lists, Sets, etc.) and Arrays

Luckily, Oracle has thought through the implementation of Streams in Java 8. Every Class that implements java.util.Collection<T> has a new method called `stream()`, which converts the collection into a Stream. Also, Arrays can be converted easily with `Arrays.stream(array)`. It's as easy as it gets.

```
public void convertStuff() {
    String[] array = {"apple", "banana"};
    Set<String> emptySet = new HashSet<>();
    List<Integer> emptyList = new LinkedList<>();

    Stream<String> arrayStream = Arrays.stream(array);
    Stream<String> setStream = emptySet.stream();
    Stream<Integer> listStream = emptyList.stream();
}
```

However, normally, you won't store a Stream in an object. You just work with them and convert them back into your desired data structure.

Working With Streams

As I already said, Streams are the way to work with data structures functional. And now we will learn about the most common methods to use.

As a side note: In the next sections, I will use `T` as the type of the objects in the Stream.

Methods We Already Know

You can also use some methods we already heard about when we learned about Optionals with Streams.

Map

This is pretty straightforward. Instead of manipulating one item, which might be in the Optional, we manipulate all items in a stream. So if you have a function that squares a number, you can use Map to use this function over multiple numbers without writing a new function for lists.

```
public void showMap() {
    Stream.of(1, 2, 3)
        .map(num -> num * num)
        .forEach(System.out::println); // 1 4 9
}
```

```
}
```

flatMap

Like with Optionals, we use flatMap to go, for example, from a Stream<List<Integer>> to Stream<Integer>. If you want to know more about them, look into part two.

Here, we want to concat multiple Lists into one big List.

```
public void showFlatMapLists() {
    List<Integer> numbers1 = Arrays.asList(1, 2, 3);
    List<Integer> numbers2 = Arrays.asList(4, 5, 6);

    Stream.of(numbers1, numbers2) //Stream<List<Integer>>
        .flatMap(List::stream)  //Stream<Integer>
        .forEach(System.out::println); // 1 2 3 4 5 6
}
```

And in these examples, we already saw another Stream method, `forEach()`, which I will describe now.

Common Stream Methods

forEach

The `forEach` method is like the `ifPresent` method from Optionals, so you use it when you have side effects. As already shown, you use it to, for example, print all objects in a stream. forEach is one of the few Stream methods that doesn't return the Stream, so you use it as the last method of a Stream and only once.

You should be careful when using forEach because it causes side effects, which we don't want to have. So think twice if you could replace it with another method without side effects.

```
public void showForEach() {
    Stream.of(0, 1, 2, 3)
        .forEach(System.out::println); // 0 1 2 3
}
```

Filter

Filter is a really basic method. It takes a 'test' function that takes a value and returns boolean. So it tests every object in the Stream. If it passes the test, it will stay in the Stream. Otherwise, it will be taken out.

This 'test' function has the type `Function<T, Boolean>`. In the JavaDoc, you will see that the test function really is of the type `Predicate<T>`. But this is just a short form for every function that takes one parameter and returns a boolean.

```
public void showFilter() {
```

```
    Stream.of(0, 1, 2, 3)
        .filter(num -> num < 2)
        .forEach(System.out::println); // 0 1
}
```

Functions that can make your life way easier when creating 'test' functions are `Predicate.negate()` and `Objects.nonNull()`.

The first one basically negates the test. Every object that doesn't pass the original test will pass the negated test and vice versa.

The second one can be used as a method reference to get rid of every null object in the Stream. This will help you to prevent `NullPointerExeptions` when, for example, mapping functions.

```
public void negateFilter() {
    Predicate<Integer> small = num -> num < 2;

    Stream.of(0, 1, 2, 3)
        .filter(small.negate()) // Now every big number passes
        .forEach(System.out::println); // 2 3
}

public void filterNull() {
    Stream.of(0, 1, null, 3)
        .filter(Objects::nonNull)
        .map(num -> num * 2) // without filter, you would've got a NullPointerExeception
        .forEach(System.out::println); // 0 2 6
}
```

Collect

As I already said, you want to transform your stream back into another data structure. And that is what you use Collect for. And most of the time, you convert it into a List or a Set.

```
public void showCollect() {
    List<Integer> filtered = Stream.of(0, 1, 2, 3)
        .filter(num -> num < 2)
        .collect(Collectors.toList());
}
```

But you can use collect for much more. For example, you can join Strings. Therefore, you don't have the nasty delimiter at the end of the string.

```
public void showJoining() {
    String sentence = Stream.of("Who", "are", "you?")
        .collect(Collectors.joining(" "));
```

```
            .collect(Collectors.joining(" "));

        System.out.println(sentence); // Who are you?

    }
```

Shortcuts

These are methods that you could mostly emulate by using map, filter, and collect, but these shortcut methods are meant to be used because they declutter your code.

Reduce

Reduce is a very cool function.

It takes a start parameter of type T and a Function of type `BiFunction<T, T, T>`. If you have a BiFunction where all types are the same, `BinaryOperator<T>` is a shortcut for that.

It basically stores all objects in the stream to one object. You can concat all Strings into one String, sum all numbers, and so on. There, your start parameters would be the empty String or zero. This function helps make your code more readable if you know how to use it.

```
public void showReduceSum() {
    Integer sum = Stream.of(1, 2, 3)
        .reduce(0, Integer::sum);

    System.out.println(sum); // 6
}
```

Now I will give a little bit more information on how reduce works. Here, it sums:

- The first number with...
- ...the sum of the second and...
- ...the sum of the third and start parameter.

As you can see, this produces a long chain of functions. In the end, we have sum(1, sum(2, sum(3, 0))). And this will be computed from right to left, or from the inside out. This is also the reason we need a start parameter — because otherwise, the chain wouldn't have a point where it could end.

Sorted

You can also use Streams to sort your data structures.

The class type of the objects in the Stream doesn't even have to implement `Comperable<T>`, because you can write your own `Comperator<T>`. This is basically a `BiFunction<T, T, int>`, but Comperator is a shortcut for all BiFunctions that take two arguments of the same type and return an int.

And this int, like in the `compareTo()` function, shows us if the first object is "smaller" than the first one (int < 0), is as big as the second (int == 0), or is bigger than the second one (int > 0). The sorted function of the Stream will interpret these ints and will sort the elements with the help of them.

```
public void showSort() {
    Stream.of(3, 2, 4, 0)
        .sorted((c1, c2) -> c1 - c2)
        .forEach(System.out::println); // 0 2 3 4
}
```

Other Kinds of Streams

There are also special types of Streams that only contains numbers. With these new Streams, you also have a new set of methods.

Here, I will introduce IntStream and Sum, but there are also LongStream, DoubleStream, etc. You can read more about them in the JavaDoc.

To convert a normal Stream into an IntStream, you have to use `mapToInt` . It does exactly the same as the normal map, but you get a IntStream back. Of course, you have to give the mapToInt function another function which will return an int.

In the example, I will show you how to sum numbers without reduce, but with an IntStream.

```
public void sumWithIntStream() {
    Integer sum = Stream.of(0, 1, 2, 3)
        .mapToInt(num -> num)
        .sum();
}
```

Use Streams for Tests

Tests can also be used to test your methods. I will use the method `anyMatch` here, but `count` , `max` and so on can help you too.

If something goes wrong in your program, use `peek` to log data. It's like forEach, but also returns the stream. As always, look into the JavaDoc to find other cool methods.

`anyMatch` is a little bit like filter, but it tells you if anything passes the filter. You can use this in `assertTrue()` tests, where you just want to look if at least one object has a specific property.

In the next example, I will test whether a specific name was stored in the DB.

```
@Test
public void testIfNameIsStored() {
    String testName = "Albert Einstein";

    Datebase names = new Datebase();
```

```
        names.drop();

        db.put(testName);
        assertTrue(db.getData()
            .stream()
            .anyMatch(name -> name.equals(testName)));
    }
```

Shortcuts of Shortcuts

And now that I've showed you some shortcut methods, I want to tell you that there are many more. There are even shortcuts of shortcuts! One example would be `forEachOrdered`, which combines `forEach` and `sorted`.

If you are interested in other helpful methods, look into the JavaDoc. I'm sure you are prepared to understand it and find the methods that you need. Always remember: If your code looks ugly, there's a better method to use.

A Bigger Example

In this example, we want to send a message to every user whose birthday's today.

The User Class

A user is defined by their username and birthday. The birthdays will be in the format "day.month.year", but we won't do much checking for this in today's example.

```
public class User {

    private String username;
    private String birthday;

    public User(String username, String birthday) {
        this.username = username;
        this.birthday = birthday;
    }

    public String getUsername() {
        return username;
    }

    public String getBirthday() {
        return birthday;
    }

}
```

To store all users, we will use a List here. In a real program, you might want to switch to a DB.

```
public class MainClass {

    public static void main() {
        List<User> users = new LinkedList<>();

        User birthdayChild = new User("peter", "20.02.1990");
        User otherUser = new User("kid", "23.02.2008");
        User birthdayChild2 = new User("bruce", "20.02.1980");

        users.addAll(Arrays.asList(birthdayChild, otherUser, birthdayChild2));

        greetAllBirthdayChildren(users);
    }

    private static void greetAllBirthdayChildren(List<User> users) {
        // Next Section
    }

}
```

The Greeting

Now, we want to greet the birthday boys and girls.

So first off, we have to filter out all Users whose birthday is today. After this, we have to message them. So let's do this.

I won't implement `sendMessage(String message, User receiver)` here, but it just sends a message to a given user.

```
public static void greetAllBirthdayChildren(List<User> users) {
    String today = "20.02"; //Just to make the example easier. In production, you would us

    users.stream()
        .filter(user -> user.getBirthday().startsWith(today))
        .forEach(user -> sendMessage("Happy birthday, ".concat(user.getUsername()).concat(

}

private static void sendMessage(String message, User receiver) {
    //...
}
```

And now we can send greetings to the users. How nice and easy was that?!

Parallelism

Streams can also be executed in parallel. By default, every Stream isn't parallel, but you can use `.parallelStream()` with Streams to make them parallel.

Although it can be cool to use this to make your program faster, you should be careful with it. As shown on this site, things like sorting can be messed up by parallelism.

So be prepared to run into nasty bugs with parallel Streams, although it can make your program significantly faster.

Conclusion

That's it for today!

We have learned a lot about Streams in Java. We learned how to convert a data structure into a Stream, how to work with a Stream, and how to convert your Stream back into a data structure.

I have introduced the most common methods and when you should use them.

In the end, we tested our knowledge with a bigger example where we greeted all birthday children.

In the next part of this series, we will have a big example where we are going to use Streams. But I won't tell you the example yet, so hopefully, you'll be surprised.

Are there any Stream methods that you miss in the post? Please let me know in the comments. I'd love to hear your feedback, too!

Download Building Reactive Microservices in Java: Asynchronous and Event-Based Application Design. Brought to you in partnership with Red Hat.

Like This Article? Read More From DZone

An Introduction to Functional Programming in Java 8 (Part 4): Splitter

Java 8 Concepts: FP, Streams, and Lambda Expressions

Functional Programming in Java 8 (Part 1): Functions as Objects

Free DZone Refcard
Reactive Microservices With Lagom and Java

Topics: JAVA , JAVA 8 , FUNCTIONAL PROGRAMING , STREAMS , TUTORIAL

Published at DZone with permission of Niklas Wuensche, DZone MVB. See the original article here.

Opinions expressed by DZone contributors are their own.

Get the best of Java in your inbox.

Stay updated with DZone's bi-weekly Java Newsletter. SEE AN EXAMPLE

SUBSCRIBE

Java Partner Resources

Market Guide for Application Platforms
Lightbend

Developing Reactive Microservices: Enterprise Implementation in Java
Lightbend

Jenkins, Docker and DevOps: The Innovation Catalysts
CloudBees

Advanced Linux Commands [Cheat Sheet]
Red Hat Developer Program