DZone

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

by Niklas Wuensche MVB · Mar. 20, 17 · Java Zone

In the previous articles, we learned about the most basic ways to program functionally in Java. We used functions as Objects, Streams, and much more.

Today, we want to create a useful example with them. You will train with the paradigms of the previous parts and learn how to optimize the runtime of your program.

Example: Split a Stream via a Specific Filter

You might have stumbled over a problem like this before: You have a collection of objects and want to split them with a specific filter. After that, you want to perform some action with all elements that passed the test and another action with the ones that didn't pass the test.

The Basic (and Slow) Approach

```
public <T> void splitAndPerform(Collection<T> items, Predicate<T> splitBy, Consumer<T> pas
    items.stream()
        .filter(splitBy)
        .forEach(passed);

    items.stream()
        .filter(splitBy.negate())
        .forEach(notPassed);
}
```

This approach works, but it is pretty slow. The runtime for splitting alone is O(2n) because we go through the whole collection two times: once to get all the items that passed the test and once to get all the items that didn't pass it.

But what if we would create a splitter on our own? It'd sort all items in the collection, depending on whether or not they pass the test, into one List or another. This would lower the run time to split the collection to O(n) because you just have to call filter once, not twice.

collection to O(n) because you just have to call filter once, not twice.

So let's do this.

The Better Approach

1. Split the Collection

The first step to building our splitter is to, you guessed it, split it.

In our `splitBy()` function, we want to take a `Predeciate<T>` as a parameter and return a new `Splitter` object, which is basically an object that consists of two Lists. One List consists of all the objects that passed the test, and the other one of all the objects that didn't pass the test.

```
public class Splitter<T> {

    private List<T> passed;
    private List<T> notPassed;

    private Splitter(List<T> passed, List<T> notPassed) {
        this.passed = passed;
        this.notPassed = notPassed;
    }

    public static <T> Splitter<T> splitBy(Collection<T> items,Predicate<T> test) {
        List<T> passed = new LinkedList<T>();
        List<T> notPassed = new LinkedList<T>();

        items.stream()
                .forEach(item -> {
                    if(test.test(item)){
                        passed.add(item);
                        return;
                    }
                    notPassed.add(item);
                });

        return new Splitter<T>(passed, notPassed);
    }

}
```

As you can see, we used the factory method pattern to make the creation of the Splitter nicer.

Now that we can create a Splitter object, we want to give it some functionality.

2. Work With the Split Lists

We want to work with the Lists in the same way that we can work with Streams. But we don't want to recreate every function that a Stream has for our two lists. That's where a nice design pattern comes in handy. I heard about it in this talk, and it's a very cool way to use lambdas. We basically create two new functions called `workOnPassedItems` and `workOnNotPassedItems` . They take a `Consumer<Stream<T>>` . Therefore, we can create a lambda and work inside it with a normal looking stream. This method will then be applied onto the List of passed and not-passed items.

```java
public class Splitter<T> {


    //...


    public Splitter<T> workWithPassed(Consumer<Stream<T>> func) {
        func.accept(passed.stream());
        return this;
    }


    public Splitter<T> workWithNotPassed(Consumer<Stream<T>> func) {
        func.accept(notPassed.stream());
        return this;
    }


}
```

We used the cascade design pattern to make the use of multiple methods nicer. You will see it in the example down below.

And that's basically our Splitter! Now, we can try out some examples on how to use it.

Example 1: Showing Numbers, but Squaring All Odd Numbers

In this very first example, we want to operate over a list of numbers. We just want to print each even number out, but we want to square every odd number before we print it.

So first off, we want to split the list into even and odd numbers. After that, we can work on the lists as already mentioned.

```java
public void workOnNumbers() {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);


    Splitter.splitBy(numbers, num -> num%2 == 0)
            .workWithPassed(passed ->
                    passed.forEach(even -> System.out.println("" + even + " -> " + even))

            .workWithNotPassed(notPassed ->
                    notPassed.map(odd -> odd * odd)
                            .forEach(odd -> System.out.println("" + Math.sqrt(odd) + " ->

        ));
}
```

Example 2: Sending All Winners a Confirmation and All Losers a Cancellation

Now, we have a List of `Candidates`. All of these people have a method called `hasWon()`, which gives back a boolean. We want to split the List into Winners and Losers. After that, we want to send all Winners a confirmation that they won and all losers a cancellation that they lost. So let's do this!

```
public void sendEmails(List<Candidates> candidates) {
    Splitter.splitBy(candidates, Candidates::hasWon)
            .workWithPassed(winners ->
                    winners.forEach(winner -> Email.send(winner.getEmail(), "You won!"))
    )
    .workWithNotPassed(losers ->
            losers.forEach(loser -> Email.send(loser.getEmail(), "You lost, sorry!"))
    );
}
```

On Using partitioningBy

While getting some feedback on this article, something that came up was the use of `Stream.collect(Collectors.partitioningBy(Predicate<T> test))`. And I totally agree that it could be useful in our scenario.

It partitions, the Stream depends on a test function. So for us, the map would look something like this: `{true: passed, false: notPassed}`. After that, we just take the two lists out of the map and go on.

So as `Philboyd_Studge` mentioned on Reddit, the new `splitBy` method could look something like this:

```
public static <T> Splitter<T> splitBy(Collection<T> items,Predicate<T> test) {

        Map<Boolean, List<T>> map = items.stream()
                .collect(Collectors.partitioningBy(test));

        return new Splitter<T>(map.get(true), map.get(false));
}
```

And I have to admit that this takes out a lot of noise. It just looks nicer than the `splitBy` method above. So thank you for that!

So What's the Splitter's Right to Exist?

The Splitter's purpose is to demonstrate how you can work with functions as objects. Its purpose is *not* to replace methods in the JDK.

It's a class for learning and playing around. If you want to tweak around, just do it. Please leave a comment on what you have learned or where you have optimized the class so that others can learn from it, too.

In addition, we learned about design patterns.

The patterns help make the syntax nicer. When you split your collection into two parts, it's nicer to use the cascade pattern in the Splitter than to handle a Map.

Conclusion

That's it for today!

We have learned how to create our first useful class with the help of Streams. We also optimized the runtime of our program with it. Therefore, we have trained our knowledge about design patterns like the factory method and the cascade pattern.

Lastly, we have tried out our splitter with some examples.

Next time, we will talk about RxJava. It's a framework that uses the observer pattern for asynchronous tasks and builds upon functional programming.

Are there any things that your splitter should do, too? Please let me know in the comments. I'd love to hear your feedback, too!

Topics: JAVA , JAVA 8 , FUNCTIONAL PROGRAMMING , STREAMS , TUTORIAL , SPLITTER