



DZone > Java Zone > Java 8 Concepts: FP, Streams, and Lambda Expressions

Java 8 Concepts: FP, Streams, and Lambda Expressions

by Radek Krawiec · May. 28, 17 · Java Zone · Tutorial

If you've been raised on object-oriented programming and have used Java (or C#, C++) all your life, the sudden turn to functional programming that comes with JDK 1.8 may seem confusing and intimidating.

Are there really any benefits of this new paradigm? In which situations can you leverage the new features?

The purpose of this article series is to give you a smooth introduction with hands-on examples into functional programming. Since Java 8 comes with the set of basic constructs to support FP, this tutorial will also contain a brief summary of new features added to Java with the 1.8 release such as:

- Lambda expressions
- Streams
- Optional object

To make the transition intuitive and painless, it'll be best to start with simple examples, discuss the basic usage, and get familiar with the new syntax.

Disclaimer

So you might be thinking — now this is another guy who's going to try to convert me to give up my old ways of doing code. That's not me. I myself am still exploring FP, and in each use case, I try to evaluate the benefits of using both OOP, FP, or a mix of the two. I'm by no means biased, and if a little, then in favor of OOP, which still seems more intuitive and familiar. My goal is to discover the possibilities that FP brings and provide you with all the knowledge to make your won choice.

A Basic Example

I'll start with a complete example. Take a look at it and try to figure out what it does, but don't worry if it seems complex and unintuitive. I had the same impression not so long ago, but it's easy to overcome with a proper dose of careful analysis.

```
1 Map < String, List < String >> phoneNumbers = new HashMap < String, List < String >> ();
2
3 phoneNumbers.put("John Lawson", Arrays.asList("3232312323", "8933555472"));
4 phoneNumbers.put("Mary Jane", Arrays.asList("12323344", "492648333"));
5 phoneNumbers.put("Mary Lou", Arrays.asList("77323344", "938448333"));
6
7 Map < String, List < String >> filteredNumbers = phoneNumbers.entrySet().stream()
8     .filter(x -> x.getKey().contains("Mary"))
9     .collect(Collectors.toMap(p -> p.getKey(), p -> p.getValue()));
0
1 filteredNumbers.forEach((key, value) -> {
2     System.out.println("Name: " + key + ": ");
3     value.forEach(System.out::println);
4 });
```

The first part of the code is pretty obvious — we're creating a map that maps a person's name to a list of their phone numbers, all stored as strings for simplicity (not that integer is a complicated concept, but...)

Then it gets a bit obscure. By the names of the methods (or rather, *functions*), you can tell that some filtering and iteration is done to the input map (`filter()` , `forEach()`), but predicting the actual output is rather difficult. Let alone seeing why you should use this instead of the classic approach.

Let me break down this code to you line by line and explain what's going on there.

Translating Code From Functional to Imperative

If you've used only OOP so far (as most of us have, and that's not a bad thing, don't get me wrong), you've used *statements* — commands that express some action to be carried out. These statements usually change the state of the program. E.g. if you're incrementing a variable...

```
1 Integer x = 0;
2 x++;
```

...then you're overwriting its previous state. That's simple. It's called **imperative programming**, because, much like the imperative mood in speech, it tells the program to perform some command.

By contrast, you have **functional programming**, which is a subset of the **declarative programming** paradigm. It also has commands, but the commands are treated more like mathematical functions. Their task is to take some input, perform some action, and return a result. The essential part here is that they *do not modify* the state of the program. The input variable(s) remain unchanged, and the returned result is always a new variable.

Here's an example from JavaScript:

```
1 function max(a, b) {  
2     return a > b ? a : b;  
3 }  
4  
5 var x = 10;  
6 var y = 5;  
7 var maximum = max(x, y);
```

After we call the `max` function, the value of variables `x` and `y` remains the same. The `max` function doesn't modify the input, nor does it depend on or modify any external (global) variables.

Code Analysis

So, let's get back to the example:

```
1 // create a map, filter it by key values  
2 Map < String, List < String >> filteredNumbers = phoneNumbers.entrySet().stream()  
3     .filter(x -> x.getKey().contains("Mary"))  
4     .collect(Collectors.toMap(p -> p.getKey(), p -> p.getValue()));
```

Let's discuss each operation here. For starters, I'll focus on what the code does, not how it does it, and what specific constructs of this expression mean from the theoretical standpoint.

We're taking the input map `phoneNumbers`, transforming it in some way, and also getting a map as an output. First, we call the `entrySet()` method to get a set of entries, each consisting of a key and a value.

Then, the `stream()` method is called. That created a **stream** (`java.util.stream.Stream`). This is a new concept introduced in Java 8, and for now, let's just stay with the definition offered by the Javadoc:

A stream is a sequence of elements supporting sequential and parallel aggregate

A stream is a sequence of elements supporting sequential and parallel aggregate operations.

The operations in question can be filtering, modification, and different types of transformations. Our next function is `filter()`, and its purpose is, of course, to filter the items in the stream with some criteria. The criteria are expressed as:

```
1 x -> x.getKey().contains("Mary")
```

This new syntactic construct is called a **lambda expression** and is also one of the concepts introduced to Java in version 1.8. In this specific case, the lambda expression is a **predicate** — a boolean function. Its goal is to evaluate the filtering criteria and tell whether each specific item in the collection (stream) should be kept or removed. The condition is quite simple:

```
1 // get the key of the map item and check if it contains the word "Mary"  
2 x.getKey().contains("Mary")
```

It takes the key of the currently processed map item and checks whether it contains a substring "Mary". If yes, the predicate (lambda expression) returns `true`, and the item is kept in the stream.

The next method is `collect()`. As you can see, the methods are chained, and they all belong to the `Stream` interface, so this suggests that each of them returns the stream to be accessible for subsequent methods in the invocation process.

The `collect()` method simply takes the elements of the stream (which are of type `java.util.Map.Entry`) and converts them back into a regular collection.

So far so good. There are some new concepts (stream, lambda expression, predicate) which we'll discuss in more details later, but all in all the code has become much more transparent. That being said, I expect the benefits of using this new approach may not be obvious yet, but bear with me.

Iterating a Collection

Now let's take a look at the second snippet:

```
1 filteredNumbers.forEach((key, value) -> {  
2     System.out.println("Name: " + key + ": ");  
3     value.stream().forEach(System.out::println);  
4 })
```

Instead of using good old-fashioned loops, let's try the new `forEach` introduced in Java 8. What it does should be comprehensible — it performs some action on each element of the underlying collection. The actual operation applied to the items is defined, again, by a lambda expression:

```
1 // expression takes two parameters
2 (key, value) -> {
3     // print person's name
4     System.out.println("Name: " + key + ": ");
5     // iterate over the person's phone numbers and print each of them
6     value.forEach(System.out::println);
7 }
```

The expression is a function that takes two parameters, `key` and `value`, (since it operates on map entries) and for each such pair performs two operations. First, it outputs the person's name (the `key` parameter):

```
1 System.out.println("Name: " + key + ": ");
```

Then it calls another `forEach` function to print all phone numbers in the `value` list.

The output of the whole operation to the console will be:

```
1 Name: Mary Lou:
2 77323344
3 938448333
4 Name: Mary Jane:
5 12323344
6 492648333
```

This was a lengthy analysis, but if you get the idea now, you'll never have to get back to the basics later.

What We Know so Far (and What We Don't)

The key takeaways from this article are:

- Java 8 supports functional programming with the help of streams and lambda expressions.
- Lambda expressions are much like anonymous functions.
- Streams are sequences of elements that support sequential or parallel operations on their items (in a very different way from classic collections).
- Functional programming is a paradigm that favors stateless operations and avoids modifications to the program state.
- Operations on streams are very concise — but are they more efficient or clear?

At this point, you probably have more questions and doubts and you're far from being convinced to the new way of thinking. What you might be asking yourself is:

- The new notation is rather obscure — I prefer the good old imperative style, which is more verbose.
- Are streams in any way more efficient?
- How is it possible to write a program that does not modify its state?

These are all fair questions and I owe you an explanation — which you'll get.

Further Articles

This article is intended as an introduction. It discusses the ideas of FP and Java 8 features in little detail. Subsequent parts of the series will deal with the comparison between traditional and OOP programming on specific examples, technical explanations of how streams and lambda expressions work, as well as some theory from imperative and functional programming paradigms and lambda calculus.

Adopting a microservices architecture? Don't forget the data layer! [Register for the we](#)

Presented by Redis Labs

Like This Article? Read More From DZone



Don't Fear the Lambda



Java Lambda Expressions: Functions as First-Class Citizens




A Little Lambda Tutorial



Free DZone Refcard Getting Started With Headless CMS

Topics: JAVA , LAMBDA EXPRESSIONS , FUNCTIONAL PROGRAMMING , STREAMS , TUTORIAL

Published at DZone with permission of Radek Krawiec . [See the original article here.](#) 
Opinions expressed by DZone contributors are their own.