A Beginners Guide to Functional Programming in Java

By Evan Hellman

For the Computer Science and Software Engineering Club at ISU

Overview

Functional Programming: What?

The term functional code is often vague if not a complete mystery to newer programmers. If you're now thinking: "Functional? All the code I write is Functional!", you're in the latter group (and that's okay)! Let's start with a few definitions:

1. Functional programming is a style of writing computer programs that treat computations as evaluating mathematical functions (Baeldung)
2. Functional programming is a programming paradigm where programs are constructed by applying and composing functions (Wikipedia)

I find neither of these to be sufficiently clear, so here's one of my own:

1. Functional programming is an approach to programming that focuses on chaining and nesting the execution of functions to control program flow rather than the more traditional imperative programming approach which utilizes modifications of local or global program state to control flow.

In short, we use a series of chained or nested functions to control the flow of our program rather than changing the state of the program. Another word for functional program control is "declarative" program control. In contrast, the traditional stateful approach is known as "imperative" program control.

There are two flavors of functional programming: purely functional and… not purely functional. This guide focuses on the latter (we are using Java, which only supports the latter). For the sake of discussion, purely functional programming is strictly declarative -- there is no program state at all (no local or global variables, no databases, essentially nothing mutable). There are languages dedicated to developing code in this manner, but that is out of the scope of this guide.

Functional Programming: Why?

By now, you're likely wondering why anyone would abandon the tried-and-true imperative programming approach for functional programming. The reason is simple: functional programming is often easier to read, understand, maintain, and optimize. Let's take a look at a quick example of imperative vs functional programming:

```
package com.test;

import java.util.List;

public class Adder {

   public Integer addImperative(List<Integer> integerList) {
       Integer sum = 0;

       for (Integer i : integerList) {
           sum += i;
       }

       return sum;
   }

   public Integer addFunctional(List<Integer> integerList) {
       return integerList.stream().mapToInt(Integer::intValue).sum();
   }

}
```

You can see that the chaining ".stream().mapToInt(...).sum()" and passing ".mapToInt(Integer::intValue)" of functions clearly in this example. We can also see that the imperative approach is not only more lengthy but also requires the creation of a placeholder variable (sum) and a loop. In contrast, the functional approach is a one-liner that is much cleaner and more readable. Even if you don't fully understand what is going on in the

addFunctional method, note how using functional programming can drastically improve code quality.

Functional Programming: How?

Now that you understand what functional programming is and how it can improve your code, we can get into the meat of this guide. We will learn how to properly implement functional code using Java 11 alongside your favorite IDE to develop a makeshift Point-Of-Sale system (POS). Before we start writing any code, I'll show you some important Java APIs that will allow us to make effective use of functional programming in Java.

The **Function** Interface

We first need to be able to replace some of the most common imperative programming approaches with functional approaches. One API java offers is java.util.**Function** interface and it's related interfaces/classes. The **Function** interface allows us to turn functions into "First-Class Citizens". In other words, we can perform any operation on a **Function** that we could on any other entity. For example, I can instantiate a **Function**, pass it as a parameter to other functions or use it as a field in class as seen below:

```
import java.util.Objects;
import java.util.function.Function;
import java.util.stream.Stream;

public class PriceSystem {

   // Note that getPrice is a member variable of PriceSystem
   private static final Function<CartItem, Double> getPrice = new Function<CartItem, Double>() {
      @Override
      public Double apply(CartItem cartItem) {
          return cartItem.blah();;
```

```
    }
};

// Note that the previously declared member function is passed to the .map() function
public static Function<Stream<CartItem>, Double> getCartTotal = (cartStream) ->
        cartStream
        .map(PriceSystem.getPrice)
        .filter(Objects::nonNull)
        .mapToDouble(Double::valueOf)
        .sum();

}
```

As you can see in these examples, the **Function** interface requires two type parameters, one for it's input type and one for it's output type. It contains the method **Function.apply()** which contains the code that the function represents. This method is what allows other **Function**s to call each other.

Lambda Expressions

Another important and helpful construct that Java offers is the lambda expression. This term was born from lambda calculus which is blissfully unnecessary to grasp in order to understand how to use lambda expressions in Java. Lambdas are commonly known as "anonymous functions" because they operate the same as the **Function** interface under the hood at runtime, they lack the **Function** type definition as well as it's specification of the input and output types. The syntax for a lambda expression in java is this:

```
(param1, param2, ... , paramN) -> {
    //Code
}
```

To put this into context, here is the previous example's **PriceSystem.getPrice** member variable rewritten as a lambda expression:

```
  // Note that getPrice is a member variable of PriceSystem, and now a lambda expression!
  private static final Function<CartItem, Double> getPrice = (cartItem) -> {
    return cartItem.blah();
  };
```

This approach is often much cleaner than fully declaring a **Function** type, which is often only necessary when implementing or extending **Function**. Another great feature of lambdas is that they do not need to be bound to a field. Because they are anonymous, you could just as easily skip the definition of the **PriceSystem.getPrice** member variable and instead pass a lambda function directly to the **.map()** function of **PriceSystem.getCartTotal** like so:

```
  // Note that we are now passing a locally declared lambda function to the .map() function
  public static Function<Stream<CartItem>, Double> getCartTotal = (cartStream) ->
          cartStream
          .map((cartItem) -> {
              return cartItem.blah();
          })
          .filter(Objects::nonNull)
          .mapToDouble(Double::valueOf)
          .sum();

}
```

Monads, Composition, and Currying

There are a few more key concepts before we can dive into writing functional Java code. Monads are another concept carried over from mathematics and serve as a slightly different form of encapsulation. Monads can be used to wrap up some data, such as an integer or list, perform transformations on the data, and then return it to its data form (although not necessarily data of the same type). One example of monads in Java are the **Optional** class which serves as a wrapper for data values that might be null and allows us to safely perform operations despite

the chance of a null value. Another example is the **Stream** interface which allows us to wrap an entire flow of data whether it be a list, queue, buffer, etc.

Next up is the concept of composition. Composition is exactly what it sounds like: the composing of multiple functions into one higher-order function. This allows us to chain together functions without making a syntactical mess. A great example of this is for use with string manipulation and testing; You could compose two separate **Functions** endsWithA and containsX (I will leave their implementation to your imagination) into a single **Function** containsXAndEndsWithA like so:

```
Function<String, Boolean> endsWithA = (input) -> {
    // code
};

Function<String, Boolean> containsX = (input) -> {
    // code
};

Function<String, Boolean> containsXAndEndsWithA = (input) -> {
    return containsX.apply(input) && endsWithA.apply(input)
}
```

In fact, there are even built-in functions to support this type of composition for some types, such as **Predicates** (which are what should be used here in place of **Function<String, Boolean>**).

Last but not least we have the concept of currying which allows us to take complex, multi-parameter functions and isolate/reduce them down into multiple less-complex functions. We do this by creating new functions that supply one of the parameters to the more complex function within them.

Your Turn

Now that you have a basic understanding of functional programming in Java, go experiment! A good starting resource is the Computer Science and Software Engineering Club at ISU's GitHub demo for this guide which is linked below with another of other great resources:

1. GitHub Repo: https://github.com/isu-cse-club/java-functional-programming-workshop/
2. Baeldung Tutorial: https://www.baeldung.com/java-functional-programming
3. Jenkov Tutorial: http://tutorials.jenkov.com/java-functional-programming/index.html
4. Baeldung Streams API Tutorial: https://www.baeldung.com/java-8-streams
5. Baeldung Functional Interfaces Tutorial: https://www.baeldung.com/java-8-functional-interfaces
6. GeeksForGeek Functional Interfaces Tutorial: https://www.geeksforgeeks.org/functional-interfaces-java/

Some fun practices using the GitHub repo:

1. Can you implement a SaleSystem that can apply sales to individual items? What about multiple items or even groups of items? Can you do it in a functional manner?
2. Experiment with redoing the syntax for some of the functional interfacing (there are a few ways to do it). Which way looks the cleanest to you?
3. Experiment with using some of the other functional interfaces Java offers, such as BiFunctions, Suppliers, Consumers, Predicates, and Operators.
4. Can you create your own functional interfaces? Try your hand at it using the @FunctionalInterface annotation. When would this be useful?