Functional Programming with Java 8 Functions

by Edwin Dalorzo · Oct. 20, 14 · Java Zone · Tutorial

Preliminaries

Ok, let's start with something simple. The following is a lambda expression (i.e. an anonymous function) that takes an argument `x` and increments it by one. In other words a function that receives, apparently an integer, and returns a new incremented integer:

```
x -> x + 1
```

And what is the type of this function in Java?

Well, the answer is that it depends. In Java the same lambda expression could be *bound* to variables of different types. For instance, the following two are valid declarations in Java:

```
Function<Integer,Integer> add1 = x -> x + 1;
Function<String,String> concat = x -> x + 1;
```

The first one increments an integer `x` by one, whereas the second one concatenates the integer 1 to any string `x`.

And how can we invoke these functions?

Well, now that they are bound to a reference we can treat them pretty much like we treat any object:

```
Integer two = add1.apply(1); //yields 2
String answer = concat1.apply("0 + 1 = "); //yields "0 + 1 = 1"
```

So, as you can see every function has a method `apply` that we use to invoke it and pass it an argument.

And what if I already have a method that does that, can I use it as a function?

Yes, since an alternative way to create functions is by using methods we had already defined and that are compatible with our function definition compatible with our function definition.

Suppose that we have the following class definition with methods as defined below:

```
public class Utils {
   publicstatic Integer add1(Integer x) { return x + 1; }
   publicstatic String concat1(String x) { return x + 1; }
}
```

As you can see the methods in this class are compatible with our original function definitions, so we could use them as "method references" to create the same functions we did before with lambda expressions.

```
Function<Integer,Integer> add1 = Utils::add1;
Function<String,String> concat1 = Utils::concat1;
```

These two are just the same thing as the ones we did before.

High Order Programming

The cool thing about functions is that they encapsulate behavior, and now we can take a piece of code, put it into a function and pass it around to other functions or methods for them to use it. This type of functions that operate on (or produce new) functions are typically called high order functions and the programming style based on exploiting this powerful feature is called, unsurprisingly, high order programming.

About Functions that Create Functions

Let's see a couple of examples of how we can do this using function objects. Let's consider the following example:

```
Function<Integer, Function<Integer,Integer>> makeAdder = x -> y -> x + y;
```

Above we have a function called `makeAdder` that takes an integer `x` and creates a new function that takes an integer `y` and when the latter is invoked, it adds `x` to `y`. We can tell this is a high order function because it produces a new function.

Now, we use this to create a new version of the `add1` function:

```
Function<Integer,Integer> add1 = makeAdder.apply(1);
Function<Integer,Integer> add2 = makeAdder.apply(2);
Function<Integer,Integer> add3 = makeAdder.apply(3);
```

With our high order function, however, we could also create `add2`, `add3`, ..., `addn`, right?

Can't we define this in a simpler way as we did before with the `Utils` class methods?

Yes, we can. Consider the following addition to the `Utils` class:

```
public class Utils {
    public static Function<Intger, Integer> adder(Integer x) {
       return y -> x + y;
    }
}
```

This signature is a bit simpler to read than that in our lambda expression, but as you can see it's pretty much the same thing. The function continues to receive the same number and type of arguments and continues to return the same type of result.

We can now use this simpler function factory to create our `makeAdded` and `add1` functions again:

```
Function<Integer, Function<Integer,Integer>> makeAdder = Utils::adder;
Function<Integer,Integer> add1 = makeAdder.apply(1);
```

And there we have it again, this is exactly the same thing as we had before.

About Functions that Receive Functions as Arguments

Let's suppose we had the following two functions defined:

```
Function<Integer,Integer> add1 = x -> x + 1;
Function<Integer,Integer> mul3 = x -> x * 3;
```

Now, naively, we could invoke this two functions together to increment and multiply a number by 3, right?. Like this:

```
Integer x = 10;
Integer res = mul3.apply(add1.apply(x)); //yields 33
```

But what if we created a function that did both things instead?

Consider the following pseudocode:

```
(f,g) -> x -> g( f(x) )
```

This would be a function that takes two other unary functions and creates yet another function that applies the original two in certain order. This is a classical example of what is called function composition.

In some languages there is even a binary operator to compose two functions in this way:

```
h = f o g
```

Where `o` would be an operator that would compose functions `f` and `g` pretty much as we did in pseudocode above and produce a new function `h`.

How can we do function composition in Java?

I can think of two ways to do this in Java, one more difficult than the other. Let's start with the more difficult strategy first, because that will let us appreciate the value of the simpler solution later on.

Function Composition Strategy 1

First, we must start by realizing that the function in pseudocode above is a binary function (i.e. a function that receives two arguments). But all our examples so far have dealt only with unary functions.

It would seem this is not important, but in Java it is, since functions of different arities have different target functional interfaces. In Java, a function that receives two arguments is called `BinaryOperator`.

For instance, using a `BinaryOperator` we could implement a sum operator:

```
BinaryOperator<Integer> sum = (a,b) -> a + b;
Integer res = sum.apply(1,2); // yields 3
```

Well, just as easily we could implement the `compose` operator, right? Only that in this case, instead of two simple integers, we receive two unary functions:

```
BinaryOperator<Function<Integer,Integer>> compose = (f,g) -> x -> g.apply(f.apply(x));
```

Now we can easily use this to fabricate a compund function that adds 1 and multiplies by 3, one after the other.

```
Function<Integer,Integer> h = compose.apply(add1,mul3);
Integer res = h.apply(10); //yields 33
```

And now we can beautifully, and in really simple way, combine two unary integer functions.

Function Composition Strategy 2

Now, function composition is something so common that it would have been a mistake if the Java Expert Group would have not considered it in their API design, and so, to make our lives simpler, all `Function` objects have a method called `compose` that allows us to very easily compose two functions together.

The following code produces the exact same result as above:

```
Function<Integer,Integer> h = mul3.compose(add1);
Integer res = h.apply(10);
```

Partial Function Application or Currying

In most functional programming languages it is possible to create partially applied functions. That is, if a function is receiving multiple arguments, we can partially invoke the function providing just a few arguments and receive a partially applied function out of it. This is typically called currying.

Although you have not noticed it, we have already covered that in this article, but now we are going to make it much more evident :-)

So, consider the following pseudocode

```
sum = x -> y -> x + y
```

Then we say that `sum` is a function that accepts one parameter `x` and fabricates another anonymous function that, in turn, accepts one parameter `y` that, when invoked, sums `x` and `y`.

In many functional programming languages a construct like this can be invoked as if this was just one simple function by doing:

```
sum 10 5 //yields 15
```

But the truth is that this is just syntactic sugar to do:

```
sum(10)(5) //yields 15
```

Since `sum` is a function that returns a function.

The beauty of this idiom is that now we could partially apply sum:

```
plus10 = sum 10
```

And now `plus10` is a partially applied function of sum, bound to a first argument 10, and yet expecting a second argument `y`.

Can you see now where we had already talked about a similar idea in this article?

```
plus10(5) //yields 15
```

Can we do this with Java?

The truth is that we have already done it above, we just probably did not notice. Unfortunately, in Java we do not have the syntactic sugar that some other language have, and therefore this is a bit more verbose:

```
Function<Integer, Function<Integer, Integer>> sum = x -> y -> x + y;
```

Well, you can see `sum` is declared in a "currified" way. And now we can partially apply it:

```
Function<Integer, Integer> plus10 = sum.apply(10);
Integer res = plus10.apply(5); //yields 15
```

Unary Functions

So, as mentioned above, Java uses different functional interfaces for different function arities. And so `Function<T,R>` is a functional interface for any unary function where the domain and the codomain of the function may be of different types.

For instance, we could define a function that receives a string value and parses it as an integer:

```
Function<String,Integer> atoi = s -> Integer.valueOf(s);
```

But most of our examples above are for integer functions whose argument and return value are of this same type. For those cases we could alternatively use the `UnaryOperator<T>` instead. This is just a `Function<T,T>` .

Thus, some our declarations above could be slightly simplified with this functional interface:

```
UnaryOperator<Integer> add1 = n -> n + 1;
UnaryOperator<String> concat1 = s -> s + 1;
Function<Integer, UnaryOperator<Integer>> sum = x -> y -> x + y;
UnaryOperator<Integer> sum10 = sum.apply(10);
```

I have already written another article that explains Why Java 8 has Interface Pollution like this in case you are interested in an explanation.

Value Types and Primitive Type Functions?

Evidently using a type like `Integer` incurs into the costs of boxing and unboxing when our functions have to deal with a primitive type like `int` .

As you know, Java does not support value types as type arguments in generic declarations, so to deal with this problem we can use alternative functional interfaces like `ToIntFunction` , `IntFunction` or `IntUnaryOperator` .

Or we can define our own primitive function.

```
interface IntFx {
    publicintapply(intvalue);
}
```

Then we can do:

```
IntFx add1 = n -> n + 1;

```

```
IntFunction<IntFx> sum = x -> y -> x + y;
IntFx sum10 = sum.apply(10);
sum10.apply(4); //yields 14
```

Similar functional interfaces can be found for types `double` and `long` as well. This topic is also covered in the alternative article mentioned above about interface pollution.

Further Reader

- High Order Programming
- High Order Functions
- Function Composition
- Currying
- Java 8 Interface Pollution

Topics: JAVA , ENTERPRISE-INTEGRATION , FUNCTIONAL PROGRAMMING , CLIENT-SIDE , JAVA8

DZone Research: The Problems With Java

by Tom Smith STAFF · Apr 30, 18 · Java Zone · Research

To gather insights on the current and future state of the Java ecosystem, we talked to executives from 14 companies. We began by asking, "What are the most common problems with the Java ecosystem today?" Here's what the respondents told us:

Verbosity

- **It's a bit verbose.** Other languages are simpler. A lot of libraries – hard to choose which one. The JDK is complete but there are a lot of dependencies.
- The most common complaint about Java is that it **tends to be long-winded**, meaning you have to type more characters to get what you want done than some other, more modern languages. That's something newer languages have been trying to tackle, with the goal of trying to get the quality, features, and power of Java without needing to do quite as much typing. The ability to write code faster is one of the biggest reasons that some people prefer languages that are actually worse languages when compared at side-by-side with Java.
- Java developers have no problems with Java. **Those on the outside will complain that it's too verbose.** The Java 9 upgrade is not seamless, you must add dependencies to adopt. It will not be supported long term. Java 10 will be out this month and may result in a lot of people skipping the Java 9 upgrade.
- **Java itself is verbose** but there are alternatives like Kotlin, Scala, and project Lombok.

Nothing

- **I think the ecosystem is pretty healthy right now,** relatively speaking. A benevolent dictator and an engaged community is probably the best mix. Feels like good energy stemming from more frequent releases.
- If you had asked me this same question one year ago, I would have suggested that the uncertainty of Oracle's intentions around Java EE was a substantial problem. As previously mentioned, Java dominates the enterprise. While we did have folks like IBM, Tomitribe, Red Hat, Payara and others working to create MicroProfile, it was unclear what would

Red Hat, Payara and others working to create MicroProfile, it was unclear what would happen if Oracle remained neutral. **Now, we have Jakarta EE at Eclipse and we are open for innovation yet again.**

Other

- Software engineering quality is degrading with less care and pride of workmanship. No real engineering processes. **No one is nailing down well-known methodologies.**
- **It can be hard to consume all that's contained in a release every six months.**
- 1) The Maven repository was revolutionary in its time as a central repository for Java libraries, but it is starting to show its age, especially compared with the npm library. Compare the simplicity of searching for quality libraries in the npm libraries, which includes a great search tool, a way to grade the quality of a library, and a culture of readmes that help you figure out what the library is all about at a glance. 2) **Java libraries tend to be too large** and try to include as much stuff as possible in a library, with, in the best case, huge documentation that is very difficult to comprehend due to the number of things to understand, and to the worst case where all you get is a huge Javadoc. Compare and contrast that with the JS library ecosystem that tends to favor small libraries, a "pick and choose" methodology, which enables easy understanding of a library. 3) The slowness of the evolution of the language is a big advantage, but because it was *too* slow, it has driven many developers, especially thought leaders, away from the language.
- Governance of the JVM. Eclipse is a good steward of open source. **The rate of innovation into the JVM is stalling.** Java 9 represents the last attempt to push technology into the platform. I'm not sure where the product roadmap goes after that. The language goes in circles. Every generation needs their own language. It's nicer to be part of a vibrant ecosystem. Java's guarantee of compatibility gives you a wide variety of languages to choose from.
- **Participation and engagement,** growing real-world developer interest. We want to hear from every user, not just architects. We encourage developers to contribute as a group. The Brazilian User Groups have adopted the JSR concept to forward feedback as a group. To continue the momentum, we have an open JDK adoption group.
- **The freedom can sometimes become a curse.** In languages such as .Net where you are more "within boundaries" it is easier to not make the wrong decisions. The different permutations of what dependencies you can use together can make your system become an unproven snowflake. There's also the notion of Java being an "old and grumpy" language. Though I don't agree with this notion the rumor can be a bit hurtful. Hopefully, this will change with the new release cadence.
- **It lags behind because it's used by large enterprises.** With slowness comes stability. It lacks some of the niceties of other languages; however, it provides quick wins with fast coding.
- **The main problem will be release fatigue.** The increase in cadence means developers have to keep up with new versions of Java. So much going on in open source community it's hard to get a handle on new APIs, components, projects. Every time you try to learn something new you're placing a bet with your mindshare on whether or not it will be relevant in a couple of years.

Here's who we spoke to:

- Gil Tayar, Senior Architect and Evangelist, Applitools
- Frans van Buul, Commercial Developer, Evangelist, AxonIQ
- Carlos Sanchez, Software Engineer, CloudB

- Carlos Sanches, Software Engineer, CloudBees
- Jeff Williams, Co-founder and CTO, Contrast Security
- Doug Pearson, CTO, FlowPlay
- John Duimovich, Distinguished Engineer and Java CTO, IBM
- Brian Pontarelli, CEO, Inversoft
- Wayne Citrin, CTO, JNBridge
- Ray Augé, Sr. Software Architect, Liferay
- Matt Raible, Java Champion and Developer Advocate, Okta
- Heather VanCura, Chair of the Java Community Process Program, Oracle
- Burr Sutter, Director Developer Experience, Red Hat
- Ola Petersson, Software Consultant, Squeed
- Roman Shoposhnik, Co-founder, V.P. Product and Strategy, Zededa

Like This Article? Read More From DZone

CodeTalk: Red Hat CTO on Jakarta EE, Cloud Native, Kubernetes, and Microservices [Podcast]

Java EE Has a New Name....

CodeTalk: Jakarta EE's Cloud Native Opportunities [Podcast]

Free DZone Refcard
Getting Started With Kotlin

Topics: JAVA, JAVA EE, JAKARTA EE, VERBOSITY

Opinions expressed by DZone contributors are their own.