ABOUT YATIN

The author is graduated in Electronics & Telecommunication. During his studies, he has been involved with a significant number of projects ranging from programming and software engineering to telecommunications analysis. He works as a technical lead in the information technology sector where he is primarily involved with projects based on Java/J2EE technologies platform and novel UI technologies.

Java 9 Functional Programming Tutorial

Posted by: Yatin in Core Java July 7th, 2017 0 Views

Hello, in this tutorial we will learn about the introduction to Functional Programming in Java 9. The idea behind this approach was to combine Haskell programming language in Java.

Table Of Contents

1. Java 9 Functional Programming

In computer science, **functional programming** is a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical *functions* and avoids changing state and *mutable data*. It is a *declarative* programming paradigm, which means programming is done with *expressions*.

Developers may have heard about the functional programming and how great it is to reduce the lines of code and enhance the readability of code. But what does it really mean to program functions and what are the main differences to Object Oriented Programming (OOP)?

1.1 All variables are final

Let's look at below function to welcome some users. First off, it is written in object oriented programming.

Test.java

```
public String welcome(List<String> names) {
    String greeting = "Welcome ";
    for(String name : names) {
        greeting += name + " ";
    }
    greeting += "!";
    return greeting;
}
```

This is a perfectly valid function to create such a welcome String in Java. But if you are using functional programming, this won't work. You change the state of greeting, which is not allowed in functional programming. So if you try to make the welcome final, you would get an error. Every time you use

```
+=
```

with that String, you change its state.

What developers basically do in functional programming is the concatenation of all names in one line into one String.

Test.java

```
public String welcome(List<String> names) {
    String greeting = "Welcome ";
    for(String name : names) {
        greeting += name + " ";
    }
    greeting += "!";
    return greeting;
}
```

If you think that this looks nasty, you're right! But there is a functional programming function to make this nicer. I will give you the right functional programming function here:

Test.java

```
public String greet(List<String> names) {
    String greeting = names.stream().map(name -> name + " ").reduce("Welcome ",(acc, name) -> acc + name);
    return greeting + "!";
}
```

1.2 Don't use global variables (and forget about side effects)

I've chosen the example of a global time object. You write a static function, which returns the current time as a String. An object oriented function could look like this:

Utils.java

```
public class Utils {
    private static Time time;
    public static String currTime() {
        return time.getTime().toString();
    }
}
```

If developers use

```
currTime
```

twice, the result will be different, because the time will be different. Although we had the same input,

```
currTime
```

had two different results!

This can't happen in functional programming. Every method only depends on its parameters and on nothing else! So if we want to do something like this, the

```
Time
```

object, which should be a set time, has to be a parameter of

```
currTime
```

:

Utils.java

```
public class Utils {
    public static String currTime(FixedTime time) {
        return fixedTime.now().toString();
    }
}
```

This might seem odd in the object oriented world, but it has some benefits.

On one hand, it is much easier to read the code. If you know that a method only relies on its parameter, you don't have to look for global variables that do the magic in your method. On the other hand, testing is much easier too! When you want to test the functional programming

```
currTime
```

method, you can mock the

```
Time
```

object. In the object-oriented version, it's really difficult to mock the static

```
Time
```

object.

1.3 Use functions as parameters

In functional programming, functions can be arguments of another function! How cool is that? Just think of a function which adds 1 to every number of a List. How would you do that object oriented? Here's a snippet:

Test.java

```
public List<Integer> addOne(List<Integer> numbers) {
    List<Integer> plusOne = new LinkedList<>();
    for(Integer number : numbers) {
        plusOne.add(number + 1);
    }
    return plusOne;
}
```

Now you have to handle two lists. This can be very confusing and leads to errors. There is also the chance to change the state of numbers. This could lead to problems in later parts of the program.

In functional programming, you can map a function to every element of a List. In this example, this means that you want to *map*

```
number+1
```

to every item in the list and store this in a new List. The functional programming method would look like this:

Test.java

```
public List<Integer> addOne(List<Integer> numbers) {
    return numbers.stream().map(number -> number + 1).collect(Collectors.toList());
}
```

This reduces the number of variables and therefore the places where you can make errors. Here, you create a new list and leave numbers like it is.

1.4 Lambda Expressions

Anonymous function (also function literal or lambda abstraction) is a function definition that is not bound to an identifier. Lambdas are often:

1. **Passed as arguments** to higher-order functions
2. Used to construct the result of a higher- order function that needs to return a function
3. Passed as an argument (common usage)

To make our functional programming code useful, developers have to introduce a second way to store a function in an object. And this is done by using anonymous functions, or so-called *Lambdas*.

1.4.1 How to work with Lambdas?

To work with Lambdas in Java 8, we have to look at a new syntax to handle them properly.

Example: Adding Two Integers

In good old Java 7, you can write a method to add two Integers like this:

Test.java

```
public Integer add(Integer a, Integer b) {
    return a + b;
}
```

And this is a Java 8 Lambda which does exactly the same:

Test.java

```
BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
```

That's pretty straightforward, isn't it?

```
BiFunction
```

is another Interface in

```
java.util
```

to represent a function with two arguments and one return object. In the brackets of the Lambda, developers define the arguments. Developers don't have to give them a type, they just have to say how many there are and how each should be called. This is equivalent to

```
(Integer a, Integer b)
```

in the Java 7 method. Next off, we have the "->" arrow. It is equivalent to the curly brackets and separates the function's head from its body. And after the arrow, developers can work with the arguments. If we have just one calculation to make, a return isn't necessary because it returns the result. You can also make the function's body bigger by using curly brackets. Let's take a look at the same example:

Test.java

```
BiFunction<Integer, Integer, Integer> add = (a,b) -> {
    Integer result = a + b;
```

```
    return result;
};
```

But most of the times, developers just need one line and therefore no brackets and no

```
return
```

keyword.

1.5 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 developers can continue to work with it.

Streams are also **immutable**. So every time developers manipulate it, they create a new Stream. Another nice thing about them is that they respect the properties of functional programming. If developers convert a Data Structure into a Stream and work on it, the original data structure won't be changed. So no side effects here!

1.5.1 How to Convert Data Structures into Streams

- Convert Multiple Objects into a Stream

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

```
Stream.of()
```

Test.java

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

- Converting Collections (Lists, Sets, …) 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).

Test.java

```
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();
}
```

1.5.2 Working with Streams

Streams are the way to work with data structures functional. And now we will see some of the most common methods to use:

- map

This works pretty straight forward. 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 a map to use this function over multiple numbers without writing a new function for lists.

Test.java

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

- flatMap

Like with *Optional*, we use flatMap to going e.g. from a Stream<List<Integer>> to Stream<Integer>. Here, we want to concatenate multiple Lists into one big List.

Test.java

```
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
}
```

1.5.3 Common Stream methods

- forEach

The

```
forEach
```

method is like the

```
ifPresent
```

method from Optional, so you use it when you have side effects. As already shown, you use it to e.g. print all objects in a stream.

```
forEach
```

is one of the few Stream methods that don'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 won't have. So think twice if you could replace it with another method without side effects.

Test.java

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

- filter

The 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 or 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 the type Predicate<T>. But this is just a short form for every function that takes one parameter and returns a boolean.

Test.java

```
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 which 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 e.g. mapping functions.

Test.java

```
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, developers want to transform your stream back into another data structure. And that is what you use Collect for. And most of the times, developers convert it into a List or a Set.

Test.java

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

But developers can use collect for much more. For example, they can join Strings. Therefore, developers or programmers don't have the nasty delimiter in the end of the string.

Test.java

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

1.5.4 Parallelism

Streams can also be executed 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.

1.5.5 Stream vs. Collection

Let's take a look and understand the basic differences between Stream and collection,

1. Like a collection, a stream provides an interface to a sequenced set of values of a specific element type
2. Because collections are data structures, they're mostly about storing and accessing elements with specific time/space complexities
3. Streams are about expressing computations such as filter, sorted, and map
4. Collections are about data; streams are about computations

1.6 Optionals

In Java 8, java.util.Optional<T> was introduced to handle objects which might not exist better. It is a container object that can hold another object. The Generic T is the type of the object you want to contain.

Test.java

```
Integer i = 5;
Optional<Integer&gr; optinal = Optional.of(i);
```

The Optional class doesn't have any public constructor. To create an optional, you have to use

```
Optional.of(object)
```

or

```
Optional.ofNullable(object)
```

. Developers use the first one if the object is never null. The second one is used for nullable objects.

1.6.1 How do Optionals work?

Options have two states. They either hold an object or they hold null. If they hold an object, Optionals are called present. If they hold null, they are called *empty*. If they are not empty, developers can get the object in the optional by using

```
Optional.get()
```

. But be careful, because a

```
get()
```

on an empty optional will cause a

```
NoSuchElementException
```

. Developers can check if an optional is present by calling the method

```
Optional.isPresent()
```

.

Example: Playing with Optional

Test.java

```
public void playingWithOptionals() {
    String s = "Hello World!";
    String nullString = null;

    Optional<String> optionalS1 = Optional.of(s);                     // Will work
    Optional<String> optionalS2 = Optional.ofNullable(s);             // Will work too
    Optional<String> optionalNull1 = Optional.of(nullString);         // -> NullPointerException
    Optional<String> optionalNull2 = Optional.ofNullable(nullString); // Will work

    System.out.println(optionalS1.get());    // prints "Hello World!"
    System.out.println(optionalNull2.get()); // -> NoSuchElementException
    if(!optionalNull2.isPresent()) {
        System.out.println("Is empty");      // Will be printed
    }
}
```

1.6.2 Common problems when using Optionals

- Working with Optional and null

Test.java

```
public void workWithFirstStringInDB() {
    DBConnection dB = new DBConnection();
    Optional<String> first = dB.getFirstString();

    if(first != null) {
        String value = first.get();
        //...
    }
}
```

This is just the wrong use of an Optional! If you get an Optional (In the example you get one from the DB), developers don't have to look whether the object is null or not! If there's no string in the DB, it will return

```
Optional.empty()
```

, not

```
null
```

! If you got an empty Optional from the DB, there would also be a

```
NoSuchElementException
```

in this example.

1.6.3 When Should you use Nullable Objects and when Optionals?

Developers can find a lot of books, talks, and discussions about the question: Should you use null or Optional in some particular case. And both have their right to be used. In the linked talk, developers will find a nice rule which they can apply in most of the cases. Use Optionals when *"there is a clear need to represent 'no result' or where null is likely to cause errors"*

So you shouldn't use Optionals like this:

Test.java

```
public String defaultIfOptional(String string) {
    return Optional.ofNullable(string).orElse("default");
}
```

Because a null check is much easier to read.

Test.java

```
public String defaultIfOptional(String string) {
    return (string != null) ? string : "default";
}
```

Developers should use Optionals just as a return value from a function. It's not a good idea to create new ones to make a cool method chain like in the example above. Most of the times, null is enough.

2. Conclusion

The main goal of this article is to discuss the functional programming in JDK 9. I hope developers can see the benefits of functional programming and can truly appreciate them once the official JDK is released to a larger audience.

Final variables are a big help in terms of multi-threading, the lack of global variables improves the testability and functions as parameters improve the code quality. And don't worry, in the beginning, you can mix OOP and functional programming in your code.

3. Download the Eclipse Project

Download
You can download the full source code of this example here: **Java9 Functional Programming**

Tagged with: CORE JAVA FUNCTIONAL PROGRAMMING

LIKE THIS ARTICLE? READ MORE FROM JAVA CODE GEEKS

Leave a Reply

Start the discussion...

Subscribe