Functional Programming in Java

What Is Functional Programming?

Basically, functional programming is **a style of writing computer programs that treat computations as evaluating mathematical functions.**

In mathematics, a function is an expression that relates an input set to an output set.

Importantly, the output of a function depends only on its input. More interestingly, we can compose two or more functions together to get a new function.

1. Lambda Calculus

To understand why these definitions and properties of mathematical functions are important in programming, we'll have to go back in time a bit.

In the 1930s, mathematician Alonzo Church developed **a formal system to express computations based on function abstraction.** This universal model of computation came to be known as lambda calculus.

Lambda calculus had a tremendous impact on developing the theory of programming languages, particularly functional programming languages. Typically, functional programming languages implement lambda calculus.

Since lambda calculus focuses on function composition, functional programming languages provide expressive ways to compose software in function composition.

2. Categorization of Programming Paradigms

Of course, functional programming is not the only programming style in practice. Broadly speaking, programming styles can be categorized into imperative and declarative programming paradigms.

The **imperative approach defines a program as a sequence of statements that change the program's state** until it reaches the final state.

Procedural programming is a type of imperative programming where we construct programs using procedures or subroutines. One of the popular programming paradigms known as Object-Oriented Programming (OOP) extends procedural programming concepts.

In contrast, the **declarative approach expresses the logic of a computation without describing its control flow** in terms of a sequence of statements.

Simply put, the declarative approach's focus is to define what the program has to achieve rather than how it should achieve it. Functional programming is a subset of the declarative programming languages.

These categories have further subcategories, and the taxonomy gets quite complex, but we won't get into that for this tutorial.

2.3. Categorization of Programming Languages

Now we'll try to understand how programming languages are divided based on their support for functional programming for our purposes.

Pure functional languages, such as Haskell, only allow pure functional programs.

Other languages allow both **functional and procedural programs** and are considered impure functional languages. Many languages fall into this category, including Scala, Kotlin and Java.

It's important to understand that most of the popular programming languages today are general-purpose languages, so they tend to support multiple programming paradigms.

3. Fundamental Principles and Concepts

This section will cover some of the basic principles of functional programming and how to adopt them in Java.

Please note that many features we'll be using haven't always been part of Java, and it's **advisable to be on Java 8 or later to exercise functional programming effectively.**

3.1. First-Class and Higher-Order Functions

A programming language is said to have first-class functions if it treats functions as first-class citizens.

This means that **functions are allowed to support all operations typically available to other entities.** These include assigning functions to variables, passing them as arguments to other functions and returning them as values from other functions.

This property makes it possible to define higher-order functions in functional programming. **Higher-order functions are capable of receiving functions as arguments and returning a function as a result.** This further enables several techniques in functional programming such as function composition and currying.

Traditionally, it was only possible to pass functions in Java using constructs such as functional interfaces or anonymous inner classes. Functional interfaces have exactly one abstract method and are also known as Single Abstract Method (SAM) interfaces.

Let's say we have to provide a custom comparator to Collections.sort method:

```
Collections.sort(numbers, new Comparator<Integer>() {
    @Override
    public int compare(Integer n1, Integer n2) {
        return n1.compareTo(n2);
    }
});
```
As we can see, this is a tedious and verbose technique — certainly not something that encourages developers to adopt functional programming.

Fortunately, Java 8 brought many **new features to ease the process, such as lambda expressions, method references and predefined functional interfaces.**

Let's see how a lambda expression can help us with the same task:

```
Collections.sort(numbers, (n1, n2) -> n1.compareTo(n2));
```
Copy

This is definitely more concise and understandable.

However, please note that while this may give us the impression of using functions as first-class citizens in Java, that's not the case.

Behind the syntactic sugar of lambda expressions, Java still wraps these into functional interfaces. So, **Java treats a lambda expression as an Object**, which is the true first-class citizen in Java.

3.2. Pure Functions

The definition of pure function emphasizes that **a pure function should return a value based only on the arguments and should have no side effects.**

This can sound quite contrary to all the best practices in Java.

As an object-oriented language, Java recommends encapsulation as a core programming practice. It encourages hiding an object's internal state and exposing only necessary methods to access and modify it. So, these methods aren't strictly pure functions.

Of course, encapsulation and other object-oriented principles are only recommendations and not binding in Java.

In fact, developers have recently started to realize the value of defining immutable states and methods without side effects.

Let's say we want to find the sum of all the numbers we've just sorted:

```
Integer sum(List<Integer> numbers) {
    return numbers.stream().collect(Collectors.summingInt(Integer::intValue));
}
```

This method depends only on the arguments it receives, so it's deterministic. Moreover, it doesn't produce any side effects.

Side effects can be anything apart from the intended behavior of the method. For instance, **side effects can be as simple as updating a local or global state** or saving to a database before returning a value. (Purists also treat logging as a side effect.)

So, let's look at how we deal with legitimate side effects. For instance, we may need to save the result in a database for genuine reasons. There are techniques in functional programming to handle side effects while retaining pure functions.

We'll discuss some of them in later sections.

3.3. Immutability

Immutability is one of the core principles of functional programming, and it **refers to the property that an entity can't be modified after being instantiated.**

In a functional programming language, this is supported by design at the language level. But in Java we have to make our own decision to create immutable data structures.

Please note that **Java itself provides several built-in immutable types**, for instance, String. This is primarily for security reasons because we heavily use String in class loading and as keys in hash-based data structures. There are also several other built-in immutable types such as primitive wrappers and math types.

But what about the data structures we create in Java? Of course, they are not immutable by default, and we have to make a few changes to achieve immutability.

The **use of the final keyword** is one of them, but it doesn't stop there:

```
public class ImmutableData {
    private final String someData;
    private final AnotherImmutableData anotherImmutableData;
    public ImmutableData(final String someData, final AnotherImmutableData
anotherImmutableData) {
        this.someData = someData;
        this.anotherImmutableData = anotherImmutableData;
    }
    public String getSomeData() {
        return someData;
    }
    public AnotherImmutableData getAnotherImmutableData() {
        return anotherImmutableData;
    }
}
public class AnotherImmutableData {
    private final Integer someOtherData;
    public AnotherImmutableData(final Integer someData) {
        this.someOtherData = someData;
    }
    public Integer getSomeOtherData() {
        return someOtherData;
    }
}
```

Note that we have to diligently observe a few rules:

- All fields of an immutable data structure must be immutable.
- This must apply to all the nested types and collections (including what they contain) as well.
- There should be one or more constructors for initialization as needed.
- There should only be accessor methods, possibly with no side effects.

It's **not easy to get it completely right every time**, especially when the data structures start to get complex.

However, several external libraries can make working with immutable data in Java easier. For instance, Immutables and Project Lombok provide ready-to-use frameworks for defining immutable data structures in Java.

3.4. Referential Transparency

Referential transparency is perhaps one of the more difficult principles of functional programming to understand, but the concept is pretty simple.

We **call an expression referentially transparent if replacing it with its corresponding value has no impact on the program's behavior.**

This enables some powerful techniques in functional programming such as higher-order functions and lazy evaluation.

To understand this better, let's take an example:

AD

```
public class SimpleData {
    private Logger logger = Logger.getGlobal();
    private String data;
    public String getData() {
        logger.log(Level.INFO, "Get data called for SimpleData");
        return data;
    }
    public SimpleData setData(String data) {
        logger.log(Level.INFO, "Set data called for SimpleData");
        this.data = data;
        return this;
    }
}
```

This is a typical POJO class in Java, but we're interested in finding if this provides referential transparency.

Let's observe the following statements:

```
String data = new SimpleData().setData("Baeldung").getData();
logger.log(Level.INFO, new SimpleData().setData("Baeldung").getData());
logger.log(Level.INFO, data);
logger.log(Level.INFO, "Baeldung");
```

The three calls to logger are semantically equivalent but not referentially transparent.

The first call is not referentially transparent since it produces a side effect. If we replace this call with its value as in the third call, we'll miss the logs.

The second call is also not referentially transparent since SimpleData is mutable. A call to data.setData anywhere in the program would make it difficult for it to be replaced with its value.

So, **for referential transparency, we need our functions to be pure and immutable.** These are the two preconditions we discussed earlier.

As an interesting outcome of referential transparency, we produce context-free code. In other words, we can run them in any order and context, which leads to different optimization possibilities.