Theory: Lambda expressions

28 minutes 0 / 5 problems solved Skip this topic Start practicing

402 users solved this topic. Latest completion was about 1 hour ago.

As you already know, Java is primarily an object-oriented programming language. It supports classes, methods, fields, and other concepts from this paradigm. Here, methods are the main way to represent the behavior of objects, classes, and whole programs. You can write absolutely any code inside their bodies and then invoke this code from other parts of your program using the names of the methods. This approach allows developers to create very structured and well-readable programs, but sometimes it's not enough and we should use other ways to represent behavior rather than methods.

§1. Functional programming in Java

This topic begins to explain another programming paradigm called **functional programming (FP)** that uses **functions** as the primary concept to provide an alternative way to solve many programming challenges. Like methods, functions are used to decompose code into small pieces. Sometimes these concepts are even interchangeable. However, unlike methods, functions can also behave like regular Java objects (e.g., be passed/returned to/from a method).

Of course, it is impossible to explain the whole paradigm at once, so there will be a lot of engaging topics. The first concept we will learn is **lambda expressions** which is the closest one to the standard Java methods. Let's take a look at what they are and why do we use them.

§2. Lambda expressions

By **lambda expression** (or just "lambda"), we mean a **function** that isn't bound to its name (an anonymous function) but can be assigned to a variable.

The most general form of a lambda expression looks like this: `(parameters) -> { body };`. Here, the part before `->` is the list of parameters (like in methods), and the part after that is the body that can return a value. The brackets `{ }` are required only for multi-line lambda expressions.

Sometimes, lambdas don't have parameters or return values or even both. Even if a lambda doesn't have a value to return, it has a body that does some useful actions (e.g. prints or saves something). You will encounter practical examples with such lambdas in the next topics.

Another important thing — like a regular Java object, a lambda expression always has a special type. There are a lot of types presented in the Java Standard Library. In this topic, we will only mention two of them: `Function` and `BiFunction`. Both of the classes are located in the `java.util.function` package among others.

You don't need to find and remember all possible types of lambda expressions at once. You will gradually do this as you learn.

Let's consider a single-line lambda expression that just multiplies its two parameters.

```java
BiFunction<Integer, Integer, Integer> mult = (x, y) -> x * y;
```

The expression has the type `BiFunction<Integer, Integer, Integer>` which means, that it takes two `Integer` values and returns another `Integer` value.

There are a lot of ways to write lambda expressions. Let's consider more examples.

Current topic:

Lambda expressions ···

Topic depends on:

✓ Functional decomposition ··· Stage 3

✓ Final variables ···

✗ Generic programming ···

Topic is required for:

Method references ···

Functional interfaces ···

Table of contents:

```java
// if it has only one argument "()" are optional
Function<Integer, Integer> adder1 = x -> x + 1;

// without type inference
Function<Integer, Integer> mult2 = (Integer x) -> x * 2;

// with multiple statements
Function<Integer, Integer> adder5 = (x) -> {
    x += 2;

    x += 3;

    return x;

};
```

Although Java provides a lot of ways to write lambda expressions, you always need to choose the shortest and readable way to do this.

§3. Invoking lambda expressions

Once lambda expression is created, it can be used in other places of your program like a regular Java object. You can invoke the body of an expression using special methods like `apply` as many times as you need. The name of the method depends on the type of lambda expression.

```java
int result2x5 = mult.apply(2, 5); // 10
int result3x1 = mult.apply(3, 1); // 3
```

So, we can invoke a lambda expression like a regular method passing arguments and obtaining results!

§4. Passing lambda expressions to methods

One of the most popular cases is to pass a lambda expression to a method and then call it there.

Look at the method below. It takes an object of the standard generic `Function` type.

```java
private static void printResultOfLambda(Function<String, Integer> function) {
    System.out.println(function.apply("HAPPY NEW YEAR 3000!"));
}
```

This function can take a `String` argument and return an `Integer` result.

To test the method, let's create an object and pass it into the method:

```java
// it returns the length of a string
Function<String, Integer> f = s -> s.length();
printResultOfLambda(f); // it prints 20
```

You can also pass a lambda expression to the method directly without an intermediate reference:

```java
// passing without a reference
printResultOfLambda(s -> s.length()); // the result is the same: 20
```

As you can see, we can pass our function presented by an object to a method as its argument, if the method takes an object of a suitable type. Then, inside the method, the given function will be invoked.

But why do we need it? First, let's look at another example, which uses a more complex lambda expression and calculates the number of digits on the string.

```
// It print the number of digits: 4
printResultOfLambda(s -> {
    int count = 0;
    for (char c : s.toCharArray()) {
        if (Character.isDigit(c)) {
            count++;
        }
    }
    return count;

});
```

What is important here? We pass to the `printResultOfLambda` not data, but some piece of code as data. So, we can parameterize the same method with a different behavior at runtime. This is what typical uses of lambda expressions look like. Many standard methods can accept lambda expressions. This will be discussed in more detail in the following topics.

Let's introduce an important term according to the examples. In functional programming theory, a function that accepts or returns another function is called a **higher-order function**. In terms of Java, we're talking about methods or functions which take / return `Function<T, R>`, `BiFunction<T, R>` or other types we will consider soon.

§5. Closures

Another important trick with lambda expressions is the possibility to capture values from a context where the lambda is defined and use the values within the body. This technique is called **closure**.

Capturing is possible only if a context variable has the `final` keyword or it's **effectively final**, i.e. the variable isn't changed in further code. Otherwise, an error happens.

Let's see an example.

```
final String hello = "Hello, ";
Function<String, String> helloFunction = (name) -> hello + name;

System.out.println(helloFunction.apply("John"));
System.out.println(helloFunction.apply("Anastasia"));
```

The lambda expression captured the final variable `hello`.

The result of this code.

```
Hello, John
Hello, Anastasia
```

Let's consider the example with an effectively final variable.

```
int constant = 100;
Function<Integer, Integer> adder100 = x -> x + constant;

System.out.println(adder100.apply(200));
System.out.println(adder100.apply(300));
```

The variable `constant` is effectively final and being captured by the lambda expression.

§6. Conclusion

In this topic, you started learning a new programming paradigm. You've seen the syntax of lambda expressions and how to pass them to methods to vary their behavior at runtime. You've also learned how lambda expressions can use context variables within the body. While you are still at the beginning of the journey in the world of functional programming, you will see soon how many new possibilities this paradigm brings to solve complex practical problems.

Report a typo

46 users liked this theory. **1** didn't like it. **What about you?**

Start practicing

Comments (0) Hints (0) Useful links (0) Show discussion