Writing Your First Lambda Expression

→

Using Lambdas Expressions in Your Application

Writing Your First Lambda Expression

In 2014, Java SE 8 saw the introduction of the concept of lambda expressions. If you remember the days before Java SE 8 was released, then you probably remember the anonymous classes concept. And maybe you have heard that lambda expressions are another, simpler way of writing instances of anonymous classes, in some precise cases.

If you do not remember those days, then you may have heard or read about anonymous classes, and are probably afraid of this obscure syntax.

Well, the good news is: you do not need to go through anonymous classes to understand how to write a lambda expression. Moreover, in many cases, thanks to the addition of lambdas to the Java language, you do not need anonymous classes anymore.

Writing a lambda expression breaks down to understanding three steps:

- identifying the type of the lambda expression you want to write
- finding the right method to implement
- implementing this method.

This is really all there is to it. Let us see these three steps in detail.

Identifying the Type of a Lambda Expression

Everything has a type in the Java language, and this type is known at compile time. So it is always possible to find the type of a lambda expression. It may be the type of a variable, of a field, of a method parameter, or the returned type of a method.

There is a restriction on the type of a lambda expression: it has to be a functional interface. So an anonymous class that does not implement a functional interface cannot be written as a lambda expression.

The complete definition of what functional interfaces are is a little complex. All you need to know at this point is that a functional interface is an interface that has only one *abstract* method.

You should be aware that, starting with Java SE 8, concrete methods are allowed in interfaces. They can be instance methods, in that case, they are called *default methods*, and they can be static methods. These methods do not count, since they are not *abstract* methods.

> *Do I need to add the annotation `@FunctionalInterface` on an interface to make it functional?*
>
> *No you don't. This annotation is here to help you to make sure that your interface is indeed functional. If you put this annotation on a type that is not a functional interface, then the compiler will raise an error.*

Examples of Functional Interfaces

Let us see some examples taken from the JDK API. We have just removed the comments from the source code.

```
1  @FunctionalInterface
2  public interface Runnable {
3      public abstract void run();
4  }
```

The `Runnable` interface is indeed functional, because it has only one abstract method. The `@FunctionalInterface` annotation has been added as a helper,

but it is not needed.

```
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after) {
        // the body of this method has been removed
    }
}
```

The `Consumer<T>` interface is also functional: it has one abstract method and one default, concrete method that does not count. Once again, the `@FunctionalInterface` annotation is not needed.

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

    default Predicate<T> and(Predicate<? super T> other) {
        // the body of this method has been removed
    }

    default Predicate<T> negate() {
        // the body of this method has been removed
    }

    default Predicate<T> or(Predicate<? super T> other) {
        // the body of this method has been removed
    }

    static <T> Predicate<T> isEqual(Object targetRef) {
        // the body of this method has been removed
    }

    static <T> Predicate<T> not(Predicate<? super T> target) {
        // the body of this method has been removed
    }
```

```
25 }
```

The `Predicate<T>` interface is a little more complex, but it is still a functional interface:

- it has one abstract method
- it has three default methods that do not count
- and it has two static methods that do not count neither.

Finding the Right Method to Implement

At this point you have identified the type of the lambda expression you need to write, and the good news is: you have done the hardest part: the rest is very mechanical and easier to do.

A lambda expression is an implementation of the only abstract method in this functional interface. So finding the right method to implement is just a matter of finding this method.

You can take a minute to look for it in the three examples of the previous paragraph.

For the `Runnable` interface it is:

```
1 public abstract void run();
```

For the `Predicate` interface it is:

```
1 boolean test(T t);
```

And for the `Consumer<T>` interface it is:

```
1 void accept(T t);
```

Implementing the Right Method with a Lambda Expression

Writing a First Lambda Expression that implements `Predicate<String>`

Now for the last part: writing the lambda itself. What you need to understand is that the lambda expression you are writing is an implementation of the abstract method you found. Using the lambda expression syntax, you can nicely inline this implementation in your code.

This syntax is made of three elements:

- a block of parameters;
- a little piece of ASCII art depicting an arrow: `->`. Note that Java uses *meager arrows* (`->`) and not *fat arrows* (`=>`);
- a block of code which is the body of the method.

Let us see examples of this. Suppose you need an instance of a `Predicate` that returns `true` for strings of characters that have exactly 3 characters.

1. The type of your lambda expression is `Predicate`
2. The method you need to implement is `boolean test(String s)`

Then you write the block of parameters, which is simple copy / paste of the signature of the method: `(String s)`.

You then add a meager arrow: `->`.

And the body of the method. Your result should look like this:

```
1  Predicate<String> predicate =
2      (String s) -> {
3          return s.length() == 3;
4
```

```
        };
```

Simplifying the Syntax

This syntax can then be simplified, thanks to the compiler that can guess many things so that you do not need to write them.

First, the compiler knows that you are implementing the abstract method of the `Predicate` interface, and it knows that this method takes a `String` as a argument. So `(String s)` can be simplified to `(s)`. In that case, where there is only one argument, you can even go one step further by removing the parentheses. The block of arguments then becomes `s`. You should keep the parentheses if you have more than one argument, or no argument.

Second, there is just one line of code in the body of the method. In that case you do not need the curly braces nor the `return` keyword.

So the final syntax is in fact the following:

```
1 | Predicate<String> predicate = s -> s.length() == 3;
```

And this leads us to the first good practice: keep your lambdas short, so that they are just one line of simple, readable code.

Implementing a `Consumer<String>`

At some point, people may be tempted to take shortcuts. You will hear developers saying "a consumer takes an object and returns nothing". Or "a predicate is true when the string has exactly three characters". Most of the time, there is a confusion between the lambda expression, the abstract method it implements, and the functional interface that holds this method.

But since a functional interface, its abstract method, and a lambda expression that implements it are so closely linked together, this way of speaking actually makes total sense. So that's OK, as long as it does not lead to any ambiguity.

Let us write a lambda that consumes a `String` and prints on `System.out`. The syntax can be this one:

```
Consumer<String> print = s -> System.out.println(s);
```

Here we directly wrote the simplified version of the lambda expression.

Implementing a Runnable

Implementing a `Runnable` turns out to write an implementation of `void run()`. This block of arguments is empty, so it should be written with parentheses. Remember: you can omit the parentheses only if you have one argument, here we have zero.

So let us write a runnable that tells us that it is running:

```
Runnable runnable = () -> System.out.println("I am running");
```

Calling a Lambda Expression

Let us go back to our previous `Predicate` example, and suppose that this predicate has been defined in a method. How can you use it to test if a given string of characters is indeed of length 3?

Well, despite the syntax you used to write a lambda, you need to keep in mind that this lambda is an instance of the interface `Predicate`. This interface defines a method called `test()` that takes a `String` and returns a `boolean`.

Let us write that in a method:

```
List<String> retainStringsOfLength3(List<String> strings) {

    Predicate<String> predicate = s -> s.length() == 3;
    List<String> stringsOfLength3 = new ArrayList<>();
```

```
    for (String s: strings) {
        if (predicate.test(s)) {
            stringsOfLength3.add(s);
        }
    }
    return stringsOfLength3;
}
```

Note how you defined the predicate, just as you did in the previous example. Since the `Predicate` interface defines this method `boolean test(String)`, it is perfectly legal to call the methods defined in `Predicate` through a variable of type `Predicate`. This may look confusing at first, since this predicate variable does not look like it defines methods.

Bear with us, there are much better way to write this code, that you will see later in this tutorial.

So everytime you write a lambda, you can call any method defined on the interface this lamdba is implementing. Calling the abstract method will call the code of your lambda itself, since this lambda is an implementation of that method. Calling a default method will call the code written in the interface. There is no way a lambda can override a default method.

Capturing Local Values

Once you get used to them, writing lambdas becomes very natural. They are very well integrated in the Collections Framework, in the Stream API and in many other places in the JDK. Starting with Java SE 8, lambdas are everywhere, for the best.

There are constraints in using lambdas and you may come across compile-time errors that you need to understand.

Let us consider the following code:

```
int calculateTotalPrice(List<Product> products) {

    int totalPrice = 0;
    Consumer<Product> consumer =
        product -> totalPrice += product.getPrice();
    for (Product product: products) {
        consumer.accept(product);
    }
}
```

Even if this code may look nice, trying to compile it will give you the following error on the use of `totalPrice` in this `Consumer` implementation:

> *Variable used in lambda expression should be final or effectively final*

The reason is the following: lambdas cannot modify variables defined outside their body. They can read them, as long as they are `final`, that is, immutable. This process of accessing variable is called *capturing*: lambdas cannot *capture* variables, they can only *capture* values. A final variable is in fact a value.

You have noted that the error message tells us that the variable can be *final*, which is a classical concept in the Java language. It also tells us that the variable can be *effectively final*. This notion was introduced in Java SE 8: even if you do not explicitly declare a variable `final`, the compiler may do it for you. If it sees that this variable is read from a lambda, and that you do not modify it, then it will nicely add the `final` declaration for you. Of course this is done in the compiled code, the compiler does not modify your source code. Such variables are not called *final*; they are called *effectively final* variables. This is a very useful feature.

Serializing Lambdas

Lambda expressions have been built so that they can be serialized.

Why would you serialize lambda expressions? Well, a lambda expression may be stored in a field, and this field may be accessed through a constructor or a setter method. Then you may have a lambda expression in the state of your object at runtime, without being aware of it.

So to preserve backward compatibility with existing serializable classes, serializing a lambda expression is possible.

More Learning

Last update: *October 26, 2021*

Writing Your First Lambda Expression

➜

Using Lambdas Expressions in Your Application

Home > Tutorials > Lambda Expressions > Writing Your First Lambda Expression