Theory: Functional interfaces

32 minutes 0 / 0 problems solved

Skip this topic Start practicing

In this topic, you will learn a new concept called **functional interfaces**. This is vital knowledge because these kinds of interfaces allow developers to use functional programming over the concepts from OOP and, in a sense, connect both these worlds together. It is functional interfaces that make it possible to use lambda expressions, method references, and other functional stuff. We suppose that you are already familiar with interfaces, lambda expressions, and anonymous classes. Now we are going to show you how they are all connected with each other.

§1. Functions and interfaces

If an interface contains only a **single abstract method** (such an interface sometimes is called a **SAM type**), it can be considered as a function. In addition to standard ways of implementing interfaces through inheritance or anonymous classes, these interfaces can be implemented by using a lambda expression or a method reference.

Here is a clone of the standard `Function<T, R>` interface that demonstrates the basic idea:

```
@FunctionalInterface
interface Func<T, R> {
    R apply(T val); // single abstract method
}
```

The annotation `@FunctionalInterface` is used to mark functional interfaces and to raise a compile-time error if an interface doesn't satisfy the requirements of a functional interface. The use of this annotation is not mandatory but recommended.

The `Func<T, R>` interface meets the requirements to be a functional interface because it has a single `apply` method, which takes a value of the type `T` and returns a result of the type `R`.

Here is an example of a lambda expression that implements this custom interface:

```
Func<Integer, Integer> multiplier10 = n -> n * 10;
System.out.println(multiplier10.apply(5)); // 50
```

In a similar way, all standard functions can be represented as functional interfaces, including `BiFunction<T, U, R>`. The concept of functional interfaces is another way to model functions instead of using regular methods.

It is worth noticing, that `static` and `default` methods are allowed in functional interfaces because these methods are not abstract:

```
@FunctionalInterface
interface Func<T, R> {
    R apply(T val);

    static void doNothingStatic() { }

    default void doNothingByDefault() { }
}
```

This interface is a valid functional interface as well.

§2. Implementing functional interfaces

4 required topics

- ✗ Default methods
- ✗ Anonymous classes
- ✓ Lambda expressions
- ✓ Method references

6 dependent topics

- Standard functional interfaces
- Optional
- Executors
- Callable and Future
- Comparator
- Swing components

Table of contents:

There are several ways to implement a functional interface. As you know from the previous OOP theory, it is impossible to directly create an instance of an interface. Then what should we do?

First of all, we should implement the interface to create a concrete class. Then, create an instance of this concrete class. The main requirement is to implement the `apply` method to get a concrete behavior.

Let's consider three ways to do it.

1) Anonymous classes.

Of course, like any other interface, a functional interface can be implemented by using an anonymous class or regular inheritance.

To implement a functional interface let's create an anonymous class and override the `apply` method. The overridden method calculates the square of a given value:

```
Func<Long, Long> square = new Func<Long, Long>() {
    @Override
    public Long apply(Long val) {
        return val * val;
    }
};

long val = square.apply(10L); // the result is 100L
```

In this example, we model a math function that squares a given value. This code works perfectly but it is a bit unclear since it contains a lot of extra characters to perform a single line of useful code.

We won't give you an example of a regular class because it has the same (and even more) disadvantages.

2) Lambda expressions.

A functional interface can also be implemented and instantiated by using a lambda expression.

Here is a lambda expression that has the same behavior as the anonymous class above:

```
 Func<Long, Long> square = val -
> val * val; // the lambda expression

 long val = square.apply(10L); // the result is 100L
```

The type of the functional interface (left) and the type of the lambda (right) are the same from a semantic perspective. Parameters and the result of a lambda expression correspond to the parameters and the result of **a single abstract method** of the functional interface.

> The code that creates a lambda expression may look as if the object of an interface type is created. As you know, it is impossible. Actually, the Java compiler automatically creates a special intermediate class that implements the functional interface and then creates an object of this class rather than an object of an interface type. The name of such a class may look like `Functions$$Lambda$14/0x0000000100066840` or something similar.

3) Method references.

Another way to implement a functional interface is by using method references. In this case, the signature of a method should match the signature of the single abstract method of a functional interface.

Suppose, there is a method `square` that takes and returns a `long` value:

```
public static long square(long val) {
    return val * val;
}
```

The signature of this method fits the `Func<Long, Long>` functional interface. This means we can create a method reference and assign it to an object of the `Func<Long, Long>` type:

```
1  Func<Long, Long> square = Functions::square;
```

Keep in mind, that the compiler creates an intermediate hidden class that implements the `Func<Long, Long>` interface in a similar way to the case of lambda expressions.

> Usually, method references are more readable than the corresponding lambda expressions. Try to prefer this way of implementing and instantiating functional interfaces when possible.

§3. Conclusion

Here are the key points of this topic:

- a functional interface is a special kind of an interface containing a single abstract method, though it can also contain `static` and `default` methods;
- a functional interface can be implemented and instantiated by a lambda expression or a method reference;
- Java compiler automatically creates special hidden classes for lambda expressions and method references;
- it is possible to use the `@FunctionalInterface` annotation to force checking whether an interface satisfies the requirement of the functional interfaces;
- in Java, functional programming has been built on top of OOP, especially, interfaces and polymorphism.