Functional Interfaces in Java

In this tutorial, we'll explain how to use functional interfaces.

A functional interface has a single abstract method. It's recommended that we decorate it with the @FunctionalInterface annotation. This way, we can inform other developers that the interface should contain only one method, which will likely be used in a lambda expression.

Let's see some standard functional interfaces that come with Java.

The Function Interface

The Function interface is a generic interface that accepts one argument and returns a result:

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);
```

```
}
```

It contains the apply() method, which applies a function to a given argument and produces a result. The type of an argument is defined with T, while the result is described with type R.

Let’s see how to implement it. Suppose we’d like to define a function that will take a String and return its length:

```
Function<String, Integer> length = new Function<String, Integer>() {
    @Override
    public Integer apply(final String s) {
        return s.length();
    }
};
```

We can simplify it further with lambda expression:

```
Function<String, Integer> length = s -> s.length();
```

Next, we can call the apply() method on any String:

```
length.apply(“Tom"); // returns 3
```

The Supplier Interface

Unlike the Function interface, the Supplier interface doesn’t accept any parameter but returns a value:

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

    T get();
}
```

We usually use it when we want to produce some result without taking any input. Let’s see how to implement it to generate random numbers:

```
Supplier<Double> generatedNumber = new Supplier<Double>() {
    @Override
    public Double get() {
        Random random = new Random();
        return random.nextDouble();
    }
};
```

Using lambda expression, the implementation would look like this:

```
Supplier<Double> generatedNumber = () -> {
    Random random = new Random();
    return random.nextDouble();
};
```

Then, we can call the get() method to return a randomly generated number:

generatedNumber.get()

The Consumer Interface

Moving on to the Consumer interface. It accepts the value but doesn't return the result:

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

    void accept(T t);
}
```

We can use the Consumer interface when we need to perform some action on the value without returning the result.

For instance, we can use it to print a given value in the standard output:

```
Consumer<String> printable = new Consumer<String>() {
    @Override
    public void accept(final String s) {
        System.out.println(s);
    }
};
```

Lambda expression equivalent:

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

We can call the accept() method and pass the String value we want to print:

```
printable.accept("This is cool.”);
```

The Predicate Interface

Lastly, let's see the Predicate functional interface. It consists of one method that accepts an argument and returns a boolean as a result:

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

    boolean test(T t);
}
```

We often use this interface when we need to filter some data.

For example, we can create an implementation that checks whether a given value length equals 5:

```
Predicate<String> checkLength = new Predicate<String>() {
    @Override
    public boolean test(final String s) {
        return s.length() == 5;
    }
```

};

If we use a lambda expression, the implementation looks like the following:

Predicate<String> checkLength = s -> s.length() == 5;

Let’s call the test() method:

boolean b = checkLength.test(“Animal"); // returns true

Conclusion

In this article, we learned the most common functional interfaces in Java.

To summarize, each functional interface serves a different need. We use them when working on streams of different data structures, such as collections.

Read More