Functional Interfaces in Java - HowToDoInJava

3:11 Estimated 667 Words EN Language

Lokesh Gupta

February 26, 2022

Java Streams
Functional Interface, Java 8 Stream

Introduced in Java 8, **a functional interface is simply an interface that has exactly one abstract method**. Learn more about functional interfaces in this tutorial.

1. What is a Functional Interface?

1.1. Only one `abstract` method is allowed

Functional interfaces are new additions in Java 8. **As a rule, a functional interface can contain exactly one abstract method.** These functional interfaces are also called **Single Abstract Method interfaces (SAM Interfaces)**.

Apart from one abstract method, a **functional interface can also have the following methods that do not count** for defining it as a functional interface.

- Default methods
- Static methods
- Public methods inherited from the *Object* class

1.2. Implemented by Lambda Expressions

In Java, *lambda expressions* can be used to represent an instance of a functional interface. For example, Comparator interface is a functional interface.

```
@FunctionalInterface
public interface Comparator<T> {
        int compare(T o1, T o2);
        boolean equals(Object obj);
```

```
        //and multiple default methods...
}
```

Comparator interface has only two abstract methods `compare()` and `equals()` . But *equals()* has been inherited from the *Object* class, so it is not counted. Other than these two methods, all other methods are *default methods*. So *Comparator* is qualified to be declared as a functional interface.

Java program to implement *Comparator* using a lambda expression.

```
//Compare by Id
Comparator<Employee> compareById = Comparator.comparing(e -> e.getId());

Comparator<Employee> compareByFirstName = Comparator.comparing(e -> e.getFir
stName());
```

2. @FunctionalInterface Annotation

Java 8 introduced the annotation `@FunctionalInterface` to mark an interface as a functional interface. The primary use of this annotation is **for compiler-level errors when the interface violates the contracts of precisely one abstract method**.

> Note that **using the annotation *@FunctionalInterface* is optional**.

If the interface has one abstract method and does not have *@FunctionalInterface* annotation, the interface is still a functional interface, and it can be the target type for lambda expressions.

The presence of the annotation protects us from inadvertently changing a functional interface into a non-functional interface, as the compiler will catch it.

Let's build our first functional interface. Note that methods in an interface are, by default, *abstract*.

```
@FunctionalInterface
public interface MyFirstFunctionalInterface
{
    public void firstWork();
}
```

Let's try to add another abstract method:

```
@FunctionalInterface
public interface MyFirstFunctionalInterface
{
    public void firstWork();
    public void doSomeMoreWork();   //error
}
```

The above code will result in a compiler error:

```
Unexpected @FunctionalInterface annotation
@FunctionalInterface ^ MyFirstFunctionalInterface is not a functional interf
ace
multiple non-overriding abstract methods found in interface MyFirstFunctiona
lInterface
```

```
1   package functionalInterfaceExample;
2
3   Unexpected @FunctionalInterface annotation
4    MyFirstFunctionalInterface is not a functional interface
5     multiple non-overriding abstract methods found in interface MyFirstFunctionalInterface
6   ----
7   (Alt-Enter shows hints)
    @FunctionalInterface
9   public interface MyFirstFunctionalInterface {
10      public void firstWork();
11      public void doSomeMoreWork();
12  }
13
```

> Read More : Generic Functional Interfaces

3. Functional Interfaces in JDK

The following is a list of Java's most commonly used functional interfaces.

- Runnable: contains only the *run()* method.
- Comparable: contains only the *compareTo()* method.
- ActionListener: contains only the *actionPerformed()* method.

- *Callable*: contains only the *call()* method.
- *Predicate*: a boolean-valued function that takes an argument and returns true or false.
- *BiPredicate*: a predicate with two arguments.
- *Consumer*: an operation that takes an argument, operates on it, and returns no result.
- *BiConsumer*: a consumer with two arguments.
- *Supplier*: a supplier that returns a value.
- *Function<T, R>*: takes an argument of type T and returns a result of type R.
- *BiFunction<T, U, R>*: takes two arguments of types T and U and returns a result of type R.

4. Demo

Let's see a quick example of creating and using functional interfaces in Java.

We are using a functional interface *Function* to create the formula for mathematical squares.

```
Function<Integer, Integer> square = x -> x * x;
```

The Function interface has one abstract method `apply()` that we have implemented above. we can execute the above method as follows:

```
System.out.println( square.apply(5) );  //Prints 25
```

5. Conclusion

In this tutorial, we learned to create and manage functional interfaces in Java. We learned that a *functional interface* has only one *abstract* method and they can be implemented by the lambda expressions.

We also saw the JDK provided existing functional interfaces, and finally how to create an use a functional interface.

Happy Learning !!

> Sourcecode on Github

Further reading:

- Python Interview Questions and Answers
- Difference between Interface and Abstract Class in Java
- Java Concurrency Interview Questions
- Java List Sorting: Comparable and Comparator Examples
- Lambda Expressions in Java
- Java OOP Interview Questions with Answers