Functional Interfaces in Java

Difficulty Level : Hard Last Updated : 16 Jan, 2022

Java has forever remained an Object-Oriented Programming language. By object-oriented programming language, we can declare that everything present in the Java programming language rotates throughout the Objects, except for some of the primitive data types and primitive methods for integrity and simplicity. There are no solely functions present in a programming language called Java. Functions in the Java programming language are part of a class, and if someone wants to use them, they have to use the class or object of the class to call any function.

A **functional interface** is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. ***Runnable***, ***ActionListener***, ***Comparable*** are some of the examples of functional interfaces.

Functional Interface is additionally recognized as **Single Abstract Method Interfaces**. In short, they are also known as **SAM interfaces**. Functional interfaces in Java are the new feature that provides users with the approach of fundamental programming.

Functional interfaces are included in Java SE 8 with Lambda expressions and Method references in order to make code more readable, clean, and straightforward. Functional interfaces are interfaces that ensure that they include precisely only one abstract method. Functional interfaces are used and executed by representing the interface with an **annotation called *@FunctionalInterface***. As described earlier, functional interfaces can contain only one abstract method. However, they can include any quantity of default and static methods.

In Functional interfaces, there is no need to use the abstract keyword as it is optional to use the abstract keyword because, by default, the method defined inside the interface is abstract only. We can also call Lambda expressions as the instance of functional interface.

Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

```
// Java program to demonstrate functional interface

class Test {
```

```
    public static void main(String args[])
    {
        // create anonymous inner class object
        new Thread(new Runnable() {
            @Override public void run()
            {
                System.out.println("New thread created");
            }
        }).start();
    }
}
```

Output

```
New thread created
```

Java 8 onwards, we can assign lambda expression to its functional interface object like this:

```
// Java program to demonstrate Implementation of
// functional interface using lambda expressions

class Test {
    public static void main(String args[])
    {

        // lambda expression to create the object
        new Thread(() -> {
            System.out.println("New thread created");
        }).start();
    }
}
```

Output

```
New thread created
```

@FunctionalInterface Annotation

@FunctionalInterface annotation is used to ensure that the functional interface can't have more than one abstract method. In case more than one abstract methods are present, the compiler flags an 'Unexpected @FunctionalInterface annotation' message. However, it is not mandatory to use this annotation.

```
// Java program to demonstrate lambda expressions to
// implement a user defined functional interface.

@FunctionalInterface

interface Square {
    int calculate(int x);
}

class Test {
    public static void main(String args[])
    {
        int a = 5;

        // lambda expression to define the calculate method
        Square s = (int x) -> x * x;

        // parameter passed and return type must be
        // same as defined in the prototype
        int ans = s.calculate(a);
        System.out.println(ans);
    }
}
```

Output

```
25
```

Some Built-in Java Functional Interfaces

Since Java SE 1.8 onwards, there are many interfaces that are converted into functional interface. All these interfaces are annotated with @FunctionalInterface. These interfaces are as follows –

- **Runnable –>** This interface only contains the run() method.
- **Comparable –>** This interface only contains the compareTo() method.

- **ActionListener –>** This interface only contains the actionPerformed() method.
- **Callable –>** This interface only contains the call() method.

Java SE 8 included four main kinds of functional interfaces which can be applied in multiple situations. These are:

1. Consumer
2. Predicate
3. Function
4. Supplier

Amidst the previous four interfaces, the first three interfaces,i.e., Consumer, Predicate, and Function, likewise have additions that are provided beneath –

1. Consumer -> Bi-Consumer
2. Predicate -> Bi-Predicate
3. Function -> Bi-Function, Unary Operator, Binary Operator

1. Consumer

The consumer interface of the functional interface is the one that accepts only one argument or a gentrified argument. The consumer interface has no return value. It returns nothing. There are also functional variants of the Consumer — DoubleConsumer, IntConsumer, and LongConsumer. These variants accept primitive values as arguments.

Other than these variants, there is also one more variant of the Consumer interface known as Bi-Consumer.

Bi-Consumer – Bi-Consumer is the most exciting variant of the Consumer interface. The consumer interface takes only one argument, but on the other side, the Bi-Consumer interface takes two arguments. Both, Consumer and Bi-Consumer have no return value. It also returns noting just like the Consumer interface. It is used in iterating through the entries of the map.

Syntax / Prototype of Consumer Functional Interface –

```
Consumer<Integer> consumer = (value) -> System.out.println(value);
```

This implementation of the Java Consumer functional interface prints the value passed as a parameter to the print statement. This implementation uses the Lambda function of Java.

2. Predicate

In scientific logic, a function that accepts an argument and, in return, generates a boolean value as an answer is known as a predicate. Similarly, in the java programming language, a predicate functional interface of java is a type of function which accepts a single value or argument and does some sort of processing on it, and returns a boolean (True/ False) answer. The implementation of the Predicate functional interface also encapsulates the logic of filtering (a process that is used to filter stream components on the base of a provided predicate) in Java.

Just like the Consumer functional interface, Predicate functional interface also has some extensions. These are IntPredicate, DoublePredicate, and LongPredicate. These types of predicate functional interfaces accept only primitive data types or values as arguments.

Bi-Predicate – Bi-Predicate is also an extension of the Predicate functional interface, which, instead of one, takes two arguments, does some processing, and returns the boolean value.

Syntax of Predicate Functional Interface –

```
public interface Predicate<T> {

    boolean test(T t);

}
```

The predicate functional interface can also be implemented using a class. The syntax for the implementation of predicate functional interface using a class is given below –

```
public class CheckForNull implements Predicate {

    @Override
    public boolean test(Object o) {

        return o != null;

    }
}
```

The Java predicate functional interface can also be implemented using Lambda expressions. The example of implementation of Predicate functional interface is given below –

```
Predicate predicate = (value) -> value != null;
```

This implementation of functional interfaces in Java using Java Lambda expressions is more manageable and effective than the one implemented using a class as both the implementations are doing the same work, i.e., returning the same output.

3. Function

A function is a type of functional interface in Java that receives only a single argument and returns a value after the required processing. There are many versions of Function interfaces because a primitive type can't imply a general type argument, so we need these versions of function interfaces. Many different versions of the function interfaces are instrumental and are commonly used in primitive types like double, int, long. The different sequences of these primitive types are also used in the argument.

These versions are:

Bi-Function – The Bi-Function is substantially related to a Function. Besides, it takes two arguments, whereas Function accepts one argument.

The prototype and syntax of Bi-Function is given below –

```
@FunctionalInterface
public interface BiFunction<T, U, R>
{

   R apply(T t, U u);
    .......

}
```

In the above code of interface, T, U are the inputs, and there is only one output that is R.

Unary Operator and Binary Operator – There are also two other functional interfaces which are named as Unary Operator and Binary Operator. They both extend the Function and Bi-Function, respectively. In simple words, Unary Operator extends Function, and Binary Operator extends Bi-Function.

The prototype of the Unary Operator and Binary Operator is given below –

1. Unary Operator

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, U>
{
```

```
    ……...
}
```

2. Binary Operator

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, U, R>
{
    ……...
}
```

We can understand front the above example that the Unary Operator accepts only one argument and returns a single argument only. Still, in Unary Operator both the input and output values must be identical and of the same type.

On the other way, Binary Operator takes two values and returns one value comparable to Bi-Function but similarly like Unary Operator, the input and output value type must be identical and of the same type.

4. Supplier

The Supplier functional interface is also a type of functional interface that does not take any input or argument and yet returns a single output. This type of functional interface is generally used in the lazy generation of values. Supplier functional interfaces are also used for defining the logic for the generation of any sequence. For example – The logic behind the Fibonacci Series can be generated with the help of the Stream.generate method, which is implemented by the Supplier functional Interface.

The different extensions of the Supplier functional interface hold many other supplier functions like BooleanSupplier, DoubleSupplier, LongSupplier, and IntSupplier. The return type of all these further specializations is their corresponding primitives only.

Syntax / Prototype of Supplier Functional Interface is –

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

// gets a result
…….

// returns the specific result
```

```
.....

T.get();

}
```

```
// A simple program to demonstrate the use
// of predicate interface

import java.util.*;
import java.util.function.Predicate;

class Test {
    public static void main(String args[])
    {

        // create a list of strings
        List<String> names = Arrays.asList(
            "Geek", "GeeksQuiz", "g1", "QA", "Geek2");

        // declare the predicate type as string and use
        // lambda expression to create object
        Predicate<String> p = (s) -> s.startsWith("G");

        // Iterate through the list
        for (String st : names) {
            // call the test method
            if (p.test(st))
                System.out.println(st);
        }
    }
}
```

Output

```
Geek
GeeksQuiz
Geek2
```

Important Points/Observations:

Here are some significant points regarding Functional interfaces in Java:

1. In functional interfaces, there is only one abstract method supported. If the annotation of a functional interface, i.e., @FunctionalInterface is not implemented or written with a function interface, more than one abstract method can be declared inside it. However, in this situation with more than one functional interface, that interface will not be called a functional interface. It is called a non-functional interface.
2. There is no such need for the @FunctionalInterface annotation as it is voluntary only. This is written because it helps in checking the compiler level. Besides this, it is optional.
3. An infinite number of methods (whether static or default) can be added to the functional interface. In simple words, there is no limit to a functional interface containing static and default methods.
4. Overriding methods from the parent class do not violate the rules of a functional interface in Java.
5. The **java.util.function** package contains many built-in functional interfaces in Java 8.

This article is contributed by **Akash Ojha**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.