Functional Interface in detail introduced from Java 8 explained

Functional interfaces are introduced as part of java 8. It is implemented using the annotation called @FunctionalInterface. It ensures that the interface should have only one abstract method. The usage of the abstract keyword is optional as the method defined inside interface is by default abstract. It is important to note that a functional interface can have multiple default methods (it can be said concrete methods which are default) but only one abstract method. The default method has been introduced in interface so that a new method can be appended in the class without affecting the implementing class of the existing interfaces. Because prior to java 8 the implementing class of an interface has to implement all the abstract methods defined in the interface.

The functional interface has been introduced in java 8 to support the lambda expression in java 8 on the other hand it can be said lambda expression is the instance of functional interface.

For example:-

@FunctionalInterface

Interface Area<T,S,R>

{

R calculateArea(T a, S b);

}

It is well known that how the above interface is implemented prior to java 8. But in java 8 it can be implemented in a smarter way. Find the below implementation.

static Area<Double, Double, Double> area = (a, b) ->(a\*b);

System.out.println("The total area is "+ area.calculateArea(1.1,1.1));

The full implementation of the above functional interface details is available in GitHub, the link of which will be provided at the end of this tutorial.

In Java 8 there are 4 main functional interfaces are introduced which could be used in different scenarios. These are given below.

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

Among the above four interfaces, the first three interfaces also have extensions also which are given below.

Consumer - BiConsumer

Predicate – BiPredicate

Function – BiFunction, UnaryOperator, BinaryOperator

1. CONSUMER

Let’s discuss about Consumer.

* The consumer interface accepts one argument but there is no return value.
* The name of function inside this interface is accept.

@FunctionalInterface

public interface Consumer<T> {

void accept(T t);

…

}

Consumer<String> ucConsumer = (s) -> System.out.println(s.toUpperCase());

ucConsumer.accept("consumer");

Output: CONSUMER

The above consumer is accepting one argument and print it in upper case. But there is no return value.

* + BICONSUMER

The extension of the Consumer which is BiConsumer accepts two arguments and return nothing.

@FunctionalInterface

public interface BiConsumer<T, U> {

void accept(T t, U u);

...

}

BiConsumer<String, String> biConsumer = (x,y) -> {

System.out.println(" x : " + x + " y : " + y );

};

biConsumer.accept("Sun" , "Moon");

Output: x: Sun y: Moon

1. PREDICATE

Predicate

Predicate will accept one argument, do some processing and then return boolean.

@FunctionalInterface

public interface Predicate<T> {

boolean test(T t);

...

}

static Predicate<Integer> p = (i) -> {return i%2 ==0;};

System.out.println("Result is p : " + p.test(4));

Output: true

* + BIPREDICATE

BiPredicate

Instead of one argument BiPredicate will accept two arguments and return nothing.

1. FUNCTION

This interface accepts one argument and return a value after the required processing. It is defined as below. The required processing logic will be executed on invocation of the apply method.

@FunctionalInterface

public interface Function<T, R> {

R apply(T t);

…..

}

For example,

static Function<String,String> upperCase = (name) -> name.toUpperCase();

System.out.println("Result is : " + upperCase.apply("functional"));

Output: FUNCTIONAL

In the above example the Function is accepting one string and also returns one string.

* + BIFUNCTION

The BiFunction is similar to Function except it accepts two inputs whereas Function accepts one argument. The sample code for the BiFunction interface is given below. In the below interface code T, U are the inputs and R is the single output.

@FunctionalInterface

public interface BiFunction<T, U, R> {

R apply(T t, U u);

……

}

* + UNARYOPERATOR & BINARYOPERATOR

Another two interfaces are UnaryOperator and BinaryOperator which extends the Function and BiFunction respectively. The following interface code snippets are given below for these two interfaces.

@FunctionalInterface

public interface UnaryOperator<T> extends Function<T, T> {

…

}

@FunctionalInterface

public interface BinaryOperator<T> extends BiFunction<T,T,T> {

...

}

From the above interfaces it is easy to understand that the UnaryOperator accepts a single argument and return a single argument but both the input and output argument should be of same of similar type.

On the other hand BinaryOperator accepts two arguments and return one argument similar to BiFunction but type of all the input and output argument should be of similar type.

The following examples for UnaryOperator and BinaryOperator interfaces are given respectively.

UnaryOperator<String> unaryOperator = (s)->s.concat("Operator");

System.out.println(unaryOperator.apply("Unary"));

Output: UnaryOperator

BinaryOperator<Integer> binaryOperator = (a,b) -> a+b;

System.out.println(binaryOperator.apply(3,4));

Output: 7

1. SUPPLIER

Supplier functional interface does not accept any input rather return single output. The following interface code is given for understanding.

@FunctionalInterface

public interface Supplier<T> {

/\*\*

\* Gets a result.

\*

\* @return a result

\*/

T get();

}

Find the below code example:-

public static Supplier<String> helloWorld = () -> {

return "Hello World";

};

String greeting = helloWorld.get();

System.out.println("The greeting message: "+greeting)

The above is just a simple example but it may be used in complex business scenario. For all the above scenarios download the sample code from <https://github.com/amit-datta/advancedJava.git>.

Lambda expression in Java 8

In my [previous](https://dzone.com/articles/functional-interface-explained-in-detail-introduce) article I have explained all the functional interfaces introduced in java 8. Now I am going to explain some more details in lambda expression. The github link for all the code examples used in this article is give at the end of this article.

The plain and simple definition of Lambda expression – It is a function without having any name. It can be written exactly in place, even it can be used as a parameter in a function.

Initially, it sounds like vague or little confusing. Let’s look into some syntax followed by examples.

(parameter to the function) -> {body of the function}

In the above syntax function parameter is present, function body is also present but the function name is not present.

For example I would like to write one function called “addition” which will take two parameters and after adding those two parameters it will return the result. We can easily write this function without using lambda expression in the following way.

static int addition(int a, int b)

{

return a+b;

}

But we can declare the same using Lambda expression also. First I will show using the BiFunction functional interface and then I will show the same by creating a separate functional interface, respectively.

static BiFunction<Integer,Integer,Integer> addition\_Lambda = (a,b) -> {

return a+b;

};

The following is the custom functional interface declared for this addition.

static AdditionLambda<Integer,Integer,Integer> additionLambda = (a,b) -> {

return a + b;

};

In the same way we can implement our different function using lambda expression. But it seems there is not much benefit rather making the simple things more complex. We will see how this lambda is beneficial for the java developer.

Let’s take the Runnable interface which is being used to create new thread for asynchronous task. Before Lambda we have used the Runnable in our multiple applications to create new thread and assign a particular task to it. The code snippet is given below.

//Define Runnable prior to Java8

static Runnable runnable = new Runnable() {

@Override

public void run() {

System.out.println("Runnable Task");

}

};

//Invocation of Runnable prior to Java8

new Thread(runnable).start();

But after Lambda expression has been introduced we can write the Runnable in the following way.

//Define Runnable using Runnable

static Runnable runnableLambda = () -> {

System.out.println("Runnable Task 1");

System.out.println("Runnable Task 2");

};

//Invocation of Runnable Lambda

new Thread(runnableLambda).start();

Or, we can define the above in simpler way, given below.

//Define and invocation of Runnable using lambda in more precise way

new Thread(() -> {

System.out.println("Runnable Task 1");

System.out.println("Runnable Task 2");

}).start();

Similarly, we can show how the comparator interface can be implemented using lambda expression in more precise way.

The following code snippet is shown how comparator is being used without lambda expression.

static Comparator<Integer> comparator = new Comparator<Integer>() {

@Override

public int compare(Integer intVar1, Integer intVar2) {

return intVar1.compareTo(intVar2);

}

};

System.out.println(comparator.compare(6,4));

Then, find the below code snippet to show how comparator can be used using lambda expression.

static Comparator<Integer> comparatorByLambda = (a,b) -> a.compareTo(b);

System.out.println("Comparator Result using lambda: "+comparatorByLambda.compare(6,4));

Now, we can see how the lambda expression is beneficial for the java developers.

Now, I am going provide some of the example using lambda. These are very useful and java developers use these in frequent manner to build their java application.

* How we can use the ArrayList using lambda

List<String> cities = Arrays.asList("Delhi", "Kolkata", "Chennai", "Mumbai");

System.out.println("Print name of cities without using lambda");

for(int i=0;i<cities.size();i++){

System.out.println(cities.get(i));

}

System.out.println("Print name of cities using lambda");

cities.forEach((city)->{

System.out.println(city);

});

* How HashMap can be used using Lambda

Map<String,String> cityMap = new HashMap<>();

cityMap.put("id","456329");

cityMap.put("name","Tushar Sharma");

cityMap.put("designation","Manager");

cityMap.forEach((k,v)->{

System.out.print("Key: "+k+"\t");

System.out.print("Value: "+v+"\n");

});

* How lambda function can be passed as function parameter. Please find the below code example. In the below code the first argument of the function “getEmployeeName” is a argument of type functional interface and to pass the parameter value we have used lambda expression while invoking the function.

public static String getEmployeeName(Function<Employee,String> getEmpInfoFn,Employee emp){

return getEmpInfoFn.apply(emp);

}

System.out.println("How to use Lambda as function parameter. ");

List<Employee> empList = EmployeeDB.getEmployees();

Employee emp = EmployeeDB.getEmployees().get(0);

String empName = getEmployeeName(empVar->empVar.getName(),emp);

String empDesignation = getEmployeeName(empVar->empVar.getDesignation(),emp);

System.out.println("Employee Name: "+empName);

System.out.println("Employee Designation: "+empDesignation);

The supporting code examples used in the above article is given [here](https://github.com/amit-datta/advancedJava). Please look into the package called lambda for the above code examples.