Java 8 Features

# Lambda Expression

# Functional Interface

# Static and Default Methods in Interfaces in Java

# 1. Lambda Expression

#### 1.1 What is a lambda expression in Java?

* Lambda expression (or function) is just an **anonymous function**,
* i.e., a function with **no name** and without being bounded to an identifier.
* They are written exactly in the place where it’s needed, typically as a parameter to some other function.
* The most important features of Lambda Expressions is that **they execute in the context of their appearance**.
* So, a similar lambda expression can be executed differently in some other context (i.e. logic will be same but results will be different based on different parameters passed to function).

#### So, it is clear that lambda is some kind of function without name and identifiers. Well, what’s big deal? Why everyone is so excited?

* The answer lies in the benefits involved in functional programming over object oriented programming (OOP). Most OOP languages evolve around objects and instances and treat only them their first class citizens. Another important entity i.e. functions take back seat. This is specially true in java, where functions can’t exist outside an object. A function itself does not mean anything in java, until it is related to some object or instance.
* But in functional programming, you can define functions, give them reference variables and pass them as method arguments and much more. JavaScript is a good example of this where you can pass callback methods e.g. to Ajax calls. It’s very useful feature and has been lacking in java from beginning. Now with java 8, we can also use these lambda expressions.
* The basic syntax of a lambda expression is:

either

(parameters) -> expression

or

(parameters) -> { statements; }

or

() -> expression

* A typical lambda expression example will be like this:

|  |
| --- |
| (x, y) -> x + y  // This function takes two parameters and return their sum. |

* Please note that based on type of x and y, method may be used in multiple places.
* Parameters can match to int, or Integer or simply String also.
* Based on context, it will either add two integers or concat two strings.

#### 1.2 Rules for writing lambda expressions

1. A lambda expression can have zero, one or more parameters.
2. The type of the parameters can be explicitly declared or it can be inferred from the context.
3. Multiple parameters are enclosed in mandatory parentheses and separated by commas. Empty parentheses are used to represent an empty set of parameters.
4. When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses. e.g. a -> return a\*a.
5. The body of the lambda expressions can contain zero, one or more statements.
6. If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression. When there is more than one statement in body than these must be enclosed in curly brackets.

#### 1.3 Lambda examples

Let’s see some examples of lambda expressions as well:

|  |
| --- |
| (int a, int b) ->    a \* b               // takes two integers and returns their multiplication    (a, b)          ->   a - b               // takes two numbers and returns their difference    () -> 99                                // takes no values and returns 99    (String a) -> System.out.println(a)     // takes a string, prints its value to the console, and returns nothing    a -> 2 \* a                               // takes a number and returns the result of doubling it    c -> { //some complex statements }   // takes a collection and do some procesing |

#### 1.4 What’s functional Interface in Java 8

* Single Abstract Method interfaces (SAM Interfaces) is not a new concept.
* It means interfaces with **only one single method**.
* In java, we already have many examples of such SAM interfaces.
* From java 8, they will also be referred as functional interfaces as well.
* Java 8, enforces the rule of single responsibility by marking these interfaces with a new annotation i.e. **@FunctionalInterface.**
* For example, new definition of Runnable interface is like this:

|  |
| --- |
| @FunctionalInterface  public interface Runnable {      public abstract void run();  } |

* If you try to add a new method in any functional interface, compiler would not allow you to do this and will **throw compile time error**.

So far so good. But, **how they are related to Lambda expressions?** Let’s find out the answer.

* We know that Lambda expressions are anonymous functions with no name and they are passed (mostly) to other functions as parameters.
* Well, in java method parameters always have a type and this type information is looked for to determine which method needs to be called in case of method overloading or even simple method calling.
* So, basically every lambda expression also must be convertible to some type to be accepted as method parameters. Well, that **type in which lambda expressions are converted, are always of functional interface type**.

Let’s understand it with an example. If we have to write a thread which will print “eMexo Technologies” in console then simplest code will be:

|  |
| --- |
| new Thread(new Runnable() {      @Override      public void run() {          System.out.println("eMexo Technologies");      }  }).start(); |

If we use the lambda expression for this task then code will be :

|  |
| --- |
| new Thread(              () ->   {                          System.out.println("My Runnable");                      }           ).start(); |

* We have also see that Runnable is an functional interface with single method run().
* So, when you pass lambda expression to constructor of Thread class, compiler tries to convert the expression into equivalent Runnable code as shown in first code sample.
* If compiler succeed then everything runs fine, if compiler is not able to convert the expression into equivalent implementation code, it will complain.
* Here, in above example, lambda expression is converted to type Runnable.

In simple words, a lambda expression is an instance of a functional interface. But a lambda expression itself does not contain the information about which functional interface it is implementing; that information is deduced from the context in which it is used.

# 2. Functional Interface

## 2.1 What is Functional Interface?

* Functional interfaces are new additions in [java 8](https://howtodoinjava.com/category/java8/) which permit **exactly one abstract method inside them**. These interfaces are also called Single Abstract Method interfaces (SAM Interfaces).
* In Java 8, functional interfaces can be represented using lambda expressions, method reference and constructor references as well.
* Java 8 introduces an annotation i.e. @FunctionalInterface too, which can be used for compiler level errors when the interface you have annotated violates the contracts of exactly one abstract method.
* Let’s build our first functional interface:

|  |
| --- |
| Functional Interface Definition |
| @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  } |

* Above will result into compiler error

## 2.2 Do’s and Don’t’s in functional interfaces

Below is list of things which are allowed and which are not in a functional interface.

* As discussed above, only one abstract method is allowed in any functional interface. Second abstract method is not not permitted in a functional interface. If we remove @FunctionInterface annotation then we are allowed to add another abstract method, but it will make the interface non-functional interface.
* A functional interface is valid even if the @FunctionalInterface annotation would be omitted. It is only for informing the compiler to enforce single [abstract method](https://howtodoinjava.com/object-oriented/exploring-interfaces-and-abstract-classes-in-java/) inside interface.
* Conceptually, a functional interface has exactly one abstract method. Since [default methods](https://howtodoinjava.com/java8/default-methods-in-java-8/) have an implementation, they are not abstract. Since default methods are not abstract you’re free to add default methods to your functional interface as many as you like.
* Below is valid functional interface:

|  |
| --- |
| @FunctionalInterface  public interface MyFirstFunctionalInterface  {      public void firstWork();        default void doSomeMoreWork1(){      //Method body      }        default void doSomeMoreWork2(){      //Method body      }  } |

* If an interface declares an abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface’s abstract method count since any implementation of the interface will have an implementation from java.lang.Object or elsewhere. e.g. [Comparator](https://howtodoinjava.com/search-sort/when-to-use-comparable-and-comparator-interfaces-in-java/) is a functional interface even though it declared two abstract methods. Why? Because one of these abstract methods “equals()” which has signature equal to public method in Object class.
* e.g. Below interface is a valid functional interface.

|  |
| --- |
| @FunctionalInterface  public interface MyFirstFunctionalInterface  {      public void firstWork();        @Override      public String toString();                //Overridden from Object class        @Override      public boolean equals(Object obj);        //Overridden from Object class  } |

* The java.util.function package contains many builtin functional interfaces in Java 8.

## 2.2 java.util.function Package

The java.util.function package in Java 8 contains many builtin functional interfaces like-

**Predicate:**

* The Predicate interface has an abstract method test which gives a Boolean value as a result for the specified argument. Its prototype is

public Predicate

{

public boolean test(T t);

}

**BinaryOperator:**

* The BinaryOperator interface has an abstract method apply which takes two argument and returns a result of same type. Its prototype is

public interface BinaryOperator

{

public T apply(T x, T y);

}

**Function:**

* The Function interface has an abstract method apply which takes argument of type T and returns a result of type R. Its prototype is

public interface Function

{

public R apply(T t);

}

**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

# 3. Static and Default methods in Interface

## 3.1 Why default methods in interfaces are needed?

* Like regular interface methods, default methods are implicitly public — there's no need to specify the public modifier.
* Unlike regular interface methods, they are declared with the default keyword at the beginning of the method signature, and they provide an implementation.

Let's see a simple example:

|  |  |
| --- | --- |
|  | public interface MyInterface {        // regular interface methods        default void defaultMethod() {          // default method implementation      }  } |

* The reason why default methods were included in the Java 8 release is pretty obvious.
* In a typical design based on abstractions, where an interface has one or multiple implementations, if one or more methods are added to the interface, all the implementations will be forced to implement them too. Otherwise, the design will just break down.
* Default interface methods are an efficient way to deal with this issue. They allow us to add new methods to an interface that are automatically available in the implementations. Thus, there's no need to modify the implementing classes.
* In this way, backward compatibility is neatly preserved without having to refactor the implementers.

## 3.2 Default Interface Methods in Action

* To better understand the functionality of *default* interface methods, let's create a simple example.
* Say that we have a naive *Vehicle* interface and just one implementation. There could be more, but let's keep it that simple:

public interface Vehicle {

    String getBrand();

    String speedUp();

    String slowDown();

    default String turnAlarmOn() {

        return "Turning the vehicle alarm on.";

    }

    default String turnAlarmOff() {

        return "Turning the vehicle alarm off.";

    }

}

And let's write the implementing class:

public class Car implements Vehicle {

    private String brand;

    // constructors/getters

    @Override

    public String getBrand() {

        return brand;

    }

    @Override

    public String speedUp() {

        return "The car is speeding up.";

    }

    @Override

    public String slowDown() {

        return "The car is slowing down.";

    }

}

Lastly, let's define a typical *main* class, which creates an instance of*Car* and calls its methods:

public static void main(String[] args) {

    Vehicle car = new Car("BMW");

    System.out.println(car.getBrand());

    System.out.println(car.speedUp());

    System.out.println(car.slowDown());

    System.out.println(car.turnAlarmOn());

    System.out.println(car.turnAlarmOff());

}

* Please notice how the *default* methods *turnAlarmOn()* and *turnAlarmOff()* from our *Vehicle* interface are **automatically available in the***Car***class**.
* Furthermore, if at some point we decide to add more *default* methods to the *Vehicle* interface, the application will still continue working, and we won't have to force the class to provide implementations for the new methods.
* The most typical use of default methods in interfaces is **to incrementally provide additional functionality to a given type without breaking down the implementing classes.**
* In addition, they can be used to **provide additional functionality around an existing abstract method**:

public interface Vehicle {

    // additional interface methods

    double getSpeed();

    default double getSpeedInKMH(double speed) {

       // conversion

    }

}

## 3.3 Multiple Interface Inheritance Rules

* Default interface methods are a pretty nice feature indeed, but with some caveats worth mentioning. Since Java allows classes to implement multiple interfaces, it's important to know **what happens when a class implements several interfaces that define the same***default***methods**.
* To better understand this scenario, let's define a new*Alarm* interface and refactor the *Car* class:

public interface Alarm {

    default String turnAlarmOn() {

        return "Turning the alarm on.";

    }

    default String turnAlarmOff() {

        return "Turning the alarm off.";

    }

}

* With this new interface defining its own set of *default* methods, the *Car* class would implement both *Vehicle* and *Alarm*:

public class Car implements Vehicle, Alarm {

    // ...

}

* In this case, **the code simply won't compile, as there's a conflict caused by multiple interface inheritance** (a.k.a the [Diamond Problem](https://en.wikipedia.org/wiki/Multiple_inheritance)). The *Car* class would inherit both sets of *default* methods. Which ones should be called then?
* **To solve this ambiguity, we must explicitly provide an implementation for the methods:**

@Override

public String turnAlarmOn() {

    // custom implementation

}

@Override

public String turnAlarmOff() {

    // custom implementation

}

* We can also have our class use the default methods of one of the interfaces.
* Let's see an example that uses the default methods from the Vehicle interface:

@Override

public String turnAlarmOn() {

    return Vehicle.super.turnAlarmOn();

}

@Override

public String turnAlarmOff() {

    return Vehicle.super.turnAlarmOff();

}

* Similarly, we can have the class use the *default* methods defined within the *Alarm* interface:

@Override

public String turnAlarmOn() {

    return Alarm.super.turnAlarmOn();

}

@Override

public String turnAlarmOff() {

    return Alarm.super.turnAlarmOff();

}

* Furthermore, it's even possible to make the Car class use both sets of default methods:

@Override

public String turnAlarmOn() {

    return Vehicle.super.turnAlarmOn() + " " + Alarm.super.turnAlarmOn();

}

@Override

public String turnAlarmOff() {

    return Vehicle.super.turnAlarmOff() + " " + Alarm.super.turnAlarmOff();

}

## 3.4 Static Interface Methods

* Aside from being able to declare default methods in interfaces, Java 8 allows us to define and implement static methods in interfaces.
* Since static methods don't belong to a particular object, they are not part of the API of the classes implementing the interface, and they have to be called by using the interface name preceding the method name.
* To understand how static methods work in interfaces, let's refactor the Vehicle interface and add to it a static utility method:

public interface Vehicle {

    // regular / default interface methods

    static int getHorsePower(int rpm, int torque) {

        return (rpm \* torque) / 5252;

    }

}

* **Defining a***static***method within an interface is identical to defining one in a class.** Moreover, a *static* method can be invoked within other *static* and *default* methods.
* Now, say that we want to calculate the [horsepower](https://en.wikipedia.org/wiki/Horsepower) of a given vehicle's engine. We just call the *getHorsePower()* method:

|  |  |
| --- | --- |
|  | Vehicle.getHorsePower(2500, 480)); |

* The idea behind *static* interface methods is to provide a simple mechanism that allows us to **increase the degree of**[cohesion](https://en.wikipedia.org/wiki/Cohesion_(computer_science)) of a design by putting together related methods in one single place without having to create an object.
* Pretty much **the same can be done with abstract classes.** The main difference lies in the fact that **abstract classes can have constructors, state, and behavior**.