**Java 8 tutorial :**

you'll learn how to use default interface methods, lambda expressions, method references and repeatable annotations.

you'll be familiar with the most recent [API](http://download.java.net/jdk8/docs/api/) changes like streams, functional interfaces, map extensions and the new Date API.

**What are new features which got introduced in Java 8?**

There are lots of new features which were added in Java 8. Here is the list of important features:

> Lambda Expression

> Stream API

> Default methods in the interface

> Functional Interface

> Optional

> Method references

> Date API

> Nashorn, JavaScript Engine

**What are main advantages of using Java 8?**

> More compact code

> Less boiler plate code

> More readable and reusable code

> More testable code

> Parallel operations

**Lambda Expressions in Java 8**

Java 8 has introduced a new feature called Lambda expressions. It is considered to be a major change in java. As this change will bring functional programming into Java. Other languages such as Scala already have this feature so this is not new to programming world, it is new to java.

Before understanding Lambda expressions, Lets first understand **Functional Interface.**

**What is Functional Interface?**

Functional interfaces are those interfaces that have only one abstract method in it. It can have more than one default or static method and can override the method from java.lang.object.

Let’s create a functional interface:

@FunctionalInterface

public interface Decorable {

// one abstract method

void decorateWithCurtains();

// default method

default void decorateWithPaints() {

System.out.println("Decorating using paints");

}

// Overriding method of java.lang.Object

@Override

public int hashCode();

}

Java can itself identify Functional Interface but you can also denote interface as Functional Interface by annotating it with @FunctionalInterface.

Some popular Functional Interfaces are:

> java.lang.Runnable

> java.util.concurrent.Callable

> java.awt.event.ActionListener

> java.util.Comparator

**Lambda expression** represents an anonymous function. It comprises of a set of parameters, a lambda operator (->) and a function body . You can call it function without name,

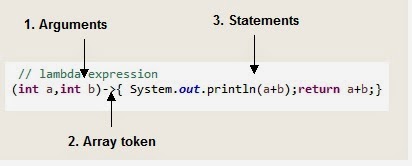
The connection between Lambda Expression and Functional Interface:

You might be thinking I have introduced the functional Interface above but how it is connected to Lambda. So Lambda expression can be applied for the abstract method of functional Interface which is being implemented or being instantiated anonymously.

**Structure of Lambda Expressions**

(Argument List) ->{expression;} or

(Argument List) ->{statements;}



1. **Argument list or parameters**

> Lambda expression can have zero or more arguments.

()->{System.out.println(“Hello”)}; //Without argument, will print hello

(int a)->{System.out.println(a)}; // One argument, will print value of a

(int a,int b)-> {a+b};//two argument, will return sum of these two integers

> You can choose to not declare the type of arguments as it can be inferred from context.

(a,b)->{a+b}; // two argument, will return sum of these two numbers

> you can not declare one argument’s type and do not declare type for other argument.

(int a,b)->{a+b}; // Compilation error

> When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses.

a->{System.out.println(a)}; // Will print value of number a

1. **Array token (->)**
2. **Body**

Body can have expression or statements.

If there is only one statement in body,curly brace is not needed and return type of the anonymous function is same as of body expression

If there are more than one statements, then it should be in curly braces and return type of anonymous function is same as value return from code block, void if nothing is returned.

*// old way*

**new** Thread(**new** Runnable() {

   @Override

   public **void** run() {

    System.out.println("Thread is started");

   }

  }).start();

*// using lambda Expression*

**new** Thread(()->System.out.println("Thread is started")).start();

}

**Code to sort list of movies by name using comparator**

Old way :

List<Movie> listOfMovies = **new** ArrayList<>();

    listOfMovies.add(m1);

    listOfMovies.add(m2);

    listOfMovies.add(m3);

    listOfMovies.add(m4);

    System.out.println("Before Sort by name : ");

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

      Movie movie = (Movie) listOfMovies.get(i);

      System.out.println(movie);

    }

*// Sort by movieName*

*// Anonymous Comparator*

*// old way*

    Collections.sort(listOfMovies, **new** Comparator<Movie>() {

      @Override

      public **int** compare(Movie o1, Movie o2) {

**return** o1.getMovieName().compareTo(o2.getMovieName());

      }

    });

**Lambda way :**

The problem with Anonymous Comparator is of syntax. Each time you want to sort the list using a comparator, you have to remember the bulky syntax.

So generally the main problem with Anonymous classes is syntax. For very simple operation, we need to write complex code. To solve this problem, JDK has introduced a new feature called Lambda Expressions. how lambda expression will reduce this complex code.

*// Sort by movieName*

*// Anonymous Comparator*

*// old way*

    Collections.sort(listOfMovies, **new** Comparator<Movie>() {

      @Override

      public **int** compare(Movie o1, Movie o2) {

**return** o1.getMovieName().compareTo(o2.getMovieName());

      }

    });

*// Using lambda expression*

    Collections.sort(listOfMovies, (o1, o2) -> o1.getMovieName().compareTo(o2.getMovieName()));

    System.out.println("After Sort by name: ");

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

      Movie movie = (Movie) listOfMovies.get(i);

      System.out.println(movie);

    }

  }

for using Comparator. So in spite of writing Anonymous comparator, our expression became very easy.



So we have passed 2 arguments o1 and o2, we didn’t pass type because it can be inferred from context.

We have only one statement here, so no need to put it in curly braces.

**HelloWorld Lambda Expression Example**

   public **interface** HelloWorld {

**void** sayHello();}

public **class** HelloWorldMain {

public static **void** main(**String** args[])

{

*// Lambda Expression*

     HelloWorld helloWorld=()->System.out.println("Hello using Lambda Expression");

    helloWorld.sayHello();

}

}

Java lambda expression is consisted of three components.

**1) Argument-list:** It can be empty or non-empty as well.

**2) Arrow-token:** It is used to link arguments-list and body of expression.

**3) Body:** It contains expressions and statements for lambda expression.

**Without Lambda Expression**

1. **interface** Drawable{
2. **public** **void** draw();
3. }
4. **public** **class** LambdaExpressionExample {
5. **public** **static** **void** main(String[] args) {
6. **int** width=10;
8. //without lambda, Drawable implementation using anonymous class
9. Drawable d=**new** Drawable(){
10. **public** **void** draw(){System.out.println("Drawing "+width);}
11. };
12. d.draw();
13. }
14. }

**With Lambda expression**

1. @FunctionalInterface  //It is optional
2. **interface** Drawable{
3. **public** **void** draw();
4. }
5. **public** **class** LambdaExpressionExample2 {
6. **public** **static** **void** main(String[] args) {
7. **int** width=10;
8. //with lambda
9. Drawable d2=()->{
10. System.out.println("Drawing "+width);
11. };
12. d2.draw();
13. }  }
14. **interface** Sayable{
15. **public** String say();
16. }
17. **public** **class** LambdaExpressionExample3{
18. **public** **static** **void** main(String[] args) {
19. Sayable s=()->{
20. **return** "I have nothing to say.";
21. };
22. System.out.println(s.say());
23. }
24. }
25. **interface** Addable{
26. **int** add(**int** a,**int** b);
27. }
29. **public** **class** LambdaExpressionExample5{
30. **public** **static** **void** main(String[] args) {
32. // Multiple parameters in lambda expression
33. Addable ad1=(a,b)->(a+b);
34. System.out.println(ad1.add(10,20));
36. // Multiple parameters with data type in lambda expression
37. Addable ad2=(**int** a,**int** b)->(a+b);
38. System.out.println(ad2.add(100,200));
39. }
40. }
41. **interface** Addable{
42. **int** add(**int** a,**int** b);
43. }
45. **public** **class** LambdaExpressionExample6 {
46. **public** **static** **void** main(String[] args) {
48. // Lambda expression without return keyword.
49. Addable ad1=(a,b)->(a+b);
50. System.out.println(ad1.add(10,20));
52. // Lambda expression with return keyword.
53. Addable ad2=(**int** a,**int** b)->{
54. **return** (a+b);
55. };
56. System.out.println(ad2.add(100,200));
57. }
58. }

**Java Lambda Expression Example: Foreach Loop**

1. list.forEach(
2. (n)->System.out.println(n)
3. );

**Multiple Statements**

**Creating Thread**

1. //Thread Example with lambda
2. Runnable r2=()->{
3. System.out.println("Thread2 is running...");
4. };

**Comparator**

1. // implementing lambda expression
2. Collections.sort(list,(p1,p2)->{
3. **return** p1.name.compareTo(p2.name);
4. });

**Filter Collection Data**

1. // using lambda to filter data
2. Stream<Product> filtered\_data = list.stream().filter(p -> p.price > 20000);
3. // using lambda to iterate through collection
4. filtered\_data.forEach(
5. product -> System.out.println(product.name+": "+product.price)
6. );

**Event Listener**

  b.addActionListener(e-> {tf.setText("hello swing");});

**Important points:**

> The body of a lambda expression can contain zero, one or more statements.

> When there is a 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 are more than one statements, then these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.

**Java 8 functional interface example**

Functional interfaces are those interfaces which have only one abstract method, it can have default methods, static methods and it can also override java.lang.Object class method.  
There are many functional interfaces already present.  
For example: Runnable , Comparable.  
You can implement functional interfaces using [lambda expressions](https://www.java2blog.com/2014/06/lambda-expressions-in-java-8.html" \t "https://java2blog.com/java-8-functional-interface-example/_blank).

*// Using lambda expression*

Thread t1=**new** Thread(

()->System.out.println("In Run method")

);

Example of functional interface,

@FunctionalInterface

public **interface** Decorable {

*// one abstract method*

**void** decorateWithCurtains();

*// default method*

**default** **void** decorateWithPaints()

{

  System.out.println("Decorating using paints");

}

*// Overriding method of java.lang.Object*

@Override

public **int** hashCode();

}

Java can itself identify Functional Interface but you can also denote interface as Functional Interface by annotating it with @FunctionalInterface. If you annotate @FunctionalInterface, you should have only one abstract method otherwise you will get compilation error.

public **class** DecorableMain {

public static **void** main(**String**[] args) {

*// Using lambada expression*

Decorable dec=()->{System.out.println("Decorating with curtains");};

dec.decorateWithCurtains();

}

}

**How lambda expression and functional interfaces are related?**

Lambda expressions can only be applied to abstract method of functional interface.

Runnable has only one abstract method called run, so it can be used as below:

*// Using lambda expression*

Thread t1=**new** Thread(

()->System.out.println("In Run method")

);

Here we are using Thread constructor which takes Runnable as parameter. As you can see we did not specify any function name here, as Runnable has only one abstract method, java will implicitly create anonymous Runnable and execute run method.  
It will be as good as below code.

Thread t1=**new** Thread(**new** Runnable() {

   @Override

   public **void** run() {

    System.out.println("In Run method");

   }

  });

**Can you create your own functional interface?**

Yes, you can create your own functional interface. Java can implicitly identify functional interface but you can also annotate it with @FunctionalInterface.

**Java Method References**

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

## **Types of Method References**

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.

**Example : Reference to a constructor**

1. InstanceMethodReference methodReference = **new** InstanceMethodReference(); // Creating object
2. // Referring non-static method using reference
3. Sayable sayable = methodReference::saySomething;
4. // Calling interface method
5. sayable.say();
6. // Referring non-static method using anonymous object
7. Sayable sayable2 = **new** InstanceMethodReference()::saySomething; // You can use anonymous object also
8. // Calling interface method
9. sayable2.say();

**Reference to a constructor.**

1. Messageable hello = Message::**new**;
2. hello.getMessage("Hello");

**What is Optional? Why and how can you use it?**

Java 8 has introduced new class Called Optional. This class is basically introduced to avoid NullPointerException in java.  
Optional class encapsulates optional value which is either present or not.  
It is a wrapper around object and can be use to avoid NullPointerExceptions.

In Java 8, we have a newly introduced Optional class in java.util package. This class is introduced to avoid NullPointerException that we frequently encounters if we do not perform null checks in our code. Using this class we can easily check whether a variable has null value or not and by doing this we can avoid the NullPointerException. In this guide, we will see how to work with Optional class and the usage of various methods of this class.

public class Example {

public static void main(String[] args) {

  String[] str = new String[10];

Optional<String> isNull = Optional.ofNullable(str[9]);

  if(isNull.isPresent()){

 //Getting the substring

 String str2 = str[9].substring(2, 5);

 //Displaying substring

System.out.print("Substring is: "+ str2);

  }

  else{

  System.out.println("Cannot get the substring from an empty string");

  }

str[9] = "AgraIsCool";

Optional<String> isNull2 = Optional.ofNullable(str[9]);

  if(isNull2.isPresent()){

 //Getting the substring

String str2 = str[9].substring(2, 5);

//Displaying substring

System.out.print("Substring is: "+ str2);

}

else{

System.out.println("Cannot get the substring from an empty string");

}

}  }

public class Example {

public static void main(String[] args) {

//Creating Optional object from a String

Optional<String> GOT = Optional.of("Game of Thrones");

//Optional.empty() creates an empty Optional object

Optional<String> nothing = Optional.empty();

System.out.println(GOT.map(String::toLowerCase));

System.out.println(nothing.map(String::toLowerCase));

Optional<Optional<String>> anotherOptional = Optional.of(Optional.of("BreakingBad"));

System.out.println("Value of Optional object"+anotherOptional);

System.out.println("Optional.map: "

+anotherOptional.map(gender -> gender.map(String::toUpperCase)));

//Optional<Optional<String>>    -> flatMap -> Optional<String>

System.out.println("Optional.flatMap: "

 +anotherOptional.flatMap(gender -> gender.map(String::toUpperCase)));

}}

public class Example {

public static void main(String[] args) {

//Creating Optional object from a String

Optional<String> GOT = Optional.of("Game of Thrones");

  /\* Filter returns an empty Optional instance if the output doesn't

\* contain any value, else it returns the Optional object of the

\* given value.

\*/

System.out.println(GOT.filter(s -> s.equals("GAME OF THRONES")));

System.out.println(GOT.filter(s -> s.equalsIgnoreCase("GAME OF THRONES")));

}}

import java.util.Optional;

  public class Example {

public static void main(String[] args) {

//Creating Optional object from a String

Optional<String> GOT = Optional.of("Game of Thrones");

//Optional.empty() creates an empty Optional object

Optional<String> nothing = Optional.empty();

//orElse() method

System.out.println(GOT.orElse("Default Value"));

System.out.println(nothing.orElse("Default Value"));

//orElseGet() method

System.out.println(GOT.orElseGet(() -> "Default Value"));

System.out.println(nothing.orElseGet(() -> "Default Value"));

}}

import java.util.Optional;

  public class Example {

public static void main(String[] args) {

//Creating Optional object from a String

Optional<String> GOT = Optional.of("Game of Thrones");

//Optional.empty() creates an empty Optional object

Optional<String> nothing = Optional.empty();

/\* isPresent() method: Checks whether the given Optional

\* Object is empty or not.

\*/

if (GOT.isPresent()) {

  System.out.println("Watching Game of Thrones");

}

else {

System.out.println("I am getting Bored");

  }

/\* ifPresent() method: It executes only if the given Optional

\* object is non-empty.

\*/

//This will print as the GOT is non-empty

GOT.ifPresent(s -> System.out.println("Watching GOT is fun!"));

//This will not print as the nothing is empty

nothing.ifPresent(s -> System.out.println("I prefer getting bored"));

}}

public class Example {

public static void main(String[] args) {

  String[] str = new String[10];

Optional<String> isNull = Optional.ofNullable(str[9]);

  if(isNull.isPresent()){

 //Getting the substring

 String str2 = str[9].substring(2, 5);

 //Displaying substring

System.out.print("Substring is: "+ str2);

  }

  else{

  System.out.println("Cannot get the substring from an empty string");

  }

str[9] = "AgraIsCool";

Optional<String> isNull2 = Optional.ofNullable(str[9]);

  if(isNull2.isPresent()){

 //Getting the substring

String str2 = str[9].substring(2, 5);

//Displaying substring

System.out.print("Substring is: "+ str2);

}

else{

System.out.println("Cannot get the substring from an empty string");

}

}  }

**What are defaults methods?**

Default method are those methods in interface which have body and use default keywords.

Default method are introduced in Java 8 mainly because of backward compatibility.