**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,

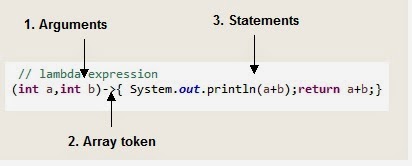
The connection between Lambda Expression and Functional Interface:

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();

}

}