Lambda Expression was first used in LISP programming language.

Benefits of Lambda Expressions

1. To Enable Functional programming in Java
2. Write more readable, maintainable and concise code
3. To use API's, very easily and effectively
4. To enable parallel programming

Lecture 3:

Lambda Expression is an Anonymous function.

Anonymous function:

* Not having a name
* Not having modifiers
* Not having a return type:

To print hello world:

//to print hellp world

**public** **void** printHelloWorld() {

System.***out***.println("Hello World");

}

Since we are not having a name

**public** **void** ~~printHelloWorld~~() {

System.***out***.println("Hello World");

}

Not having any modifiers

**~~public~~** **void** ~~printHelloWorld~~() {

System.***out***.println("Hello World");

}

Not having any return type

**~~public~~** **~~void~~** ~~printHelloWorld~~() {

System.***out***.println("Hello World");

}

The above resulting expression is lambda expression:

() -> {

System.***out***.println("Hello World");

}

Eg2: To print sum of 2 numbers.

//to print the sum of two numbers

**public** **void** addTwoNumbers(**int** num1, **int** num2) {

System.***out***.println("The sum of two numbers are "+(num1+num2));

}

Since we are not having a name

**public** **void** ~~addTwoNumbers~~(**int** num1, **int** num2) {

System.***out***.println("The sum of two numbers are "+(num1+num2));

}

Not having any modifiers

**~~public~~** **void** ~~addTwoNumbers~~(**int** num1, **int** num2) {

System.***out***.println("The sum of two numbers are "+(num1+num2));

}

Not having any return type

**~~public~~** **~~void~~** ~~addTwoNumbers~~(**int** num1, **int** num2) {

System.***out***.println("The sum of two numbers are "+(num1+num2));

}

The above resulting expression is lambda expression:

(**int** num1, **int** num2) -> {

System.***out***.println("The sum of two numbers are "+(num1+num2));

}

Example 3: Similarly to print the length of String.

//to print return the length of a string

**public** **int** getLength(String string) {

**return** string.length();

}

Lambda Experssion:

//to print return the length of a string

(String string) -> {

**return** string.length();

}

Lecture 4:

//to print hellp world

**public** **void** printHelloWorld() {

System.***out***.println("Hello World");

}

For this example the lambda expression is

//to print hellp world

()-> {

System.***out***.println("Hello World");

}

Simplified since its having only one statement we can ignore “{”

//to print hellp world

()-> System.***out***.println("Hello World");

Another Lambda Expression:

//to print the sum of two numbers

**public** **void** addTwoNumbers(**int** num1, **int** num2) {

System.***out***.println("The sum of two numbers are "+(num1+num2));

}

Lambda Expression:

Since we know that there is only one statement we can remove the “{”

//to print the sum of two numbers

(**int** num1, **int** num2) ->

System.***out***.println("The sum of two numbers are "+(num1+num2));

The types can be guessed by the compiler, this guessing is done through type inference. Hence we can remove the data types:

(num1, num2) ->

System.***out***.println("The sum of two numbers are "+(num1+num2));

One more Lambda Expression:

//to print return the length of a string

**public** **int** getLength(String string) {

**return** string.length();

}

From the first example:

(String string) -> **return** string.length();

From second example

(string) -> **return** string.length();

We are not required to write return statement explicitly, hence the expression will look like the below.

(string) -> string.length();

Since we are having only one input value we can even ignore the parentheses “(”

string -> string.length();

Note if there is no value then () must be present:

Eg:

()-> System.***out***.println("Hello World");

Lecture 5:

Functional Interface:

* Will have only one abstract method.
  + Eg: Runnable Contains only run() method
  + Callable Contains only call() method
  + ActionListener Contains only actionPerformed()
  + Comparable Contains only compareTo()

The restriction for Functional interface is only for abstract method not for any other method types. i.e.. not for any Default or static methods.

Not Fucntional Interface:

**interface** Interf2{

**public** **void** m1();

**public** **void** m2();

}

Functional Interface:

**interface** Intrf1 {

**public** **void** m1();

**public** **static** **void** m2() {

}

**default** **void** m3() {

}

}

Lecture 6:

**@FunctionalInterface** Indicates explicitly that the below interface is functional interface. This information is for the compiler and it will check if we are making any errors.

Example of **@FunctionalInterface**

Eg:

@FunctionalInterface

**interface** Intrf1 {

**public** **void** m1();

**public** **static** **void** m2() {

}

**default** **void** m3() {

}

}

//Not functional Interface

@FunctionalInterface

**interface** Interf2{

Error Thrown in Eclipse

**public** **void** m1();

**public** **void** m2(); Invalid '@FunctionalInterface' annotation; Interf2 is not a

functional interface

Explanation:

Must have only one abstract method

Compilation error:

unexpected @FunctionalInterface annotation:

Reason: Multiple non-overriding abstract methods found in Interface Interf2

}

//Not functional Interface

@FunctionalInterface

**interface** Interf3{

Error Thrown in Eclipse

Invalid '@FunctionalInterface' annotation; Interf3 is not a

functional interface

}

Explanation:

It must have atleast one abstract interface.

Compilation error:

unexpected @FunctionalInterface annotation:

Reason: No Abstract method found in interface Interf2

Lecture 7: Functional Interface with respect to Inheritance:

Case1: If an interface extends Functional Interface and child interface does not contain any abstract method, then child Interface is always Functional Interface.

Eg:

@FunctionalInterface

**interface** One{

**public** **void** m1();

}

@FunctionalInterface

**interface** OnesChild **extends** One{

}

Case2: If a child interface is defining exactly the same parent interface abstract method. Then the child interface is always Functional Interface.

@FunctionalInterface

**interface** Two{

**public** **void** m1();

}

@FunctionalInterface

**interface** TwosChild **extends** One{

**public** **void** m1();

}

Case3: If a child Interface is defining a new abstract method. Then the child interface is not Functional Interface.

@FunctionalInterface

**interface** Three{

**public** **void** m1();

}

@FunctionalInterface

**interface** ThreesChild **extends** One{

**public** **void** m2();

}

Compilation Error: unexpected @FunctionalInterface Annotation,

Multiple non-overriding abstract methods found in interface C.

Case4: If a child Interface is defining a new abstract method. Then the child interface is not Functional Interface.

@FunctionalInterface

**interface** Four{

**public** **void** m1();

}

**interface** FoursChild **extends** One{

**public** **void** m2();

}

Note: Perfectly valid, as there is not @FunctionalInterface method

Lecture 8: Example of Functional Interface

**public** **class** Example1 {

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

/// Old way of writing

Interf demo1 = **new** Demo();

demo1.m1();

// Using Lambda Expression

Interf demo2 = ()-> System.***out***.println(" Lambda Expression");

demo2.m1();

}

}

// Old way of writing

**interface** Interf{

**public** **void** m1();

}

**class** Demo **implements** Interf{

@Override

**public** **void** m1() {

// **TODO** Auto-generated method stub

System.***out***.println("m1() method Implementation");

}

}

Lecture 9: Example of Functional Interface:

Example1:

**public** **class** Example1 {

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

Interf interf1 = **new** Demo();

interf1.add(10, 20);

interf1.add(100, 200);

//Implmentation by Lambda Expresssion

Interf interf2 = (a, b)-> System.***out***.println("The Sum is "+( a+ b));

interf2.add(10, 20);

interf2.add(100, 200);

}

}

**interface** Interf{

**public** **void** add(**int** a, **int** b);

}

**class** Demo **implements** Interf{

@Override

**public** **void** add(**int** a, **int** b) {

// **TODO** Auto-generated method stub

System.***out***.println("The Sum is "+(a+b));

}

}

Example 2:

**public** **class** Example2 {

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

//With out Lambda Expression

Interface interf1 = **new** Demo1();

System.***out***.println(interf1.getLength("hello "));

System.***out***.println(interf1.getLength("Without Lambda Expression"));

//With Lambda Expression

Interface interf2 = s -> s.length();

System.***out***.println(interf2.getLength("hello "));

System.***out***.println(interf2.getLength("Without Lambda Expression"));

}

}

**interface** Interface {

**public** **int** getLength(String string);

}

**class** Demo1 **implements** Interface {

@Override

**public** **int** getLength(String string) {

// **TODO** Auto-generated method stub

**return** string.length();

}

}

Lecture 10: Example of Functional Interface: for Runnable Interface:

Without Lambda Expression:

**package** com.durgaSoft.section2.lecture9;

**public** **class** Example4 {

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

Thread thread1 = **new** Thread(**new** Myrunnable());

thread1.start();

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Main Thread " + i);

}

}

}

**class** Myrunnable **implements** Runnable {

@Override

**public** **void** run() {

// **TODO** Auto-generated method stub

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Child Thread" + i);

}

}

}

Using Lambda Expression:

**public** **class** Example5 {

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

Runnable r = () -> {

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Child Tread" + i);

}

};

Thread thread = **new** Thread (r);

thread.start();

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Main Thread " + i);

}

}

}

Lecture 11: Summary:

Functional Expresssion:

* It should be contain exactly one abstract method (SAM- Single Abstract Method)
* It can contain any number of default and static methods
* It acts a type for lambda expressions
* It can be used to invoke lambda expressions.

Case1: Why should Functional Interface contain only one abstract method.

**interface** Interf{

**public** **void** m1();

}

**public** **class** Example1 {

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

Interf demo1 = ()-> System.***out***.println(" Lambda Expression");

Demo1.m1();

}

}

In there the lambda expression demo1 is mapped to m1 method.

Assume if the interface is having two abstract method.

**interface** Interf{

**public** **void** m1();

**public** **void** m2();

}

**public** **class** Example1 {

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

Interf demo1 = ()-> System.***out***.println(" Lambda Expression");

Demo1.m1();

}

}

In this case, the compile does not know which interface to use and will throw the below error during compilation:

Incompatible types :Interf is not a functional Interface

Multiple Non-Overriding abstract methods in interface Interf.

Case2: What is the advantage of @FunctionalInterface Annotaiton.

Assume the below example is implemented.

**interface** Interf{

**public** **void** m1();

}

**public** **class** Example1 {

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

Interf demo1 = ()-> System.***out***.println(" Lambda Expression");

Demo1.m1();

}

}

And overtime if another programmer adds a new abstract method to the Interface it will lead to code break. Hence to avoid this we can use Functional Interface.