

Lambda Expressions

- Java 1.8 onwards is Objective.
- To bring the functional programming features in Java

#

OOP

Data will be stored in the form of classes and objects and the processing is also occurring through objects.

functional programming

Data will be stored in the form of functions and variables.

- Code optimization (we can reduce the code) / simplifying the code

What is lambda expression?

It is an anonymous function (function without name, return type and no ~~return value~~ access modifier)

```
public void main() { System.out.println("Hello World"); }
```

#

This is the simple example of a simple method we can write in Java

$() \rightarrow \{ \text{System.out.println("Hello world")} \}$

$() \rightarrow \text{System.out.println("Hello world");}$

- In the lambda expressions we don't need to specify the type of the variables, it's because java compiler will automatically find the data type of the variables based on the context -

$(a, b) \rightarrow \text{sop}(a+b)$

- $(a) \rightarrow \{ \text{return}(a*a); \}$; if we want to use return statement use the curly brackets

- $(a) \rightarrow a*a;$

- $a \rightarrow a*a;$ if we have a single argument we can also avoid the parenthesis

Functional Interface ...

How to invoke lambda expressions in java, we have to know the functional interface concepts

Interface is the class which contains only abstract methods.
(methods only having definition not implementations)

Functional Interface

If any interface contains only one abstract method
it is called a Functional interface (SAM)

#

Abstract methods
Default methods
Static methods

• one abstract method
• default methods
• static methods

#

by default

Functional Interface in java which are already available in java

#

Runnable → run()

Callable → call()

Comparable → compareTo()

ActionListener → actionPerformed()

* without using functional interface we cannot use the lambda expressions.

In order to create an object of functional interface or any interface we have to create a class which implements the interface and it is called concrete class and we can create the object of the interface by creating the object of the concrete class.

So whenever there is a functional interface, we can implement it by using lambda expression.
So lambda expressions are always associated with the functional interface.

PRACTICALS ...

"We can't write lambda expression without a Functional Interface"

without using functional interfaces also we can still write lambda expressions and invoke them

...
This is possible with the help of predefined functional interfaces

Java.util. Function

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

Predicate (I)

It is one of the predefined functional interfaces available in Java, and it is found in Java.util.function package. All the predefined functional interfaces, we can access from the same package.

- method available in this functional interface is **test()** which will take some arguments and returns a boolean value

- Predicate interface is a choice if we have a conditional checks in our program and ~~then~~ we want to ~~write~~ ^{date} write a lambda expression for them.
- we can specify multiple conditions using Predicate interface

```
public class Employee {
```

```
    String name;  
    int salary;  
    int experience;
```

```
    public Employee(String String name, int salary, int experience)
```

```
    {  
        this.name = name;  
        this.salary = salary;  
        this.experience = experience;  
    }
```

```
}  
  
public class Demo {
```

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

Exp 1:

```
        Employee emp = new Employee("John", 2000, 2);
```

```
        Predicate<Employee> p1 = e -> (e.salary > 3000 && e.  
                                         experience > 3);
```

```
        System.out.println(p1.test(emp));
```

```
    }  
}
```


Why we need to go for lambda expression
Here we have two conditions and for every object we have to verify two conditions and some times we may have 10 conditions or 20 conditions. Instead of writing all the conditions in the if block if we can use a lambda expression and write our put these conditions there, then and we just call the lambda expression each time.

{ we can also form predicates using and, or, negate }

$P_1 \text{ and } P_2 \cdot \text{test}(n)$ ($P_1 \cdot \text{test}(n)$ & $P_2 \cdot \text{test}(n)$)

$P_1 \text{ or } P_2 \cdot \text{test}(n)$ ($P_1 \cdot \text{test}(n)$ || $P_2 \cdot \text{test}(n)$)

$P_1 \cdot \text{negate}() \cdot \text{test}(n)$ (we will get the answer just opposite to the condition we are giving)

PRACTICALS

```
21 public class Demo2 {
22
23     public static void main(String[] args) {
24
25         //Ex1:
26         Employee emp=new Employee("John", 20000, 2);
27
28         // emp obj---> return name if sal>50K exp>3
29         Predicate<Employee> pr= e->(e.salary>30000 && e.experience>3);
30         System.out.println(pr.test(emp));
31
32         //Ex2:
33         ArrayList<Employee> al=new ArrayList<Employee>();
34         al.add(new Employee("John", 50000, 5));
35         al.add(new Employee("David", 20000, 2));
36         al.add(new Employee("Scott", 30000, 3));
37         al.add(new Employee("Scott", 40000, 6));
38
39         for(Employee e:al)
40         {
41             if(pr.test(e))
42             {
43                 System.out.println(e.ename+" "+e.salary);
44             }
45         }
46
47
48
```




```
7 // p2 -- checks greater than 50
8
9
10 public class Demo3 {
11
12     public static void main(String[] args) {
13
14         int a[] = {5,15,20,25,30,35,40,45,50,55,60,65};
15
16         Predicate<Integer> p1 = i -> i%2==0;
17         Predicate<Integer> p2 = i -> i>50;
18
19         //and
20
21         for(int n:a)
22         {
23             //if(p1.test(n) && p2.test(n))
24
25             if(p1.and(p2).test(n))
26             {
27                 System.out.println(n);
28             }
29         }
30
31
32
33 }
```



Function

Date : _____

- It contains a method called apply(). It takes any type of parameter and will return a single value of any type.
- Here the apply() function will take argument of any type we also have to specify the return type. Then the apply method will do some processes and return some value.

Function ^(return type) <Integer, Integer> f = n → n * n;
f.apply(5)
(parameter type) (return type)

We can also use functions and predicate combination in one programme.

{ We can achieve the function chaining by
andThen()
compose() }

Function <Integer, Integer> f1 = n → n * 2;
Function <Integer, Integer> f2 = n → n * n * n;
System.out.println (f1.andThen(f2).apply(2));
(result will be pass to f2)
(54)
2 will pass to f1

$f_1 \cdot \text{compose}(f_2) \cdot \text{apply}(2)$
 (result of f_2 will go to f_1)
 (it will go to f_2)

function

{16}

{ when we need to go for a function, we want to provide something and get the result }

predicate \rightarrow Param Type \rightarrow boolean
 test()

function \rightarrow Parameter Type, Return Type \rightarrow Some type
 apply()

PRACTICALS.....


```
24 emplist.add(new Employee("David",50000));
25 emplist.add(new Employee("John",30000));
26 emplist.add(new Employee("Mary",20000));
27
28 Function<Employee,Integer> fn=e->{
29     int sal=e.salary;
30
31     if(sal>=10000 && sal<=20000)
32         return (sal *10/100);
33     else if(sal>20000 && sal<=30000)
34         return (sal *20/100);
35     else if(sal>30000 && sal<=50000)
36         return (sal *30/100);
37     else
38         return (sal*40/100);
39 };
40
41 for(Employee emp:emplist)
42 {
43     int bonus=fn.apply(emp);
44     System.out.println(emp.ename+" "+emp.salary);
45     System.out.println("Bonus is:"+bonus);
46 }
47
48
49 }
50
```



Consumer .. (means it will consume)

Date : _____

- It contains a method called accept(). It will take ~~one~~ a single parameter as the input but it doesn't return anything for us. It means it just consumes the input, doesn't return anything.

Supplier ..

- It contains a method called get() and this method will not take any parameter, but it will supply some value. So the supplier means it will supply something, but it doesn't take any input parameter.

(~) Supplier<Date> S = () -> new Date()

We can also achieve consumer chaining using andThen()


```
29      emplist.add(new Employee("John",30000,"Male"));
30      emplist.add(new Employee("Mary",20000,"FeMale"));
31      emplist.add(new Employee("Scott",60000,"Male"));
32
33      //Function
34      Function<Employee,Integer> f=emp->(emp.salary*10)/100; // task1
35
36      //Predicate
37      Predicate <Integer> p=b->b>=5000; //task2
38
39      //Consumer
40      Consumer <Employee> c=emp->{
41          System.out.println(emp.ename);
42          System.out.println(emp.salary);
43          System.out.println(emp.gender);
44      }; //task3
45
46      for(Employee e:emplist)
47      {
48          int bonus=f.apply(e);
49
50          if(p.test(bonus))
51          {
52              c.accept(e);
53          }
54
55      }
56
```



```
1 package consumers;
2
3 import java.util.function.Consumer;
4
5 //Consumer chaining..
6 public class Demo3 {
7
8     public static void main(String[] args) {
9
10         Consumer <String> c1=s->System.out.println(s+" is White");
11         Consumer <String> c2=s->System.out.println(s+" is having four legs");
12         Consumer <String> c3=s->System.out.println(s+" is eating grass");
13
14         /*c1.accept("Cow");
15         c2.accept("Cow");
16         c3.accept("Cow");*/
17
18         //or
19
20         //c1.andThen(c2).andThen(c3).accept("Cow");
21
22         //or
23         Consumer <String> c4=c1.andThen(c2).andThen(c3);
24         c4.accept("Cow");
25     }
26
27 }
28
```

