LAMBDA EXPRESSIONS (JDK -1.8 feature)

- It is a process of writing method logic in less lines of code.
- It reduces number of lines of code.
- It works faster than previous process.
- It saves memory and execution time by reducing creation of object(s).

----------**Example**----------

Method:

```
public int doSum(int x,iny y)

{

        return  x+y;

}
```

Expression:

```
Ob=public  int  doSum (int x,int y)

{

        return  x+y;

}
```

Now, remove **Access Modifier**, **Return type and Method Name** from **the Method Heading**,

Then Expression will be:

```
Ob=(int x, int y){

    return  x+y;

 }
```

Now add Lambda symbol **(->)** between (pameters) and {Block}.

Then Expression will be

```
Ob=(int x, int y)->{
   return  x+y;
}
```

Here parameter Data Types are optional, then Expression will be

```
Ob=( x, y)->{
   return x+y;
}
```

Now, if Block has only one statement then "{}" are optional, and also "return" keyword must not be written if no Blocks are given.

Hence, final Expression (**Lambda Expression**) will be

```
Ob=(x+Y)->x+y;
```

---------- **Sample Examples** ----------

#1

Method:

```
public int getCount(int p){
return P*2;
}
```

Lambda Expression:

```
Ob=(p)->p*2;
```

#2

Method:

```
Public voib show(){
System.out.println("Hi ");
}
```

Lambda Expression:

```
Ob=()->System.out.println("Hi ");
```

#3

Method:

```
public String get(int x,double y,String z){
    return   "Hi "+z+ ", "+(x-y);
}
```

Lambda Expression:

```
(x,y,z)-> "Hi "+z+ ", "+(x-y);
```

Functional Interface:

An interface which is having only one abstract method (**having multiple default methods and static methods) is called as Function Interface.

Before JDK 1.8 (JDK<=1.7), we used to define one implementation class and create object to that implementation class. Lambda Expression replaces Implementation class and creating object to that class.

----------**Example(JDK 1.7or Before)**----------

#1 Function Interface:

```
interface Process{
    String getProcess(int id)
}
```

#2 Implementation class:

```
class MyProc implements Process{
  public String getProcess(int id){
     return  "Id is: "+id;
   }
}
```

#3 Create Object :

```
Process p = new MyProc();
```

#4 Call Method:

```
String s = p.getProcess(55);
System.out.println(s);
```

Now, if we re-write this above code using **Lambda Expression** , it will replace steps **#2** and **#3**.

#1 Functional Interface:

```
interface Process{
  String getProcess(int x);
```

```
}
```

Steps#2 And #3

```
Process p=(id)-> "Id is: "+id;
```

#4 Call Method:

```
String s=p.getProcess(55);

System.out.println(s);
```

----------Examples Using Lambda Expression----------

Ex-1:

#1 Functional Interface:

```
interface Consumer{

        void print(Object ob);

    }
```

#2 Lambda Expression:

```
Consumer c=(ob)->System.out println(ob);
```

Ex-2:

#1 Functional Interface:

```
interface Consumer{

   boolean test(int id);

}
```

#2 Labmda Expression for above method as logic-

"if input (id)>=0 true else false "

Equivalent Lambda Expression (using if else control structure):

```
Consumer c=(id)->{
    If(id>=0)
      return true;
   else
      return false;
}
```

Equivalent Lambda Expression (using ternary operator):

```
Consumer c=(id)->id>=0?true:false
```

Ex-3:

#1 Functional Interface:

```
interface StringOpr{
   int findLen(Srting s);
}
```

#2 Lambda Expression for finding the input string length as logic:

```
StringOpr sob=(s)->s.length();
```

Using Generics with Labmda Expressions:

```
interface Product<T>{
   T  add(T  x, T  y);   }
```

Here , "T " is a Generic type, it means T=DataType will be decided at Runtime while creating the Lambda Expression for the above method "add() ".

T=Integer,Double,String,-------any class

Examlpes:

```
Product <String>p=(x, y)->x+y;

Product <Integer>p=(x, y)->x+y;

Product <Double>p=(x, y)->x+y;
```

Predefined Functional Interfaces:

To write Lambda Expressions, we need interfaces with one abstract method.

JDK-1.8 has all predefined functional interfaces in a package "**java.util.function** ".

Here, we need to choose one proper interface for our logic, based on **number of parameters**

and **return type** of the method.