Program on Functional interface :

```java
package com.ravi.functional_interface;

@FunctionalInterface
interface Animal
{
  void makeSound();
}

public class FunctionalInterfaceDemo
{
    public static void main(String[] args)
    {
        //Anonymous inner class
        Animal lion = new Animal()
        {
            @Override
            public void makeSound()
            {
                System.out.println("Lion is roaring");
            }
        };


            //Anonymous inner class
            Animal dog = new Animal()
            {
                @Override
                public void makeSound()
                {
                    System.out.println("Dog is barking");
                }
            };

            lion.makeSound();
            dog.makeSound();

    }

}
```

What is Lambda Expression in Java ?

----

* It is a new fetaure introduced from JDK 1.8V.

* It is an **anoymous function (Function/Method without any name)** which is used to write **concise coding.**

* It is an improvement over **anonymous inner class** where We **need not to** write the following while defining a method :

    a) Access modifier of the method is not required.
    b) Name of the Method is not required (Anonymous Function)
    c) Return type of the Method is not required.
    d) While defining the Lambda Variable, Data type is also not required.

Different Cases for Wriring Lambda Expression :

--

Case 1 :

Concrete Method :

```java
public void display()
{
  System.out.println("Display");
}
```

Converting the Concrete Method into Lambda Expression:

```java
() -> System.out.println("Display");
```

Case 2:

Concrete Method :

```java
public void doSum(int x, int y)
{
 System.out.println(x + y);
}
```

Converting the Concrete Method into Lambda Expression:

```java
(x, y) ->  System.out.println(x + y);
```

Case 3 :

Concrete Method

```java
public int getLength(String str)
{
   return str.length();
}
```

Convert concrete method into Lambda Expression :

```java
str -> str.length();
```

Anonymous inner class

```java
@FunctionalInterface
interface Printable
{
  void print();
}
public class Main
{
  public static void main(String ...x)
  {
    Printable p = new Printable()
    {
       @Override
       public void print()
       {
          System.out.println("Printing");
       }
    };
    p.print();
  }
}
```

Boiler-plate code

Anonynous Function(Lambda)

```java
@FunctionalInterface
interface Printable
{
  void print();
}
public class Main
{
  public static void main(String ...x)
  {
    Printable p = ()-> S.o.p("Printing");
    p.print();
  }
}
```

In java it is used to enable Functional Programming.

It is used to concise our code as well as we can remove boilerplate code.

Lambda will work only with functional interface.

If the body of the Lambda Expression contains only one statement then curly braces are optional.

We can also remove the variables type while defining the Lambda Expression parameter.

If the lambda expression method contains only one parameter then we can remove () symbol also.

In lambda expression return keyword is optional but if we use return keyword then {} are compulsory.

Independently, Lamda Expression is not a statement.

It requires a target variable i.e functional interface reference only.

Lamda target can't be class or abstract class, it will work with functional interface only.

//Programs :

```java
package com.ravi.lambda_expression;

interface Printable
{
    void print();
}

public class LambdaDemo1
{
    public static void main(String[] args)
    {
        Printable p = ()-> System.out.println("Printing");
        p.print();

    }

}
```

```java
package com.ravi.lambda_expression;

@FunctionalInterface
interface Vehicle
{
   void run();
}
public class LambdaDemo2
{
   public static void main(String[] args)
   {
      Vehicle car = () -> System.out.println("Car is running");
      Vehicle bike = () -> System.out.println("Bike is running");
      Vehicle bus = () -> System.out.println("Bus is running");

      car.run();  bike.run();  bus.run();
   }

}
```

```java
package com.ravi.lambda_expression;

@FunctionalInterface
interface Calculator
{
    void doSum(int x, int y);
}
public class LambdaDemo3
{
    public static void main(String[] args)
    {
        Calculator c1 = (x, y) -> System.out.println("Sum is :"+(x+y));
         c1.doSum(12, 48);
    }

}
```

```java
package com.ravi.lambda_expression;

import java.util.Scanner;

@FunctionalInterface
interface Length
{
    int getLength(String str);
}

public class LambdaDemo4
{
    public static void main(String[] args)
    {
        Length length =   str ->
        {
          return str.length();
        };

        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your City Name :");
        String city = sc.nextLine();

        System.out.println("Length of "+city+" is :"+length.getLength(city));
        sc.close();
    }

}
```

The following Program explains the target of Lambda must be a Functional interface only.

```java
abstract class Drawable
{
    abstract void draw();
}

public class LambdaTarget
{
    public static void main(String[] args)
    {
     Drawable d1 = ()-> System.out.println("Drawing"); //error
     d1.draw();
    }
}
```

What is Type Parameter <T> in java ?

----

* This concept was originally developed by C++ by the name Template.

Example :

```java
void swap(int x, int y)
{
  //Can swap only two integer value
}

void swap(double x, double y)
{
  //Can swap double values
}


void swap(String x, String y)
{
  //Can swap String  values
}
```

Instead of Writing these 3 method we can write a single method with **Template**

```java
void swap(T x, T y)
{
  //Can swap all different types
}
```

* Java has introduced Type parameter <T> from JDK 1.5V to make java application "Independent of Data Type".

* By using Type parameter, We can accept only Wrapper classes and User defined classes.

* We **cannot accept primitive data type.**