Functional Interfaces

If an interface contain only one abstract method, such type of interfaces are called functional interfaces and the method is called functional method or single abstract method (SAM).

Ex:

1) Runnable → It contains only run() method
2) Comparable → It contains only compareTo() method
3) ActionListener → It contains only actionPerformed()
4) Callable → It contains only call() method

Inside functional interface in addition to single Abstract method (SAM) we write any number of default and static methods.

Ex:

```
1)  interface Interf {
2)          public abstract void m1();
3)          default void m2() {
4)                  System.out.println ("hello");
5)          }
6)  }
```

In Java 8, Sun Micro System introduced @Functional Interface annotation to specify that the interface is Functional Interface.

Ex:

```
@Functional Interface
        Interface Interf {
            public void m1();
    }
```

This code compiles without any compilation errors.

Inside Functional Interface we can take only one abstract method, if we take more than one abstract method then compiler raise an error message that is called we will get compilation error.

Ex:

```
@Functional Interface {
        public void m1();
        public void m2();
}
```

This code gives compilation error.

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Inside Functional Interface we have to take exactly only one abstract method.If we are not declaring that abstract method then compiler gives an error message.

Ex:

```
@Functional Interface {
        interface Interface {
}
```

compilation error

Functional Interface with respect to Inheritance:

If an interface extends Functional Interface and child interface doesn't contain any abstract method then child interface is also Functional Interface

Ex:

```
1)  @Functional Interface
2)  interface A {
3)        public void methodOne();
4)  }
5)  @Functional Interface
6)  Interface B extends A {
7)  }
```

In the child interface we can define exactly same parent interface abstract method.

Ex:

```
1)  @Functional Interface
2)  interface A {
3)        public void methodOne();
4)  }
5)  @Functional Interface
6)  interface B extends A {
7)        public void methodOne();
8)  }
```

No Compile Time Error

In the child interface we can't define any new abstract methods otherwise child interface won't be Functional Interface and if we are trying to use @Functional Interface annotation then compiler gives an error message.

```
1)  @Functional Interface {
2)  interface A {
3)        public void methodOne();
4)  }
5)  @Functional Interface
6)  interface B extends A {
7)        public void methodTwo();
8)  }
```

Compiletime Error

Ex:

```
@Functional Interface
interface A {
      public void methodOne();
}
interface B extends A {
       public void methodTwo();
error
}
```

No compile time error

This's Normal interface so that code compiles without

In the above example in both parent & child interface we can write any number of default methods and there are no restrictions. Restrictions are applicable only for abstract methods.

Functional Interface Vs Lambda Expressions:

Once we write Lambda expressions to invoke it's functionality, then Functional Interface is required. We can use Functional Interface reference to refer Lambda Expression.
Where ever Functional Interface concept is applicable there we can use Lambda Expressions

Ex:1 Without Lambda Expression

```
1)  interface  Interf {
2)        public void methodOne() {}
3)        public class Demo implements Interface {
4)            public void methodOne() {
5)                System.out.println("method one execution");
6)            }
7)            public class Test {
8)                public static void main(String[] args) {
9)                    Interfi = new Demo();
10)                   i.methodOne();
11)               }
12) }
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Above code With Lambda expression

```
1)  interface Interf {
2)        public void methodOne() {}
3)        class Test {
4)             public static void main(String[] args) {
5)                  Interfi = () → System.out.println("MethodOne Execution");
6)                  i.methodOne();
7)              }
8)  }
```

Without Lambda Expression

```
1)  interface Interf {
2)         public void sum(inta,int b);
3)  }
4)  class Demo implements Interf {
5)          public void sum(inta,int b) {
6)                System.out.println("The sum:"  +(a+b));
7)          }
8)  }
9)  public class Test {
10)         public static void main(String[] args) {
11)                Interfi = new Demo();
12)                i.sum(20,5);
13)         }
14) }
```

Above code With Lambda Expression

```
1)  interface Interf {
2)        public void sum(inta, int b);
3)  }
4)  class Test {
5)        public static void main(String[] args) {
6)             Interfi = (a,b) → System.out.println("The Sum:" +(a+b));
7)             i.sum(5,10);
8)        }
9)  }
```

Without Lambda Expressions

```
1)  interface Interf {
2)          publicint square(int x);
3)  }
4)  class Demo implements Interf {
5)              public int square(int x) {
6)                    return x*x;  OR  (int x) → x*x
7)              }
8)  }
9)  class Test {
10)         public static void main(String[] args) {
11)             Interfi = new Demo();
12)             System.out.println("The Square of 7 is: " +i.square(7));
13)         }
14) }
```

Above code with Lambda Expression

```
1)  interface Interf {
2)          public int square(int x);
3)  }
4)  class Test {
5)          public static void main(String[] args) {
6)                  Interfi = x → x*x;
7)                  System.out.println("The Square of 5 is:"+i.square(5));
8)          }
9)  }
```

Without Lambda expression

```
1)  class MyRunnable implements Runnable {
2)              public void main() {
3)                    for(int i=0; i<10; i++) {
4)                          System.out.println("Child Thread");
5)                    }
6)              }
7)  }
8)  class ThreadDemo {
9)              public static void main(String[] args) {
10)                   Runnable r = new myRunnable();
11)                   Thread t = new Thread(r);
12)                   t.start();
13)                   for(int i=0; i<10; i++) {
14)                         System.out.println("Main Thread")
15)                   }
16)         }
```


```
17) }
```

With Lambda expression

```
1)  class ThreadDemo {
2)          public static void main(String[] args) {
3)                  Runnable r = () → {
4)                          for(int i=0; i<10; i++) {
5)                                  System.out.println(“Child Thread”);
6)                          }
7)                  };
8)                  Thread t = new Thread(r);
9)                  t.start();
10)                 for(i=0; i<10; i++) {
11)                         System.out.println(“Main Thread”);
12)                 }
13)         }
14) }
```

Anonymous inner classes vs Lambda Expressions

Wherever we are using anonymous inner classes there may be a chance of using Lambda expression to reduce length of the code and to resolve complexity.

Ex: With anonymous inner class

```
1)  class Test {
2)      public static void main(String[] args) {
3)              Thread t = new Thread(new Runnable() {
4)                      public void run() {
5)                              for(int i=0; i<10; i++) {
6)                                      System.out.println("Child Thread");
7)                              }
8)                      }
9)              });
10)             t.start();
11)            for(int i=0; i<10; i++)
12)                System.out.println("Main thread");
13)     }
14) }
```

With Lambda expression

```
1)  class Test {
2)        public static void main(String[] args) {
3)                Thread t = new Thread(() → {
4)                                                for(int i=0; i<10; i++) {
5)                                                    System.out.println("Child Thread");
6)                                                }
7)                });
8)                t.start();
9)                for(int i=0; i<10; i++) {
10)                   System.out.println("Main Thread");
11)               }
12)       }
13) }
```

What are the advantages of Lambda expression?

- We can reduce length of the code so that readability of the code will be improved.
- We can resolve complexity of anonymous inner classes.
- We can provide Lambda expression in the place of object.
- We can pass lambda expression as argument to methods.

Note:

- Anonymous inner class can extend concrete class, can extend abstract class, can implement interface with any number of methods but
- Lambda expression can implement an interface with only single abstract method (Functional Interface).
- Hence if anonymous inner class implements Functional Interface in that particular case only we can replace with lambda expressions. Hence wherever anonymous inner class concept is there, it may not possible to replace with Lambda expressions.
- Anonymous inner class! = Lambda Expression

- Inside anonymous inner class we can declare instance variables.
- Inside anonymous inner class "this" always refers current inner class object(anonymous inner class) but not related outer class object

Ex:

- Inside lambda expression we can't declare instance variables.
- Whatever the variables declare inside lambda expression are simply acts as local variables

- Within lambda expression 'this" keyword represents current outer class object reference (that is current enclosing class reference in which we declare lambda expression)

Ex:

```
1)  interface Interf {
2)          public void m1();
3)  }
4)  class Test {
5)          int x = 777;
6)          public void m2() {
7)                  Interfi = ()→ {
8)                          int x = 888;
9)                          System.out.println(x); 888
10)                         System.out.println(this.x); 777
11)                 };
12)                 i.m1();
13)         }
14)         public static void main(String[] args) {
15)                 Test t = new Test();
16)                 t.m2();
17)         }
18) }
```

- From lambda expression we can access enclosing class variables and enclosing method variables directly.
- The local variables referenced from lambda expression are implicitly final and hence we can't perform re-assignment for those local variables otherwise we get compile time error.

Ex:

```
1)  interface Interf {
2)          public void m1();
3)  }
4)  class Test {
5)          int x = 10;
6)          public void m2() {
7)                  int y = 20;
8)                  Interfi = () → {
9)                          System.out.println(x); 10
10)                         System.out.println(y); 20
11)                         x = 888;
12)                         y = 999; //CE
13)                 };
14)                 i.m1();
15)                 y = 777;
16)         }
```


```
17)     public static void main(String[] args) {
18)             Test t = new Test();
19)             t.m2();
20)     }
21) }
```

Differences between anonymous inner classes and Lambda expression

Anonymous Inner class	Lambda Expression
It's a class without name	It's a method without name (anonymous function)
Anonymous inner class can extend Abstract and concrete classes	lambda expression can't extend Abstract and concrete classes
Anonymous inner class can implement An interface that contains any number of Abstract methods	lambda expression can implement an Interface which contains single abstract method (Functional Interface)
Inside anonymous inner class we can Declare instance variables.	Inside lambda expression we can't Declare instance variables, whatever the variables declared are simply acts as local variables.
Anonymous inner classes can be Instantiated	lambda expressions can't be instantiated
Inside anonymous inner class "this" Always refers current anonymous Inner class object but not outer class Object.	Inside lambda expression "this" Always refers current outer class object. That is enclosing class object.
Anonymous inner class is the best choice If we want to handle multiple methods.	Lambda expression is the best Choice if we want to handle interface With single abstract method (Functional Interface).
In the case of anonymous inner class At the time of compilation a separate Dot class file will be generated (outerclass$1.class)	At the time of compilation no dot Class file will be generated for Lambda expression. It simply converts in to private method outer class.
Memory allocated on demand Whenever we are creating an object	Reside in permanent memory of JVM (Method Area).