```
Lambda Expression
+++++++++++++++++
@FunctionalInterface
interface Interf
{
      public int square(int x);
}

class Demo implements Interf
{
      @Override
      public int square(int x)
      {
            return (x*x);
      }
}
public class Test{
      //JVM -> main thread created and started
      public static void main(String[] args){

            //Traditional Approach
            Interf i = new Demo();
            int result=i.square(5);
            System.out.println("The square is :: "+result);

            System.out.println();

            //Interface called Interf square(int x) :: void
            //Binding Lambda-Expression
            Interf i1 = x->x*x;
            System.out.println("The square is :: "+i1.square(100));
      }
}

Output
The square is :: 25
The square is :: 10000
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

eg#2.
/*
public interface java.lang.Runnable {
      public abstract void run();
}
*/
class MyRunnable implements Runnable
{
      @Override
      public void run()
      {
            //logic for thread
            for (int i = 0;i<10 ;i++ )
            {
                  System.out.println("Child Thread....");
            }
      }
}
public class Test{
      //JVM -> main thread created and started
```

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

		//Traditional Approach of working with OOPS
		Runnable r = new MyRunnable();
		Thread t = new Thread(r);
		t.start();
		for (int i = 1;i<=10 ;i++ )
		{
			System.out.println("Main Thread");
		}

		System.out.println();

		System.in.read();

		//New Approach in java :: Functional Programming
		//Interface called Runnable run() :: void
		//Binding Lambda-Expression
			Runnable r1=()->{
				for (int i =0;i<5 ;i++ )
				{
					System.out.println("Lambda Expression :: Thread");
				}
			};
			Thread t1 = new Thread(r1);
			t1.start();
			for (int i = 1;i<=10 ;i++ )
			{
				System.out.println("Main Thread");
			}
	}
}
```

Anonymous Inner class
++++++++++++++++++++

Inner classes
++++++++++++

```
class Outer
{
	//static varaibles
	//instance variables
	//blocks  :: instance, static
	//methods :: instance, static

	class Inner
	{

	}
}

class Outer
{
	class Inner
	{
		public void m1()
		{
			System.out.println("From Inner class");
		}
```

```
      }
}
Output:
Outer.class
Outer$Inner.class
```

=> Sometimes we declare inner class wihtout name such type of inner classes are called as "Ananymous Inner class".
=> Main object of Ananymous Inner class is "just for instance use".
=> There are 3 types of Ananymous Inner class
      1. Anonymous inner class extends a class.
      2. Anonymous inner class implements an interface.
      3. Anonymous inner class that is defined inside method argument.

1. Anonymous inner class extends a class.

eg#1.
```
class PopCorn
{
      public void taste()
      {
            System.out.println("Spicy...");
      }
}
public class Test{
      //JVM -> main thread created and started
      public static void main(String[] args)throws Exception{

            PopCorn p = new PopCorn()
            {
                  @Override
                  public void taste()
                  {
                        System.out.println("Salty...");
                        brandName();
                  }

                  public void brandName()
                  {
                        System.out.println("MacD");
                  }
            };
            p.taste();
            //p.brandName();

            System.out.println();

            PopCorn p1 = new PopCorn();
            p1.taste();

      }
}
output
Salty...
MacD

Spicy...
```

eg#2.

```
/*
      @FunctionalInterface
      interface Runnable
      {
            void run();
      }
      public class Thread implements Runnable
      {
            public void start(){
                  1. register the thread with T.S
                  2. perform low level activities
                  3. invoke run()
            }

            @Override
            public void run(){

            }
      }
*/

class MyThread extends Thread
{
            @Override
            public void run(){
                  //logic for a thread
                  for (int i =0;i<5 ;i++ )
                  {
                        System.out.println("child thread...");
                  }
            }
}
public class Test{
      //JVM -> main thread created and started
      public static void main(String[] args)throws Exception{
            Thread t = new MyThread();
            t.start();

            //logic for main thread
            for (int i =0;i<5 ;i++ )
            {
                  System.out.println("parent thread...");
            }

            System.out.println();

            System.in.read();

            Thread t1 =new Thread()
            {
                  @Override
                  public void run(){
                        //logic for a thread
                        for (int i =0;i<5 ;i++ )
                        {
                              System.out.println("Child thread::Anonymous Inner
class");
                        }
                  }
            };
            t1.start();

            //logic for main thread
            for (int i =0;i<5 ;i++ )
            {
                  System.out.println("Parent thread:: Ananymous Inner class");
            }

            System.out.println();

            System.in.read();
            Runnable r = ()->{
                        //logic for a thread
                        for (int i =0;i<5 ;i++ )
                        {
                              System.out.println("Child thread::Lambda
Expression");
                        }
            };
            new Thread(r).start();
            //logic for main thread
            for (int i =0;i<5 ;i++ )
            {
                  System.out.println("Parent thread:: Lambda Expression");
            }


      }
}
```
Output
```
D:\OctBatchMicroservices>java Test
parent thread...
parent thread...
parent thread...
parent thread...
parent thread...

child thread...
child thread...
child thread...
child thread...
child thread...

Parent thread:: Ananymous Inner class
Parent thread:: Ananymous Inner class
Parent thread:: Ananymous Inner class
Parent thread:: Ananymous Inner class
Child thread::  Anonymous Inner class
Child thread::  Anonymous Inner class
Child thread::  Anonymous Inner class
Child thread::  Anonymous Inner class
Child thread::  Anonymous Inner class
Parent thread:: Ananymous Inner class

Parent thread:: Lambda Expression
Parent thread:: Lambda Expression
Parent thread:: Lambda Expression
Parent thread:: Lambda Expression
Parent thread:: Lambda Expression
Child thread::Lambda Expression
Child thread::Lambda Expression
Child thread::Lambda Expression
Child thread::Lambda Expression
Child thread::Lambda Expression
```

2. Anonymous inner class implements an interface.

```
/*
      @FunctionalInterface
      interface Runnable
      {
            void run();
      }
      public class Thread implements Runnable
      {
            public void start(){
                  1. register the thread with T.S
                  2. perform low level activities
                  3. invoke run()
            }

            @Override
            public void run(){

            }
      }
*/
class MyRunnable implements Runnable
{
            @Override
            public void run(){
                  //logic for a thread
                  for (int i =0;i<5 ;i++ )
                  {
                        System.out.println("child thread...");
                  }
            }
}

public class Test{
      //JVM -> main thread created and started
      public static void main(String[] args)throws Exception{
            Runnable r = new MyRunnable();
            Thread t = new Thread(r);
            t.start();

            //logic for main thread
            for (int i =0;i<5 ;i++ )
            {
                  System.out.println("parent thread...");
            }

            System.out.println();
            System.in.read();
            /*
                  here we are creating an object of a class which impements
                  Runnable interface and there is no name for that class.
            */
            Runnable r1 = new Runnable()
            {
                  @Override
                  public void run()
                  {
                        //logic for a thread
                        for (int i =0;i<5 ;i++ )
                        {
                              System.out.println("Child thread::Ananymous Inner
class");
                        }
                  }
            };
            new Thread(r1).start();

            //logic for main thread
            for (int i =0;i<5 ;i++ )
            {
                  System.out.println("Parent thread::Ananymous Inner class");
            }

            System.out.println();
            System.in.read();

            Runnable r2 = ()->{
                        //logic for a thread
                        for (int i =0;i<5 ;i++ )
                        {
                              System.out.println("Child thread::Lambda
Expression");
                        }
            };
            new Thread(r2).start();
            //logic for main thread
            for (int i =0;i<5 ;i++ )
            {
                  System.out.println("Parent thread:: Lambda Expression");
            }
      }
}
```

```
Output
D:\OctBatchMicroservices>java Test
parent thread...
parent thread...
parent thread...
parent thread...
parent thread...

child thread...
child thread...
child thread...
child thread...
child thread...
```

```
Parent thread:: Ananymous Inner class
Parent thread:: Ananymous Inner class
Parent thread:: Ananymous Inner class
Parent thread:: Ananymous Inner class
Child thread::Anonymous Inner class
Child thread::Anonymous Inner class
Child thread::Anonymous Inner class
Child thread::Anonymous Inner class
Child thread::Anonymous Inner class
Parent thread:: Ananymous Inner class

Parent thread:: Lambda Expression
Parent thread:: Lambda Expression
Parent thread:: Lambda Expression
Parent thread:: Lambda Expression
Parent thread:: Lambda Expression
Child thread::Lambda Expression
Child thread::Lambda Expression
Child thread::Lambda Expression
Child thread::Lambda Expression
Child thread::Lambda Expression
```

3. Anonymous inner class that is defined inside method argument

```
public class Test{
      //JVM -> main thread created and started
      public static void main(String[] args)throws Exception{

            new Thread(
                  new Runnable()
                  {
                              @Override
                              public void run()
                              {
                                          //logic for a thread
                                          for (int i =0;i<5 ;i++ )
                                          {
                                                System.out.println("Child
thread::Ananymous Inner class");
                              }
                  }
            }).start();

            //logic for main thread
            for (int i =0;i<5 ;i++ )
            {
                  System.out.println("Parent thread::Ananymous Inner class");
            }


      }
}
```

Output

```
Parent thread::Ananymous Inner class
Parent thread::Ananymous Inner class
Parent thread::Ananymous Inner class
Parent thread::Ananymous Inner class
Parent thread::Ananymous Inner class
Child thread::Ananymous Inner class
Child thread::Ananymous Inner class
Child thread::Ananymous Inner class
Child thread::Ananymous Inner class
Child thread::Ananymous Inner class
```

Working with Lambda Expression
++++++++++++++++++++++++++++++
1. Inside lambda expression we can declare varaibles those variables are treated as local variables.
2. Within Lambda expression we can access instance variables of that class using "this" keyword.
3. Inside Lambda expression "this" would refer to Current class object.

eg#1.
```
@FunctionalInterface
interface Interf
{
      void m1();
}
public class Test{

      //instance variable
      int x= 777;

      //instance method
      public void m2()
      {

            Interf i = () -> {
                              int x = 888;
                              System.out.println(x);//888
                              System.out.println(this.x);//777
                        };
            i.m1();
      }

      //JVM -> main thread created and started
      public static void main(String[] args)throws Exception{
            new Test().m2();

      }
}
```

eg#2.
 Inside a method the variables are local variables, but if we write a lambda expression inside a method,then those local variables inside
 lambda expression will be treated as "final" variables, if we try to change the value it would result in "CE".

eg#1.
```
@FunctionalInterface
interface Interf
{
      void m1();
}

public class Test{
      //instance variable
      int x= 10;

      //instance method
      public void m2()
      {
            //local variable[Inside lambda they are final]
            int y = 20;

            Interf i = () -> {
                              System.out.println(x);//10
                              System.out.println(y);//20

                              x = 100;
                              System.out.println(x);//100

                              y = 200;//CE: y is final
                              System.out.println(y);
                        };

                  i.m1();
                  y = 200;
                  System.out.println(y);//200
      }

      //JVM -> main thread created and started
      public static void main(String[] args)throws Exception{
            new Test().m2();
      }
}
```

```
Special features in interface from JDK1.8V
++++++++++++++++++++++++++++++++++++++++++
 Till JDK1.7V
 a. Inside interface we can write
      1. method   -> by default they are public and abstract
      2. variable -> by default they are public static final.

 From JDK1.8V
=> It is possible to write concrete methods also in interface.
=> To write concrete methods we need to use "default" keyword.
=> These methods are called as "Defeneder methods/Virtual methods[To give support
for Backward Compatibility]".
=> By writing these methods the implementation class won't get affected.
=> These methods will be available to implementation class object directly, if the
child class is not happy with the implementation then
   we can change the implemenation of default methods[means we can override the
method].
=> These methods will have some dummy implementation which might be required for
implementation class.
```

```
eg#1.
//JDK1.8V
@FunctionalInterface
interface Car
{
      public int noOfWheels();
      default void engineMake()
      {
            System.out.println("ENGINE MAKE GOOD FROM :: TATA");
      }
}
class Nexon implements Car
{
      @Override
      public int noOfWheels()
      {
            return 6;
      }

      @Override
      public void engineMake()
      {
            System.out.println("ENGINE MAKE GOOD FROM :: MARUTHI");
      }
}
public class Test{
      public static void main(String[] args){
            Car car = new Nexon();
            car.engineMake();
            int wheels=car.noOfWheels();
            System.out.println("No of wheels is :: "+wheels);
      }
}
```

Output

```
ENGINE MAKE GOOD FROM :: MARUTHI
No of wheels is :: 6
```

eg#2.

```
@FunctionalInterface
interface Car
{
      public int noOfWheels();

      default void engineMake()
      {
            System.out.println("ENGINE MAKE GOOD FROM :: TATA");
      }

      default String toString()
      {
            return "Hey Default method from Interface";
      }
}
```

Note: Methods of object class will be by default avaialable to every implementation class, so we should not bring those methods through
"default" methods of an interface.

Note: Default methods in interface would lead to "DiamondShaped" problem in Multiple inheritance

eg#3

```
interface Left{
      default void info(){
            System.out.println("From Left");
      }
}
interface Right{
      default void info(){
            System.out.println("From Right");
      }
}
class Demo implements Left,Right{}
public class TestApp {
      public static void main(String[] args) {
            Demo d =new Demo();
            d.info();//CE: ambiguity
      }
}
```

Solution :: Compulsorily we need to override default method in implementation class.

Note: To get the facility of interface default methods in overdiden method we use the following syntax

```
            interfaceName.super.methodName();
```

eg#4.

```
interface Left{
      default void info(){
            System.out.println("From Left");
      }
}
interface Right{
      default void info(){
            System.out.println("From Right");
      }
}
class Demo implements Left,Right{

      @Override
      public void info()
      {
            Left.super.info();
            Right.super.info();
            System.out.println("From Implementation class...");
      }
}
public class TestApp {
      public static void main(String[] args) {
            Demo d =new Demo();
            d.info();
      }
}
```

Output

```
From Left
From Right
From Implementation class...
```

Conclusions :: Functional interface can have any no of default methods, but we need to have only one "abstract method".

static methods inside interface
++++++++++++++++++++++++++++++++
=> It is possible to write static methods inside interface.
=> These methods are called as "Helper/utility" methods.
=> These methods by default won't be available to implementation class, to use this methods we need to use "InterfaceName".
=> Static methods won't be inherited to Implementation class, so Overriding is not possible.

eg#1.
```java
interface Vehicle
{
	//public abstract methods
	String getBrand();
	String speedUp();
	String speedDown();

	//default methods
	default String turnAlarmOn()
	{
		return "Turning the Vehicle alaram on...";
	}

	default String turnAlarmOff()
	{
		return "Turning the Vehicle alaram of...";
	}

	//static methods :: utility methods/helper methods
	public static void cleanVehicle()
	{
		System.out.println("Clean the Vehicle Properly....");
	}
}
class Car implements Vehicle
{
	private String brand;

	Car(String brand){
		this.brand = brand;
	}

	@Override
	public String getBrand(){
		return brand;
	}

	@Override
	public String speedUp(){
		return "The car is speeding up...";
	}

	@Override
	public String speedDown(){
		return "The car is speeding down...";
	}
}

public class TestApp {
      public static void main(String[] args) {
            Vehicle car= new Car("Nexon");

            //abstract methods
            System.out.println(car.getBrand());
            System.out.println(car.speedUp());
            System.out.println(car.speedDown());

            //default methods
            System.out.println(car.turnAlarmOn());
            System.out.println(car.turnAlarmOff());

            //Utility method
            Vehicle.cleanVehicle();
      }
}
Nexon
The car is speeding up...
The car is speeding down...
Turning the Vehicle alaram on...
Turning the Vehicle alaram of...
Clean the Vehicle Properly...

Case1:
interface Interf1
{
      public static void m1(){}
}
public class TestApp implements Interf1{
      @Override
      public static void m1(){}
      public static void main(String[] args) {

      }
}
Output:: CE

Case2:
interface Interf1
{
      public static void m1(){}
}
public class TestApp implements Interf1{
      @Override
      public void m1(){}
      public static void main(String[] args) {

      }
}
Output :: CE

Case3::
interface Interf1
{
      private void m1(){}
}
public class TestApp implements Interf1{
      @Override
      public void m1(){}
      public static void main(String[] args) {

      }
}
```
Output: CE (private modifiers are not allowed)

Note: Since static methods can be a part of interface, we can write main method which is static inside the interface

```
interface Interf
{
      //utility methods :: JDK1.8
      public static void main(String[] args)
      {
            System.out.println("Main method in interface");
      }
}
```
output
```
D:\OctBatchMicroservices>javac Interf.java

D:\OctBatchMicroservices>java Interf
Main method in interface
```