Inner Classes

A class inside a class is called Inner Class.

Types:

1] Member Inner Class [Non-static Class]
2] Static Inner Class
3] Anonymous Inner Class

Member Inner Class:

```
package com.basics;

public class Test {

	public static void main(String[] args) {
		A obj=new A();
		obj.i=5;

		A.B obj1=obj.new B();
		obj1.d=7;
	}
}

class A{
	int i;
	class  B{
		int d=5;
	}
}
```

And the class files corresponding to this are as follows:

- A$B.class
- A.class
- Test.class

Static Inner Class:

If we make the inner class static, we will not require the object of A to access B object now.

```
package com.basics;

public class Test {

	public static void main(String[] args) {
		A.i=5;

		A.B obj1=new A.B();
		obj1.d=14;
	}
}
class A{
	static int i;
	static class  B{
		int d=5;
	}
}
```

Annonymous Inner Class:

```
package com.basics;

public class AnnonymousInnerClassDemo {
	public static void main(String[] args) {
		Phone obj=new Phone() {//Anno
			public void show() {
				System.out.println("Call,sms,camera");
			}
		};
		obj.show();

	}
}

class Phone{
	public void show() {
		System.out.println("Call");
	}
}
```

And the class files corrosponding to this are as followes:

AnnonymousInnerClassDemo$1.class	4/12/2018 15:23	CLASS File	1 KB
AnnonymousInnerClassDemo.class	4/12/2018 15:23	CLASS File	1 KB
Phone.class	4/12/2018 15:23	CLASS File	1 KB

Lambda Expression:

```
public class Test {

	public static void main(String[] args) {
		Phone phone=new Phone() {
			@Override
			public void show(String key, String Value) {
				System.out.println("Key: "+key+" Value: "+Value);
			}
		};
		phone.show("abc", "123");
	}
}

interface Phone{
	public void show(String key,String Value);
}
```

Here we already know that the show method has **public void** type as defined in the interface. So it is a waste to once again right the same thing in the anonymous inner class.
Note: We can replace that with simple `(key,value)-> {` provided that it is a functional Interface.

Q] What is functional Interface?
Ans: An interface with exactly one abstract method is called Functional Interface. **@FunctionalInterface** annotation is added so that we can mark an interface as functional interface.
Note: The instances of functional interfaces can be created with lambda expressions, method references, or constructor references.

```java
public class Test {

	public static void main(String[] args) {
		Phone phone=(key,value)-> {
			System.out.println("Key: "+key+" Value: "+value);
		};
		phone.show("abc", "123");
	}
}

interface Phone{
	public void show(String key,String Value);

}
```