INNER CLASSES

1. Introduction

➔ One can create inner class as shown below

```
package ch18InnerClasses;

public class Ex1Outer
{
	class Inner
	{
		public void InMeth()
		{
			System.out.println("Inner Method");
		}
	}

	public Inner outMeth()
	{
		return new Inner();
	}

	public static void main(String[] args)
{
		Ex1Outer out = new Ex1Outer();
		Ex1Outer.Inner in = out.outMeth();
		in.InMeth();
		//Inner in2 = new Inner();////No enclosing instance of type Ex1Outer is accessible. Must qualify the allocation with an enclosing instance of type Ex1Outer (e.g. x.new A() where x is an instance of Ex1Outer). Ex1Outer.java/CoreJavaTutorial/src/ch18InnerClasses line 24 Java Problem
	}
}
Output:
Inner Method
```

➔ Above we can see that inner class object is created by OuterClassName.InnerClassName.

➔ Object of inner class can only be created in association with the object of the outer class(whenever inner class is non-static).

➔ Also if one try to create inner class object in a general way using `Inner in2 = new Inner()` then compiler will give error as shown in the programme.

➔ Lets see inner classes in more detail using below example.

```
package ch18InnerClasses;

public class InAccessOutMemOuter
{
	private int [] intArr = {1, 2, 3, 4, 5};

	class InAccessOutMemInner
	{
		public void inMethod()
		{
			for(int i=0; i<intArr.length; i++)
				System.out.print(intArr[i]+ " ");
		}
	}

	public InAccessOutMemInner inRefMeth()
	{
		return new InAccessOutMemInner();
	}
```

```
	public static void main(String[] args)
	{
		InAccessOutMemOuter out = new InAccessOutMemOuter();
		InAccessOutMemInner in = out.inRefMeth();

		in.inMethod();
	}
}
Output:
1 2 3 4 5
```

➔ Here we can see inner class `InAccessOutMemInner` can access private members of enclosing class `InAccessOutMemOuter`. How is it possible? Its possible because inner class secretly captures reference to the enclosing class during construction. Construction of inner class object requires access to the enclosing class object else compiler gives error.

➔ Whenever inner class refer to the members of enclosing class, it uses enclosing class reference.

➔ Lets see one more example of above with Iterator design pattern.

```
package ch18InnerClasses;

interface Selector
{
	boolean end();
	String current();
	void next();
}

class Word
{
	String word;
	Word(String word) { this.word = word; }
	public String toString()
	{
		return word;
	}
}

public class Ex2StringSequence
{
	private String [] items = new String[3];
	int num;

	Ex2StringSequence(int num) { this.num = num; }
	int i = 0;

	void add(String str)
	{

		if(i<num)
		items[i++] = str;
	}

	class SequenceSelector implements Selector
	{
		public int j=0;
		public boolean end() { return j==items.length; }
		public String current()   { return items[j];  }
		public void next() { if(j<items.length) j++; }
	}

	public Selector getSel()
	{
		return new SequenceSelector();
	}
```

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

		Word [] wordArr = {new Word("One"), new Word("Two"), new Word("Three")};
		Ex2StringSequence strseq = new Ex2StringSequence(wordArr.length);

		for(int i=0; i<wordArr.length; i++)
		strseq.add(wordArr[i].toString());

		Selector sel = strseq.getSel();
		while(!sel.end())
		{
			System.out.println(sel.current());
			sel.next();
		}
	}
}
Output:
One
Two
Three
```

➔ Here class `StringSequence` taking strings and put them in a sequence using a private string array `items`. Inner class `SequenceSelector` is used to access items elements. Here 'Iterator' design pattern is used.

2. Using .this and .new

➔ Outer and Inner class objects can also be created as shown below.

```
package ch18InnerClasses;
public class DotNewNThisOuter {

	void outf() {System.out.println("outer fn()"); }

	class DotNewNThisInner
	{
		public DotNewNThisOuter outRef()
		{
			return DotNewNThisOuter.this;//.this
		}
	}

	public DotNewNThisInner inRef()
	{
		return new DotNewNThisInner();
	}

	public static void main(String[] args)
	{
		DotNewNThisOuter out = new DotNewNThisOuter();
		DotNewNThisOuter.DotNewNThisInner in = out.new DotNewNThisInner();//.new
		in.outRef().outf();
	}
}
Outupt:
outer fn()
```

➔ Above we can see outer class object creation has happened through .this and inner class object creation took place using .new.

3. Inner Class and Upcasting

➔ Upcasting is nothing but assigning class reference to base class or an interface.

➔ When inner class is upcasted to base class or an interface then its implementation is not visible hence upcasting is useful to hide the implementation.

```
package ch18InnerClasses;
interface IntInner
{
	void IntInFn();
}
public class UpcastOuter
{
	void outFn()
	{
		System.out.println("Outer Fn");
	}

	class UpcastInner implements IntInner
	{
		void inFn()
		{
			System.out.println("Inner Fn");
		}

		public void IntInFn()
		{
			System.out.println("Inner Interface Fn");
		}
	}

	public IntInner intRef()
	{
		return new UpcastInner();
	}

	public static void main(String[] args)
	{
		UpcastOuter uout = new UpcastOuter();
		IntInner iin= uout.intRef();
		iin.IntInFn();
	}
}
Output:
Inner Interface Fn
```

➔ We know classes can have only public or package access but inner classes can have both private and protected access too as shown below.

```
package ch18InnerClasses;
interface PrivateInt
{
	void IntFn();
}

class PrivateInnerOuter1
{

	void outFn()
	{
		System.out.println("Inside outer method");
	}

	private class PrivateInner1
```

```
	{
		void inFn1()
		{
			System.out.println("Inside private1 inner method");
		}
	}

	private class PrivateInner2 implements PrivateInt
	{
		public void IntFn()
		{
			System.out.println("Interface Method");
		}
		void inFn2()
		{
			System.out.println("Inside private2 inner method");
		}
	}

	protected class ProtectedInner
	{
		void inFn3()
		{
			System.out.println("Inside protected inner method");
		}
	}

	public PrivateInner1 priRef1()
	{
		return new PrivateInner1();
	}

	public PrivateInt priRef2()
	{
		return new PrivateInner2();
	}

	public ProtectedInner proRef()
	{
		return new ProtectedInner();
	}
}

public class PrivateInnerOuter
{
	public static void main(String[] args)
	{
		PrivateInnerOuter1 out =  new PrivateInnerOuter1();
		//PrivateInnerOuter1.PrivateInner1 priin1 = out.new PrivateInner1();//Cannot Access
		Private class, its not visible.
		//PrivateInnerOuter1.PrivateInner priin2 = out.new PrivateInner2();//cannot access
		private class, its not visible.
		PrivateInnerOuter1.ProtectedInner proin = out.new ProtectedInner();
		out.outFn();
		//out.priRef1().inFn1();//ERR : The type PrivateInnerOuter1.PrivateInner1 is not
		visible
		out.priRef2().IntFn();
		proin.inFn3();
	}
}
Output:
Inside outer method
Interface Method
Inside protected inner method
```

Above we can see private inner classes can only be accessed inside its enclosing class. Protected classes can only be accessed in the same package or from classes who inherit them.

4. Inner Classes In Methods and Scopes

➔ Inner classes can be created within a method or even in an arbitrary scope. There are two reasons for doing it.

i. As shown in previous example, if any interface is implemented then its reference can be returned through it.

ii. You want a class for aid, but dont want it to be publicly available.

Now we are going to created classes as

1. A class defined within a method
2. A class defined within a scope inside a method
3. An *anonymous* class implementing an interface
4. An anonymous class extending a class that has a non-default constructor
5. An anonymous class that performs field initialization
6. An anonymous class that performs construction using instance initialization (anonymous inner classes cannot have constructors)

4.1 A class defined within a method

➔ Inner class created inside a method is called local inner class.

```
package ch18InnerClasses;

interface InInt
{
	void IntFn();
}

public class InnerInsideMethodOuter
{
	public InInt OutFn()
	{
		class InnerInsideMethodInner implements InInt
		{
			public void IntFn()
			{
				System.out.println("Inside Interface Method");
			}
		}
		return new InnerInsideMethodInner();
	}

	public static void main(String[] args)
	{
		InnerInsideMethodOuter out = new InnerInsideMethodOuter();
		out.OutFn().IntFn();

	}
}
Output:
Inside Interface Method
```

4.2 A class defined within a scope inside a method

➔ As they are called local inner classes i.e. they are local to the method. Hence they cannot be accessed outside method as shown below.

```
package ch18InnerClasses;

public class InnerInsideMethodOuter
{
	public void outFn()
	{
		class InnerInsideMethodInner
		{
			public void IntFn()
			{
				System.out.println("Inside Interface Method");
			}
		}
	}

	public static void main(String[] args)
	{
		InnerInsideMethodOuter out = new InnerInsideMethodOuter();
		//InnerInsideMethodOuter.InnerInsideMethodInner = out.new
InnerInsideMethodInner();//InnerInsideMethodInner cannot recognize type
	}
}
```

Therefore upcasting is used through interface to get access to it as shown below.

```
package ch18InnerClasses;

interface InInt
{
	void IntFn();
}

public class InnerInsideMethodOuter
{
	public InInt outFn()
	{
		class InnerInsideMethodInner implements InInt
		{
			public void IntFn()
			{
				System.out.println("Inside Interface Method");
			}
		}
		return new InnerInsideMethodInner();
	}

	public static void main(String[] args)
	{
		InnerInsideMethodOuter out = new InnerInsideMethodOuter();
		out.outFn().IntFn();
	}
}
Output:
Inside Interface Method
```

4.2 A class defined within a scope inside a method

```
package ch18InnerClasses;

public class InnerInsideScopeOuter
{
	public void outFn1(boolean b)
	{
		if(b)
		{
			class InnerInsideScopeInner
			{
				private String id;
				InnerInsideScopeInner(String str)
```

```
                        {
                                id = str;
                        }

                        public String getId()
                        {
                                return id;
                        }
                }
                InnerInsideScopeInner in = new InnerInsideScopeInner("Inner");
                System.out.println(in.getId());
        }
    }

    public void outFn2()
    {
        outFn1(true);
    }

    public static void main(String[] args)
    {
        InnerInsideScopeOuter out = new InnerInsideScopeOuter();
        out.outFn2();
    }
}
Output:
Inner
```

Above we can see class `InnerInsideScopeInner` is nested inside the scope of `if` statement. This doesn't mean class is conditionally created. It gets compiled along with everything else but not available outside scope.

4.3 Anonymouse Inner Classes implementing an interface

➔ Its an inner class which is anonymous i.e. don't have any name.

➔ It is useful in case one want to overload a method without subclass a class.

```
package ch18InnerClasses;

interface AnonymousInnerInner
{
    void intFn();
}

public class AnonymousInnerOuter {

    public Anthropomorphising inRef()//Here anonymous inner class have been created using
interface
    {
        return new AnonymousInnerInner()
                        {
                                public void intFn()
                                {
                                        System.out.println("Inside Anonymous Interface Method");
                                }
                        };
    }

    public static void main(String[] args)
    {
        AnonymousInnerOuter out = new AnonymousInnerOuter();
        AnonymousInnerInner in = out.inRef();
        in.intFn();
    }
}
Output:
Inside Anonymous Interface Method
```

➔ Semicolon at the end of anonymous inner class doesn't mark end of class body rather its marks the expression which happens to contain anonymous class body. Its identical to the use of semicolon anywhere else.

➔ The above programme is a short hand for

```
public class AnonymousInnerOuter {

	public class AnonymousInnerInner implements InInt
	{
				public void intFn()
				{
					System.out.println("Inside Anonymous Interface Method");
				}
	}

	public InInt intRef() { return new AnonymousInnerInner(); }

	public static void main(String[] args)
	{
		AnonymousInnerOuter out = new AnonymousInnerOuter();
		AnonymousInnerInner in = out.inRef();
		in.intFn();
	}
}
```

4.4 Anonymous class extending a class with non-default constructor

➔ The anonymous inner class `AnonymousInnerInner` created above uses default constructor. If constructor with an argument is needed then one have to extend a class than implementing an interface as shown below.

```
package ch18InnerClasses;

class AnonArgConstInner1
{
	private int i;
	AnonArgConstInner1(int x)
	{
		i = x;
	}
	public int value()
	{
		return i;
	}
}

public class AnonArgConstOuter
{
	public AnonArgConstInner1 inRef()
	{
		return new AnonArgConstInner1(5)
				{
					public int value()
					{
						return super.value()*5;
					}
				};
	}
	public static void main(String args [])
	{
		AnonArgConstOuter out = new AnonArgConstOuter();
		System.out.println(out.inRef().value());
	}

}
Output: 25
```

4.5 Anonymous class with field initialization

➔ One can also do field initialization in an anonymous inner class as shown below.

```
package ch18InnerClasses;

class AnonFieldInner1
{
	String str1;
	AnonFieldInner1(String str)
	{
		str1 = str;
	}
	public String retStr()
	{
		System.out.println("super method");
		return str1;
	}
}

public class AnonFieldOuter
{
	public AnonFieldInner1 inRef(String str)
	{
		return new AnonFieldInner1("Concrete Class")
				{
					private String str2 = str;

					public String retStr()
					{
						return str2;
					}
				};
	}
	public static void main(String[] args)
	{
		AnonFieldOuter out = new AnonFieldOuter();
		System.out.println(out.inRef("Anonymous Class").retStr());
	}
}
Output:
Anonymous Class
```

➔ Above we can also see how a constructor(base class) can be used to initialize variables.

5. Nested Classes

➔ When one dont want a connection between inner class object and outer class object then inner class should be made static. It is commonly called as Nested Class.

➔ For nested classes

i. Outer class object is no longer needed to create inner class object.

ii. Non-static outer class members cannot be accessed by nested class.

```
package ch18InnerClasses;
public class NestedClassOuter
{
  int i = 5;
  static int j = 10;
  static class NestedClassInner
  {
    int k = 15;
```

```
    public void inFn1()
    {
      //System.out.println(i);//non-static variable i cannot be referenced from a static context
      System.out.println(j);
      System.out.println(k);
    }
    public void inFn2()
    {
      System.out.println("Inner fn....");
    }
  }

  public NestedClassInner inRef()
  {
    return new NestedClassInner();
  }

  public void outCheck()
  {
    //NestedClassInner.inFn1();//ERR(exc) : cannot make a static reference to the non-static
    //method inFn1() from NesstedClassInner
  }

  public static void main(String [] args)
  {
    NestedClassOuter out = new NestedClassOuter();
    out.outCheck();
    NestedClassInner in = new NestedClassInner();//nestsed classes can be created directly
    in.inFn1();
    in.inFn2();
    NestedClassOuter.NestedClassInner in1 = out.inRef();
    in1.inFn2();
  }
}
Output:
10
15
Inner fn....
Inner fn....
```

➔ Above we can see how NestedClassInner object is created without the reference of NestedClassOuter class.

➔ Also note static classes cannot access non-static variables/objects. Hence it analogous to a static method.

➔ Also nested classes are run from console using $ operator.

```
public class NestedOuter
{
	public void outFn()
	{
		System.out.println("Outer Function");
	}

	static class NestedInner
	{
		public void inFn()
		{
			System.out.println("Inner Function");
		}

		public static void main(String args [])
		{
			NestedOuter out = new NestedOuter();
			NestedInner in = new NestedInner();

			out.outFn();
```

```
                    in.inFn();
             }
      }
}
Output:
Outer Function
Inner Function
```

➔ The above programme compile normally. But to run it one should use '`java NestedOuter$NestedInner`'. This is because inner class is having main() method inside it. If main() method was present in outer class then it will execute normally using command '`java NestedOuter`'.

➔ The above is necessary because

5.1 Classes Inside Interface

➔ One cannot put any code inside an interface, but nested classes can be part of it.

➔ Any class one puts insides an interface is automatically public and static.

➔ Nested class can even implement the surrounding interface.

➔ (*) Class inside an interface is used to provide default implementations of interface methods.(?we have abstract classes too for that).

```
package ch18InnerClasses;

//To run this class one should use
//1. compile using javac NestedInterface.java
//2. run class using java NestedInterface$NestedInterfaceClass
public interface NestedInterface
{
      void intFn();

      class NestedInterfaceClass implements NestedInterface
      {
            public void intFn()
            {
                  System.out.println("Inside Interface Method");
            }

            public static void main(String [] args)
            {
                  new NestedInterfaceClass().intFn();
            }
      }
}
Output:
Inside Interface Method
```

➔ Above we can see main() method is present in nested class, hence after compilation one should use '`java NestedInterface$NestedInterfaceClass`' on command line.

➔ One can either implement the interface or extends the class using interface.className as shown below. Check out more on working of nested class inside interface.

```
package ch18InnerClasses;
interface NestedInterface1
{
```

```
	void intFn();

class NestedInterfaceCls implements NestedInterface1
	{
		public void intFn()
		{
			System.out.println("Interface Method");
		}
	}
}

class NestedInterfaceClass1 implements NestedInterface1
{
	public void intFn()
	{
		System.out.println("Inside NestedInterafaceClass1 intFn()");
	}

	public void nc1Fn()
	{
		NestedInterface1.NestedInterfaceCls ni = new NestedInterface1.NestedInterfaceCls();
		ni.intFn();
	}
}

class NestedInterfaceClass2 extends NestedInterface1.NestedInterfaceCls
{
	public void nc2Fn()
	{
		intFn();
	}
}

public class NestedInterfaceWorking extends NestedInterface1.NestedInterfaceCls
{
	public static void main(String [] args)
	{
		NestedInterfaceClass1 ni1 = new NestedInterfaceClass1();
		NestedInterfaceClass2 ni2 = new NestedInterfaceClass2();

		ni1.intFn();
		ni1.nc1Fn();

		ni2.nc2Fn();
	}
}
Output:
Inside NestedInterafaceClass1 intFn()
Interface Method
Interface Method
```

5.2 Multiple Nested Classes

➔ It doesn't matter how deeply an inner class may be nested—it can transparently access all of the members of all the classes it is nested within, as shown below.

```
package ch18InnerClasses;

class MNA
{
	private void f() {System.out.println("Inside f()");}
	class A
	{
		private void g() {System.out.println("Inside g()");}
		public class B
		{
			void h()
			{
				g();
```

```
                        f();
                    }
                }
        }
}

public class MultiNestingAccess
{
        public static void main(String[] args)
        {
                MNA mna = new MNA();
                MNA.A mnaa = mna.new A();
                MNA.A.B mnaab = mnaa.new B();
                mnaab.h();
        }
}
Output:
Inside g()
Inside f()
```

6. <u>Why Inner Classes ?</u>

➔ (*) Above example clearly explains why inner class is needed. An inner class can independently inherit from an implementation. That is the <u>problem of multiple inheritance</u> is partly solved by interfaces. But inner classes effectively allow one to inherit from more than one non-interface.

1) If we have two interfaces then one can implement both of them or can also use inner class as shown below.

```
package ch18InnerClasses;

interface A {}
interface B {}
class X implements A, B {}
class Y implements A
{
  B makeB()
  {
  // Anonymous inner class:
    return new B() {};
  }
}

public class MultiInterfaces
{
  static void takesA(A a) {}
  static void takesB(B b) {}
  public static void main(String[] args)
  {
    X x = new X();
    Y y = new Y();
    takesA(x);
    takesA(y);
    takesB(x);
    takesB(y.makeB());
  }
}
```

2) But if we have two classes, one is concrete and another is abstract then inner class is the best option.

```
package ch18InnerClasses;
```

```
class D {}
abstract class E {}

class Z extends D
{
E makeE() { return new E() {}; }
}

public class MultInhThruMultImpl
{
static void takesD(D d) {}
static void takesE(E e) {}
public static void main(String[] args)
	{
		Z z = new Z();
		takesD(z);
		takesE(z.makeE());
	}
}
```

➔ If one don't want to address the problem of multi inheritance then he can do away with inner classes.

7. <u>Inheriting From Inner Classes</u>

➔ Because inner class constructor is attached to outer class reference hence inheriting inner class is sightly complicated.

➔ Because secret outer class reference must be initialized, hence there is a special syntax for this.

```
package ch18InnerClasses;

class InnerInheritanceOuter
{
	public void outFn()
	{
		System.out.println("Inside Outer Function");
	}

	class InnerInheritanceInner
	{
		public void inFn()
		{
			System.out.println("Inside Inner Function");
		}
	}
}

public class InnerInheritanceMain extends InnerInheritanceOuter.InnerInheritanceInner
{
	//InnerInheritanceMain(){}//default constructor is not allowed.

	InnerInheritanceMain(InnerInheritanceOuter out)
	{
		out.super();
	}

	public static void main(String[] args)
	{
		InnerInheritanceOuter out = new InnerInheritanceOuter();

		InnerInheritanceMain inm = new InnerInheritanceMain(out);

		inm.inFn();
	}
```

```
}
Output:
Inside Inner Function
```

➔ But Inner class needs reference to the enclosing class hence it is passed through constructor using a special syntax. `enclosingClassReference.super()`

8. Overridding Inner Class

➔ What happens when we extends the outer class and then completely overrides the inner class. Checkout the example shown below.

```
package ch18InnerClasses;

class Egg
{
	private Yolk y;

	protected class Yolk
	{
		public Yolk() { System.out.println("Egg.Yolk()"); }
	}

	public Egg()
	{
		System.out.println("New Egg()");
		y = new Yolk();
	}
}

public class OverrideInnerBigEgg extends Egg
{
	public class Yolk
	{
		public Yolk()
		{
			System.out.println("BigEgg.Yolk()");
		}
	}
	public static void main(String[] args)
	{
	new OverrideInnerBigEgg();
	}
}
Output:
New Egg()
Egg.Yolk()
```

➔ The above output proves that all inner and outer classes are completely separate entities. But one can explicitly inherit from the inner class as shown below.

```
package ch18InnerClasses;

class Egg2
{
	protected class Yolk
	 {
	 public Yolk()
	 {
		System.out.println("Egg2.Yolk()");
	 }
	 public void f()
	 {
		System.out.println("Egg2.Yolk.f()");
```

```
        }
}

private Yolk y = new Yolk();

        public Egg2()
        {
         System.out.println("New Egg2()");
        }
        public void insertYolk(Yolk yy)
        {
                y = yy;
        }
        public void g()
        {
                y.f();
        }
}

public class OverrideInnerBigEgg2 extends Egg2
{
        public class Yolk extends Egg2.Yolk
        {
        public Yolk()
        {
                System.out.println("BigEgg2.Yolk()");
        }
        public void f()
        {
                System.out.println("BigEgg2.Yolk.f()");
        }
}
        public OverrideInnerBigEgg2()
        {
                insertYolk(new Yolk());
        }
        public static void main(String[] args)
        {
        Egg2 e2 = new OverrideInnerBigEgg2();
        e2.g();
        }
}
Output:
Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
```