

Lesson 7

Nested and Inner Class

Wholeness of the Lesson






- Inner and nested classes allow classes to play the roles of instance variable, static variable and local variable, providing more expressive power to the Java language. Likewise, it is the hidden, unmanifest dynamics of consciousness that are responsible for the huge variety of expressions in the manifest world.

Introduction

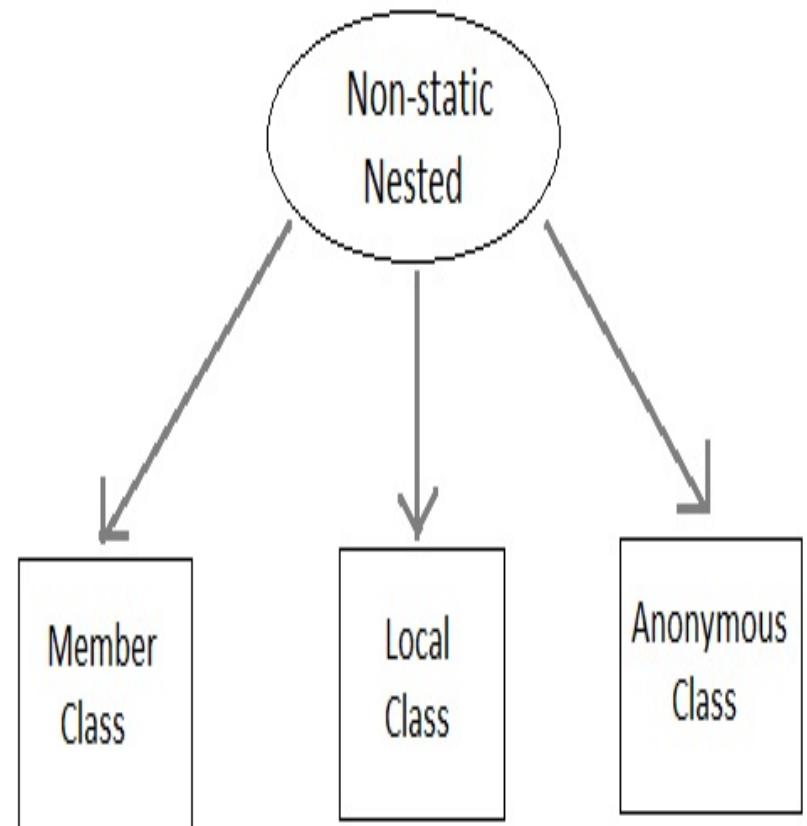
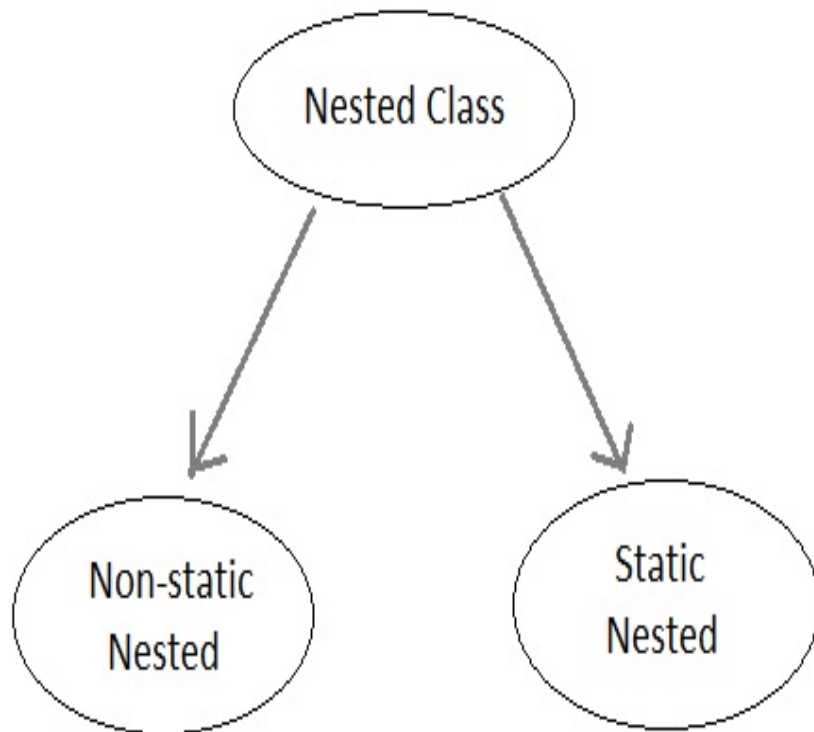
- A nested class *is a class defined within the scope of another class.*
- A nested class is an *inner class* if it has access to all members and methods of its enclosing class.
- The inside class is called an **inner class** and the **enclosing class** is called an **outer class**.
- A simple use of inner classes is to combine dependent classes into a primary class.
- *Inner classes are useful for defining listener classes.*
- Mostly used in GUI programming to handle events.

Nested and Inner classes

The compiler adds code to nested class definitions and to the enclosing class to support the features of nested classes. When you compile the `UserIO` class (which has two member inner classes), you will see the inner classes explicitly named in `.class` files, using a '\$' in their names to distinguish them as nested classes.

 <code>UserIO\$1.class</code>	1 KB	CLASS File
 <code>UserIO\$ClearListener.class</code>	2 KB	CLASS File
 <code>UserIO\$SubmitListener.class</code>	1 KB	CLASS File
 <code>UserIO.class</code>	5 KB	CLASS File
 <code>UserIO.java</code>	6 KB	JAVA File

Nested class types



Four kind of nested classes

1. Inside the Class (Member Class)
2. Inside the Method(Local Class)
3. Anonymous Class
4. Static Inner Class

Main point

- Classes are the fundamental notion in Java – programs are built from classes. With nested classes, Java makes it possible for this fundamental construct to play the roles of instance variable (member inner classes), static variable (static nested classes), and local variable (local inner classes). Likewise, in the unfoldment of creation, pure intelligence assumes the role of creative intelligence – in all of creation we find pure intelligence.

Member Class

- Member inner classes, like other members of the class, can be declared public or private, or may have package level access or may be protected.
- Member inner classes have access to all fields and methods of the enclosing class, including private fields and methods. **No explicit reference to an enclosing class instance is needed.**
- Likewise, the outer class can access private variables and methods in the inner class, **but only with reference to an inner class instance.**
- In a member inner class, no variable or method may be declared static. (By contrast, static members are allowed in a **static nested class.**)

Syntax - Member Class

```
class OuterClass
```

```
{
```

```
    Data members;
```

```
    Method functions;
```

```
class InnerClass
```

```
{
```

```
    Data members;
```

```
    Method functions;
```

```
}
```

```
}
```

```
OuterClass$InnerClass.Class
```

Example

//Blue outer class & Black inner class

```
public class MyClass {  
    private String s = "hello";  
    public static void main(String[] args){  
        new MyClass();  
    }  
    MyClass() {  
        MyInnerClass myInner = new MyInnerClass();  
        System.out.println(myInner.intval);  
        myInner.innerMethod();  
    }  
    private class MyInnerClass{  
        private int intval = 3;  
        private void innerMethod(){  
            System.out.println(s);  
        }  
    }  
}
```

Output:

3

hello

Member class

- If the member inner class is sufficiently accessible (i.e., not private), it can be instantiated by a class other than the enclosing class.

Example

```
class Outer
{
    private String text1 = "I am Outer
        Class!";
    public void callinner()
    {
        Inner iobj = new Inner();
        iobj.printText();
    }
}
class Inner
{
    private String text2 = " I am Inner
        Class!";
    void printText()
    {
        System.out.println(text1);
        System.out.println(text2);}
}
```

```
class Main()
{
    public static void main(String args[])
    {
        // Object for outer class
        Outer ot = new Outer();
        ot.callinner();
        // object for inner class
        Outer.Inner obj = new Outer().new
            Inner();
        Obj.printText();
    }
}
```

Output

```
I am Outer Class!
I am Inner Class!
I am Outer Class!
I am Inner Class!
```

this and constructors

- Like ordinary classes, when a member inner class is instantiated, it has an implicit parameter `this`.
- The “`this`” of the enclosing class is accessible from within itself (as ever), and also from within the inner class.
- The “`this`” of the inner class is accessible from within itself, but *not* from the enclosing class.

this and constructors

```
class MyOuterClass {
    MyInnerClass inner;
    private String param;
    MyOuterClass(String param) {
        inner = new MyInnerClass("innerStr");
        this.param = param; // the outer class
        version of this
    }
    void outerMethod() {
        System.out.println(inner.param);
        inner.innerMethod();
    }
    private class MyInnerClass{
        private String param;
```

```
        MyInnerClass(String param) {
            //the inner class version of 'this'
            this.param = param;
        }
        void innerMethod() {
            //accessing enclosing class's version of this
            System.out.println(MyOuterClass.this.param);
            //Invoke inner class parameter
            System.out.println(param);
        }
    }
    public static void main(String[] args) {
        (new MyOuterClass("outerStr")).outerMethod();
        // the above line is similar to the below
        //commented line
        /*MyOuterClass obj = new
        MyOuterClass("outerStr");
        obj.outerMethod();*/
    }
    Output
    innerStr
    outerStr
    innerStr
```

Member class

- **Best Practice.** A member inner class is typically a small specialized “assistant” that is exclusively “owned by” its enclosing class.
- Often used in a GUI class to enclose event handlers, though listeners.
- Consequently, it is not good practice to access a member inner class from outside the enclosing class, since this undermines the purpose.

Main Point

- Inner classes – a special kind of nested class -- have access to the private members of their enclosing class. The most commonly used kind of inner class is a *member* inner class. Likewise, when individual awareness is awake to its fully expanded, self-referral state, the memory of its eternal and infinitely dynamic nature becomes lively.

Quiz – 1 – What is the Output?

```
public class ShadowTest {  
    public int x = 0;  
    class FirstLevel {  
        public int x = 1;  
        void methodInFirstLevel(int x) {  
            System.out.println("x = " + x);  
            System.out.println("this.x = " + this.x);  
            System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);  
        }  
    }  
  
    public static void main(String[] args) {  
        ShadowTest st = new ShadowTest();  
        ShadowTest.FirstLevel fl = st.new FirstLevel();  
        fl.methodInFirstLevel(23);  
    }  
}
```

Output :

Local class

- Inner classes are defined entirely within the body of a method.
- Access specifiers (public, private, etc) are not used to affect access of the inner class since access to the inner class is always restricted to the local access within the method body.
- Inner classes have access to instance variables and methods in the enclosing class; they also have access to *local variables*.

Example :

```
class Outer {  
    public void printText() {  
        class Local {...}  
        Local local = new Local();  
    }  
}
```

LocalClass.java

Local class - Example

```
class Out
{
    int y=20;
    public void display()
    {
        class Inner //Inner class defined inside method
        {
            int x=10;
            public void show()
            {
                System.out.println("x + y : "+ (x+y));
            }
        }
        Inner in = new Inner();
        in.show();
    }
}
```

```
public class LocalClass {
    public static void main(String[]
args) {
        Out ot = new Out();
        ot.display();
    }
}
```

Output :
x + y : 30

Anonymous classes

- *An anonymous inner class is an inner class without a name. It combines defining an inner class and creating an instance of the class into one step.*

Implementing an ActionListener with an Anonymous Inner Class

```
public class AnonymousClassDemo
extends JFrame {
    public AnonymousClassDemo() {
        // Create a button
        JButton jbtPrint = new JButton("Print");
        // Create a panel to hold buttons
        JPanel panel = new JPanel();
        panel.add(jbtPrint);
        add(panel);
        jbtPrint.addActionListener(new
        ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                System.out.println("Process Print");
            }
        });
    }
}
```

```
public static void main(String[]
args) {
    AnonymousClassDemo frame = new
    AnonymousClassDemo();

    frame.setTitle("AnonymousListenerDe
mo");

    frame.setDefaultCloseOperation(JFra
me.EXIT_ON_CLOSE);
    frame.setVisible(true);
}
}
```

Interface and abstract implementation using Anonymous class

```
interface Printable{  
    void print(String msg);  
}  
  
public class Ainterface{  
    public static void main (String[] args){  
        Printable printer = new Printable(){  
            public void print(String msg) {  
                System.out.println(msg);  
            }  
        };  
        printer.print("Have a nice Day!");  
    }  
}
```

Static Inner class

- An inner class can be declared using the static modifier are called nested static classes in java.
- The instance of static inner class is created without the reference of Outer class, which means it does not have enclosing instance.
- A static nested class cannot invoke non-static methods or access non-static fields.
- It can have static and non-static fields and method.
- Eg : StaticDemo.java

```
public class Main {  
    public int i = 4;  
    public int getInt() {  
        return 3;  
    }  
    static class NestedClass {  
        public void innerMethod() {  
            int j = i;    //compiler error  
            int k = getInt(); //compiler error  
        }  
    }  
}
```


Static - Example

```
public class StaticDemo {
    static int data=30;
    int data1 = 100;
    void print()
    {
        System.out.println("data is "+data1);
    }
    static class Inner{
        static int x=50;
        static void msg()
        {
            System.out.println("data is "+x);
            System.out.println("data is "+data);
        }
        // System.out.println("data is "+data1); //Error
    }
    void display()
    {
        System.out.println("Static Method Demo");
    }
}
```

```
public static void main(String[] args) {
    new StaticDemo().print();
    Inner.msg();
    Inner obj1=new Inner();
    obj1.display();
}
}
```

Output

```
data is 100
data is 50
data is 30
Static Method Demo
```

Application: Implementing Singleton Using Nested Class

There are several ways to implement the singleton pattern:

- Use lazy initialization to instantiate a private static instance variable. (Not thread safe.) (Concept we learned in Lesson-3) – [\(SingleThreadedSingleton.java\)](#)
- Store instance as a public static constant, constructed when class is loaded. [\(SingletonAsPublicStatic.java\)](#)
- Implement as an Enum. Also constructed when enum is loaded. [\(SingletonAsEnum.java\)](#)
- Use Spring's Singleton Holder pattern, whereby the singleton is stored as a static variable of a static nested class. Permits lazy initialization and is thread safe. [\(SingletonAsInnerClass.java\)](#)

Lambda Expressions

1. An interface that contains just one (abstract) method is called a *functional interface*.
2. An implementation of a functional interface is called a *functor*
3. A *closure* is a functor that remembers the state of its enclosing environment
4. Example: ActionListener
 - ActionListener is an interface with just one method – actionPerformed, so it is a **functional interface**
 - An anonymous inner class **implementation of ActionListener** is a **functor**
 - An anonymous inner class **implementation of actionPerformed** is a **closure**
 - A lambda expression can be used in place of an anonymous inner class
5. The Java runtime determines the type of the function argument evt by context – a mechanism that is called *target typing*. Note: Target typing relies on the fact that lambda expressions are used only with *functional interfaces*; there is just one possible function that is to be implemented, so the JVM can figure out the necessary typing on this basis.

Example

// Without Lambda

```
button.addActionListener(new ActionListener() {  
public void actionPerformed(ActionEvent e) {  
    System.out.println("Process Print");  
} });
```

// Using Lambda

```
button.addActionListener(  
    evt -> {  
        System.out.println("Process Print");  
    });
```

Demo Code

- MemberClass.java
- LocalClass.java
- StaticDemo.java
- AnonymousInterfaceDemo.java
- MyFrameAnonymous.java

UNITY CHART

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Inner class is an integral part of the unlimited potential

A nested class is a class that is defined inside another class.

An inner class has full access to its context, its enclosing class.

Transcendental Consciousness: TC is the unbounded context for individual awareness.

Impulses within the Transcendental field: *The impulses within the transcendental field are the simplest (and most powerful) ones to achieve the desired result.*

Wholeness moving within Itself: *When individual awareness is permanently and fully established in its transcendental "context" – pure consciousness – every impulse of creation is seen to be an impulse of one's own awareness.*



PRACTICE
