Nested class

Nested Class

The Java programming language allows us to define a class within another class. Such a class is called a *nested class*.

The purpose of a nested class is to clearly group the nested class with its surrounding class, signaling that these two classes are to be used together. Or perhaps that the nested class is only to be used from inside its enclosing (owning) class.

Example:

```
class OuterClass
{
                ...
    class NestedClass
    {
                ...
    }
}
```

Why Use Nested Classes?

- Logical grouping of classes—If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together.

- Increased encapsulation—Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

- More readable, maintainable code—Nesting small classes within top-level classes places the code closer to where it is used.

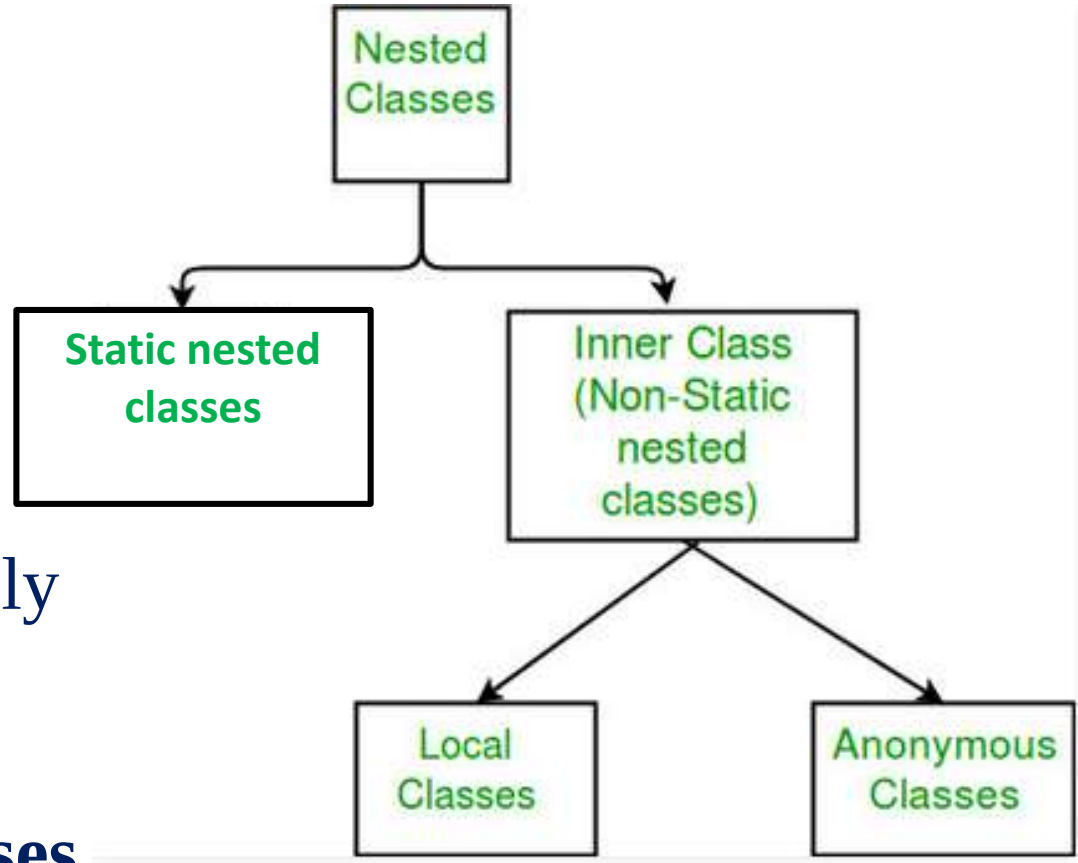
Nested Class Benefits

The benefits of Java nested classes are that you can group classes together that belong together. You could do so already by putting them in the same package, but putting one class inside another makes an even stronger grouping.

A nested class is typically only used *by* or *with* its enclosing class. Sometimes a nested class is only visible to the enclosing class, is only used internally, and is thus never visible outside the enclosing class. Other times the nested class is visible outside its enclosing class, but can only be used in conjunction with the enclosing class.

Types of Nested Classes

- A nested class is a member of its enclosing class.
- Nested classes are divided into two categories:
  - static
  - non-static
- Nested classes that are declared static are simply called **static nested classes**.
- Non-static nested classes are called **inner classes**.



Static Nested Classes

- A static nested class is associated with its outer class similar to class methods and variables.
- A static nested class cannot refer directly to instance variables or methods defined in its enclosing class.
- It can use them only through an object reference.
- Static nested classes are accessed using the enclosing class name:

  OuterClass.StaticNestedClass
- For example, to create an object for the static nested class, use this syntax:

  OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

Example

```
public class test1
{
    public static void main(String args[])
    {
        Outer.Inner inn = new Outer.Inner();    //creating object of nested class which is static
        inn.display();                          //calling function of nested class
    }
}


class Outer
{
    static class Inner          //static nested class
    {
        public void display()
        {
            System.out.println("This is nested class function");
        }
    }
}
```

Example 1

```
class Outside
{
    int outVar;
    static class Inside           //static nested class
    {
        void show() {
            //outVar = 23;        //can't access non-static members directly
            Outside o = new Outside();
            o.outVar = 23;
            System.out.println(o.outVar);
        }
    }
}

class Temp
{
      public static void main(String args[])
      {
           Outside.Inside inn =  new Outside.Inside();  //nested class object
           inn.show();
      }
}
```

Example 2

```
class Outside
{
    static int outVar;
    static class Inside          //static nested class
    {
        void show() {
            outVar = 23;
            System.out.println(outVar);
        }

        static void fun() {
            System.out.println("Fun method");
        }
    }
}

class Temp
{
      public static void main(String args[])
      {
           Outside.Inside inn =  new Outside.Inside();  //nested class object
           inn.show();
           Outside.Inside.fun();
      }
}
```

Inner Classes (non-static nested classes)

- An inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields.

- Because an inner class is associated with an instance, it cannot define any static members itself.

- Objects that are instances of an inner class exist *within* an instance of the outer class.
- Consider the following classes:

```
class OuterClass {
        ...
        class InnerClass { ... }
}
```

- An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance.

- To instantiate an inner class, we must first instantiate the outer class. Then, create the inner object within the outer object.

- Syntax:

```
OuterClass.InnerClass  innerObject  =   outerObject.new  InnerClass();
```

Example

```
class Employee {
    int id;
    int salary;

    class PersonalInformation {
        String name;
        long phoneNo;
    }
}

class Temp {
    public static void main(String args[])
    {
        Employee ep = new Employee();
        ep.id = 1034;
        ep.salary = 20000;

        Employee.PersonalInformation epInfo = ep.new PersonalInformation();
        epInfo.name = "Max";
        epInfo.phoneNo = 123456789;
    }
}
```

Example 1

```
class Outside
{
    int outVar;
    class Inside     //Inner class (non-static nested class)
    {
        void show() {
            outVar = 23;
            System.out.println(outVar);
        }
    }
}

class Temp
{
      public static void main(String args[])
      {
            Outside out = new Outside();              //outer-class object
            Outside.Inside inn =  out.new Inside();  //nested class object
            inn.show();
      }
}
```

Example 2

```
class Outside
{
    int outVar;
    class Inside     //Inner class (non-static nested class)
    {
        void show() {
            outVar = 23;
            System.out.println(outVar);
        }

        //can't create static method in non-static nested class
        static void fun() {

        }
    }
}

class Temp
{
      public static void main(String args[])
      {
           Outside out = new Outside();              //outer-class object
           Outside.Inside inn =  out.new Inside();  //nested class object
           inn.show();
      }
}
```

Example 3 (accessibility of nested classes in method)

```
class Outer
{
    void outerDisplay()
    {
        Inner inn = new Inner();
        Nested nest = new Nested();
        nest.innerDisplay();
    }

    class Inner     //Inner class (non-static nested class)
    {
        void nestedDisplay()
        {
            System.out.println("Inner function");
        }
    }

    static class Nested    //static nested class
    {
        void innerDisplay()
        {
            System.out.println("Nested function");
        }
    }
}
```

Example 4 (accessibility)

```
class Outer
{
    class Inner     //Inner class (non-static nested class)
    {
        void innerDisplay()
        {
            //Nested obj = new Nested();     //no error
            System.out.print("Inner");
        }
    }

    static class Nested     //static nested class
    {
        void nestedDisplay()
        {
            //Inner inn = new Inner();       //error
            Outer o = new Outer();
            Inner obj = o.new Inner();
            obj.innerDisplay();
        }
    }
}

public class Test
{
    public static void main(String args[])
    {
        Outer.Nested obj = new Outer.Nested();
        obj.nestedDisplay();
    }
}
```

Example 5 (deeply nesting classes)

```
class Outer
{
    class Inner     //Inner class (non-static nested class)
    {
        class Inner2   //Inner class (non-static nested class)
        {
            void nestedDisplay()
            {
                System.out.println("Max");
            }
        }
    }
}

public class Test
{
    public static void main(String args[])
    {
        Outer o = new Outer();
        Outer.Inner inn = o.new Inner();
        Outer.Inner.Inner2 obj = inn.new Inner2();
        obj.nestedDisplay();
    }
}
```

Classification of Inner classes

- Additionally, there are two special kinds of inner classes:
  - local classes and
  - anonymous classes (also called anonymous inner classes).

Local Classes

- Local classes are *classes that are defined in a block*, which is a group of zero or more statements between balanced braces.
- For example, we can define a local class in a method body, a for loop, or an if clause.
- A local class has access to the members of its enclosing class.
- A local class has access to local variables. However, a local class can only access local variables that are declared final.

Example 1 (creating and accessing local class)

```
public class test1
{
    public static void main(String args[])
    {
        Outer out = new Outer();
        out.display();
    }
}


class Outer
{
    int outVar = 10;
    public void display()
    {
            class Inner                     //local class inside a function
            {
                public void show()
                {
                    System.out.println("This is local class function");
                    outVar = 100 * 2;   //outer class variable can be accessed
                }
            }

            Inner inn = new Inner();    //creating object of local class
            inn.show();                 //calling function of local class
    }
}
```

Example 2 (accessing local variables)

```
class Outer
{
    public void display()
    {
        int localVar1 = 2;      //local variable

        class Inner             //local class
        {
            public void show()
            {
                /* Only final or effectively final variables
                   can be accessed in local-class */
                //localVar1 = localVar1 * 20;   //error, can't modify

                System.out.println(localVar1);
            }
        }

        //localVar1 = localVar1 * 20;   //error, can't modify
        Inner inn = new Inner();   //creating object of local class
        inn.show();                //calling local class method
    }
}
```

Example 3 (accessing other local class)

```
class Outer
{
    public void display()
    {
        class local1                    //local class
        {
            public void show1(){
                System.out.println("local-1");
            }
        }

        class local2                    //local class
        {
            public void show2(){
                local1 obj = new local1();  //creating local class object
                obj.show1();                //calling method of local class
            }
        }

        local2 obj = new local2();  //creating local class object
        obj.show2();                //calling method of local class
    }
}
```

Anonymous Class

Anonymous Classes

- A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:
  - class (may be abstract or concrete).
  - Interface

- Anonymous classes enable us **to declare and instantiate a class at the same time.**

- They are like local classes except that they do not have a name.

- The anonymous class expression consists of the following:

  1. The new operator
  2. The name of an interface to implement or a class to extend.
  3. Parentheses that contain the arguments to a constructor, just like a normal class instance creation expression.
  4. A body, which is a class declaration body. More specifically, in the body, method declarations are allowed but statements are not.

Example

```
class Shape
{
    void show() {
        System.out.println("Parent-show");
    }
}

class Test
{
    public static void main(String a[])
    {
        Shape s = new Shape() {
            void show() {
                System.out.println("Child-show");
            }
        };

        s.show();
    }
}
```

- "**s**" is the reference variable of superclass **Shape**
- It is used to hold the instance (i.e. object) of subclass which is anonymous.
- In other words, the **Shape** reference variable "**s**" refers *not* to an instance of **Shape** class but to an instance of an anonymous inner subclass of **Shape**.
- It overrides one or more methods of the super class on the fly.

Reference variable of parent

Anonymous class (i.e. child class) object being created

Anonymous class (i.e. child class having no name) being created and overriding parent-class method

Exercise

Write a program to create an class **Color** having abstract method:

void showColor()

which will be used to print a particular color. Now make 3 anonymous class objects, each of which prints different color.

Anonymous inner class

- An anonymous inner class is not an independent inner class.
- It is a *sub-class* of either a *class* type or an anonymous *implementer* of the specified *interface* type.
- So, when anonymous inner classes are in picture, *polymorphism* must be there. And when polymorphism is there, you can only call members that are defined in the parent using parent class reference.
- Java's anonymous inner classes being sub-classes or implementers do strictly adhere to the polymorphism rules.

Exercise

Create an interface **Colors** having method: **void showColor()**

Implement the method of this interface using anonymous class object and invoke it.

Now create at least 3 anonymous class objects and pass them one-by-one to a method called display() which will invoke the overridden methods of anonymous class objects.

Example (using interface)

```
interface Shape
{
    void showColor();
}

public class Test
{
    public static void main(String args[])
    {
        Shape s = new Shape(){
                        public void showColor()
                        {   System.out.println("Red");  }
                    };

        s.showColor();
    }
}
```

Key points

➢ Anonymous classes have the same access to local variables of the enclosing scope as local classes:

- An anonymous class has access to the members of its enclosing class.
- An anonymous class cannot access local variables in its enclosing scope that are not **effectively final**.

➢ Anonymous classes also have the same restrictions as local classes with respect to their members:

- We cannot declare static initializers or member interfaces in an anonymous class.
- An anonymous class can have static members provided that they are constant variables.

➢ Note that we can declare the following in anonymous classes:

- Fields
- Extra methods (even if they do not implement any methods of the supertype)
- Local classes
- we cannot declare constructors in an anonymous class.

Example 1 (accessing parent's variable)

```
public class Computer
{
    public static void main(String []args)
    {
        Student s = new Student()
        {
            void show() {System.out.println("This is anonoymous class function");}
        };                                  //creating anonymous inner class
                                            //it is a like creating a sub-class
        s.a = 100;
        System.out.println(s.a);            //displays 100
        s.show();                           //displays overridden function
    }
}


class Student
{
    int a=10;
    void show()
    {
        System.out.println("This is student class function");
    }
}
```

Example 2 (declaring, instantiating & passing anonymous class object)

```
class Peripheral
{
    int var = 10;
    void show()
    {
        System.out.println("Parent function");
    }
}

public class Test
{
    public static void main(String[] args)
    {
        Test t = new Test();
        t.compute(  new Peripheral()
                    {
                        void show(){ System.out.println("Overriden function"); }
                        void newFunction(){}
                    }     //declaring and instantiating class (sub-class)
                );
    }

    void compute(Peripheral obj)
    {
        obj.show();          //accessing overriden function
        obj.newFunction();   //error, new member not accessible
    }
}
```

Anonymous inner class

- Remember, **anonymous inner classes are inherited ones**, and we always use a superclass reference variable to refer to an anonymous subclass object.

- And, we can only call methods on an anonymous inner class object that are defined in the superclass.

- Though, we can introduce new methods in anonymous inner class, but we cannot access them through the reference variable of superclass because superclass does not know anything about new methods or data members introduced in subclass.