Nested Classes

AUGUST 20, 2015

A class inside another class is called a **Nested Class**. In other words, as a class has member variables and member methods it can also have member classes.

Nested classes are divided into two categories: **static** and **non-static**. Nested classes that are declared static are called **static nested classes**. Non-static nested classes are called **inner classes**.

You can say inner classes are of 3 types:

- “Regular” Inner classes
- Method-local inner classes
- Anonymous inner classes

Why Use Nested Classes?

- **It is a way of logically grouping classes that are only used in one place**: 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. Nesting such “helper classes” makes their package more streamlined.
- **It increases 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.
- **It can lead to more readable and maintainable code**: Nesting small classes within top-level classes places the code closer to where it is used.

Inner classes

These are just “regular” inner classes which are not method-local, anonymous or static.

Little Basics

Suppose you have an inner class like this:

```
class MyOuter {
    class MyInner { }
}
```

When you compile it:

```
javac MyOuter.java
```

you will end up with two class files:

```
MyOuter.class
MyOuter$MyInner.class
```

The inner class is still, in the end, a separate class, so a separate class file is generated for it. But the inner class file isn't accessible to you in the usual way. You can't say

```
java MyOuter$MyInner
```

in hopes of running the `main()` method of the inner class, because a regular inner class cannot have static declarations of any kind. The only way you can access the inner class is through a live instance of the outer class. In other words, only at runtime, when there's already an instance of the outer class to tie the inner class instance to.

Now with the basics done, let's see an inner class performing some actions:

```java
class MyOuter {
    private int x = 7;

    // inner class definition
    class MyInner {
        public void seeOuter() {
            System.out.println("Outer x is " + x);
        }
    } // close inner class definition
} // close outer class
```

The output of the above program would be `Outer x is 7`. This happens because an **inner class can access private members of its outer class even those which are declared as `static`**. The inner class is also a member of the outer class. So just as any member of the outer class (say, an instance method) can access any other member of the outer class, private or not, the inner class (also a member) can do the same.

Instantiate the inner class

To create an instance of an inner class, you must have an instance of the outer class to tie to the inner class. There are no exceptions to this rule: **An inner class instance can never stand alone without a direct relationship to an instance of the outer class**.

Instantiating an Inner Class from Within the Outer Class

```java
class MyOuter {
    private int x = 7;

    public void makeInner() {
        MyInner in = new MyInner();  // make an inner instance
        in.seeOuter();
    }

    class MyInner {
        public void seeOuter() {
            System.out.println("Outer x is " + x);
        }
    }
}
```

The reason the above syntax works is because the outer class instance method code is doing the instantiating. In other words, there's already an instance of the outer class i.e, the instance running the makeInner() method.

Instantiating an Inner Class from Outside the Outer Class Instance Code

```java
public static void main(String[] args) {
    MyOuter mo = new MyOuter(); // gotta get an instance of outer class
    MyOuter.MyInner inner = mo.new MyInner(); // instantiate inner class from outer class instar
    inner.seeOuter();
}
```

As we know that we must have an outer class instance to create an inner class instance, the above code would be the way to go. *Instantiating an inner class is the only scenario in which you'll invoke new on an instance as opposed to invoking new to construct an instance.*

If you are a one liner then the below code is for you:

```java
public static void main(String[] args) {
     MyOuter.MyInner inner = new MyOuter().new MyInner(); // all in one line
     inner.seeOuter();
}
```

Referencing the Inner or Outer Instance from Within the Inner Class

An object refers to itself normally by using the `this` reference. The `this` reference is the way an object can pass a reference to itself to some other code as a method argument:

```
public void myMethod() {
    MyClass mc = new MyClass();
    mc.doStuff(this); // pass a ref to object running myMethod
}
```

So within an inner class code, the `this` reference refers to the instance of the inner class, as you'd probably expect, since `this` always refers to the currently executing object. But when the inner class code wants an explicit reference to the outer class instance that the inner instance is tied to, it can access the outer class `this` like shown below:

```java
class MyOuter {
    private int x = 7;

    public void makeInner() {
        MyInner in = new MyInner();
        in.seeOuter();
    }

    class MyInner {
        public void seeOuter() {
            System.out.println("Outer x is " + x);
            System.out.println("Inner class ref is " + this);
            System.out.println("Outer class ref is " + MyOuter.this);
        }
    }

    public static void main(String[] args) {
        MyOuter.MyInner inner = new MyOuter().new MyInner();
        inner.seeOuter(); // works fine
        System.out.println(inner.x); // compiler error as you can't directly
                                     // access outer class member through a
                                     // inner class reference (you can have
                                     // getter & setter in the inner class)
    }
}
```

Some points to note

- You can refer to this StackOverflow question to understand how you can and can't access outer class members from inner class.
- Inner class can access **private members** of the outer enclosing class even those **which are declared `static`**.
- Normally the inner class code doesn't need a reference to the outer class, since it already has an implicit one it's using to access the members of the outer class, it would need a reference to the outer class if it needed to pass that reference to some other code as in the above example.

Method-Local Inner Classes

A regular inner class scoped inside another class's curly braces, but outside any method code (in other words, at the same level that an instance variable is declared) is called a **method-local inner class**.

```java
class Outer {
    private String x = "Outer";

    void doStuff() {

        class Inner {

            public void seeOuter() {
                System.out.println("Outer x is " + x);
            } // close inner class method

        } // close inner class definition
    }
}
```

In the above example, `class Inner` is the method-local inner class. But the inner class is useless because you are never instantiating the inner class. Just because you declared the class doesn't mean you created an instance of it. So to use the inner class, you must make an instance of it somewhere within the method but **below the inner class definition (or the compiler won't be able to find the inner class)**. You can even refer my StackOverflow question.

The following legal code shows how to instantiate and use a method-local inner class:

```
class Outer {
    private String x = "Outer";

    void doStuff() {

        class Inner {

            public void seeOuter() {
                System.out.println("Outer x is " + x);
            } // close inner class method

            MyInner mi = new MyInner();  // This line must come
                                         // after the class

            mi.seeOuter();

        } // close inner class definition
    }
}
```

What a Method-Local Inner Object Can and Can't Do

A method-local inner class can be instantiated only within the method where the inner class is defined. In other words, no other code running in any other method inside or outside the outer class can ever instantiate the method-local inner class.

Can	**Cannot**
Access private members of the outer/enclosing class.	Cannot use the local variables of the method the inner class is in.
The only modifiers you can apply to a method-local inner class are `abstract` and `final`, but, as always, never both at the same time.	You can't mark a method-local inner class `public`, `private`, `protected`, `static` or `transient` (just like local variable declaration).

Important Points

- Method-local inner class **cannot access the local variables of the method inside which it is defined unless the variables are marked `final`**. This is because the local variables of the method live on the stack and exist only for the lifetime of the method. The scope of a local variable is limited to the method the variable is declared in. When the method ends, the stack frame is blown away and the variable is history. But even after the method completes, the inner class object created within it might still be alive on the heap if, for example, a reference to it was passed into some other code and then stored in an instance variable. Because the local variables aren't guaranteed to be alive as long as the method-local inner class object is, the inner class object can't use them. Unless the local variables are marked final.

```java
class MyOuter {
    private String x = "Outer";

    void doStuff() {
        String z = "local variable";

        class MyInner {
            public void seeOuter() {
                System.out.println("Outer x is " + x);
                System.out.println("Local var z is " + z);  // won't compile, making 'z' fir
                                                            // will solve the problem
            }
        } // close method-local inner class
    }
}
```

- A local class declared in a `static` method has access to only `static` members of the enclosing class, since there is no associated instance of the enclosing class. If you're in a `static` method, there is no `this`, so an inner class in a `static` method is subject to the same restrictions as the `static` method. In other words, no access to instance variables.

Anonymous Inner Classes

Inner class declared without any class name at all is known as **Anonymous Inner Class**. These can be seen as two types:

- Plain/Normal Anonymous Inner Class
- Argument Defined Anonymous Inner Class

Plain/Normal Anonymous Inner Class also comes in two flavors:

- Flavor 1:

```java
class Popcorn {
    public void pop() {
        System.out.println("popcorn");
    }
}
class Food {
    Popcorn p = new Popcorn() {
        public void pop() {
            System.out.println("this method overrides pop() in Popcorn class");
        }
        public void push() {
            System.out.println("new method in anonymous inner class");
        }
    };
}
```

In the above code, the `Popcorn` reference variable **DOES NOT** refer to an instance of `Popcorn`, but to an **instance of an anonymous (unnamed) subclass of `Popcorn`**.

Polymorphism comes to play in Anonymous Inner Class as in the above example, we're using a superclass reference variable type to refer to a subclass object. And when you call `p.pop()` then the `pop()` inside `Food` class will be called instead of the one in `Popcorn` class.

But we need to keep one thing in mind, i.e, you can only call methods on an anonymous inner class reference that are defined in the reference variable type. So, in the above code, you cannot call `p.push()` as `p` is a reference variable of type `Popcorn` and `Popcorn` class does not have any method named `push()`.

- Flavor 2:

The only difference between flavor one and flavor two is that flavor one creates an anonymous subclass of the specified class type, whereas flavor two creates an **anonymous implementer of the specified interface type**.

```java
interface Cookable {
    public void cook();
}

class Food {
    Cookable c = new Cookable() {
        public void cook() {
            System.out.println("anonymous cookable implementer");
        }
    };
}
```

This is the only time you will ever see the syntax `new Cookable()` where `Cookable` is an interface rather than a non-abstract class type. Think about it: You can't instantiate an interface, yet that's what the above code looks like it's doing. But, of course, it's not instantiating a `Cookable` object, it's creating an instance of a new anonymous implementer of `Cookable`.

Some obvious points:

- Anonymous Inner Class can implement only one interface. There simply isn't any mechanism to say that your anonymous inner class is going to implement multiple interfaces.
- Anonymous Inner Class can't extend a class and implement an interface at the same time.
- If the Anonymous Inner Class is a subclass of a class type, it automatically becomes an implementer of any interfaces implemented by the superclass.

Argument Defined Anonymous Inner Class

Consider the below code:

```java
 1  class MyWonderfulClass {
 2      void go() {
 3          Bar b = new Bar();
 4          b.doStuff(new Foo() {
 5              public void foof() {
 6                  System.out.println("foofy");
 7              } // end foof method
 8          }); // end inner class def, arg, and b.doStuff stmt.
 9      } // end go()
10  }
11
12  interface Foo {
13      void foof();
14  }
15
16  class Bar {
17      void doStuff(Foo f) {
18      }
19  }
```

The `doStuff(Foo f)` in `Bar` class takes an object of a class that implements `Foo` interface. So, we just passed an anonymous class which is an implementation of the `Foo` interface as an argument to the `doStuff()` method (in line 4). This we call it as **Argument Defined Anonymous Class**.

Please note that argument defined anonymous class end like `});` but normal anonymous class end like `};`.

Static Nested Class

Static Nested Classes are sometimes referred to as static inner classes, but they really aren't inner classes at all based on the standard definition of an inner class. While an inner class (regardless of the flavor) enjoys that special relationship with the outer class (or rather, the instances of the two classes share a relationship), a static nested class does not. It is simply **a non-inner (also called "top-level") class scoped within another**.

So with static classes, it's really more about name-space resolution than about an implicit relationship between the two classes. A static nested class is **simply a class that's a static member of the enclosing class**:

```
class BigOuter {
    static class Nested { }
}
```

The class itself isn't really "`static`", there's no such thing as a `static` class. The `static` modifier in this case says that the nested class is a `static` member of the outer class. That means it can be accessed, as with other `static` members, without having an instance of the outer class.

```
class Outer {
    static class Nest {
        void go() {
            System.out.println("hi outer");
        }
    }
}

class Inner {
    static class Nest {
        void go() {
            System.out.println("hi inner");
        }
    }

    public static void main(String[] args) {
        Outer.Nest outerNest = new Outer.Nest(); // access static inner class
        outerNest.go();                          // present inside a different class

        Nest innerNest = new Nest(); // access static inner class
        innerNest.go();              // present inside the same class
    }
}
```

Q&A

Q1.

```java
1   public class HorseTest {
2       public static void main(String[] args) {
3           class Horse {
4               public String name;
5               public Horse(String s) {
6                   name = s;
7               }
8           }
9           Object obj = new Horse("Zippo");
10          System.out.println(obj.name);
11      }
12  }
```

What is the result?

A. An exception occurs at runtime at line 10
B. Zippo
C. Compilation fails because of an error on line 3
D. Compilation fails because of an error on line 9
E. Compilation fails because of an error on line 10

Q2.

```java
public abstract class AbstractTest {
    public int getNum() {
        return 45;
    }

    public abstract class Bar {
        public int getNum() {
            return 38;
        }
    }

    public static void main(String[] args) {
        AbstractTest t = new AbstractTest() {
            public int getNum() {
                return 22;
            }
        };
        AbstractTest.Bar f = t.new Bar() {
            public int getNum() {
                return 57;
            }
        };
        System.out.println(f.getNum() + " " + t.getNum());
    }
}
```

What is the output?

A. 57 22

B. 45 38

C. 45 57

D. An exception occurs at runtime

E. Compilation fails

Q3.

```java
class A {
    void m() {
        System.out.println("outer");
    }
}

public class MethodLocalVSInner {

    public static void main(String[] args) {
        new MethodLocalVSInner().go();
    }

    void go() {
        new A().m();
        class A {
            void m() {
                System.out.println("inner");
            }
        }
    }

    class A {
        void m() {
            System.out.println("middle");
        }
    }
}
```

This is an interesting question, so think and determine the output?

A. inner
B. outer
C. middle
D. Compilation fails
E. An exception is thrown at runtime

Q4.

```java
class Car {
    class Engine {
        // insert code here
    }

    public static void main(String[] args) {
        new Car().go();
    }

    void go() {
        new Engine();
    }

    void drive() {
        System.out.println("hi");
    }
}
```

Which, inserted independently at line 3, produce the output “hi”? (Choose all that apply.)

A. { Car.drive(); }

B. { this.drive(); }

C. { Car.this.drive(); }

D. { this.Car.this.drive(); }

E. Engine() { Car.drive(); }

F. Engine() { this.drive(); }

G. Engine() { Car.this.drive(); }

Q5.

```
 1  class City {
 2      class Manhattan {
 3          void doStuff() throws Exception {
 4              System.out.print("x ");
 5          }
 6      }
 7
 8      class TimesSquare extends Manhattan {
 9          void doStuff() throws Exception {
10          }
11      }
12
13      public static void main(String[] args) throws Exception {
14          new City().go();
15      }
16
17      void go() throws Exception {
18          new TimesSquare().doStuff();
19      }
20  }
```

What is the result?

A. x

B. x x

C. No output is produced

D. Compilation fails due to multiple errors

E. Compilation fails due only to an error on line 4

F. Compilation fails due only to an error on line 7

Q6.

```java
1   class OuterClassAccess {
2       private int size = 7;
3       private static int length = 3;
4
5       public static void main(String[] args) {
6           new OuterClassAccess().go();
7       }
8
9       void go() {
10          int size = 5;
11          System.out.println(new Inner().adder());
12      }
13
14      class Inner {
15          int adder() {
16              return size * length;
17          }
18      }
19  }
```

What is the result?

A. 15

B. 21

C. An exception is thrown at runtime

D. Compilation fails due to multiple errors

E. Compilation fails due only to an error on line 4

F. Compilation fails due only to an error on line 5

Q7.

```java
import java.util.*;

public class Pockets {
    public static void main(String[] args) {
        String[] sa = {"nickel", "button", "key", "lint"};
        Sorter s = new Sorter();
        for (String s2 : sa) System.out.print(s2 + " ");
        Arrays.sort(sa, s);
        System.out.println();
        for (String s2 : sa) System.out.print(s2 + " ");
    }

    class Sorter implements Comparator<String> {
        public int compare(String a, String b) {
            return b.compareTo(a);
        }
    }
}
```

What is the result?

A. Compilation fails

B. button key lint nickel

nickel lint key button

C. nickel button key lint

button key lint nickel

D. nickel button key lint

nickel button key lint

E. nickel button key lint

nickel lint key button

F. An exception is thrown at runtime

Statics

« Previous

2 Comments Java Notes 1 Login

Recommend Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Amit Satpathy • 8 months ago

Ram, it's really great to find your short and precise writeups on tech. Keep up the passion.

1 • Reply • Share ›

Ram swaroop Mod Amit Satpathy • 5 months ago

Thanks @Amit Satpathy but I haven't been writing lately, will resume soon.

• Reply • Share ›