Nested Classes (Inner Classes)

Sang Shin
Michèle Garoche
www.javapassion.com
"Learn with Passion!"

Topics

- Nested class
- Static nested class
- Accessibility scope
- Inner class
- Local (inner) class and Anonymous (inner) class
- Performance implication

Nested Classes

What is a Nested Class?

- The Java programming language allows you to define a class within another class. Such a class is called a nested class
- Nested classes are divided into two categories:
  - Static nested classes - Nested classes that are declared *static* are simply called “Static Nested” classes.
  - Non-static (Inner) classes - Non-static nested classes are called “Inner” classes

Examples of Nested Classes

```
class OuterClass {
    ...
    // Static nested class has static modifier
    static class StaticNestedClass {
        ...
    }

    // Non-static nested class does not have static modifier. It is called Inner Class.
    class InnerClass {
        ...
    }
}
```

Why Use Nested Classes?

- Simplifies coding
  - By logically grouping classes that are only used in one place
- 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 still private and B can access them. In addition, B itself can be hidden from the outside world. This is possible through the use of Inner class.
- Increases readability

Static Nested Class

When to Use Static Nested Class?

- A static nested class cannot access the the instance members of its outer class (and other classes) just like any other top-level class
  - In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

Accessibility Scope

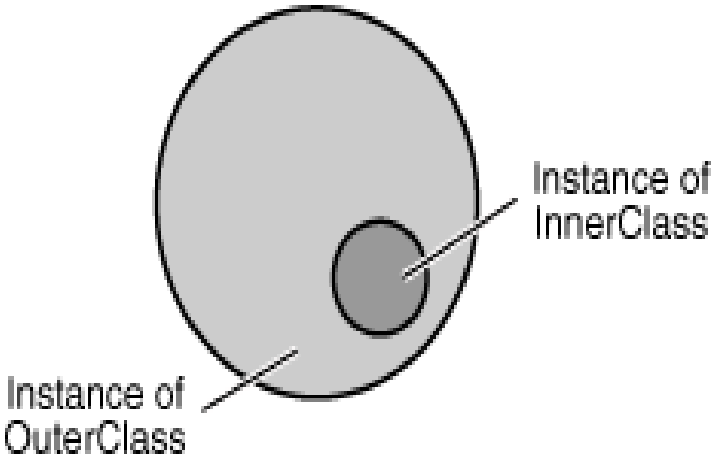
Accessibility Scope

- Inner classes (Non-static nested classes) have access to other members of the enclosing class (outer class), even if they are declared private.
  - As was mentioned, it enhances the encaptulation
- Static nested classes do not have access to other members of the enclosing class.

Inner Class (Non-Static Nested Class)

Object Instances of an Inner Class

- As with instance methods and instance variables, 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.



How to instantiate a Inner class?

- To instantiate an inner class (to create object instance of an inner class), you must first instantiate the outer class. Then, create the inner object within the outer object

```
// Create an object instance of outer class.
OuterClass outerObject = new OuterClass();

// Create an object instance of an inner class.  Please
// note that you create it from an object instance of
// OuterClass.
OuterClass.InnerClass innerObject =
                                    outerObject.new InnerClass();
```

Use case #1 of Inner Class

- Inner classes can be used to implement helper classes that are not used by other classes

```
public class DataStructure {
    private int[] arrayOfInts = new int[SIZE];

    public void printEven() {
        //print out values of even indices of the array
        InnerEvenIterator iterator = this.new InnerEvenIterator();  // this represents Outer class object
        while (iterator.hasNext()) {
            System.out.println(iterator.getNext() + " ");
        }
    }

    // inner class as a helper class
    private class InnerEvenIterator {

        public int getNext() {
            int retValue = arrayOfInts[next];   // Inner class method can access private
                                                // member of outer class
            next += 2;
            return retValue;
        }
    }
```

Use case #2 of Inner Class

- What if you want to use an adapter class, but do not want your public class to inherit from (extend) the adapter class?
- For example, suppose you write an applet, and you want your MyApplet subclass (which extends Applet class) to contain some code to handle mouse events. Since the Java language does not permit multiple inheritance, your class cannot extend both the *Applet* and *MouseAdapter* classes. A solution is to define an inner class — a class inside of your *Applet* subclass — that extends the *MouseAdapter* class.

Example: Use case #2 of Inner Class

```
// MyAppletClass cannot extend MouseAdaptor class since it
// already extends Applet
public class MyAppletClass extends Applet {
    ...
    someObject.addMouseListener(new MyAdapter());
    ...
    class MyAdapter extends MouseAdapter {
        public void mouseClicked(MouseEvent e) {
            //Event listener implementation goes here...
        }
    }
}
```

Use Case #3 of Inner Classes

- If you find that you are passing a ton of stuff in the constructor to a class and that class is only used by one other class, inner classes may come to the rescue, by giving direct reference to all the outer class's fields.
- Similarly if you find you are doing a huge amount of referencing of the outer class's fields, think of inner classes.

Local Inner Class & Anonymous Inner Class

Two Special Types of Inner Classes

- Local inner class (also called as Local class)
  - An inner class within the body of a method
- Anonymous inner class (also called as Anonymous class)
  - An inner class within the body of a method without naming it.

Local (Inner) Class

- A local class is declared locally within a block of Java code, rather than as a member of a class.
  - Typically, a local class is defined within a method, but it can also be defined within a static initializer or instance initializer of a class.
- The defining characteristic of a local class is that it is local to a block of code
  - Like a local variable, a local class is valid only within the scope defined by its enclosing block
- Use it when a class is used only within a method
  - Might improve code readability

Example: Local (Inner) Class

```
public void init() {
  /* ... some code */

  // Define a class right in the middle of the init method
  class MyListener implements ActionListener {

    // Method of MyListener local inner class
    public void actionPerformed(ActionEvent e) {
      // insultText is a field of the outer class,
      // in which this anonymous class is embedded.
      // generateResult is a method of the outer class.
      Outer.this.insultText.setText(Outer.generateResult());
    }
  }


  myButton.addActionListener(new MyListener());

  /* ...  some code */

}  // End of init() method
```

Anonymous (Inner) Class

- Because they don't have a name, there is no way to refer to them. For this reason, their declaration must be given at creation time, as part of the new statement.
- That requires yet another form of the *new* statement

```
new <class or interface>() {
        <Anonymous class' body>
}
```

- This form of the new statement declares a new anonymous class that extends a given class or implements a given interface.
- It also creates a new instance of that class or interface and returns it as the result of the statement

Use case of Anonymous Inner Class

- One common use for an anonymous class is to provide a simple implementation of an adapter class.
  - An adapter class is one that defines code that is invoked by some other object.
- The object gets created only once

```
// An example of using an anonymous inner class.
button.addActionListener(
        // An object instance of Anonymous class, which implements
        // the ActionListener interface gets returned.  This object
        // is then attached to the button object as an ActionListener.
       new ActionListener() {   // ActionListener is an interface in this example
          public void actionPerformed(ActionEvent e) {
             frame.dispose();
          }
       }
);
```

Performance Implication

Performance Implication

- When considering whether to use an inner class, keep in mind that application startup time and memory footprint are typically directly proportional to the number of classes you load.
- The more classes you create, the longer your program takes to start up and the more memory it will take.
- As an application developer you have to balance this with other design constraints you may have.

Thank you!

Check JavaPassion.com Codecamps!
http://www.javapassion.com/codecamps
"Learn with Passion!"