The Essence of OOP using Java, Nested Top-Level Classes

Baldwin explains nested top-level classes, and illustrates a very useful polymorphic structure where nested classes extend the enclosing class and override methods declared in the enclosing class.

Published: May 25, 2004
By Richard G. Baldwin

Java Programming Notes # 1642

Preface

This series of lessons is designed to teach you about the essence of Object-Oriented Programming *(OOP)* using Java.

The first lesson in this overall series on OOP was entitled The Essence of OOP Using Java, Objects, and Encapsulation.

Inner classes

This lesson is the last lesson in a six-lesson miniseries designed to teach you about inner classes. The topics covered by the lessons in this miniseries are:

- Static initializer blocks
- Instance initializers
- Member classes
- Local classes
- Anonymous classes
- Nested top-level classes

The first lesson in the six-lesson miniseries on inner classes was entitled The Essence of OOP using Java, Static Initializer Blocks. The previous lesson was entitled The Essence of OOP using Java, Anonymous Classes.

Another browser window

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different figures and listings while you are reading about them.

Further reading

For further reading, see my extensive collection of online Java tutorials at Gamelan.com. A consolidated index is available at www.DickBaldwin.com.

Preview

What can you include in a class definition?

There are several different kinds of items that can be included in a class definition. As you learned in the earlier lessons in this series, the list includes:

- Static variables
- Instance variables
- Static methods
- Instance methods
- Constructors
- Static initializer blocks
- Instance initializers

Can also contain other class definitions

You have also learned that a class definition can contain the following four kinds of *inner classes:*

- Member classes
- Local classes
- Anonymous classes
- Nested top-level classes and interfaces

Previous lessons explained member classes, local classes, and anonymous classes. This lesson will explain nested top-level classes and interfaces.

> *(Note that it is questionable whether a nested top-level class should be referred to as an inner class. Unlike an object of a member class, local class, or anonymous class, an object of a nested top-level class can exist in the absence of an object of the enclosing class. Regardless of whether the term inner class applies, a nested top-level class is defined within the definition of another class, so its definition is internal to the definition of another class.)*

What is a nested top-level class or interface?

I'm going to begin my discussion with a quotation from one of my favorite authors, David Flanagan, author of Java in a Nutshell.

> *"A nested top-level class or interface is defined as a static member of an enclosing top-level class or interface. The definition of a nested top-level class uses the static modifier ... Nested interfaces are implicitly static ... and so are always top-level. A nested top-level class or interface behaves just like a 'normal' class or interface ... The difference is that the name of a nested top-level class or interface includes the name of the class in which it is defined."*

Why use nested top-level classes or interfaces?

Again, according to Flanagan,

> *"Nested top-level classes and interfaces are typically used as a convenient way to group related classes."*

Can be particularly useful when ...

A particularly useful implementation of top-level classes occurs when the nested classes extend the enclosing class and override methods that are declared or defined in the enclosing class. This makes it very convenient to construct a hierarchical API, which exhibits very useful polymorphic behavior, and which cannot easily be expanded.

> *(Without getting into the technical details, I will cite the Java2D API as an example, which makes heavy use of nested top-level classes for this purpose. See, for example, the classes named* ***Point2D****,* ***Poind2D.Double****, and* ***Point2D.Float****. According to Sun, the* ***Point2D*** *class is "the abstract superclass for all objects that store a 2D coordinate. The actual storage representation of the coordinates is left to the subclass." With these classes, when you perform an operation on one of the subclass objects, whose reference has been stored as the superclass type, runtime polymorphism kicks in and the appropriate method is invoked for the actual type of object on which the method is invoked.)*

Purpose of this lesson

This lesson explains top-level nested classes from a practical viewpoint, and discusses a sample program that creates and exercises a simple class hierarchy as described above.

Miscellaneous comments

The following are a few of the characteristics of nested top-level classes, which are not necessarily illustrated by the sample program that follows later.

A nested top-level class must be declared static within another top-level class. Methods in a nested top-level class have access to the static members of its containing class.

Nested top-level classes can only be nested within other top-level classes. They cannot be defined inside member classes, local classes, or anonymous classes. However, nested top-level classes can be nested to any depth.

Typically nested top-level classes are referred to by their fully-qualified name, such as **Poind2D.Double**, where **Point2D** is the name of the enclosing class and **Double** is the name of the nested class. According to Flanagan, in the same sense that it is possible to use an import directive beginning with a package name to eliminate the requirement to include the package name in a reference to a class, it is also possible to use an import directive beginning with an enclosing class name to eliminate the requirement to include the enclosing class name in a reference to a nested class. However, I have never been able to make this work. Perhaps I don't fully understand the required syntax for the import directive.

Smoke and mirrors

In an earlier lesson, I explained that every class definition in a Java program, including nested top-level classes, member classes, local classes, and anonymous classes, produces a class file when the program is compiled. According to Flanagan,

> *"The Java Virtual Machine knows nothing about nested top-level classes and interfaces or the various types of inner classes. Therefore, the Java compiler must convert these new types into standard non-nested class files that the Java interpreter can understand. This is done through source code transformations that insert $ characters into nested class names. These source code transformations may also insert hidden fields, methods, and constructor arguments into the affected classes."*

Example class file names

For example, compilation of the sample program discussed later in this lesson produces the following class files:

- InnerClasses09.class
- Shape.class
- Shape$Circle.class
- Shape$Rectangle.class

The first file in the above list is the driver program that is used to exercise the three class files that follow the first one in the list. The second file named Shape.class represents the enclosing class named **Shape**. The remaining two files represent the two static classes named **Circle** and **Rectangle**, which are nested within the class named **Shape**. *(Note how the file name is constructed from the name of the enclosing class and the nested class.)*

Enough talk, let's see some code

The paragraphs that follow will explain a program named **InnerClasses09**, which is designed specifically to illustrate nested top-level classes. I will discuss the program in fragments. A complete listing of the program is provided in Listing 7 near the end of the lesson.

Discussion and Sample Code

This program named **InnerClasses09**, illustrates static top-level classes that extend their containing class.

An abstract class named **Shape** is defined, which encloses two static classes named **Rectangle** and **Circle**. **Rectangle** and **Circle** each extend **Shape**.

Shape declares an abstract method named **area**, which is overridden in each of the static classes.

Each of the overridden methods contains the appropriate code to calculate and display the area of a **Shape** object of that particular subclass type *(**Rectangle** or **Circle**)*.

An object is instantiated from each of the static classes. The object's references are saved as type **Shape**.

Polymorphic behavior

The **area** method is invoked on each of the references. Polymorphic behavior causes the appropriate overridden version of the **area** method to be invoked in each case, causing the correct area for each type of shape to be displayed.

The dimensions of the **Rectangle** object are 2x3. The radius of the **Circle** object is 3. The output from the program, showing the area of each object, is displayed in Figure 1.

```
Rectangle area = 6
Circle area = 28.274333882308138
```

Figure 1

The program was tested using SDK 1.4.2 under WinXP.

The Shape class

The beginning of the definition for the class named **Shape** is shown in Listing 1. *(Once again, see Listing 7 for a listing of the complete program.)* This class contains two nested static classes named **Rectangle** and **Circle**.

> *(Normally a class like this would be declared **public**. However, declaring a class **public** requires the source code for the class to be in a separate file. In order to keep all the code in this program in a single source code file, I caused this class to be **package-private** instead of **public**.)*

```
abstract class Shape{
  public abstract void area();

Listing 1
```

The code in Listing 1 declares an **abstract** method named **area**. This method is overridden in each of the enclosed classes, which are subclasses of the **Shape** class.

Note that the **Shape** class is also declared **abstract**. Any class that contains an abstract class must itself be declared abstract.

The Rectangle class

Listing 2 shows the beginning of the enclosed static class named **Rectangle**. *(This class appears inside the definition of the **Shape** class.)*

```
  public static class Rectangle
extends Shape{
    int length;
    int width;

    public Rectangle(int length,int
width){
      this.length = length;
      this.width = width;
    }//end constructor


Listing 2
```

Listing 2 shows the constructor for the **Rectangle** class, which receives and saves values for the length and width of a rectangle.

The overridden area method

Continuing with the definition of the **Rectangle** class, Listing 3 shows the overridden **area** method, which is inherited from the **Shape** class.

```
    public void area(){//override the
area method
      System.out.println(
             "Rectangle area = " +
length*width);
    }//end overridden area() method
  }//end class Rectangle

Listing 3
```

The overridden **area** method calculates and displays the area of a rectangle based on the length and width values that were saved when the object was instantiated.

Listing 3 also signals the end of the static class named **Rectangle**.

The Circle class

The entire class definition for the enclosed static **Circle** class is shown in Listing 4.

```
  public static class Circle extends
Shape{
   int radius;

   public Circle(int radius){
     this.radius = radius;
   }//end constructor

    public void area(){//override the
area method
      System.out.println("Circle area
= "
                    + Math.PI * radius
* radius);
    }//end overridden area() method
  }//end class Circle

}//end class Shape

Listing 4
```

Overridden area methods are appropriate for their classes

The **Circle** class is very similar to the **Rectangle** class, except that the overridden **area** method uses a different formula for calculating the area of a **Circle** object.
*(The overridden **area** method in each of the two enclosed classes uses a formula for calculation of the area that is appropriate for an object of that type. That is the beauty of runtime polymorphism. When the reference to an object of either class is saved as type **Shape**, and the **area** method is invoked on that reference, the version of the **area** method executed is appropriate for the actual type of object on which the method is invoked. However, the using programmer doesn't have to worry about the actual type of the object.)*
Listing 4 also signals the end of the **Shape** class, which encloses the definitions of the **Rectangle** and **Circle** classes.

The driver class

Listing 5 shows the beginning of the **main** method for the class named **InnerClasses09**. The purpose of this class is to instantiate and exercise objects of the **Shape.Rectangle** and **Shape.Circle** classes discussed above.

```
public class InnerClasses09{
  public static void main(String[]
args){

    Shape aShape = new
Shape.Rectangle(2,3);
```

```
    aShape.area();//Get and display
the area

Listing 5
```

The code in Listing 5 begins by instantiating a new object of the nested top-level class named **Shape.Rectangle** *(with a length of 2 and a width of 3),* and saving that object's reference in a local variable of type **Shape**.

In order to instantiate a new object of the **Rectangle** class, it must be referred to by the fully-qualified name **Shape.Rectangle**. This is not because **Rectangle** is a subclass of **Shape**. Rather, it is because **Rectangle** is a static class that is defined inside the definition of the **Shape** class. *(Note that this is similar to accessing a static variable of the **Math** class as Math.PI.)*

Invoke the area method on the object

Having instantiated the new **Rectangle** object, and having saved the object's reference as type **Shape**, the code in Listing 5 invokes the **area** method on that reference. This produces the output shown in Figure 2.

```
Rectangle area = 6
Figure 2
```

A rectangle with a length of 2 and a width of 3 has an area of 6. The output shown in Figure 2 confirms that even though the object's reference was saved as the superclass type **Shape**, the correct version of the **area** method was executed to calculate and display the area of the object of the nested **Rectangle** class.

A Circle object

The code in Listing 6

- Instantiates a new object of type **Shape.Circle** with a radius of 3 *(once again note the use of the fully-qualified class name).*
- Saves the object's reference as type **Shape**, overwriting the reference previously stored in the local variable named **aShape**.
- Invokes the **area** method on that reference.

```
    aShape = new Shape.Circle(3);
    aShape.area();//Get and display
the area
  }//end main

}//end controlling class
InnerClasses09

Listing 6
```

Execution of the code in Listing 6 produces the output shown in Figure 3, showing the correct area for a circle with a radius of 3, once again illustrating runtime polymorphism.

```
Circle area = 28.274333882308138
```
Figure 3

Listing 6 also signals the end of the class named **InnerClasses09**, and the end of the program.

No object of the enclosing class is required

Once again, let me emphasize that unlike objects instantiated from member classes, local classes, and anonymous classes, the existence of an object of a nested top-level class does not require the existence of an object of the enclosing class.

Run the Program

At this point, you may find it useful to compile and run the program shown in Listing 7 near the end of the lesson.

Summary

In addition to a number of other items, a class definition can contain:

- Member classes
- Local classes
- Anonymous classes
- Nested top-level classes and interfaces

Member classes, local classes, and anonymous classes were explained in previous lessons. This lesson explains nested top-level classes and interfaces *(although an example of a top-level interface was not presented).*

A nested top-level class or interface is defined as a static member of an enclosing top-level class or interface.

The name of a nested top-level class or interface includes the name of the class in which it is defined.

Nested top-level classes are often used as a way to group related classes. This is particularly useful when the nested classes extend the enclosing class and override methods that are declared or defined in the enclosing class. This makes it very convenient to construct a hierarchical API, which exhibits useful polymorphic behavior, and which cannot easily be expanded.

A nested top-level class must be declared static within another top-level class. Methods in a nested top-level class have access to the static members of the containing class.

Nested top-level classes can only be defined within other top-level classes. They cannot be defined inside member classes, local classes, or anonymous classes. However, nested top-level classes can be nested to any depth.

Complete Program Listing

A complete listing of the program discussed in this lesson is show in Listing 7 below.

```
/*InnerClasses09.java
Copyright 2003, R.G.Baldwin
Revised 08/26/03

Illustrates static top-level classes that extend
their containing class.  Also illustrates the
design of a closed polymorphic system.

An abstract class named Shape is defined, which
contains two static top-level classes named
Rectangle and Circle.

Rectangle and Circle each extend Shape.

Shape declares an abstract method named area(),
which is overridden in each of the static
top-level classes contained within the
definition of Shape.

Each of the overridden methods contains the
appropriate code to calculate and display the
area of a Shape object of that particular
subclass type (Rectangle or Circle).

An object is instantiated from each of the
static top-level classes.  That object's
reference is saved as type Shape.

The area() method is invoked on each of the
references.  Polymorphic behavior causes the
appropriate overridden version of the area()
method to be invoked in each case, causing the
correct area for each type of shape to be
displayed.

The output from the program is:

Rectangle area = 6
Circle area = 28.274333882308138

Compilation of the program produces the following
class files:
```

```
InnerClasses09.class
Shape$Circle.class
Shape$Rectangle.class
Shape.class

Tested using JDK 1.4.2 under Win XP
*********************************************/

//This class contains two nested static
// top-level classes. I made this class package-
// private instead of public so that I could
// contain everything in a single source file.
abstract class Shape{
  //This abstract method is overridden in each
  // of the nested classes
  public abstract void area();

  //---------------------------------------//
  //The definitions of two nested top-level
  // classes follow

  //Nested top-level class named Rectangle
  public static class Rectangle extends Shape{
    int length;
    int width;

    public Rectangle(int length,int width){
      this.length = length;
      this.width = width;
    }//end constructor

    public void area(){//override the area method
      System.out.println(
              "Rectangle area = " + length*width);
    }//end overridden area() method
  }//end class Rectangle

  //Nested top-level class named Circle
  public static class Circle extends Shape{
   int radius;

   public Circle(int radius){
     this.radius = radius;
   }//end constructor

    public void area(){//override the area method
      System.out.println("Circle area = "
                    + Math.PI * radius * radius);
    }//end overridden area() method
  }//end class Circle

}//end class Shape

//=========================================//
//This controlling class instantiates and
// processes an object from each of the nested
```

```
// top-level classes defined above.
public class InnerClasses09{
  public static void main(String[] args){

    //Instantiate and process an object of the
    // class named Shape.Rectangle.  Save the
    // object's reference as the superclass
    // type Shape.
    Shape aShape = new Shape.Rectangle(2,3);
    aShape.area();//Get and display the area

    //Instantiate and process an object of the
    // class named Shape.Circle.  Save the
    // object's reference as the superclass
    // type Shape.
    aShape = new Shape.Circle(3);
    aShape.area();//Get and display the area
  }//end main

}//end controlling class InnerClasses09
//===========================================//
```

Listing 7

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, Texas) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.*

Richard has participated in numerous consulting projects, and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which has gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

baldwin@DickBaldwin.com

-end-