Richard G Baldwin (512) 223-4758, baldwin@austin.cc.tx.us,
http://www2.austin.cc.tx.us/baldwin/

Java 2D Graphics, Nested Top-Level Classes and Interfaces

Java Programming, Lecture Notes # 300, Revised 01/30/00.

Getting Started with Java 2D Graphics

I don't know of a better way to get started with Java 2D Graphics than to run, and enjoy the demonstration program provided by Sun with the JDK 1.2.x download package. If you don't do anything else with Java 2D, you should at least run the demo to get a feel for what is possible.

I am running JDK 1.2.2 as of this writing. In my installation of the JDK, I can find the demo stored in the following path and file:

jdk1.2.2\demo\jfc\Java2D\Java2Demo.jar.

Your path may be different, but hopefully the demo will be contained in the file named **Java2Demo.jar**. (Note that the file is named Java2Demo.jar and not Java2DDemo.jar.)

You must have JDK 1.2.x installed in order to run the demo. Apparently, it is also possible to run it as an applet if you have a browser that is JDK 1.2 compatible, but I haven't tried to do that. See the **README.txt** file in the same directory for more information in this regard.

I can execute the demo by changing directories to the directory containing the file named Java2Demo.jar and then executing the following command at the command prompt:

java -jar Java2Demo.jar

What happens next is truly amazing; so amazing in fact that I won't even try to describe it. Give it a shot. It will be well worth the effort.

Introduction

You might wonder why a tutorial lesson that purports to cover the Java 2D Graphics API is really a tutorial on what some may consider to be advanced Inner Classes. This is the first in a series of tutorials on the Java 2D Graphics API. That API makes some rather interesting uses of Nested Top-Level Classes. Therefore, I decided to get that behind us at the outset, and then it won't be necessary for me to continue explaining it as we get into the meat of the 2D Graphics API.

References

The best reference on Inner Classes that I am aware of is Java in a Nutshell, Second Edition or later, by David Flannagan, published by O'Reilly.

Types of Inner Classes

Flannagan tells us that beginning with JDK 1.1, Java has supported the following types of Inner Classes:

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

*Technically, this is not really an Inner Class. Rather, it is a special form of a top-level class. However, the syntax for defining it causes it to look like an Inner Class, and Flanagan discusses it in the Inner Class section of his book.

Member classes and anonymous classes have been discussed in detail in other tutorial lessons in this series, and therefore, won't be discussed in any detail here. As of this writing, I haven't discussed local classes in any of the tutorial lessons. Since they are not of interest with regard to the 2D Graphics API, I won't discuss them in detail here either.

As of this writing, nested top-level classes also haven't been discussed in any of my other tutorial lessons. They are particularly interesting with regard to the 2D API. Therefore, they will be the primary topic of this tutorial lesson.

What Is an Inner Class

Classes can be defined as members of other classes, just as fields and methods can be defined within classes. Classes can also be defined within a block of Java code.

The following quotation from Flanagan provides one of the most important insights that I know of regarding Inner Classes (and nested top-level classes that look like Inner Classes from a syntax viewpoint):

"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 affected classes."

Flanagan also tells us that "All nested and inner classes are converted to" top-level classes and interfaces. (We will see an example of this at the end of this lesson.) Top-level classes and interfaces are the ordinary classes and interfaces that are direct members of packages. These are the basic Java classes and interfaces that are understood by the Java Virtual Machine.

In other words, this says to me that anything that we can do with Inner Classes and top-level nested classes, we should be able to do without the use of Inner Classes if we knew how to write the source code to do it. However, the use of Inner Classes provides a very convenient mechanism for accomplishing certain tasks, and I use them frequently.

Brief Discussion of Different Types

I will provide a brief discussion of each of the types of Inner Classes in the following sections, and will then concentrate on nested top-level classes for the remainder of the lesson.

Member Class

A *member class* is defined as a member of an enclosing class. It is not prefaced by the *static* keyword. Therefore, it is truly an Inner Class definition and is not a *top-level nested class* as discussed later.

There is no such thing as a *Member Interface* or *Inner Interface*. Probably the most important thing about a member class is that the code within a member class can implicitly refer to any of the fields and methods, including *private* fields and methods, of its enclosing class. This can often greatly simplify the source code to accomplish a particular task. The use of member classes is particularly common with respect to the *source/listener* event model introduced in JDK 1.1.

Local Class

According to Flanagan, a *local class* is "an inner class defined within a block of Java code; it is visible only within that block." Interfaces cannot be defined locally. According to Flanagan, "local classes are not member classes, but can still use the fields and methods of enclosing classes. See Flanagan's book for additional discussion on local classes.

Anonymous Class

An anonymous class is an extension to the local class described above. According to Flanagan, "Instead of declaring a local class with one Java statement, and then instantiating and using it in

another statement, an anonymous class combines the two steps in a single Java expression." Interfaces cannot be defined anonymously.

I have discussed the definition and use of anonymous classes in several previous lessons where I use them for defining, instantiating and registering listener objects. See those lessons, or see Flanagan's book for further discussion of anonymous classes.

Nested Top-Level Class or Interface

The use of nested top-level classes provides a convenient way to group related classes. A nested top-level class or interface must be defined as a *static* member of an enclosing top-level class or interface. Nested interfaces are implicitly *static*.

According to Flanagan, "A nested top-level class or interface behaves just like a 'normal' class or interface that is a member of a package. 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." (Again, we will see an example of this naming convention at the end of this lesson.)

They can only be nested within other top-level classes and interfaces (they cannot be nested within inner classes), and they can be nested to any depth.

As mentioned earlier, nested top-level classes are of particular interest to us here because they are heavily used in the 2D Graphics API.

Sample Program

The behavior of this sample program is similar to the manner in which the 2D Graphics API makes use of nested top-level classes. The program is designed to illustrate the use of a *static* class that is contained within another class. It is important to note that the contained *static* class *extends* the class in which it is contained. As a result, an object of the contained class can be treated as though it is of the containing class type, with appropriate downcasting being performed when required.

Because of the length of the program, I will break it up and discuss it in fragments. A complete listing of the program is provided at the end of the lesson.

The class named OuterClass

The first fragment shows the beginning of an *abstract* class named **OuterClass** that contains two different nested top-level classes. Each of the contained classes *extends* the containing class named **OuterClass**.

This class contains a method named **talk()** that is overridden in both of the contained classes. Note that this method is included here simply to make it possible to identify and separate the behavior of the containing class (which is a superclass) and the contained classes, each of which is a subclass of the containing class.

This identification and separation of behavior is accomplished by invoking the superclass version of the **talk()** method in each of the overridden versions of the method, prior to providing specific behavior in the overridden versions.

```
abstract class OuterClass{
  void talk(){
   System.out.println("Hello from superclass OuterClass");
  }//end talk
```

The class named OuterClass.NestedDoubleClass

The next fragment shows the definition of a *static* class named **OuterClass.NestedDoubleClass**, which is the first of the two nested top-level classes contained within the class named **OuterClass**.

Note in particular the name of this class. Although the class is named **NestedDoubleClass** in its source code definition, it must be referred to by the name **OuterClass.NestedDoubleClass** because it is nested within the class named **OuterClass**.

As mentioned above, this class definition contains an overridden version of the method named **talk()**, which is inherited from the superclass named **OuterClass**.

Also, as mentioned above, the overridden version of the method named **talk()** invokes the superclass version of the same method through the use of the **super** keyword. This results in the following sequence of output lines on the screen when the **talk()** method is invoked on an object of this class.

```
Hello from superclass OuterClass
--Goodbye from object of class OuterClass.NestedDoubleClass
```

The first line of output is produced by the superclass version of the method and the second line of output is produced by the overridden version of the method.

This class definition also contains an instance variable named **theString**, which we will see being accessed in a subsequent code fragment.

```
  static class NestedDoubleClass extends OuterClass{
   String theString = "---Hello from object of class "
                      + "OuterClass.NestedDoubleClass";
   void talk(){//override the talk method
     super.talk();//invoke superclass version
     System.out.println("---Goodbye from object of class "
                        + "OuterClass.NestedDoubleClass");
   }//end talk()
  }//end class NestedDoubleClass
```

The class named OuterClass.NestedFloatClass

The next fragment shows the definition of another nested *static* class that is very similar to the previous one. Because of the similarity, I won't discuss it further. The fragment also shows the end of the outer class named **OuterClass**.

```
  static class NestedFloatClass extends OuterClass{
   String theString = "==Hello from object of class "
                        + "OuterClass.NestedFloatClass";
   void talk(){//override the talk method
     super.talk();//invoke superclass version
     System.out.println("==Goodbye from object of class "
                       + "OuterClass.NestedFloatClass");
   }//end talk()
  }//end class NestedFloatClass

}//end class OuterClass
```

The controlling class

The next fragment shows the beginning of a controlling class that instantiates and processes an object from each of the nested top-level classes defined above. This fragment also shows the beginning of the **main()** method of the controlling class.

The main() method

The **main()** method declares a reference variable of the class type **OuterClass**. This reference variable will later be used to refer to objects instantiated from the nested classes that extend **OuterClass**. This will be possible because an object can be referred to by a reference variable of the class from which it is instantiated, or by a reference variable of any superclass of that class. However, when the reference variable is of a superclass type, it normally requires downcasting before anything useful can be done with it.

```
class InnerClasses04{
  public static void main(String[] args){
   //Declare a reference variable of type OuterClass
   OuterClass theVar;
```

Instantiating and processing OuterClass.NestedDoubleClass

The following fragment instantiates and processes an object of one of the nested classes (named **OuterClass.NestedDoubleClass**) and saves the reference to that object in the reference variable (of type **OuterClass**) declared above, which is of the superclass type.

Note that even though the name of the class in the nested class definition is **NestedDoubleClass**, it is necessary to refer to its constructor using the name **OuterClass.NestedDoubleClass** in order to instantiate the object.

In this case, processing the object consists of invoking the method named **processTheObject()** and passing to that method, the reference variable (of the superclass type) that refers to the object. I will explain the behavior of the **processTheObject()** method later.

```
    theVar = new OuterClass.NestedDoubleClass();
    processTheObject(theVar);
```

Instantiating and processing OuterClass.NestedFloatClass

The next fragment shows the instantiation and processing of an object of the other nested class. The code and behavior is essentially the same as in the previous fragment, so I won't discuss it further.

```
    theVar = new OuterClass.NestedFloatClass();
    processTheObject(theVar);
  }//end main
```

The method named processTheObject()

Finally, the next fragment shows the method named **processTheObject()**. There are several important points to consider here.

First, the method receives a reference to an object as type **OuterClass**. This is not the actual type of the object to which it refers, but rather is the superclass of the object. Therefore, in order for this method to gain access to the instance members of the object that are defined in the class from which the object was instantiated, it is necessary do downcast the reference to the actual type of the object.

Downcast required

In this case, there are two members of the object that are of interest. One member is the instance variable named **theString**. Because this member was not defined in the superclass, but rather was defined in the subclass, it is necessary to downcast the reference to the actual type to gain access to this member.

Downcast not required

The other member of interest is the method named **talk()**. However, this member was defined in the superclass and then overridden in the subclass. Due to the polymorphic behavior of Java, the overridden version of the method in the object of the subclass type can be accessed using a reference of the superclass type.

Doing the required downcast

The design of this method assumes that the object to which the reference refers must be of one of the following two types:

OuterClass.NestedDoubleClass
OuterClass.NestedFloatClass

Since these are the only possibilities that exist, the required downcasting can be accomplished using a simple pair of **if** statements. (The problem would be more complicated if the possible types of the incoming object were not known when the method is written.)

Similar to functionality in 2D Graphics API

This may not be exactly how this functionality is accomplished in the 2D Graphics API (to be discussed in subsequent lessons), but it is probably pretty close.

In effect, I have made it possible

- To instantiate an object from one of two classes that are both contained as *static* classes in their common *abstract* superclass, and
- To handle the reference to that object as the superclass type until the very last minute.

At the last minute, when the time comes to actually perform some process using the reference to the object, it is necessary to determine the actual class of the object and to downcast the reference to that type in order to access instance variables defined in the subclass. However, due to polymorphic behavior, it is not necessary to downcast the reference in order to gain access to a method that is defined in the abstract superclass and overridden in the subclass.

This is probably very similar to the treatment given to several different classes that we will see in the 2D Graphics API. You should be able to recognize those classes when you see them.

```
static void processTheObject(OuterClass theRefVariable){
//Test for the actual type of the incoming reference
// variable and downcast as appropriate before using it

 if(theRefVariable instanceof
                          OuterClass.NestedDoubleClass){
   System.out.println(
         ((OuterClass.NestedDoubleClass)theRefVariable).
                                                 theString);
 }//end if(theRefVariable instanceof ...

  if(theRefVariable instanceof
                            OuterClass.NestedFloatClass){
  System.out.println(
          ((OuterClass.NestedFloatClass)theRefVariable).
                                                 theString);
  }//end if(theRefVariable instanceof ...

  theRefVariable.talk();//invoke overridden talk() method
```

```
  }//end processTheObject()
}//end controlling class InnerClasses04
//=======================================================//
```

The screen output

The following screen output was produced by the invocation of the **processTheObject()** method on an object from each of the nested subclasses. (I manually inserted blank lines between the output produced by processing the first object and the output produced by processing the second object to make the two sections of output easier to identify.)

The first line of output in each case was produced by accessing and displaying the instance variable named **theString** in the object (after downcasting appropriately). The next two lines were produced by invoking the **talk()** method on the reference to the object with no downcasting required.

The first line of output (in the last two lines) was produced by the version of the **talk()** method that was defined in the superclass. The second line was produced by the overridden version of the **talk()** method that was defined in the nested *static*subclass.

```
---Hello from object of class OuterClass.NestedDoubleClass
Hello from superclass OuterClass
--Goodbye from object of class OuterClass.NestedDoubleClass


==Hello from object of class OuterClass.NestedFloatClass
Hello from superclass OuterClass
==Goodbye from object of class OuterClass.NestedFloatClass
```

The class files

Finally, you may be interested in knowing that the compilation of this program produced the following four class files, each of which, according to Flanagan, is a top-level class file. The two nested classes resulted in class-file names that are prefaced by the name of the containing class (**OuterClass**) and a dollar sign.

```
InnerClasses04.class
OuterClass$NestedDoubleClass.class
OuterClass$NestedFloatClass.class
OuterClass.class
```

Complete Program Listing

A listing of the complete program follows:

```
/*InnerClasses04.java 12/07/99
Illustrates a static class that extends its
```

```
containing class.  This is an illustration of the use of
nested top-level classes similar to that used in the
Graphics 2D API.

The output from the program is:

---Hello from object of class OuterClass.NestedDoubleClass
Hello from superclass OuterClass
--Goodbye from object of class OuterClass.NestedDoubleClass
==Hello from object of class OuterClass.NestedFloatClass
Hello from superclass OuterClass
==Goodbye from object of class OuterClass.NestedFloatClass

Tested using JDK 1.2.2 under WinNT Workstation 4.0
**********************************************************/

//This is a class that contains two nested top-level
// classes
abstract class OuterClass{
  //This method is overridden in each of the nested classes
  void talk(){
   System.out.println("Hello from superclass OuterClass");
  }//end talk

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

  //nested top-level class
  static class NestedDoubleClass extends OuterClass{
   String theString = "---Hello from object of class "
                      + "OuterClass.NestedDoubleClass";
   void talk(){//override the talk method
     super.talk();//invoke superclass version
     System.out.println("---Goodbye from object of class "
                        + "OuterClass.NestedDoubleClass");
   }//end talk()
  }//end class NestedDoubleClass

  //nested top-level class
  static class NestedFloatClass extends OuterClass{
   String theString = "==Hello from object of class "
                       + "OuterClass.NestedFloatClass";
   void talk(){//override the talk method
     super.talk();//invoke superclass version
     System.out.println("==Goodbye from object of class "
                        + "OuterClass.NestedFloatClass");
   }//end talk()
  }//end class NestedFloatClass

}//end class OuterClass

//=========================================================//
//This is the controlling class that instantiates and
// processes an object from each of the nested top-level
// classes defined above.
class InnerClasses04{
```

```
  public static void main(String[] args){
   //Declare a reference variable of type OuterClass
   OuterClass theVar;

   //Instantiate and process an object of the subclass
   // named OuterClass.NestedDoubleClass, but refer to
   // it as type OuterClass.
   theVar = new OuterClass.NestedDoubleClass();
   processTheObject(theVar);

   //Instantiate and process an object of the subclass
   // named OuterClass.NestedFloatClass,  but refer to
   // it as type OuterClass.
   theVar = new OuterClass.NestedFloatClass();
   processTheObject(theVar);
  }//end main

  static void processTheObject(OuterClass theRefVariable){
  //Test for the actual type of the incoming reference
  // variable and downcast as appropriate before using it

   if(theRefVariable instanceof
                            OuterClass.NestedDoubleClass){
     System.out.println(
           ((OuterClass.NestedDoubleClass)theRefVariable).
                                                 theString);
   }//end if(theRefVariable instanceof ...

    if(theRefVariable instanceof
                              OuterClass.NestedFloatClass){
    System.out.println(
            ((OuterClass.NestedFloatClass)theRefVariable).
                                                 theString);
    }//end if(theRefVariable instanceof ...

    theRefVariable.talk();//invoke overridden talk() method
  }//end processTheObject()
}//end controlling class InnerClasses04
//=========================================================//
```

Copyright 2000, Richard G. Baldwin. Reproduction in whole or in part in any form or medium without express written permission from Richard Baldwin is prohibited.

-end-