Tutorial 5 - Inheritance & Polymorphism

This tutorial discusses the second and third fundamental object oriented programming principles of ***inheritance*** and ***polymorphism***.

Inheritance

Inheritance is the capability of a class to use the properties and methods of another class while adding its own functionality. An example of where this could be useful is with an employee records system. You could create a ***generic*** employee class with states and actions that are common to all employees. Then more ***specific*** classes could be defined for salaried, commissioned and hourly employees. The generic class is known as the parent (or ***superclass*** or base class) and the specific classes as children (or ***subclasses*** or derived classes). The concept of inheritance greatly enhances the ability to ***reuse*** code as well as making design a much simpler and cleaner process.

The Object Class

The ***Object*** class is the highest superclass (ie. ***root*** class) of Java. All other classes are subclasses (children or descendants) of the Object class. The Object class includes methods such as:

clone()	finalize()	hashCode()	toString()
copy(Object src)	getClass()	notifyAll()	wait()

Inheritance in Java

Java uses the ***extends*** keyword to set the relationship between a child class and a parent class. For example using our Box class from **tutorial 4**:

```
public class GraphicsBox extends Box
```

The GraphicsBox class assumes or ***inherits*** all the properties of the Box class and can now add its own properties and methods as well as override existing methods. ***Overriding*** means creating a new set of method statements for the ***same*** method signature (name, number of parameters and parameter types). For example:

```
// define position locations
    private int left;
    private int top;
// override a superclass method
    public int displayVolume() {
      System.out.println(length*width*height);
      System.out.println("Location: "+left+", "+top);
      }
```

When extending a class constructor you can reuse the superclass constructor and overridden superclass methods by using the reserved word ***super***. Note that this reference must come first in the subclass constructor. The reserved word ***this*** is used to distinguish between the object's property and the passed in parameter.

```
    GraphicsBox(l,w,h,left,top)
    {
      super (l,w,h);
      this.left = left;
      this.top = top;
    }
    public void showObj()
    {System.out.println(super.showObj()+"more stuff here");}
```

The reserved word ***this*** can also be used to reference ***private*** constructors which are useful in initializing properties.

Special Note: You cannot override final methods, methods in final classes, private methods or static methods.

Abstract Classes

As seen from the previous example, the superclass is more general than its subclass(es). The superclass contains elements and properties common to all of the subclasses. The previous example was of a ***concrete*** superclass that objects can be made from. Often, the superclass will be set up as an ***abstract*** class which does not allow objects of its prototype to be created. In this case only objects of the subclass are used. To do this the reserved word ***abstract*** is included in the class definition.

Abstract methods are methods with no method statements. Subclasses ***must*** provide the method statements for their particular meaning. If the method was one provided by the superclass, it would require overriding in each subclass. And if one forgot to override, the applied method statements may be inappropriate.

```
public abstract class Animal  // class is abstract
{
  private String name;
  public Animal(String nm)
  { name=nm; }
  public String getName()         // regular method
  { return (name); }
  public abstract void speak();   // abstract method - note no {}
}
```

Abstract classes and methods force prototype standards to be followed (ie. they provide templates).

Interfaces

Java does not allow ***multiple inheritance*** for classes (ie. a subclass being the extension of more than one superclass). To tie elements of different classes together Java uses an ***interface***. Interfaces are similar to abstract classes but all methods are ***abstract*** and all properties are ***static final***. As an example, we will build a ***Working*** interface for the subclasses of Animal. Since this interface has the method called ***work()***, that method must be defined in any class using Working.

```
public interface Working
{
  public abstract void work();
}
```

When you create a class that uses an ***interface***, you reference the interface with the reserved word ***implements Interface_list***. ***Interface_list*** is one or more interfaces as multiple interfaces are allowed. Any class that implements an interface must include code for all methods in the interface. This ensures commonality between interfaced objects.

```
public class WorkingDog extends Dog implements Working
{
  public WorkingDog(String nm)
  {
    super(nm);    // builds ala parent
  }
  public void work()  // this method specific to WorkingDog
  {
    speak();
    System.out.println("I can herd sheep and cows");
  }
}
```

Interfaces can be inherited (ie. you can have a sub-interface). As with classes the ***extends*** keyword is used. Multiple inheritance can be used with interfaces.

Polymorphism

Overloaded methods are methods with the same ***name signature*** but either a different number of parameters or different types in the parameter list. For example 'spinning' a number may mean increase it, 'spinning' an image may mean rotate it by 90 degrees. By defining a method for handling each type of parameter you achieve the effect that you want.

Overridden methods are methods that are redefined within an **inherited or subclass**. They have the ***same***

signature and the subclass definition is used.

Polymorphism is the capability of an action or ***method*** to do different things based on the object that it is acting upon. This is the third basic principle of object oriented programming. Overloading and overriding are two types of polymorphism . Now we will look at the third type: ***dynamic method binding***.

Assume that three subclasses (Cow, Dog and Snake) have been created based on the Animal abstract class, each having their own speak() method.

```
public class AnimalReference
{
  public static void main(String args[])
  Animal ref                   // set up var for an Animal
  Cow aCow = new Cow("Bossy");  // makes specific objects
  Dog aDog = new Dog("Rover");
  Snake aSnake = new Snake("Earnie");

  // now reference each as an Animal
  ref = aCow;
  ref.speak();
  ref = aDog;
  ref.speak();
  ref = aSnake;
  ref.speak();
}
```

Notice that although each method reference was to an Animal (but no animal objects exist), the program is able to resolve the correct method related to the subclass object at runtime. This is known as dynamic (or late) method binding.

Example: Applets

Applets are good examples of both the ***inheritance*** and the ***polymorphism*** principles. All applets ***extend*** the Applet class which has several predefined methods for its **life cycle**. To make an applet unique, various life cycle methods are ***overridden***. Here is a simple example:

```
import java.awt.*; import java.applet.*;
public class NestedApplet extends Applet
{
  int width=400; int height=200; // display params
  int level=100; int inc=10;     // nesting defaults
  // first override the life cycle methods
  public void init()
  {System.out.println("Initializing");incNesting();}
  public void start()
  {System.out.println("Starting.");incNesting();}
  public void stop()
  {System.out.println("Stopping.");incNesting();}
  public void destroy()
  {System.out.println("Shutting down.");incNesting();}
  public void paint(Graphics g)
  {
    int i, shift=0; g.setColor(Color.blue);
    for (i=0;i<level;i++)
      {
        g.drawRect(shift,shift,width-2*shift-1,height-2*shift-1);
        shift = shift + inc;
      }
      g.drawString("Nesting level = "+level,width/2-50,height/2+5);
  }
  public void incNesting() {level++;repaint();}
}
```

Arrays of Objects

As with arrays of primitive types, arrays of objects allow much more efficient methods of access. Note in this example that once the array of Animals has been structured, it can be used to store objects of any subclass of Animal. By making the method speak() abstract, it can be defined for each subclass and any usage will be polymorphic (ie. adapted to the appropriate object type at runtime). It now becomes very easy to rehearse the speak() method for each object by object indexing.

```
public class AnimalArray
{
  public static void main(String args[])
  Animal ref[] = new Animal[3]; // assign space for array
  Cow aCow = new Cow("Bossy");  // makes specific objects
  Dog aDog = new Dog("Rover");
  Snake aSnake = new Snake("Earnie");

  // now put them in an array
  ref[0] = aCow; ref[1] = aDog; ref[2] = aSnake;

  // now demo dynamic method binding
  for (int x=0;x<3;++x) { ref[x].speak(); }
}
```

Casting Objects

One of the difficulties of using a superclass array to hold many instances of subclass objects is that one can only access properties and methods that are in the superclass (ie. common to all). By ***casting*** an individual instance to its subclass form one can refer to any property or method. But first take care to make sure the cast is valid by using the operation ***instanceof***. Then perform the cast. As an example using the above Animal class:

```
if (ref[x] instanceof Dog) // ok right type of object
   {
   Dog doggy = (Dog) ref[x]; // cast the current instance to its subclass
   doggy.someDogOnlyMethod();
   }
```

JR's HomePage | **Comments** [jatutor5.htm:2007 12 10]