Labs – Week 8: Inheritance

Intro to inheritance lab:

File *Dog.java* contains a declaration for a Dog class. Save this file to your directory and study it—notice what instance variables and methods are provided. Files *Labrador.java* and *Yorkshire.java* contain declarations for classes that extend Dog. Save and study these files as well.

File *DogTest.java* contains a simple driver program that creates a dog and makes it speak. Study DogTest.java, save it to your directory, and compile and run it to see what it does. Now modify these files as follows:

1. Add statements in DogTest.java after you create and print the dog to create and print a Yorkshire and a Labrador. Note that the Labrador constructor takes two parameters: the name and color of the labrador, both strings. Don't change any files besides DogTest.java. Now recompile DogTest.java; you should get an error saying something like

```
./Labrador.java:18: Dog(java.lang.String) in Dog cannot be applied to ()
       {
       ^
```

   1 error
   If you look at line 18 of Labrador.java it's just a {, and the constructor the compiler can't find (Dog()) isn't called anywhere in this file.
   a. What's going on? (Hint: What call must be made in the constructor of a subclass?)
   b. Fix the problem (which really is in Labrador) so that DogTest.java creates and makes the Dog, Labrador, and Yorkshire all speak.

2. Add code to DogTest.java to print the average breed weight for both your Labrador and your Yorkshire. Use the avgBreedWeight() method for both. What error do you get? Why?

   Fix the problem by adding the needed code to the Yorkshire class.

3. Add an abstract *int avgBreedWeight()* method to the Dog class. Remember that this means that the word *abstract* appears in the method header after *public*, and that the method does not have a body (just a semicolon after the parameter list). It makes sense for this to be abstract, since Dog has no idea what breed it is. Now any subclass of Dog must have an avgBreedWeight method; since both Yorkshire and Laborador do, you should be all set.

   Save these changes and recompile DogTest.java. You should get an error in Dog.java (unless you made more changes than described above). Figure out what's wrong and fix this error, then recompile DogTest.java. You should get another error, this time in DogTest.java. Read the error message carefully; it tells you exactly what the problem is. Fix this by changing DogTest (which will mean taking some things out).

```
// ****************************************************************
// Dog.java
// A class that holds a dog's name and can make it speak.
// ****************************************************************
public class Dog
{
    protected String name;
    // Constructor -- store name
    public Dog(String name){
      this.name = name;
    }
    // Returns the dog's name
    public String getName(){
      return name;
    }
    // Returns a string with the dog's comments
    public String speak(){
      return "Woof";
    }
}
// ****************************************************************

// Labrador.java

// A class derived from Dog that holds information about

// a labrador retriever.  Overrides Dog speak method and includes

// information about avg weight for this breed.

// ****************************************************************

public class Labrador extends Dog
{
    private String color; //black, yellow, or chocolate?
    private int breedWeight = 75;

    public Labrador(String name,  String color)
    {
      this.color = color;
    }

    // ------------------------------------------------------------
    // Big bark -- overrides speak method in Dog
    // ------------------------------------------------------------
    public String speak()
    {
      return "WOOF";
    }

    // ------------------------------------------------------------
    // Returns weight
    // ------------------------------------------------------------
    public static int avgBreedWeight()
    {
      return breedWeight;
    }
}
```

```
// ****************************************************************
// Yorkshire.java
// A class derived from Dog that holds information about
// a Yorkshire terrier. Overrides Dog speak method.
// ****************************************************************

public class Yorkshire extends Dog
{
    public Yorkshire(String name) {
      super(name);
    }

    // ------------------------------------------------------------
    // Small bark -- overrides speak method in Dog
    // ------------------------------------------------------------
    public String speak(){
      return "woof";
    }
}
// ****************************************************************
// DogTest.java
// A simple test class that creates a Dog and makes it speak.
// ****************************************************************

public class DogTest
{
    public static void main(String[] args)
    {
      Dog dog = new Dog("Spike");
      System.out.println(dog.getName() + " says " + dog.speak());

    }
}
```

Lab #2:

1. Write an inheritance hierarchy for classes Quadrilateral, Trapezoid, Parallelogram, Rectangle and Square. Use Quadrilateral as the superclass of the hierarchy. Specify the instance variables and methods for each class. The private instance variables of Quadrilateral should be the *x*-*y* coordinate pairs for the four endpoints of the Quadrilateral. Write a program that instantiates objects of your classes and outputs each object's area (except Quadrilateral).

Lab#3

A Sorted Integer List

File *IntList.java* contains code for an integer list class. Save it to your directory and study it; notice that the only things you can do are create a list of a fixed size and add an element to a list. If the list is already full, a message will be printed. File *ListTest.java* contains code for a class that creates an IntList, puts some values in it, and prints it. Save this to your directory and compile and run it to see how it works.

Now write a class SortedIntList that extends IntList. SortedIntList should be just like IntList except that its elements should always be in sorted order from smallest to largest. This means that when an element is inserted into a SortedIntList it should be put into its sorted place, not just at the end of the array. To do this you'll need to do two things when you add a new element:

- Walk down the array until you find the place where the new element should go. Since the list is already sorted you can just keep looking at elements until you find one that is at least as big as the one to be inserted.
- Move down every element that will go after the new element, that is, everything from the one you stop on to the end. This creates a slot in which you can put the new element. Be careful about the order in which you move them or you'll overwrite your data!

Now you can insert the new element in the location you originally stopped on.

All of this will go into your *add* method, which will override the *add* method for the IntList class. (Be sure to also check to see if you need to expand the array, just as in the IntList *add* method.) What other methods, if any, do you need to override?

To test your class, modify ListTest.java so that after it creates and prints the IntList, it creates and prints a SortedIntList containing the same elements (inserted in the same order). When the list is printed, they should come out in sorted order.

```
// ****************************************************************
// IntList.java
// An (unsorted) integer list class with a method to add an
// integer to the list and a toString method that returns the contents
// of the list with indices.
// ****************************************************************
public class IntList{
   protected int[] list;
   protected int numElements = 0;

   //-------------------------------------------------------------
   // Constructor -- creates an integer list of a given size.
   //-------------------------------------------------------------
   public IntList(int size) {
         list = new int[size];
   }
   //-------------------------------------------------------------
   // Adds an integer to the list.  If the list is full,
   // prints a message and does nothing.
   //-------------------------------------------------------------
   public void add(int value) {
            if (numElements == list.length)
               System.out.println("Can't add, list is full");
            else{
                        list[numElements] = value;
                        numElements++;
               }
   }

   //-------------------------------------------------------------
   // Returns a string containing the elements of the list with their
   // indices.
   //-------------------------------------------------------------
   public String toString(){
            String returnString = "";
            for (int i=0; i<numElements; i++)
               returnString += i + ": " + list[i] + "\n";
            return returnString;
   }
}
// ****************************************************************
// ListTest.java
// A simple test program that creates an IntList, puts some
// ints in it, and prints the list.
// ****************************************************************
public class ListTest{
   public static void main(String[] args) {
            IntList myList = new IntList(10);
            myList.add(100);
            myList.add(50);
            myList.add(200);
            myList.add(25);
            System.out.println(myList);
   }
}
```