Chapter 8: Inheritance
Lab Exercises

Topics	Lab Exercises
Inheritance	Exploring Inheritance
	A Sorted Integer List
	Test Questions
	Overriding the *equals* Method

Exploring Inheritance

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());

    }
}
```

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);
    }
}
```

Test Questions

In this exercise you will use inheritance to read, store, and print questions for a test. First, write an abstract class TestQuestion that contains the following:

- A protected String variable that holds the test question.
- An abstract method *protected abstract void readQuestion()* to read the question.

Now define two subclasses of TestQuestion, Essay and MultChoice. Essay will need an instance variable to store the number of blank lines needed after the question (answering space). MultChoice will not need this variable, but it will need an array of Strings to hold the choices along with the main question. Assume that the input is provided from the standard input as follows, with each item on its own line:

- type of question (character, m=multiple choice, e=essay)
- number of blank lines for essay, number of blank lines for multiple choice (integer)
- choice 1 (multiple choice only)
- choice 2 (multiple choice only) ...

The very first item of input, before any questions, is an integer indicating how many questions will be entered. So the following input represents three questions: an essay question requiring 5 blank lines, a multiple choice question with 4 choices, and another essay question requiring 10 blank lines:

```
3
e
5
Why does the constructor of a derived class have to call the constructor
of its parent class?
m
4
Which of the following is not a legal identifier in Java?
guess2
2ndGuess
_guess2_
Guess
e
5
What does the "final" modifier do?
```

You will need to write *readQuestion* methods for the MultChoice and Essay classes that read information in this format. (Presumably the character that identifies what kind of question it is will be read by a driver.) You will also need to write *toString* methods for the MultChoice and Essay classes that return nicely formatted versions of the questions (e.g., the choices should be lined up, labeled a), b), etc, and indented in MultChioce).

Now define a class WriteTest that creates an array of TestQuestion objects. It should read the questions from the standard input as follows in the format above, first reading an integer that indicates how many questions are coming. It should create a MultChoice object for each multiple choice question and an Essay object for each essay question and store each object in the array. (Since it's an array of TestQuestion and both Essay and MultChoice are subclasses of TestQuestion, objects of both types can be stored in the array.) When all of the data has been read, it should use a loop to print the questions, numbered, in order.

Use the data in *testbank.dat* to test your program.

```
testbank.dat

5
e
5
Why does the constructor of a subclass class have to call the constructor of its
parent class?
m
4
Which of the following is not a legal identifier in Java?
```

Overriding the *equals* Method

File *Player.java* contains a class that holds information about an athlete: name, team, and uniform number. File *ComparePlayers.java* contains a skeletal program that uses the Player class to read in information about two baseball players and determine whether or not they are the same player.

1. Fill in the missing code in ComparePlayers so that it reads in two players and prints "Same player" if they are the same, "Different players" if they are different. Use the *equals* method, which Player inherits from the Object class, to determine whether two players are the same. Are the results what you expect?

2. The problem above is that as defined in the Object class, *equals* does an address comparison. It says that two objects are the same if they live at the same memory location, that is, if the variables that hold references to them are aliases. The two Player objects in this program are not aliases, so even if they contain exactly the same information they will be "not equal." To make *equals* compare the actual information in the object, you can override it with a definition specific to the class. It might make sense to say that two players are "equal" (the same player) if they are on the same team and have the same uniform number.
   - Use this strategy to define an *equals* method for the Player class. Your method should take a Player object and return true if it is equal to the current object, false otherwise.
   - Test your ComparePlayers program using your modified Player class. It should give the results you would expect.

```
// ********************************************************
// Player.java
//
// Defines a Player class that holds information about an athlete.
// ********************************************************

import java.util.Scanner;

public class Player
{
    private String name;
    private String team;
    private int jerseyNumber;


    //--------------------------------------------------------
    // Prompts for and reads in the player's name, team, and
    // jersey number.
    //--------------------------------------------------------

    public void readPlayer()
    {
      Scanner scan = new Scanner(System.in);
      System.out.print("Name: ");
      name = scan.nextLine();
      System.out.print("Team: ");
      team = scan.nextLine();
      System.out.print("Jersey number: ");
      jerseyNumber = Scan.nextInt();
    }

}
```

```
// ************************************************************
// ComparePlayers
//
// Reads in two Player objects and tells whether they represent
// the same player.
// ************************************************************
import java.util.Scanner;
public class ComparePlayers
{
    public static void main(String[] args)
    {
      Player player1 = new Player();
      Player player2 = new Player();

      Scanner scan = new Scanner();

      //Prompt for and read in information for player 1

      //Prompt for and read in information for player 2

      //Compare player1 to player 2 and print a message saying
      //whether they are equal

    }
}
```