Lab 9 Polymorphism

Another Type of Employee

The files *Firm.java*, *Staff.java*, *StaffMember.java*, *Volunteer.java*, *Employee.java*, *Executive.java*, and *Hourly.java* contain a program that illustrates inheritance and polymorphism. In this exercise you will add one more employee type to the class hierarchy. The employee will be one that is an hourly employee but also earns a commission on sales. Hence the class, which we'll name Commission, will be derived from the Hourly class.

Write a class named Commission with the following features:

- It extends the Hourly class.
- It has two instance variables (in addition to those inherited): one is the total sales the employee has made (type double) and the second is the commission rate for the employee (the commission rate will be type double and will represent the percent (in decimal form) commission the employee earns on sales (so .2 would mean the employee earns 20% commission on sales)).
- The constructor takes 6 parameters: the first 5 are the same as for Hourly (name, address, phone number, social security number, hourly pay rate) and the 6th is the commission rate for the employee. The constructor should call the constructor of the parent class with the first 5 parameters then use the 6th to set the commission rate.
- One additional method is needed: public void addSales (double totalSales) that adds the parameter to the instance variable representing total sales.
- The pay method must call the pay method of the parent class to compute the pay for hours worked then add to that the pay from commission on sales. (See the pay method in the Executive class.) The total sales should be set back to 0 (note: you don't need to set the hoursWorked back to 0—why not?).
- The toString method needs to call the toString method of the parent class then add the total sales to that.

To test your class, update Staff.java as follows:

- Increase the size of the array to 8.
- Add two commissioned employees to the staffList—make up your own names, addresses, phone numbers and social security numbers. Have one of the employees earn $6.25 per hour and 20% commission and the other one earn $9.75 per hour and 15% commission.
- For the first additional employee you added, put the hours worked at 35 and the total sales $400; for the second, put the hours at 40 and the sales at $950.

Compile and run the program. Make sure it is working properly.

Painting Shapes

In this lab exercise you will develop a class hierarchy of shapes and write a program that computes the amount of paint needed to paint different objects. The hierarchy will consist of a parent class Shape with three derived classes - Sphere, Rectangle, and Cylinder. For the purposes of this exercise, the only attribute a shape will have is a name and the method of interest will be one that computes the area of the shape (surface area in the case of three-dimensional shapes). Do the following.

1. Write an abstract class Shape with the following properties:
   - An instance variable shapeName of type String
   - An abstract method area()
   - A toString method that returns the name of the shape
2. The file Sphere.java contains a class for a sphere which is a descendant of Shape. A sphere has a radius and its area (surface area) is given by the formula 4*PI*radius^2. Define similar classes for a rectangle and a cylinder. Both the Rectangle class and the Cylinder class are descendants of the Shape class. A rectangle is defined by its length and width and its area is length times width. A cylinder is defined by a radius and height and its area (surface area) is PI*radius^2*height. Define the toString method in a way similar to that for the Sphere class.
3. The file Paint.java contains a class for a type of paint (which has a "coverage" and a method to compute the amount of paint needed to paint a shape). Correct the return statement in the amount method so the correct amount will be returned. Use the fact that the amount of paint needed is the area of the shape divided by the coverage for the paint. (NOTE: Leave the print statement - it is there for illustration purposes, so you can see the method operating on different types of Shape objects.)

4. The file PaintThings.java contains a program that computes the amount of paint needed to paint various shapes. A paint object has been instantiated. Add the following to complete the program:

   - Instantiate the three shape objects: deck to be a 20 by 35 foot rectangle, bigBall to be a sphere of radius 15, and tank to be a cylinder of radius 10 and height 30.
   - Make the appropriate method calls to assign the correct values to the three amount variables.
   - Run the program and test it. You should see polymorphism in action as the amount method computes the amount of paint for various shapes.

Polymorphic Sorting

The file *Sorting.java* contains the Sorting class that implements both the selection sort and the insertion sort algorithms for sorting any array of Comparable objects in ascending order. In this exercise, you will use the Sorting class to sort several different types of objects.

1. The file *Numbers.java* reads in an array of integers, invokes the selection sort algorithm to sort them, and then prints the sorted array. *Numbers.java* won't compile in its current form. Study it to see if you can figure out why.

2. Try to compile Numbers.java and see what the error message is. The problem involves the difference between primitive data and objects. Change the program so it will work correctly
3. Write a program *Strings.java*, similar to *Numbers.java*, that reads in an array of String objects and sorts them. You may just copy and edit *Numbers.java*.
4. Modify the insertionSort algorithm so that it sorts in descending order rather than ascending order. Change *Numbers.java* and *Strings.java* to call insertionSort rather than selectionSort. Run both to make sure the sorting is correct.
5. The file *Salesperson.java* partially defines a class that represents a sales person. A sales person has a first name, last name, and a total number of sales (an int) rather than a first name, last name, and phone number. Complete the compareTo method in the Salesperson class. The comparison should be based on total sales; that is, return a negative number if the executing object has total sales less than the other object and return a positive number if the sales are greater. Use the name of the sales person to break a tie (alphabetical order).
6. The file WeeklySales.java contains a driver for testing the compareTo method and the sorting. Compile and run it. Make sure your compareTo method is correct. The sales staff should be listed in order of sales from most to least with the four people having the same number of sales in reverse alphabetical order.
7. OPTIONAL: Modify WeeklySales.java so the salespeople are read in rather than hardcoded in the program.

Searching and Sorting An Integer List

File *IntegerList.java* contains a Java class representing a list of integers. The following public methods are provided:

- IntegerList(int size)—creates a new list of size elements. Elements are initialized to 0.
- void randomize()—fills the list with random integers between 1 and 100, inclusive.
- void print()—prints the array elements and indices
- int search(int target)—looks for value target in the list using a linear (also called sequential) search algorithm. Returns the index where it first appears if it is found, -1 otherwise.
- void selectionSort()—sorts the lists into ascending order using the selection sort algorithm.

File *IntegerListTest.java* contains a Java program that provides menu-driven testing for the IntegerList class. Copy both files to your directory, and compile and run IntegerListTest to see how it works. For example, create a list, print it, and search for an element in the list. Does it return the correct index? Now look for an element that is not in the list. Now sort the list and print it to verify that it is in sorted order.

Modify the code in these files as follows:

1. Add a method void replaceFirst(int oldVal, int newVal) to the IntegerList class that replaces the first occurrence of oldVal in the list with newVal. If oldVal does not appear in the list, it should do nothing (but it's not an error). If oldVal appears multiple times, only the first

occurrence should be replaced. Note that you already have a method to find oldVal in the list; use it!

Add an option to the menu in IntegerListTest to test your new method.

2. Add a method void replaceAll(int oldVal, int newVal) to the IntegerList class that replaces all occurrences of oldVal in the list with newVal. If oldVal does not appear in the list, it should do nothing (but it's not an error). Does it still make sense to use the search method like you did for replaceFirst, or should you do your own searching here? Think about this.

   Add an option to the menu in IntegerListTest to test your new method.

3. Add a method void sortDecreasing() to the IntegerList class that sorts the list into decreasing (instead of increasing) order. Use the selection sort algorithm, but modify it to sort the other way. Be sure you change the variable names so they make sense!

   Add an option to the menu in IntegerListTest to test your new method.

4. Add a method int binarySearchD (int target) to the IntegerList class that uses a binary search to find the target assuming the list is sorted in decreasing order. Your algorithm will be a modification of the binary search algorithm.

   Add an option to the menu in IntegerListTest to test your new method. In testing, make sure your method works on a list sorted in descending order then see what the method does if the list is not sorted (it shouldn't be able to find some things that are in the list).

Timing Searching and Sorting Algorithms

In this exercise you will use an IntegerList2 class (in the file IntegerList2.java) and a driver (in the file *IntegerListTest2.java*) to examine the runtimes of the searching and sorting algorithms. The IntegerListTest2 class has several options for creating a list of a given size, filling the list with random integers or with already sorted integers, and searching or sorting the list. (NOTE: You may have used a version of these classes in the last lab.) Run IntegerListTest2 a few times to explore the options.

The runtimes of the sorting and searching algorithms can be examined using the Java method System.currentTimeMillis(), which returns the current system time in milliseconds. (Note that it returns a long, not an int.) You will have to import java.util.* to have access to this method. In IntegerListTest2, just get the system time immediately before and immediately after you perform any of the searches or sorts. Then subtract the first from the second, and you have the time required for the operation in milliseconds. WARNING: Be sure you are not including any input or output in your timed operations; these are very expensive and will swamp your algorithm times!

Add appropriate calls to System.currentTimeMillis() to your program, run it and fill out the tables below. Note that you will use much larger arrays for the search algorithms than for the sort algorithms; do you see why? Also note that the first couple of times you run a method you might get longer runtimes as it loads the code for that method. Ignore these times and use the "steady-

state" times you get on subsequent runs. On a separate sheet, explain the times you see in terms of the known complexities of the algorithms. Remember that the most interesting thing is not the absolute time required by the algorithms, but how the time changes as the size of the input increases (doubles here).

Array Size	Selection Sort (random array)	Selection Sort (sorted array)	Insertion Sort (random array)	Selection Sort (sorted array)
10,000				
20,000				
40,000				
80,000				

Array Size	Linear Search (unsuccessful)	Binary Search (unsuccessful)
100,000		
200,000		
400,000		
800,000		
1,600,000		