Lab 18 (November 8 or November 9)

Instructions: Complete the steps below. **Be sure to show your code to one of the lab TAs before you leave, so that you can receive credit for this lab.** You must also upload copies of all your source code (.java) files to the link on Blackboard by 11:59 PM on Thursday, November 9.

1. Many applications require a list of values to remain in sorted order, even if new values are added to the list (or old values are removed). It can be very expensive (in terms of CPU time) to continually re-sort the array every time a new value is added. A better approach is to insert new values in their proper place as they are added to the list, shifting elements down as necessary.

   Define a new class, SortedIntegerArray. This class contains two private instance variables: an integer array (**DO NOT** use an ArrayList!) and an integer that tracks how many values are currently stored in the array (a value between 0 and the array length).

   Implement the following public methods for your new data type:

   * A constructor that takes a single integer argument. The constructor should set the size instance variable to 0 (the array is initially empty) and creates a new integer array large enough to hold the specified number of values (for example, SortedIntegerArray(10) would create a new instance with a 10-element array). Unlike the stack from Lab 17, your internal array **DOES NOT** have to expand to accommodate additional values.

   * A size() method that returns the number of elements currently stored inside the array.

   * A get() method. This method takes an integer index as its argument and returns the value stored in that position. If the position does not exist (because the specified index is less than 0 or greater than or equal to size), return -1 instead.

   * A toString() method that returns a String containing all of the elements that are currently stored in the array. Your resulting String should only contain values located in "currently occupied" array positions (in other words, only include the values from index 0 up through index (size – 1))!

   * An add() method. This method takes a single integer argument and inserts that value into its proper place in the array, in sorted (ascending) order. You may need to shift other array elements over by one position each to make room for the new value; use a loop to do this, starting from the end of the array. If the array is already full (its size is equal to the array's length), do not insert the new value (print an error message if you wish, or just allow the operation to fail silently). You may assume that the array will only store non-negative integers (0 or greater). Be sure to update the size variable if your addition is successful!

* A remove() method. This method takes an array index (an integer in the range 0 through (size - 1)) as its argument. If the index is valid, remove the element at that position and shift any following elements forward to close the gap (don't forget to update the size variable as well). This method does not return anything.

Finally, add a main() method that creates a new SortedIntegerArray object and displays the result of inserting several values into the array. Show that your implementation correctly places values into their proper sorted positions, and that your size(), get(), and toString() methods work correctly.

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

   Complete the main() method in ComparePlayers.java so that it reads in information for two players and prints "Same player" if they are the same, or "Different players" if they are different. Use the equals() method, which Player inherits from the Object class, to determine whether the two players are the same. Are the results what you expect?

   The problem is that, as defined by the Object class, equals() performs a shallow address comparison. It says that two objects are only the same if they live at the same memory location, that is, if the variables that hold references to them are aliases. Our 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 contained in the object, you must override it with a definition specific to the class. In this case, it makes sense to say that two players are "equal" (the same player) if they are on the same team and have the same uniform number (their names might differ if one name is actually a nickname or other variant of the player's official name).

   Use this strategy to define a new equals() method for the Player class. Note that the header for equals() (which you are overriding) **MUST** be:

   ```
   public boolean equals(Object o)
   ```

   This means that Player's version of equals() must first cast its Object parameter to a Player before returning true or false based on whether that Player object's team and uniform numbers match those of the current Player's corresponding instance variables.

   Finally, test your ComparePlayers program using your modified Player class. It should now give the results you would expect.

Grading Guidelines: This lab is graded on a scale of 0-3 points, assigned as follows:

0 points: Student is absent or does not appear to have completed any work for the lab

1 point: Student has completed some work, but neither program compiles or runs.

2 points: Student has correctly completed only one of the programs. The second program is in-progress, but not yet functional (it may not even compile at this stage).

3 points: Student has correctly completed both programs, without any apparent errors.