Quiz #4: Writing Classes and methods for Fixed-sized Arrays INF-42, Winter 2011

Name _____

Lab _____

Recall the Academic Integrity statement that you signed. Write all answers clearly on these pages, ensuring your final answers are easily recognizable. The number of points for each problem is clearly marked, for a total of 25 points. I will post my solutions on the web on Monday, off the **Solutions** link, after class.

---

1. (15 pts) Write a class named **InRangeCatenate**. Each instance of this class stores four fields: the lower and upper values of the range, the catenation of all the values in this range that it is has seen (via the `seeIt` method described below), and the total number of values that is has seen (whether or not they are in the range). Its field declarations and constructor should initialize these fields appropriately, with the constructor throwing an `IllegalArgumentException` if its first parameter is bigger than its second parameter. See below for the declaration and use (via accessors and mutators) of a variable of the class **InRangeCatenate**.

Its accessor, **getCatenation**, should return the catenation of all values seen which were in the range; its accessor, **getCount**, should return a count of all of values that it has seen; its mutator, **seeIt**, is passed a **String** parameter that is conditionally catenated (and always counted), changing all appropriate fields.

For example, if we write `InRangeCatenate c = new InRangeCatenate("cat","mouse");` and execute the statements `c.seeIt("dog"); c.seeIt("zebra"); c.seeIt("gerbil"); c.seeIt("cow"); c.seeIt("ant"); System.out.println("\""+c.getCatenation()+":"+c.getCount()+"\"");` Java will print the `String "dog gerbil cow:5"` because `"dog"`, `"gerbil"` and `"cow"` are all in the range specified in the constructor's arguments (comparing strictly bigger than `"cat"` and comparing strictly less than `"mouse"` -but `"zebra"`and `"ant"` aren't in that range) and a total of `5` values were seen. Notice that spaces in the catenated `String` occur only **between** tokens (there is no space before or after the result).

Use correct syntax for defining this class and all its members (constructor, methods, and fields).

Write the fields first in the **left** column, followed by the constructor; in the **right** column, write the accessor methods first, followed by the mutator; use appropriate access modifiers. Hint: see **String**'s **compareTo** method.

For 2-6: Each of these methods can be written with one, fairly short loop (and a bit more code). Determine the pattern of scanning/movement in the body of the loop, and then write the `for` loop bounds to scan/move the appropriate values. Hand simulate a small example before typing your code. Note that none of these methods change the length of the array. Also note that none of these methods will need to check for or throw exceptions. I think they are in increasing order of complexity. Put in stub methods for each so that you can compile/run the driver, testing each new method as you write it.

In most cases, I have put simple ascending values in the array (for easily checking your methods), but your code should work independent of the values stored in the array: it should work if the values are in reverse order, or in any random order.

2. (2 pts) Write a method named `countOccurrences`, which takes two parameters: an `int[]` and an `int`. It returns the number of times that the `int` occurs in the `int[]`; it does not change the array.

3. (2 pts) Write a method named `replace`, which takes three parameters: an `int[]` and two `int` (values). It replaces every occurrence in the array of the first `int` value with the second `int` value and returns the number of replacements it made (possibly 0).

4. (2 pts) Write a method named `moveToRear`, which takes two parameters: an `int[]` and an `int`. It changes the array so that the value at the specified index is moved to the rear; all values following it are moved towards the front by one index. Calling `moveToRear` with a second argument of `7` means to move the value **at index** `7` to the rear (it does NOT mean move **the value** `7` to the rear).

5. (2 pts) Write a method named `moveToFront`, which takes two parameters: an `int[]` and an `int`. It changes the array so that the value at the specified index is moved to the front; all values preceding it are moved towards the rear by one index. Calling `moveToFront` with a second argument of `7` means to move the value at index `7` to the front (it does NOT mean move **the value** `7` to the front).

6. (2 pts) Write a method named `reverse`, which takes one parameter: an `int[]`. It changes the array, so that its values are reversed: the value at the first index and the value at the last index are exchanged; the values at the second index and the second to last index are exchanged, etc. **Do not** declared/allocate another array. Hint: it is easy to write something that ends up reversing the order twice, leaving the array unchanged: hand simulation will show you your error.

Important: There is a **downloadable** Eclipse project file that you should do all your work in (see the link on the Weekly Schedule page for Friday 1/27), for both problems. Test your code using the drivers supplied. If your code is not working (especially the array code) hand simulate it to understand where it is going wrong and how to fix it (likewise you can use the debugger and step through the code).

The description of the first problem provides lots of clues about how to declare and initialize the four required instance variables (and the constructor that reinitializes some of these instance variables) and how to write the two accessor and one mutator methods. Look at the code for the `SimpleDiceEnsemble` class (follow the Sample Programs link and download the SimpleDiceEnsemble Demontration) which shows all the features needed in your class.

Write your solution to Problem #1 on the first page. Submit your solutions to Problems 2-6 as Java files online using Checkmate. Recall that this assignment is due on Monday by the start of class.