Name: _____

USC loginid (e.g., ttrojan):_____

CS 455 Midterm Exam 1
Fall 2010 [Bono]

Sept. 29, 2010

There are 4 problems on the exam, with 55 points total available. There are 7 pages to the exam, including this one; make sure you have all of them. There is also a double-sided one-page code handout that accompanies the exam. If you need additional space to write any answers, you may use the backs of exam pages (just direct us to look there).

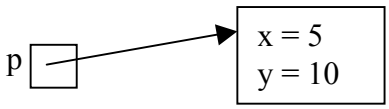
Remote DEN students only: Do not write on the backs of pages. If additional space is needed, ask proctor for additional blank page(s), put your name on them, and attach them to the exam.

Put your name and USC ID number at the top of the exam. Please read over the whole test before beginning. Good luck!

	value	score
Problem 1	10 pts.	
Problem 2	4 pts.	
Problem 3A	8 pts.	
Problem 3B	8 pts.	
Problem 4	25 pts.	
TOTAL	55 pts.	

Problem 1 [10 pts.]

Consider the following code fragment:




```
Point p = new Point(5, 10);
Point r;
Point s = new Point(12, 20);
s = p;
p.translate(50, 100);
System.out.println(s.getX());
```

Part A [6]. To the right of the code, complete the diagram started above so it shows all object variables, objects, and their state as they change during the code sequence.

Part B [2]. What is printed by the code?

Part C [2]. How many `Point` objects are created by the code above?

Problem 2 [4 pts.]

Consider the following version of the `Drunkard` class from PA1, and a code fragment that uses it.

```
public class Drunkard {
   public Drunkard(Point startLoc, int aStepSize) {
      . . . [code to init any other fields not shown]
      Point currentLoc = new Point(startLoc);
      stepSize = aStepSize;

   }
   . . . [other methods / fields not shown]
   private Point currentLoc;
   private int stepSize;
}
```

```
Drunkard joe = new Drunkard(new Point(10, 20), 5);
```

Part A [2]. The `Drunkard` class compiles, but has an error. (Hint: it's related to variable declarations.) Show the exact value for `joe` after executing the code fragment above (i.e., do not fix the error: describe what the code does as is). Use a "box-and-pointer diagram" (there was an example of such a diagram in Problem 1).

Part B [2]. Fix the code (make your changes right in the class code above – don't rewrite it).

Problem 3 [16 pts. total]

Part A [8]. Consider the following code we discussed in lecture to look up a value in an `ArrayList` using linear search. The current version of the method works on ordered or unordered `ArrayList`s and it always looks at all `size()` elements in the `ArrayList` for unsuccessful searches. Modify the code so that the new version only works on `ArrayList`s whose elements are in increasing order, and takes advantage of the ordering to, in general, perform better on unsuccessful searches than the old version. That is, it will stop looking when it gets to the part of the `ArrayList` that has values bigger than the one we are looking for.

For example: `target: joe`
`namesArr: bob carly john mary peter sam tom`
we don't need to look past `john` (at `mary`, `peter`, `sam`, or `tom`) to figure out that `joe` is not there.

Note: make your changes directly in the code below, or write any additional code to the right, adding arrows to show where the new code belongs. Also, see code handout for `compareTo` method for doing inequality comparisons between `String`s.

```
public class Names {

   /**
      implementation invariant:
      values in namesArr are unique and in increasing alphabetical order
   */
   private ArrayList<String> namesArr;


   /**
      returns location of target in namesArr or -1 if not found
   */
   private int lookupLoc(String target) {   [do not change the method header]

     int loc = 0;

     while (loc < namesArr.size() && (!target.equals(namesArr.get(loc))) {

       loc++;

     }


     if (loc == namesArr.size()) {

       return -1;

     }
     else {

       return loc;

     }

   }
   . . . [rest of Names class not shown]
```

```
}
```

Problem 3 (cont.)

Part B [8]. Come up with a good set of test cases to thoroughly test our new version of the code (i.e., this should include cases we would have used on the old version as well as ones designed to exercise all parts of the new code).

Use the following `Names` object contents in some or all of your tests:

```
namesArr:  bob   carly    john   mary    peter   sam    tom
```

For each test case,

a. give the exact input that would be used (`target`; and `namesArr`, if different from the `namesArr` given above),

b. show the expected result (i.e., expected return value of the function) for that case, and

c. describe what case is being tested by that input.

`target` **(and possibly `namesArr`)**	**expected result**	**case tested**

Problem 4 [25 pts. total]

Consider the following class to simulate a die (i.e., most commonly is 6-sided, plural is dice). Assume someone already implemented the class for us.

```
public class Die {

  // constructs a die with the given number of sides
  // pre: numSides >= 1
  public Die(int numSides) { . . . }

  // simulates one throw of the die
  // returns a random number in the range [1..numSides]
  public int throw() { . . . }

  // return number of sides of the die
  public int getNumSides() { . . . }

  . . .

}
```

Example of using the `Die` class:

```
Die die = new Die(6);             // 6-sided die
System.out.println(die.throw());  // rolls a 3 (actual results are random)
System.out.println(die.throw());  // rolls a 1
System.out.println(die.throw());  // rolls a 5
```

In this problem you are going to use the `Die` class to implement a class to simulate what happens when throwing a pair of dice many times. It will keep track of how many times each possible sum of the two dice occurs. For 6-sided dice the smallest such sum (called a *value* below) is 2 (rolling two ones), and the largest is 12 (rolling two sixes). Here is an example of using the simulator:

```
DiceSimulator sim = new DiceSimulator(6);  // two 6-sided dice

sim.run(10000);    // what happens over 10000 rolls of each die?

sim.printResults();
```

Sample output produced by `printResults` (some lines not shown):

```
Results of throwing the dice 10000 times:
value  number of throws
2  280
3  557
4  834
. . .
12  275
```

Note: To simplify this problem, you may assume each of the three `DiceSimulator` member functions will be called only once, as shown in the example above.

The class interface is given on the next page, with space for your answer. To get full credit you must solve the problem efficiently, taking advantage of Java features we have learned about.

Problem 4 (cont.)

```
public class DiceSimulator {
```

[Note: space for instance vars given first.]

```
  // constructs a dice simulator using dice with the given number of sides
  // pre: numSides >= 1
  public DiceSimulator(int numSides)
```

```
  // run the simulation for the given number of throws
  // pre: numThrows >= 0
  public void run(int numThrows)
```

```
  // prints out the results of the simulation:
  // shows for each of the values from 2 to 2*numSides
  //    how many times we rolled that value
  // [see sample output for details]
  public void printResults()
```

```
}
```