CS100J **October 1, 2001**
Prelim 1 **7:30 PM – 9:00 PM**

_____ _____
(Print last name, first name, middle initial/name) (Student ID)

Statement of integrity: I did not, and will not, break the rules of academic integrity on this exam:

_____
(Signature)

Circle Your Section:

	Monday		Tuesday		
	HO 306	UP 111	BD 140	UP 111	UP 215
10:10			13: Lu		
1:25	10: Pappu				16: Singer
2:30		11: Pappu		14: Singer	
3:35		12: Lysiuk		15: Scovetta	

Instructions:

- Read each problem *completely* before starting it!
- Do not use calculators, reference sheets, or any other material. This test is closed book.
- Solve each problem using Java, except where indicated.
- You may *not* use **`switch`**, **`for`**, **`do-while`**, nested loops, your own methods, arrays, or your own classes.
- Write your solutions directly on the test using blue/black pen or pencil. Clearly indicate which problem that you are solving. You may write on the back of each sheet. If you need scrap paper, ask a proctor.
- Provide only *one* statement, expression, modifier, type, or comment per blank!
- Do *not* alter, add, or remove any code that surrounds the blanks and boxes.
- Do *not* supply multiple answers. If you do so, we will grade only one that we will choose.
- Show all work, especially algorithms. Better that you explain how you would solve a problem than to leave it blank.
- Follow good style! When possible, keep solutions general, avoid redundant code, use descriptive variables, use named constants, indent substructures, avoid breaking out of loops, and maintain other tenets of programming philosophy.
- Comment each control structure and major variable, *briefly.*
- Do not dwell on a problem if you get stuck. Do the other problems first!
- Raise your hand if have any questions.

Core Points:

1. _____ (25 points) _____

2. _____ (35 points) _____

3a. _____ (05 points) _____

3b. _____ (35 points) _____

Total: _____/(100 points) _____

Bonus Points:

Bonus: _____/(4 points)

Problem 1 [25 points] *Selection Statements, Boolean Values & Operators, Assignment Statements*

Background: An *exclusive-or* (`xor`) is a boolean operator that has a different behavior than an *or* operator. When comparing two boolean expressions with an exclusive-or, the result is **`true`** if the expressions have different boolean values. Otherwise, the result is **`false`**. For example, **`true`** `xor` **`true`** is **`false`**, whereas **`true`** `xor` **`false`** is **`true`**.

1a [22 points] Java does have an exclusive-or operator, but you will not use it. Instead, we want you to write a series of selection statements in class **`Problem1`** to simulate **`v1`** `xor` **`v2`**. Your program will store the result of the simulated `xor` operation in the variable **`result`**, but not print it. For full credit, your code must be concise and not use any operators other than assign (**`=`**).

```
public class Problem1 {
   public static void main(String[] args) {

      // Variables for testing the code that simulates an xor:
         boolean v1 = false;   // first boolean value
         boolean v2 = true;    // second boolean value
         boolean result;       // result of ''v1 xor v2''

      // Use selection statements to simulate ''v1 xor v2'' and
      // store the result in $result$:
```

```
         if (v1)
            if (v2)
               result=false; // T xor T --> false
            else
               result=true;  // T xor F --> true
         else if (v2)
            result=true;     // F xor T --> true
         else
            result=false;    // F xor F --> false
```

```
   } // method main
} // class Problem1
```

1b [1 point] If **`v1`** and **`v2`** were both **`false`**, what value would be assigned to **`result`**?

false

1c [2 points] For two core points, what Java operator, besides Java's actual exclusive-or, produces the same result as an exclusive-or? For one bonus point, what *is* Java's exclusive-or operator? Hint: It's not `xor`.

2 core points: **`!=` (not equals)**
1 bonus point: **`^`**

Problem 2 [35 points] *Definite Iteration, Input, Arithmetic Operators, Types, Random Numbers*

Definitions: *Frequency* is the percentage of times that a particular value appears in a collection of data. If you could determine the exact amount of times that a value appears, you can determine the *ideal frequency.* For example, if you have four marbles with different colors, each marble has an ideal frequency of 25%. However, in a large collection of randomly selected marbles, the frequency of each color might vary from the ideal frequency.

Background: Suppose that you would like to know more about the randomness of the numbers that **`Math.random()`** generates. By generating several random numbers in a given range and inspecting their frequencies, you can estimate the degree of randomness. For instance, you could use **`Math.random()`** to generate two integers (1 and 2) in a loop and count the number of times each integer appears.

Tasks: To test the randomness of **`Math.random()`**, complete class **`Problem2`**, which does the following:

- Initializes test data. Your program will test **`Math.random()`** with just two integers, 1 and 2.
- Prompts the user to enter the number of test cases, which is stored in **`testCases`**. This integer value represents the number of times that the program should generate a random number. [Assume that the input is positive and legal.]
- Generates a random number for each test and increments the appropriate counter.
- Reports the percent errors of the evaluated frequencies to the ideal values.

```
public class Problem2 {
   public static void main(String[] args) {
      // Initialize data to test:

         final int FIRST = _1_;      // first number to count

         final int SECOND = _2_;     // second number to count

         int count1 = _0_;           // count of 1st number so far

         int count2 = _0_;           // count of 2nd number so far

         double idealPercent1 = _50_ ; // expected frequency of the 1st number

         double idealPercent2 = _50_ ; // expected frequency of the 2nd number

      // Initialize test case information:

         System.out.print("Enter the number of tests: ");

         int testCases = SavitchIn.readInt(); // # of times to generate random #

         int testCount = _0_ ; // number of tests performed so far
```

[This space is deliberately blank. The problem continues on the next page.]

```
      // Determine total counts of $FIRST$ and $SECOND$ by running several testcases:
         while ( testValue < testCases ) {
            // Generate a random number: either $FIRST$ or $SECOND$:
               int testValue = (int) (Math.random()*(HIGH-LOW+1)) + (int)(LOW) ;
            // Increment counters for just $FIRST$ and $SECOND$:
```

```
            if (testValue == FIRST)
               testCount1++;
            else if (testValue == SECOND)
               testCount2++;
```

```
            // Increment count of tests:
               testCount++ ;
         } // end while
      // Determine frequency of $FIRST$ and $SECOND$:
         double myPercent1 = 100.0*testCount1/testCases ; // frequency of 1st number
         double myPercent2 = 100.0*testCount2/testCases ; // frequency of 2nd number
      // Determine percent error for $FIRST$ and $SECOND$:
         System.out.println("Percent error for the 1st value: " +
                            100*Math.abs(myPercent1 - idealPercent1)/idealPercent1);
         System.out.println("Percent error for the 2nd value: " +
                            100*Math.abs(myPercent2 - idealPercent2)/idealPercent2);
   } // method main
} // class Problem2
```

[This space is deliberately blank.]

Problem 3 [40 points] *Simulation, Algorithms, Indefinite Iteration, Random Numbers*

Background: *DIS.com* needs you to simulate the number of boxes a worker can stack on eight carts in one day. A chute issues one box at a time for the worker to place on a cart. The boxes have random sizes, which are ranked from 1 to 5. A box of size 5 (#5) must be at the bottom of the stack on each cart. All other boxes may be stacked in any order. Since the worker does not want to bother with removing boxes from a cart, the simulation has these assumptions:

- A cart has no limit to the number of boxes it can store.
- The worker will not shuffle the boxes.
- Each time a #5 box appears, the worker starts stacking a new cart with the #5 box at the bottom.
- Another worker has already filled the first and second carts with three and seven boxes, respectively.
- The third cart currently has two boxes stacked on it, so far.
- The first cart has a #5 box at the bottom.

Since the company is very cheap, it has only five more empty carts for the worker to use. So, when the ninth #5 box appears, the worker cannot continue stacking boxes. However, the company will only produce forty boxes in one day. So, the worker might not need to use all of the carts.

3a [5 points]

Write a *brief* algorithm that describes how to simulate this problem. Ultimately, the program must report the following:

- the total number of carts needed
- the total number of boxes stacked on all of the carts, which is the sum of *all* boxes on *all* carts.

Refer to Problem 3b for a partial implementation of the code to help you write the algorithm.

Spew one box of random size (1 to 5) from chute
If the amount of carts (8) and amount of boxes (40) is not exhausted:

- If the current box is not a #5
  - Place the box on the current cart.
  - Increment total number of boxes on carts.
- Otherwise
  - Increment the cart count.
  - Check if the carts are exahusted.
- If the carts are not exhausted,
  - Place the current box on the new cart.
  - Increment total number of boxes on carts.
- Repeat.

If necessary, adjust the cart count and/or box count to account for whether or not they were incremented too many times.
Report the number of carts used and total boxes taken.

3b [35 points] (Problem 3 continued)

Complete class **`Problem3`** that determines the number of carts used and the number of boxes that the worker stacks on all of the carts by doing the following:

- Initialize the following variables: maximum number of boxes that the company produces (**`maxBoxes`**), total number of boxes placed on the carts (**`totalBoxes`**), total number of carts for storing boxes (**`cartTotal`**), current count of cart that is being used (**`cartCount`**), and the current box size that worker will attempt to stack (**`currentBox`**).
- Use a loop with selection statements inside of it to simulate whether or not the current box gets stacked and if a new cart must be chosen. Boxes of size #5 must be placed on a new cart, assuming that the carts have not been exhausted.
- Report the number of carts that were used and the total number of boxes stored on all carts.

```
public class Problem3 {
   public static void main(String[] args) {
      // Initialize data to test:
         final int MIN = 1;       // minimum box size
         final int MAX = 5;       // maximum box size

         int maxBoxes =   40 ;   // max # of boxes that the company produces

         int totalBoxes = 12 ;   // total # of boxes taken so far

         int cartTotal  = 8 ;    // total amount of carts

         int cartCount  = 3 ;    // cart count so far

         int currentBox = (int)(Math.random()*(MAX-MIN+1)) + (int)MIN; //random size

      // Attempt to place boxes on carts until boxes or carts are used up:

         while ( cartCount <= cartTotal && totalBoxes+1 <= maxBoxes ) {
```

```
      // Stack boxes on old cart:
         if (currentBox != MAX)
            totalBoxes++;

         // #5 box starts new cart:
            else {
               cartCount++;

            // Can #5 be stacked?
               if (cartCount <= cartTotal)
                  totalBoxes++;
         }

      // Obtain next box from chute:
         currentBox = (int) (Math.random()*(MAX-MIN+1)) + (int)MIN;
```

```
         } // end while
```

[Problem3b continues on the next page.]

```
      // Report number of carts used and total number of boxes on all carts:

      if (cartCount > cartTotal)
         cartCount--;
      System.out.println("Final total:"+totalBoxes);
      System.out.println("Final carts:"+cartCount);

   } // method main

} // class Problem3
```

[This space is deliberately blank.]

Checklist: Congratulations! You reached the last page of Prelim 1. Make sure that you clearly indicate your name, ID, and section. Also, re-read all of the problem descriptions/code comments/instructions. If you reached this part before exhausting the allotted time, check your test! Maybe you made a simple mistake? You should check the following:

____ maintained all assumptions

____ remembered punctuation, such as semicolons and braces

____ didn't confuse *equals* with *assign* operators

____ completed all tasks

____ filled in ALL required blanks

____ given comments when necessary

____ declared all variables

____ maintained case-sensitivity

____ handled "special cases" correctly

____ indicated which solution to grade if you wrote multiple attempts

---

Bonus: [3 points] You may do the following evaluation after you have finished writing and checking your prelim. We will give you extra time after the test end to complete this portion. Remember that bonus points do not count towards your core-point total! To receive bonus points, tear this sheet off from the exam, make sure the proctor records the points on the front page, and put it in a separate pile to maintain anonymity.

(1) What are 1 to 3 things we can do to improve lecture? (You may also say what you like, as well.)

(2) What are 1 to 3 things we can do to improve section? (You may also say what you like, as well.)

Circle your section instructor: LISIUK LU PAPPU SCOVETTA SINGER

(3) What are 1 to 3 things we can do to improve CS100J, overall? (You may also say what you like, as well.)