Name: ______________________________

CMSC 433 Section 0101
Spring 2014
Midterm Exam

Directions: Test is open book, open notes. Answer every question; write solutions in spaces provided. Use backs of pages for scratch work. Good luck!

Please do not write below this line.

1.	________
2.	________
3.	________
4.	________
5.	________
SCORE	________

1. (20 points) TRUE / FALSE. Answer each question below by writing "T" if you believe the statement is true and "F" if you think it is false. Each question is worth 2 points.

   (a) One of the reasons that multi-threaded programming is difficult is that concurrency does not respect procedural abstraction.

   (b) When a scheduler stops one thread so that it may execute another, the operating system must perform a context switch.

   (c) When a Java `Thread` object `t` is in the `TERMINATED` state, it may be started by executing `t.start()`.

   (d) If a program has no data races, then it is guaranteed to have no race conditions.

   (e) For a class to be thread-safe, all of its methods are required to be `synchronized`.

   (f) In Java, reads and writes of all 32-bit values are guaranteed to be atomic.

   (g) In Java, all writes to `volatile` variables are guaranteed to be visible immediately.

   (h) Constructors are guaranteed to execute atomically in Java.

   (i) `notify()` should be used rather than `notifyAll()` because the latter can induce deadlocks that the former avoids.

   (j) Initialization safety is guaranteed to hold for immutable objects in Java.

2. (20 points) Answer each of the following questions, showing your work as needed and restricting any explanations to a couple of sentences or so.

   (a) (5 points) Suppose one sequence, s_1, has 3 elements while another, s_2, has four elements, and that neither sequence shares any elements with the other. How many interleavings of s_1 and s_2 are there?

   (b) (5 points) Suppose that locks in Java were *not reentrant.* What would the following code do? Explain.

```
synchronized(obj){
  synchronized(obj){ System.out.println ("Here I am!"); }
}
```

(c) (5 points) In class we gave a strategy for deadlock avoidance in Java programs: require each thread to acquire locks in the same order. Why does this prevent deadlock?

(d) (5 points) What does it mean for an object to be "thread confined", and why is it desirable?

3. (20 points) JAVA MEMORY MODEL

   (a) (5 points) Java provides "out-of-thin-air" safety for reads of variables. What does this mean?

   (b) (5 points) Consider the following execution of a multi-threaded program with threads t_1 and t_2 (S_1 is the event sequence for t_1, and S_2 is the one for t_2).

$$\begin{array}{cc} \underline{S_1} & \underline{S_2} \\ \langle t_1, \mathsf{write}, a, 1\rangle & \langle t_2, \mathsf{write}, b, 2\rangle \\ \langle t_1, \mathsf{read}, b, 2\rangle & \langle t_2, \mathsf{read}, a, 1\rangle \end{array}$$

   Is this execution sequentially consistent? Explain.

(c) (6 points) Consider the following (partial) execution sequence of a program containing threads t_1 and t_2, with synchronization order S.

$\underline{t_1}$	$\underline{t_2}$	$\underline{S}$
$\langle t_1, \mathsf{write}, b, 1\rangle$	$\langle t_2, \mathsf{lock}, l\rangle$	$\langle t_2, \mathsf{lock}, l\rangle$
$\langle t_1, \mathsf{lock}, l\rangle$	$\langle t_2, \mathsf{write}, a, 1\rangle$	$\langle t_2, \mathsf{unlock}, l\rangle$
$\langle t_1, \mathsf{write}, a, 2\rangle$	$\langle t_2, \mathsf{unlock}, l\rangle$	$\langle t_1, \text{lock}, l\rangle$
$\langle t_1, \mathsf{unlock}, l\rangle$	$\langle t_2, \mathsf{write}, b, 2\rangle$	$\langle t_1, \mathsf{unlock}, l\rangle$

For each pair of events in the table below, write $\preceq$ between them if the first event happens-before the second, and $\not\preceq$ otherwise.

$\langle t_1, \mathsf{write}, b, 1\rangle$	$\langle t_1, \mathsf{write}, a, 2\rangle$
$\langle t_1, \mathsf{unlock}, l\rangle$	$\langle t_2, \mathsf{lock}, l\rangle$
$\langle t_2, \mathsf{write}, a, 1\rangle$	$\langle t_1, \mathsf{write}, a, 2\rangle$

(d) (4 points) In the execution given in Part 3c, is there a data race? Explain.

4. (20 points) Answer each of the following questions using 1–2 sentences.

   (a) (5 points) Why is publishing `this` in a constructor considered bad practice?

   (b) (5 points) Explain what changes would need to be made to the following class to make its objects immutable objects.

```
public class IntPair {

    private int x;
    private int y;

    IntPair (int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX () { return x; }
    public int getY () { return y; }
}
```

(c) (10 points) What is *nested monitor lockout*, and how can it be avoided?

5. (20 points) In lecture we studied a class, `BoundedCounterThreadSafe`, of *thread-safe bounded counters*. Objects in this class contained methods for querying the current state of the counter; incrementing counters up to a specified maximum; resetting the current value to 0; and asking whether or not the counter is currently at its maximum value. In `BoundedCounterThreadSafe`, increment operations were ignored when the counter was at its maximum value; in this case the method terminated immediately.

   In this question you are asked to implement a *blocking* version of bounded counters. The new class, `BlockingCounter`, provides all of the same methods as the original `BoundedCounterThreadSafe`, but the `inc()` and `reset()` operations behave differently. Threads that performing `inc()` must *wait* until the counter is no longer at its maximum value before performing the increment operation, while calls to `reset()` must notify waiting threads appropriately, as the modification to the counter value by changing it to 0 may enable pending increment operations to complete.

   Your specific task is to provide implementations for the methods `reset()` and `inc()` using the space given on the next page. You may only use synchronization constructs, such as waiting and locking, that we have studied so far this semester.

```
public class BlockingCounter {
    private int count;
    private int max;

    BlockingCounter (int max) {
        this.count = 0;
        this.max = max;
    }

    public synchronized int current () {
        return count;
    }

    // Reset count to 0; notify waiting threads appropriately.
    public synchronized void reset () {









    }

    public synchronized boolean isMaxed () {
        return (count == max);
    }

    // Wait until count is <= max, then increment count by 1.
    public synchronized void inc () {









    }
}
```