Name: ______________________________

# CMSC 433 Section 0101
# Fall 2015
# Midterm Exam

**Directions: Test is open book, open notes, open electronics; however, all electronic devices must have all wireless capability disabled ("airplane mode"). 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) The term "process" is just another word for "thread."

   (b) While a thread is performing a `Thread.sleep()` operation, the thread's state is BLOCKED.

   (c) When all user threads in an application have terminated, any running daemon threads are immediately terminated as well.

   (d) In Java, reads and writes of 64-bit reference values are atomic.

   (e) In Java, threads share both stacks and heaps.

   (f) In Java, when a thread releases a monitor lock, all writes performed by the thread are immediately visible to other threads.

   (g) Because monitor locks in Java are re-entrant, it is impossible to have deadlocks in Java programs.

   (h) When a thread enters the wait-set of an object, it releases all the locks it is holding on the monitor lock of the object.

   (i) `ConcurrentHashMap` objects in Java support weakly consistent rather than fail-fast iteration.

   (j) In Java, `CopyOnWriteArrayList` objects are best used in applications where updates to these data structures are frequent and read operations are infrequent.

2. (20 points) PUBLISHING AND VISIBILITY

   (a) (8 points) When an object is created, are all the updates made to its fields by the constructor guaranteed to be immediately visible to all threads? Explain.

   (b) (6 points) Does declaring a field within an object to be `volatile` guarantee that updates made to the field will be visible immediately to all threads? Explain.

   (c) (6 points) Several factory methods in `java.util.Collections` are provided for implementing thread-safe versions of traditional collection objects. For example, given `List<T>` object `l`, `Collections.synchronizedList(l)` returns a new `List<T>` object whose methods are synchronized and whose content is `l`. Note that a *shallow copy* of `l` is embedded inside the object created by `Collections.synchronizedList(l)`.

       Explain why publishing `l` can break the thread-safety guarantees provided by the object `Collections.synchronizedList(l)` in this case.

3. (20 points) DEADLOCK AND DEADLOCK-LIKE BEHAVIOR

   (a) (10 points) Suppose we have objects $a$, $b$ and $c$, and threads $T_1$, $T_2$ and $T_3$ given as follows.

| $\underline{T_1}$ | $\underline{T_2}$ | $\underline{T_3}$ |
|---|---|---|
| `synchronized (a) {` | `synchronized (a) {` | `synchronized (b) {` |
| `    synchronized (b) {` | `    synchronized (b) {` | `    synchronized (c) {` |
| `        synchronized (c) {` | `        synchronized (c) {` | `        synchronized (a) {` |
| `            ...` | `            ...` | `            ...` |
| `        }` | `        }` | `        }` |
| `    }` | `    }` | `    }` |
| `}` | `}` | `}` |

   Can this program deadlock? Explain your answer, either by describing a deadlock scenario that can arise or by arguing that no such scenario is possible.

(b) (5 points) Explain why using `notify()` rather than `notifyAll()` can lead to deadlocks.

(c) (5 points) Consider the following code snippet.

```
class PositiveBlockingBuffer {
    private final BlockingQueue<Integer> backingQueue;
    ...
    public synchronized boolean putIfPositive (int i)
        throws InterruptedException {
        if (i > 0) {
            backingQueue.put(i);
            return true;
        }
        else return false;
    }
}
```

Under what circumstances will a call to `putIfPositive()` cause the thread calling it to remain stuck forever? Explain.

4. (20 points) LOCKING

(a) (10 points) Suppose we want to implement a synchronized instance method `foo()` in a class `Bar`. Ordinarily we would implement it as follows.

```
public synchronized void foo() { BODY }
```

Suppose we instead want to use explicit, rather than monitor, locks. Would the following be correct? Why or why not?

```
public void foo() {
    Lock l = new ReentrantLock();
    l.lock();
    try {
        BODY
    }
    finally {
        l.unlock();
    }
}
```

(b) (10 points) Use client-side locking to implement static method `replaceIfPresent()`. Specifically, given a thread-safe `Collection` object `c`, and elements `a` and `b`, `replaceIfPresent(c,a,b)` searches for `a` in `c` and, if it finds it, removes it and adds `b` into `c`. Otherwise, it does nothing. You may assume that `c` contains at most one instance of `a`. Relevant methods from the `Collection<T>` interface include:

`contains()` Returns boolean indicating whether argument is in collection.

`remove()` Removed argument from collection.

`add()` Adds argument to collection.

Please include your code in the following template.

```
public static <T> void replaceIfPresent (Collection<T> c, T a, T b) {

}
```

5. (20 points) In this question you are asked to fill in implementation code for a *blocking buffer* that implements the following `MaxIntBuffer` interface.

```
public interface MaxIntBuffer {
    public int takeMax() throws InterruptedException;
    public void put(int i) throws InterruptedException;
}
```

The method call `put(i)` should wait until the buffer has room, then insert integer `i` into the buffer. Method `takeMax()` waits until the buffer is non-empty, then returns the largest element in the buffer, removing it from the buffer in the process.

The next page contains a skeleton implementation of a class `BlockingMaxIntBuffer`. Please fill in your implementations of the methods `put()` and `takeMax()`. You may find the following operations from the `ArrayList` class helpful.

`add(i,e)` Add element `e` at position `i` in the `ArrayList`. Elements at position `i`, `i`+1, `i`+2, are shifted to the right in the process (i.e. have 1 added to their position values).

`get(i)` Return the element stored at position `i`.

`remove(i)` Remove and return the element at position `i`. Elements at positions `i`+1, `i`+2, etc., are shifted to the left (i.e. have 1 subtracted from their position values).

You may use other `ArrayList` methods if you wish as well.

```
public class BlockingMaxIntBuffer implements MaxIntBuffer {
    private final ArrayList<Integer> buffer;
    private final int capacity;

    public BlockingMaxIntBuffer (int capacity) {
        ... // You may assume that the constructor ensures capacity > 0
    }

    public synchronized void put (int elt) throws InterruptedException {










    }

    public synchronized int takeMax() throws InterruptedException {










    }
}
```