CMSC 433 – Spring 2013 – Exam 1

Name: _ EXAMPLE SOLUTIONS ____

Directions:

Test is closed book, closed notes, closed electronics.
Answer every question; write your answers in the spaces provided.
If you need extra sheets, raise your hand.
Good luck!

"I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination."

_____

_____

_____

_____

_____

_____

Please do not write below this line on this page.

Grading:

1.	4.
2.	5.
3.	6.

1. (30 Points) Answer each of the following assuming Java and the Java Memory Model is our context. Provide one or two sentences to support your answer. A yes/no answers without a corresponding written answer will not receive points.

(a) `i++;` is an atomic operation on an `int` **NO**

Why/why not?

There is really a set of three individual low-level instructions within it; a read of the value, then an increment, then a write.

(b) It is fine to guard values stored in an `Integer` object as a way to synchronize on an integer value while its value is updated.

NO

Why/why not?

Integer objects are immutable which means that when you update it you are really creating a different object than the one on which you originally said to lock.

(c) Give shared integers **`X,Y,Z,errorCount`** all initialized to 0 and the following two threads:

Thread T1 runs: **`while(true){X=Y; Y=Z+1; Z=X+1;}`**
Thread T2 runs: **`while(true){if (X>Z) errorCount++;}`**

The value in **`errorCount`** will *always* be 0. **NO**

Why/why not?

X could be greater than Z because X and Z are not guaranteed to be flushed from local caches or read into local caches in the same way as each other.

(d) Regardless of the programmer, a **`private`** field in a class can never escape.

NO

Why/why not?

If a method returns a reference to a private field, it has now escaped.

(e) Assuming that **`SafeClass`** is a thread-safe class, is **`IsItSafe`** thread-safe as a class definition or are there any data races or deadlock risks possible?

```
public class IsItSafe {
   private  SafeClass latest;
   public  SafeClass  getter() {
      synchronized (latest) {return latest;}
   }
   public void setter(SafeClass latestIn) {
      synchronized (latest) {latest = latestIn;}
   }
}
```

NO

Why/why not?

Since *latest* is a reference and is not final and the setter can set it to a different object, it is possible that the getter and setter end up 'synchronizing' on different objects, so not really synchronizing at all!

(f) Would objects of the following class be ***immutable*** according to the rules of Java?

```
public class Point2D {
   private int xVal;
   private int yVal;

   Point2D (int x, int y) { xVal = x; yVal = y; }
   int getX() = { return xVal; }
   int getY() = { return yVal; }
}
```

NO

Why/why not?

To be immutable, it needs to have all fields marked as final.

2. (10 Points)
Main Method: White House defines:

```
LinkedList<BudgetItems> cutCandidates = new LinkedList<BudgetItems>();
```

and creates two objects that instantiate thread-based classes House and Senate that are each passed a reference to this shared list via the constructor and have a `run()` method as given below. The Main Method then calls `.start()` on both thread objects. From the starter code for the two `run()` methods below, ***fill in the code required*** (if any) to guard the shared data structure in a way that maintains good opportunities for concurrency. You may assume that the helper method `pickRandomProgram` is thread-safe.

In the **House** `run()` method, we start with the following code:

```
for (int i=0; i<5; i++) {

    BudgetItem oneToCut = House.pickRandomProgram();

    synchronized(cutCandidates) {     ←has to be here to have most
    cutCandidates.add(oneToCut);                    possible concurrency.
    }

}
```

In the **Senate** `run()` method, we start with the following code:

```
for (int i=0; i<5; i++) {

    BudgetItem oneToCut = Senate.pickRandomProgram();

    synchronized(cutCandidates) {
    cutCandidates.add(oneToCut);
    }

}
```

3. (10 Points) Answer each of the following with two to three sentences.

(a) What is the difference between “a data race” and “non-determinism” in Java?

With non-determinism, you can't predict the overall result of a program because you don't know which thread will get a lock first. With a data race you can't predict the overall result of a specific order of events due to a lack of a happens-before relationship guaranteeing visibility at a certain point.

(b) Assume you had a class **`Worker`** that had a static Integer field keeping track of how many **`Worker`** objects were created. Your program will start an arbitrary number of threads, and each wants to be able to have its **`Worker`** objects know how many were created within its own thread, but not how many were created by other threads. What Java mechanism would most easily support this and how?

Using a ThreadLocal <Integer>.

4. (20 Points)

(a) Assuming that **`SafeClass`** is a thread-safe class and we have a single shared static **`SafeClass`** object referred to by **`value`** in our class, and three methods named **`A,B,C`** each of which contains a block of code surrounded by **`synchronized(value)`**.

From within its synchronized block of code Method A calls Method B.
From within its synchronized block of code Method C calls Method B.

Given this context, answer each of the following:
(i) Assume only one thread is calling Method A. Why doesn't Method B block on value seeing as Method A already has its lock?

Intrinsic locks are re-entrant (ie: once a thread holds the lock, if the thread asks for it again, it's given it again).

(ii) Assume you have two threads; T1 which repeatedly calls Method A and T2 which repeatedly calls Method C. Can the threads ever deadlock? Briefly explain why or why not.

No, it cannot deadlock. Imagine a hierarchy for the locks as A > C > B and the above follow the hierarchy in how they ask for locks. Basically, there's no "crossing" between lock requests A and C.

(b) Assume the following three threads share objects Java, Fortran, Perl:

Thread T1	Thread T2	Thread T3

synchronized(Java) {
  synchronized(Fortran) {
  System.out.println("X");
  }
}
	synchronized(Fortran) {
	  synchronized(Perl) {
	  System.out.println("Y");
	  }
	}
		synchronized(Perl) {
		  synchronized(Java) {
		  System.out.println("Z");
		  }
		}

(i) Provide a sequence of events showing how this system could deadlock.

T1 locks Java
T2 locks Fortran
T3 locks Perl
T1 can't get Fortran
T2 can't get Perl
T3 can't get Java
so nobody advances

(ii) What principle of lock acquisition can be applied to prevent this? Show how that principle can be utilized to fix the above three threads.

We could establish a hierarchy of locks and then rewrite the order of the synchronized nesting to adhere to this hierarchy. For example, Java > Fortran > Perl and then

Thread T1	Thread T2	Thread T3

synchronized(Java) {
  synchronized(Fortran) {
  System.out.println("X");
  }
}
	synchronized(Fortran) {
	  synchronized(Perl) {
	  System.out.println("Y");
	  }
	}
		synchronized(Java) {
		  synchronized(Perl) {
		  System.out.println("Z");
		  }
		}

5. (20 points)
(a) Explain how the **`volatile`** keyword impacts how the Java Memory Model treats a class variable and explain whether a variable in a class is marked as **`volatile`**, does that remove the need to use locks if multiple threads might be setting and getting that variable?

With a volatile variable the reads are "true" reads (no cache use) and the writes are "true" writes (again no cache use).

We'd still need locks to deal with non-atomic calls using the variable, like the ++ operator.

(b) In the box to the right, give the program-order ***event sequence*** for the following code segment running in a thread named **TZ**.
Recall that events are things such as **read(T2, x, _)** or **write(T1, x, _)**.

```
int x;
int y = 10;
int z = 100;

   x = 1;
   y++;
   z += x;
```

write(TZ, y, 10)
write(TZ, z, 100)

write(TZ, x, 1)

read(TZ, y, 10)
write(TZ, y, 11)

read(TZ, x, 1)
read(TZ, z, 100)
write(TZ, z, 101)

(c) Consider the following **event sequence trace** for part of a two-threaded program that share variables x and y where ***neither*** is marked as **`volatile`**.

lock(T1, sharedLock)
write(T1, y, _)
write(T2, x, _)
unlock(T1, sharedLock)
lock(T2, sharedLock)
read(T1, x, _)
read(T2, y, _)
unlock(T2, sharedLock)

Are there any volatility-based data races in this trace? Whether you say "yes" or "no" you need to explain your answer using the "happens-before" logic we've been using in class. If there is more than one, identify them all.

Volatility race between write(T2, x, _) and read(T1, x, _) since they are in different threads and only T1 writes are forced to happen before by the unlock in T1 and lock in T2.

5. (15 points)
Image we are writing a program for a 1-Dimensional world. We want a thread-safe class called **LeftArrow** that contains two floating point values along the length of the Universe where values to the left are smaller than those to the right; one that records the coordinate of the head of the arrow and one that records the coordinate of the tail of the arrow. The class invariant is that the spot marking the head is required to be to the left of the spot marking the tail by at least 1 unit of measurement.

The class needs to have a constructor that takes in two floats and if the values passed in meet the above criteria, sets the spots of the head and tail. If the values do not meet the above criteria it needs to use the given spot for the head and then set the tail such that it is at least 1 unit of measurement to the left of the head.

There need to be two methods (**setTail** and **setHead**) that each get passed a floating point value. They need to work such that they only alter a coordinate if doing so does not break the invariant. They will return "true" if it did alter the coordinate and "false" is it did not.

There needs to be a method **toString** that returns a String of the form:
Left Arrow: From _ to _ .
where the _ spaces are filled in (respectively) with the coordinate of the head and the coordinate of the tail.

Implement the class by filling in the skeleton below:

```
public class LeftArrow {
//DATA GOES HERE


private float head;
private float tail;



//CONSTRUCTOR
   public LeftArrow(float headIn, float tailIn) {

      head = headIn;
      if (tailIn<head+1) {tail = head+1;}
      else {tail = tailIn;}


   }
```

```
//SETTERS AND GETTERS
   public boolean setTail(float newCoord) {
        ^----make the method synchronized so the test and set are atomic
                relative to each other


      if (newCoord>=head+1) {tail=newCoord; return true;}
      else {return false;}




   }


   public boolean setHead(float newCoord) {
        ^----make the method synchronized so the test and set are atomic
                relative to each other


      if (newCoord<=tail-1) {head=newCoord; return true;}
      else {return false;}







   }

   public String toString() {
        ^----make the method synchronized so the tail is the value that
                 it was when we printed the head

     System.out.println(
          "Left Arrow: From " + head
          " to " + tail
          ".");
```