

Final Exam

CMSC 433
Programming Language Technologies and Paradigms
Fall 2002

December 21, 2002

Guidelines

The exam time is from **10:30 am to 12:30 pm**, totalling 2 hours.

This exam has 11 pages; count them to make sure have all of them (the last page is blank). Put your name and class account number on each page before starting the exam. Write your answers directly on the exam sheets, using the back of the page as necessary. If you finish early, you may bring your exam to the front, but please be as quiet as possible.

If you have a question, raise your hand and I will come to you. However, to minimize interruptions, you may only do this once for the entire exam. Therefore, wait until you are sure you don't have any more questions before raising your hand. Errors on the exam will be posted on the board as they are discovered. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

Good luck, and have a great break!

Question	Points	Score
1	20	
2	15	
3	15	
4	15	
5	20	
6	15	
Total	100	

1. (Short Answer, 20 points)

(a) (5 points) Describe the difference between *white box* and *black box* testing, for a given class. Which one is JUnit more suited for, and why?

(b) (5 points) Describe as concisely as possible how a program consisting of two threads can be in a state of deadlock. In what way can you guarantee that for any multithreaded program that uses locks it will never be deadlocked?

(c) (4 points) Explain why `wait` should nearly always be called from within a loop. What undesirable situation can arise when this is not the case?

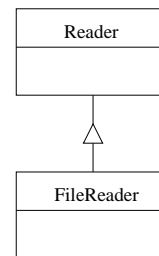
(d) (6 points) Name two features that Java RMI provides automatically when dealing with Remote objects. More to the point, what functionality did you have to implement yourself in project 1 for `RemoteLogClient` and `RemoteLogServer` that you did not have to implement when writing your `RMIRemoteLogClient` and `RMIRemoteLogServer`?

2. (Short Answer, 15 points)

(a) (4 points) Name two ways that JVM can acquire a stub to a Remote object using Java RMI. As a hint (for one of these two), think about how the `RMIRemoteLogClient` acquired a stub to a Remote `LocalLog` living on the `RMIRemoteLogServer`.

(b) (3 points) What is the effect of the Java `volatile` qualifier?

(c) (3 points) What is the relationship between the classes `Reader` and `FileReader` that is shown in the following class interaction diagram?



(d) (5 points) One of the principles of design patterns is to *encapsulate what varies*. That is, when looking at the functional and/or structural decomposition of a system, often certain parts are essentially fixed while other parts vary. Pick a design pattern and explain how it implements this principle, perhaps using a small example.

3. (Concurrency Patterns, 15 points) A simple implementation of a *copy-on-write set* is given below.

```
public class CopyOnWriteSet {
    private Object[] array_;
    public CopyOnWriteSet() { array_ = new Object[0]; }

    // look through the array for the given object
    public synchronized boolean member(Object o) {
        for (int i = 0; i < array_.length; i++) {
            if (array_[i] == o || (o != null && o.equals(array_[i])))
                return true;
        }
        return false;
    }

    // make a copy of the array, add the object, and update
    // the array reference to point to the new array
    public synchronized void add(Object o) {
        int len = array_.length;
        Object[] newArray = new Object[len+1];
        System.arraycopy(array_, 0, newArray, 0, len);
        newArray[len] = o;
        array_ = newArray;
    }
}
```

This implementation uses a *conservative update* for the `add` method; that is, it holds a lock while performing the entire `add` operation. Change this implementation to perform `add` as an *optimistic update* instead, using two additional methods `array()` and `commit()`. A partial implementation is given on the next page with space for you to insert your code; some additional background information is below.

Additional background information (i.e. hints!):

A *copy-on-write set* is one in which a write to the set results in its contents being copied first. This allows existing threads that are in the process of using the set, e.g., iterating over it, to continue to do so safely.

Because we assume for copy-on-write sets that `add` calls are rare and membership tests are frequent, we can use *optimistic updates* to increase program availability. As explained in class, an optimistic update works by (1) grabbing the existing state, (2) operating on that state in some way, and (3) committing the end result; the lock must only be held for the first and third operations, increasing availability. The commit operation ensures that no other thread has performed a commit since the current thread acquired the state. If this turns out to be the case, then the current thread must reacquire the state, redo its change, and try to commit again. This ensures that no changes are lost. The drawback of this approach is that work could be wasted if threads constantly must redo their changes due to failed commits.

```
public class CopyOnWriteSet {
    // array_, CopyOnWriteSet() and member(Object o) same as above

    private synchronized Object[] array() { return array_; }
    private synchronized boolean commit(Object[] oldArray,
                                       Object[] newArray) {
        boolean success = (oldArray == array_);
        if (success) array_ = newArray;
        return success;
    }

    public void add(Object o) { // WRITE CODE BELOW

    }
}
```

4. (Concurrency, 15 points) The code snippets below *may* allow for race conditions or deadlock. If so, describe the problem in general, and then describe an execution trace that illustrates the problem. In the first two cases, assume that multiple threads may be manipulating the object at once. In the third case, assume that `startComputations` is called with $n \geq 1$.

(a) (5 points)

```
// a user can post a message that are later retrieved
1: public class MessageBoard {
2:     String msg = null;
3:
4:     void post(String aMsg) {
5:         synchronized (this) {
6:             while (msg != null) wait();
7:         }
8:         synchronized (this) {
9:             msg = aMsg; notifyAll();
10:        }
11:    }
12:
13:    String retrieve() {
14:        synchronized (this) {
15:            while (msg == null) wait();
16:        }
17:        synchronized (this) {
18:            String aMsg = msg; msg = null;
19:            notifyAll(); return aMsg;
20:        }
21:    }
22: }
```

(b) (5 points)

```
1: public class MyInteger {
2:     private int i;
3:
4:     public MyInteger(int i) { this.i = i; }
5:
6:     public MyInteger add(MyInteger x) {
7:         return new MyInteger(x.i + this.i);
8:     }
9: }
```

(c) (5 points)

```
// assume doComputation manipulates its object without
// synchronization
1: public class CompThread extends Thread {
2:     private Object o;
3:     public CompThread(Object toComputeWith) { o = toComputeWith; }
4:
5:     public void run() {
6:         synchronized (this) {
7:             doComputation(o);
8:         }
9:     }
10:
11:     public static void startComputations(int n, Object toComputeWith) {
12:         for (int j = 0; j<n; j++) {
13:             new CompThread(toComputeWith).start();
14:         }
15:     }
16: }
```

5. (Design Patterns, 20 points) A *thread-safe* class is one in which objects of that class will work properly when handled by multiple threads. This typically means that all accesses to the object must be synchronized. However, as we saw in Dr. Pugh's Java One slides, making a class synchronized by default can impose a significant performance penalty when the class doesn't actually need the synchronization, i.e. it's used in a single-threaded program.

Write a *proxy* for objects conforming to the MyMap interface (shown below) called SynchronizedMyMap that adds synchronization to an unsynchronized MyMap object (a similar function is performed by the Collections.synchronizedMap() method). Make sure that the proxy will work even if multiple SynchronizedMyMap objects are created that wrap the same underlying MyMap object.

```
public interface MyMap {
    public void put(Object key, Object value);
    public Object get(Object key);
    public Object remove(Object key);
    public int size();
}
```

6. (Design Patterns, 15 points) It is sometimes useful when reading in a text file to prepend each line of the file with the current line number. We could do this using inheritance by overriding the `read` method of some implementation of the abstract class `Reader` (shown below) to output the line number when a newline is seen in the original text. This idea is shown in the code on the next page by extending the `FileReader` class (whose signature is shown below).

The disadvantage of using inheritance in this way is that it will only apply to a particular subclass, in this case `FileReader`. Here we would have to write a separate subclass of `BufferedReader` to implement line prepending, rather than reuse the code that subclassed the `FileReader` class.

To fix this problem, we can use the *Decorator* pattern. A decorator *wraps*, via composition, a related class's methods to perform additional functionality, similar to a proxy. This way, you need to write the decorator only once, and it will work for *all* classes that it can wrap.

Change the code on the next page to be a decorator instead, implementing line number prepending by decorating instances of the `Reader` class. Most of the code on the following page will not change. You can mark up the code provided there, and write in additional methods yourself, as necessary. Your answer must be precise enough that someone could code up the entire correct answer based on what you say.

```
public abstract class Reader {
    public abstract void close() throws IOException;
    public abstract int read() throws IOException;
}

public class FileReader extends Reader {
    public FileReader(String filename) throws FileNotFoundException;
    public void close() throws IOException;
    public int read() throws IOException;
}
```

```

1: public class FileLineReader extends FileReader {
2:     private int n = 1;           // current line number in file
3:     private String number = "1:\t"; // line number to print out
4:     private int numberIdx = 0;    // current index in number variable
5:
6:     public FileLineReader(String filename) throws FileNotFoundException {
7:         super(filename);
8:     }
9:
10:    public int read() throws IOException {
11:        // if we are outputting the line number, continue doing so
12:        if (numberIdx >= 0) {
13:            char c = number.charAt(numberIdx);
14:            if (numberIdx + 1 == number.length()) {
15:                n++;
16:                number = null;
17:                numberIdx = -1;
18:            }
19:            else numberIdx++;
20:            return (int)c;
21:        }
22:        // read the next character
23:        int c = super.read();
24:
25:        // check for an end-of-line;
26:        // if found, start outputting the line number
27:        if (c == '\n') {
28:            number = (new Integer(n).toString()) + ":\t";
29:            numberIdx = 0;
30:        }
31:        return c;
32:    }
33: }

```

This page intentionally left blank.