

Final Exam

CMSC 433
Programming Language Technologies and Paradigms
Fall 2002

December 16, 2002

Guidelines

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. I will not accept exams until I ask for them. If you finish early use the time to recheck your answers. Please be as quiet as possible.

I will not take any questions during the exam. 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.

Question	Points	Score
1	15	
2	15	
3	15	
4	10	
5	15	
6	10	
7	20	
Total	100	

1. (JUnit) 15 points. Some methods for `java.util.TreeMap` are described below. Write a class called `TreeMapTest` that contains three JUnit “reasonable” tests of this `java.util.TreeMap`. Your tests should not use any `java.util.TreeMap` methods that are not described below. Include comments in each test that describe what the purpose of each test is.

	<code>TreeMap()</code>	Constructs a new, empty map, sorted according to the keys’ natural order.
<code>void</code>	<code>clear()</code>	Removes all mappings from this <code>TreeMap</code> .
<code>Object</code>	<code>clone()</code>	Returns a shallow copy of this <code>TreeMap</code> instance.
<code>boolean</code>	<code>containsKey(Object key)</code>	Returns true if this map contains a mapping for the specified key.
<code>boolean</code>	<code>containsValue(Object value)</code>	Returns true if this map maps one or more keys to the specified value.
<code>Set</code>	<code>entrySet()</code>	Returns a set view of the mappings contained in this map.
<code>Object</code>	<code>firstKey()</code>	Returns the first (lowest) key currently in this sorted map.
<code>Object</code>	<code>get(Object key)</code>	Returns the value to which this map maps the specified key.
<code>Set</code>	<code>keySet()</code>	Returns a <code>Set</code> view of the keys contained in this map.
<code>Object</code>	<code>lastKey()</code>	Returns the last (highest) key currently in this sorted map.
<code>Object</code>	<code>put(Object key, Object value)</code>	Associates the specified value with the specified key in this map.
<code>void</code>	<code>putAll(Map map)</code>	Copies all of the mappings from the specified map to this map.
<code>Object</code>	<code>remove(Object key)</code>	Removes the mapping for this key from this <code>TreeMap</code> if present.
<code>int</code>	<code>size()</code>	Returns the number of key-value mappings in this map.
<code>Collection</code>	<code>values()</code>	Returns a collection view of the values contained in this map.

```
import junit.framework.*;

public class TreeMapTest extends TestCase
{
    public TreeMapTest( String name ) {
        super( name );
    }

    // YOUR TESTS GO HERE ...

    public static Test suite() {
        return new TestSuite( TreeMapTest.class );
    }

    public static void main(String args[]) {
        junit.textui.TestRunner.run(suite());
    }
}
```

2. (RMI) 15 points

The LockerI interface extends java.rmi.Remote. LockerI encapsulates a single lock. The LockerI interface is shown below.

```
import java.rmi.*;

public interface LockerI extends Remote {
    public void getLock() throws RemoteException ;
    public void releaseLock() throws RemoteException ;
}
```

The LockerI interface is implemented by the Locker class. This class has three methods: getLock(), releaseLock(), and main(). getLock() acquires the lock and then returns. releaseLock() releases the lock, and main does the setup necessary to make an instance of Locker accessible via java RMI. You are to write the code needed in main.

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class Locker extends UnicastRemoteObject implements LockerI {

    boolean locked = false;

    public synchronized void getLock() throws RemoteException {
        while (locked){
            try { wait();}
            catch (InterruptedException e ) {}
        }
        locked = true;
    }
    public synchronized void releaseLock()throws RemoteException {
        locked = false;
        notifyAll();
    }

    public Locker() throws RemoteException {}

    public static void main(String[] args) throws Exception {
        // WRITE THIS CODE

    }
}
```

Answer:

The inserted code is

```
System.setSecurityManager(new RMISecurityManager());
LocateRegistry.createRegistry(2005);
LockerI pt = new Locker();
Naming.rebind("rmi://milano.cs.umd.edu:2005/Locker", pt);
```

The UseLock class is an abstract class that uses a LockerI instance to make doOp() method atomic. Write the code necessary to acquire and use the LockerI instance via RMI.

```
import java.rmi.*;
import java.rmi.registry.*;

public abstract class UseLock {
    private static void doOp() { // to be overridden by subclass}

    public static void main(String[] args) throws Exception {

        // WRITE THIS CODE

        for(;;) {
            t.getLock();
            doOp();
            t.releaseLock();
        }
    }
}
```

Answer:

The inserted code is

```
System.setSecurityManager(new RMISecurityManager());
LockerI t = (LockerI)Naming.lookup("rmi://milano.cs.umd.edu:2005/Locker");
```

3. (RMI - short answer 15 points).

- (a) Explain what information the `java.rmi.server.codebase` property conveys to a JVM and how that information is used when running RMI applications.

Answer:

java.rmi.server.codebase indicates the codebase URL of classes originating from the VM. The codebase property is used to annotate class descriptors of classes originating from a VM so that the class for an object sent as a parameter or return value in a remote method call can be loaded at the receiver.

- (b) Explain the design issues that should be considered when deciding whether to make an object implement `Serializable` or `Remote` in a java RMI application.

Answer:

Serializable implements call by value while Remote implements call by reference semantics. That is, when an object is serializable it is copied before being sent, while for a Remote object, a stub is sent and all interactions are forwarded back to the object owner. The two results are: (1) changes to serializable objects are not reflected to all copies, but all execution happens on the local object; (2) changes to Remote objects are reflected to all copies, but all execution happens on the host owning the original object, requiring messages to be sent.

4. (Concurrency Patterns) 10 points. The class XY represents a point in the plane. Classes LocationV2 and LocationV3 contain an XY instance and have a method called moveBy(). The purpose of moveBy() is to safely change the coordinates of the XY instance.

From the user's point of view LocationV2 and LocationV3 are functionally equivalent. Using less than 50 words, discuss the conditions under which a user would prefer to use each class over the other.

```
class XY {
    private final double x_, y_;
    XY(double x, double y) { x_ = x; y_ = y; }
    double x() { return x_; }
    double y() { return y_; }
}

class LocationV2 {
    private XY xy_;
    LocationV2(double x, double y) {
        xy_ = new XY(x, y);
    }
    synchronized XY xy() { return xy_; }
    synchronized void moveBy(double dx, double dy) {
        xy_ = new XY(xy_.x() + dx, xy_.y() + dy);
    }
}

class LocationV3 {
    private XY xy_;
    private synchronized boolean commit(XY oldp,
    XY newp){
        boolean success = (xy_ == oldp);
        if (success) xy_ = newp;
        return success;
    }
    LocationV3(double x, double y){xy_=new XY(x,y);}
    synchronized XY xy() { return xy_; }
    void moveBy(double dx, double dy) {
        while (!Thread.interrupted()){
            XY oldp = xy();
            XY newp = new XY(oldp.x()+dx, oldp.y()+dy);
            if (commit(oldp, newp)) break;
            Thread.yield();
        }
    }
}
```

Answer:

The LocationV3 class makes use of optimistic updates. That is, rather than hold the lock for the entire moveby operation, it only grabs the lock to capture the state, and then again to commit changes it made to that state. This commit operation could fail if an intervening thread modified the state. Optimistic updates are useful for ensuring greater availability, since locks are not held as long. On the other hand, they can result in wasted work (when the commit fails), so it should be unlikely that other operations will intervene and update the state in the meantime.

5. (Design Patterns) 15 points. You are faced with the following design problem. You have already written an abstract class called Resource. Resource has two methods initialize() and cleanup() that encapsulate application-specific setUp and tearDown procedures for Resources.

Resource subclasses inherit these methods, overriding them as desired. These subclasses also implement the singleton pattern, so at runtime there is only one instance of each subclass. Users of these subclasses are expected to call initialize() and cleanup before and after using a Resource subclass instance.

Currently, your Resource subclass instances are NOT thread safe. In particular, it's no longer clear how and when to call initialize() and cleanup(). For example, if two threads start up, create an instance of ResourceSubclass1, and call initialize(), then the ResourceSubclass1 instance will be initialized twice, when it should only have been initialized once. The same goes for cleanup().

Sketch a solution to this problem (using diagrams and psuedocode as necessary) that allows you Resource subclasses to be used by multiple threads. What design pattern(s) can you use to minimize changes to existing classes?

Answer:

Use a decorator, proxy, or adapter to wrap each Resource subclass with code that adds reference counts. That is, users call initialize() and cleanup() as normal. The proxy checks to see if the caller is the first to call initialize(); if so it calls the wrapped class's initialize() method, and increments the count. Subsequent calls to initialize just result in the count being increased. Conversely, when a user calls cleanup(), the count is decreased; when the count reaches 0, the underlying Resource class's cleanup() method is called. To work in a concurrent setting, you may need to synchronize initialize() and cleanup() in the proxy/adaptor.

6. (Design Patterns) 10 points

PartFactory's instantiate part objects, with the following interface:

```
interface PartFactory {
    public Car createCar();
    public Wheel createWheel(String size);
    public Engine createEngine(int horsePower);
}
```

- (a) (2 points) Using the Abstract Factory design pattern, show the declarations of Java classes that are Factories for two different car models: ChevyPartFactory and FordPartFactory.

Answer:

```
class ChevyPartFactory implements PartFactory {
    public Car createCar();
    public Wheel createWheel(String size);
    public Engine createEngine(int horsePower);
}
class FordPartFactory implements PartFactory {
    public Car createCar();
    public Wheel createWheel(String size);
    public Engine createEngine(int horsePower);
}
```

Alternatively, you might have specialized Car to be ChevyCar for the ChevyPartFactory and FordCar for FordPartFactory, and made similar changes to the other base types.

- (b) (2 points) Provide code for a client that uses the ChevyPartFactory to create a Car, a Wheel of size "R15", and an Engine of horsePower 6000.

Answer:

```
/* note the use of base types in all declarations, making the code
   completely generic, only depending on which CarFactory is
   instantiated */
ChevyPartFactory f = new ChevyPartFactory();
Car c = f.createCar();
Wheel w = f.createWheel("R15");
Engine e = f.createEngine(6000);
```

- (c) (6 points) What restrictions/requirements are there on the types of the objects returned from the methods in the ChevyPartFactory and FordPartFactory classes?

Answer:

The return value objects have to be subclasses of the return values of the methods in interface CarFactory (e.g., if CarFactory.createCar() returns an ChevyCar object, then ChevyCar must be a subclass of Car).

7. (Java Concurrency - 20 points). The following code is intended to implement a thread-safe one-place buffer. It does not work. Explain why it does not work and give an execution trace that demonstrates the problem.

```
class Example {

    Object full = new Object();
    Object empty = new Object();
    Object obj = null;

    void produce(Object o) {
        while (obj != null) {
            synchronized (empty) {
                try {empty.wait();}
                catch (InterruptedException e){ }
            }
        }
        synchronized (this) {obj = o;}
        synchronized (full) {full.notify();}
    }

    Object consume() {
        Object o;
        while (obj == null) {
            synchronized (full) {
                try {full.wait();}
                catch (InterruptedException e){ }
            }
        }
        synchronized (this) {o = obj; obj = null; }
        synchronized (empty) {empty.notify();}
        return o;
    }
}
```

Answer:

There are two main problems with the code:

- (a) Both produce() and consume() examine the contents of the variable obj in the wait-loop without synchronizing first. Because of the Java memory model, reads of unsynchronized, non-volatile variables are not guaranteed to be the most up-to-date. A trace demonstrating this fact would be: thread t1 calls consume(), and ends up calling wait because the obj field is null. Then, some thread t2 calls produce(), thereby setting the obj field to some object. The call to notify awakens t1. Due to the lack of synchronization, when t1 re-checks its condition obj == null, the condition evaluates to true, because t1 is not looking at the most up-to-date version of obj, due to the lack of synchronization.*
- (b) Multiple threads can execute in the latter half of consume() or produce(), since there is no synchronization around the while loop and the latter half. For example, given two producer threads, the first (call it t1) can check the condition, find it to be false, and so exit the while loop, and then be context-switched before entering the next synchronized block. The second thread t2 can then do exactly the same thing but proceed through the synchronized blocks, setting obj to be its object. When thread t1 is rescheduled, it will then set obj to its object, overwriting the one that was there before.*