

Name:

Final

CMSC 433

Programming Language Technologies and Paradigms
Spring 2004

May 19, 2004

Instructions

This exam contains 15 pages, including this one. Make sure you have all the pages. Write your name on the top of this page before starting the exam.

Write your answers on the exam sheets. If you finish at least 15 minutes early, bring your exam to the front when you are finished; otherwise, wait until the end of the exam to turn it in. Please be as quiet as possible.

If you have a question, raise your hand. 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.

You may avail yourself of the *punt* rule. If you write down *punt* for any numbered or lettered part of a question, you will earn 1/5 of the points for that question (rounded down).

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

Question 1. Short Answer (20 points).

- a. (4 points)** Explain the difference between method overriding and method overloading.

Answer: Overriding is when a subclass replaces code inherited from a superclass. Overloading is when a class has multiple methods with the same name and return type but different numbers and/or types of parameters.

- b. (4 points)** What does the following program print? Does it always print the same thing? (Hint: Yes, this is a trick question.) Explain your answer.

```
public class Foo extends Thread {
    public void run() {
        System.out.print("run ");
    }
    public static void main(String[] args) {
        Foo f = new Foo();
        f.run();
        System.out.print("main ");
    }
}
```

Answer: This program always prints “run main.” This is a single-threaded program; the call to `f.run()` is just a normal method invocation. Separate threads are only started when their `start()` method is invoked, which we don’t do here.

- c. (4 points)** Explain one advantage and one disadvantage of delegation as compared to inheritance.

Answer: Delegation is more flexible than inheritance, because the inheritance hierarchy is fixed at compile time, whereas delegation relationships can change dynamically at run time. Delegation also allows for some reuse that’s not easy with inheritance, such as the kind of recursion in the decorator pattern. On the other hand, the dynamic reconfigurability of delegation can make programs harder to understand than inheritance, and it also increases the number of objects in the system, adding further complication. Delegation can also affect performance, since it increases the number of method calls.

d. (4 points) Draw a UML diagram for the adapter pattern and explain how the pattern works. If you forget the particular UML notation for something, just add a note to the side explaining what your lines and boxes mean. We've given you part of the diagram to start.



Answer: See the lecture notes.

e. (4 points) Describe the sequence of events that happens after a thread calls `o.notifyAll()` when several other threads are waiting on `o`.

Answer: When `o.notifyAll()` is called, all threads waiting on `o` are woken up. All of the newly woken threads contend for the lock on `o`. Since the thread that just called `o.notifyAll()` has that lock, they can't make progress until that thread releases the lock. As the lock becomes available, each thread waiting on `o` will, in some unspecified order, unblock and continue executing.

Question 2. Multi-threading (10 points). For each of the following multi-threaded programs, state whether the program has a problem related to synchronization. If there is a problem, explain what the problem is, and give a sequence of actions that exhibits the problem.

a. (5 points) The following code implements the observer pattern. Assume that the `attach()`, `detach()`, and `notify()` methods may be called arbitrarily by other threads.

```
public interface Observer {
    void update();
}

public class SynchronizedList {
    private LinkedList l = new LinkedList();
    public synchronized void add(Object o) { l.add(o); }
    public synchronized void remove(Object o) { l.remove(o); }
    public synchronized int size() { return l.size(); }
    public synchronized Object get(int index) { return l.get(index); }
}

public class Subject {
    private SynchronizedList observers = new SynchronizedList();

    public void attach(Observer s) {
        observers.add(s);
    }

    public void detach(Observer s) {
        observers.remove(s);
    }

    public void notify() {
        for (int i = 0; i < observers.size(); i++) {
            Observer s = (Observer) o.get(i);
            s.update();
        }
    }
}
```

Answer: The notify method is not sufficiently synchronized (attach and detach are OK). In particular, the following sequence of events could happen. Suppose there is one observer that has been attached. (1) One thread calls `notify()`, (2) the check `i < observers.size()` succeeds, (3) Another thread calls `detach()`, which removes the observer from the list, (4) The first thread calls `o.get(0)`, which fails. The basic problem is a failure of atomicity.

b. (5 points) In the following example, assume that `foo()` and `bar()` may both be called arbitrarily by other threads.

```
public class Outer {
    private class Inner {
        private int counter = 0;
        public synchronized void incCounter() { counter++; }
    }
    private Inner myCount = new Inner();

    public synchronized void foo() {
        System.out.println("Foo!");
        myCount.incCounter();
    }

    public synchronized void bar() {
        System.out.println("Bar!");
        myCount.incCounter();
    }
}
```

Answer: This program does not have any synchronization problems. The extra synchronization on the inner class object is redundant and wasteful, but it doesn't hurt anything: The inner class object lock is only acquired when the outer class lock is already held, so there's no danger of deadlock.

Question 3. Refactoring (10 points). In class we showed an example of some code that could use some refactoring. We ended with two of the classes, `Rental` and `Movie`, looking like the following:

```
public class Rental {
    private Movie _movie;
    private int _daysRented;

    public Rental(Movie movie, int daysRented) {
        _movie = movie;
        _daysRented = daysRented;
    }
    public int getDaysRented() { return _daysRented; }
    public Movie getMovie() { return _movie; }

    public double amountFor() {
        double thisAmount = 0;
        //determine amounts for each line
        switch (getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (getDaysRented() > 2)
                    thisAmount += (getDaysRented() - 2) * 15;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (getDaysRented() > 3)
                    thisAmount += (getDaysRented() - 3) * 1.5;
                break;
        }
        return thisAmount;
    }
}

public class Movie {
    public static final int CHILDRENS = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;

    private String _title;
    private int _priceCode;

    public Movie(String title, int priceCode) {
        _title = title;
        _priceCode = priceCode;
    }

    public int getPriceCode() { return _priceCode; }
    public void setPriceCode(int arg) { _priceCode = arg; }
    public String getTitle() { return _title; }
}
```

Describe at least two different refactorings we could perform on the sample code, and explain why those refactorings are useful. Your refactorings should be different not just in what source code they operate on but what they actually do (e.g., don't list three uses of Rename Variable). You may not use renaming of variables, methods, or fields by themselves as refactorings. You may use any reasonable refactoring, not just those we discussed in class. Don't worry about getting the names exactly right. Do worry about describing the refactorings clearly and precisely.

Answer:

1. Add parameter `daysRented` to replace call to `getDaysRented()` in `amountFor()`, in preparation for step 2.
2. Move method `amountFor()` from `Rental` into `Movie`, since this will differ from one kind of movie to another.
3. Replace conditional switch by polymorphism by replacing the single class `Movie` by three subclasses `ChildrensMovie`, `RegularMovie`, and `NewRelease`. Split the behavior of the `amountFor()` method accordingly. This refactoring has many advantages. For example, it means we have an enumerate of movies rather than using an integer type; it's easy to add new movies using subclassing.

There are many other possible refactorings for this program, and any refactoring, no matter how useless, was acceptable. Most people got full credit for this problem.

This page intentionally left blank.

Question 4. Locking (20 points). Using ordinary Java synchronization (and possibly wait and notify), implement a ReaderWriterLock class on the next pages. Your class should support *reader-writer* locks: Any number of readers can hold the “lock” at the same time, but a writer has exclusive access. When a writer holds the lock, no readers or other writers are allowed to hold it. Your class should have the following methods:

```
public class ReaderWriteLock {
    public void acquireReader();
    public void releaseReader();
    public void acquireWriter();
    public void releaseWriter();
}
```

- Clients call {acquire/release}{Reader/Writer} to acquire or release the reader or writer “lock.”
- The acquire methods block until it is possible to acquire the lock.
- If a thread tries to release a lock it does not hold, you should throw IllegalStateException.
- Any number of readers can hold the reader lock at the same time.
- Only one writer can hold the writer lock. While it holds the lock, no readers and no other writers may hold locks. (And it may not acquire the lock until all readers have given up their locks.)
- A thread holding the writer lock will deadlock if it tries to acquire the reader lock. (In other words, you don’t need to make a special case for when the holder of the writer lock tries to acquire a reader lock.)
- If a writer is waiting to acquire the lock, no new readers may acquire the lock, and no reader may reacquire the lock. (Note: be sure to handle the case when multiple writers are waiting for the lock. No readers should be able to acquire the lock until all the writers have had their chance.)
- Use Thread.currentThread() to get the current thread object, so you can keep track of who has the lock. Use == to compare thread objects.
- **Do not use busy waiting!**

In addition, just like in Java, ReaderWriter locks should be *re-entrant*:

- A reader can reacquire the same reader lock as many times as it likes. You’ll need to keep track of reference counts to handle this. The lock is released by the reader when it has called releaseReader() the same number of times it has called acquireReader().
- A writer can reacquire the same writer lock as many times as it likes. Again, you’ll need to keep a reference count. The lock is released by the writer when it has called releaseWriter() the same number of times it has called acquireWriter().

Here are some methods you may find useful:

Class	Method	Description
HashMap	boolean containsKey(Object key)	Returns true if this map contains a mapping for the key
	Object get(Object key)	Returns the value to which the specified key is mapped in this hash map, or null if the map contains no mapping for this key.
	boolean isEmpty();	Returns true if this map contains no key-value mappings.
	Object put(Object key, Object value);	Associates the specified value with the specified key in this map. Returns the previous value associated with the specified key, or null if there was no mapping for key.
	Object remove(Object key);	Removes the mapping for this key from this map if present. Returns the previous value associated with specified key, or null if there was no mapping for key.
Integer	int intValue()	Returns the value of this Integer

```

public class ReaderWriterLock {

    private Map readers = new HashMap();
    private Thread writer = null;
    private int writerCount = 0;
    private int writersWaiting = 0;

    synchronized public void acquireReader() {
        Thread t = Thread.currentThread();
        while (writer != null || writersWaiting > 0)
            try { wait(); } catch (InterruptedException e) { }
        Integer count = readers.get(t);
        if (count == null)
            readers.put(new Integer(1));
        else
            readers.put(t, new Integer(count.intValue() + 1));
    }

    synchronized public void releaseReader() {
        Thread t = Thread.currentThread();
        if (!readers.containsKey(t))
            throw new IllegalStateException();
        int count = readers.get(Thread.currentThread()).intValue();
        if (count == 1) {
            readers.remove(t);
            notifyAll();
        } else
            readers.put(t, new Integer(count - 1));
    }

    synchronized public void acquireWriter() {
        if (writer == Thread.currentThread()) {
            writerCount++;
            return;
        }
        writersWaiting++;
        while (writer != null || !readers.isEmpty())
            try { wait(); } catch (InterruptedException e) { }
        writersWaiting--;
        writer = Thread.currentThread();
        writerCount = 1;
    }

    synchronized public void releaseWriter() {
        if (writer != Thread.currentThread())
            throw new IllegalStateException();
        writerCount--;
        if (writerCount == 0) {
            writer = null;
            notifyAll();
        }
    }
}

```

```
// More space to continue class ReaderWriterLock
```

Answer: This was the hardest problem on the exam, and I was generous in grading it. I did not take off for ignoring `InterruptedException` or for minor typos. I did take off for logical errors or bad programming mistakes. If your answer was different than mine, don't worry. There are many different solutions to this problem, and many people had very clever answers that looked different from my solution but were correct (or almost correct).

Question 5. Dynamic Proxies (20 points). A *security proxy* ensures that all clients that call methods of an object have permission to do so. Using dynamic proxies, on the next page implement the `createSecureObject()` method, which takes an `Object o` and a `Permission p`, and returns a new object `result`. The object `result` should implement all of the interfaces of `o`, **including** interfaces implemented by any superclass of `o`. (Note: This is slightly different than project 6.) Calls to `result.method(args)` should first call `checkPermission(p)` on the current security manager to see if access is allowed. If it's not, a `SecurityException` should be thrown. Otherwise `o.method(args)` should be invoked and its result returned. If `o.method(args)` throws an exception, that exception should be thrown (don't forget to unwrap it).

We've provided you with the necessary portion of the API below.

Class	Method	Description
Object	Class getClass()	Get the Class for this object
System	SecurityManager getSecurityManager()	Return the current Security Manager
SecurityManager	void checkPermission(Permission perm)	Throws a SecurityException if the requested access, specified by the given permission, is not permitted based on the security policy currently in effect. Otherwise has no effect.
Class	ClassLoader getClassLoader()	Return the class loader for this class
	Class[] getInterfaces()	Get the interfaces declared by this class
	Class getSuperclass()	Get the superclass of this class, or null if this class has no superclass
Method	Object invoke(Object obj, Object[] args) throws InvocationTargetException	Invokes the underlying method represented by this Method object, on the specified object with the specified parameters. (To simply things, we've eliminated some exceptions from the API.)
InvocationTargetException	Throwable getCause()	Returns the cause of this exception (the thrown target exception, which may be null).
Proxy	static Object newInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h);	Returns an instance of a proxy class for the specified interfaces that dispatches method invocations to the specified invocation handler.
InvocationHandler	Object invoke(Object proxy, Method method, Object[] args) throws Throwable;	Processes a method invocation on a proxy instance and returns the result. This method will be invoked on an invocation handler when a method is invoked on a proxy instance that it is associated with.
Arrays	static List asList(Object[] a)	Returns a fixed-size list backed by the specified array.
LinkedList	void addAll(Collection c)	Appends all of the elements in the specified collection to the end of this list.
	Object[] toArray()	Returns an array containing all of the elements in this list in the correct order.

```

public class SecureObject implements InvocationHandler {

    private Object o;
    private Permission p;

    public SecureObject(Object o, Permission p) {
        this.o = o;
        this.p = p;
    }

    public static Object createSecureObject(Object o, Permission p) {
        Class c = o.getClass();
        List int = new LinkedList();
        for (Class i = c; i != null; i = i.getSuperclass())
            int.addAll(Arrays.asList(i.getInterfaces()));
        InvocationHandler h = new SecureObject(o, p);
        return Proxy.newProxyInstance(c.getClassLoader(),
            (Class[]) int.toArray(), h);
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        try {
            System.getSecurityManager().checkPermission(p);
            return method.invoke(o, args);
        }
        catch (InvocationTargetException e) {
            throw e.getCause();
        }
    }
}

```

Question 6. RMI (10 points). This question involves the following program, which uses RMI. We first give the shared interfaces followed by the Client code, which runs on the client machine, and the Server code, which runs on the server machine. The ... in the call to `Naming.lookup()` represents the appropriate URL.

```
public interface Start extends Remote { public void start(Ping a) throws RemoteException; }
public interface Ping extends Serializable { public void ping(Pong b); }
public interface Pong extends Serializable { public void pong(); }
```

<pre>public class Client implements Ping { public void ping(Pong b) { System.out.println("Ping!"); b.pong(); } } public class ClientMain { public static void main(String[] args) { Start s = (Start) Naming.lookup("../Server"); s.start(new Client()); } }</pre>	<pre>public class Server implements Pong { public void pong() { System.out.println("Pong!"); } } public class ServerMain extends UnicastRemoteObject implements Start { public void start(Ping a) { a.ping(new Server()); } public static void main(String[] args) { ServerMain s = new ServerMain(); Registry r = LocateRegistry.createRegistry(1099); r.bind("Server", s); } }</pre>
---	---

What does this program print on the server machine? What does this program print on the client machine?

Answer: The program prints “Ping! Pong!” on the server machine. It prints nothing on the client machine.

If both Ping and Pong are changed to Remote interfaces (and the appropriate other changes are made), what does this program print on the client machine? What does this program print on the server machine?

Answer: The program prints “Pong!” on the server machine. It prints “Ping!” on the client machine.

If Ping remains Serializable and Pong is changed to a Remote interface (and the appropriate other changes are made), what does this program print on the client machine? What does this program print on the server machine?

Answer: The program prints “Ping! Pong!” on the server machine. It prints nothing on the client machine.

Question 7. More Short Answer (10 points).

a. (4 points) In AspectJ, what is a join point, and what is a pointcut? Give examples of both.

Answer: A join point is a point in the program where AspectJ might insert code in its weaving process. Some examples are method call and return, field getting and setting, exception handling, and initialization. A pointcut is a predicate that identifies join points. (Most people forgot what a pointcut was or mixed up join points and pointcuts; you only lost a point if you did that.)

b. (3 points) What is a persistent object? Why would you want to use persistent objects? (Hint: This was discussed in the enterprise Java lectures.)

Answer: A persistent object is one that reflects state in a database. Reads and writes to and from the object are translated to reads and writes in the database. Persistent objects are useful because they provide a convenient means for interacting with a database. In particular, they can be much easier to use than, say, dynamically generating SQL queries.

c. (3 points) In class we talked about how to handle state-dependent actions in multi-threaded programming. Two possibilities were balking and guarding. Explain both of these. Which is/are useful for single-threaded programming? Explain.

Answer: Balking is when you raise an exception when you attempt to perform an action in the wrong state. Balking is very often used in single-threaded programming. Guarding is when you wait (possibly indefinitely) until some other thread sets the state right for the action. Guarding is useless in single-threaded programming, since there are no other threads to change the state.