

Name:

Final

CMSC 433

Programming Language Technologies and Paradigms
Spring 2005

Final Exam

Instructions

This exam contains 13 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		12
3		13
4		20
5		25
6		10
Total		100

Question 1. Short Answer (20 points).

a. (4 points) What is the difference between a regular inner class and a static inner class?

b. (4 points) Explain the difference between `o.notify()` and `o.notifyAll()`, and explain why in class we said that `o.notifyAll()` should almost always be used instead of `o.notify()`.

c. (4 points) Below are four design features of Projects 5 and 6. For each feature, list which **one** of the following design patterns it corresponds to: Singleton, Proxy, Command, Bridge, Observer. (Each pattern may be used 0, 1, or more times.)

- Whiteboards attach and detach themselves to the RemoteBoard to listen for updates.
- The Swing GUI provides an interface to the underlying GUI of the host platform.
- TimedObject wraps an object `o` to intercept calls to `o`'s methods (so that calls to them time out after *ms* milliseconds).
- Programs use RMI stubs to invoke a method remotely on another machine.

d. (4 points) In class we said that JUnit was created partially to aid refactoring. Explain how automated testing is useful for refactoring. (In your explanation, you should also explain the difference between refactoring and arbitrary code changes.)

e. (4 points) Explain what pre- and post-conditions are. In Java, methods list checked exceptions that they may throw. Are these exceptions conceptually part of the precondition or the postcondition?

Question 2. Double Dispatch, Again (12 points). The following program uses `instanceof` to simulate dynamically dispatching on multiple arguments. On the next page, rewrite `A`, `B`, and the body of `Main` to use the double dispatch of the Visitor pattern to invoke the `doXX` methods of class `Do`, following the same logic as below. We've provided you the new version of the `Thing` interface and part of `Main` to get started.

```
public interface Thing {}
public class A implements Thing {}
public class B implements Thing {}

public class Do {
    public static void doAA(A left, A right) { ... }
    public static void doAB(A left, B right) { ... }
    public static void doBA(B left, A right) { ... }
    public static void doBB(B left, B right) { ... }
}

public class Main throws Exception {
    public static void main(String[] args) {
        Thing t0 = (Thing) Class.forName(args[0]).newInstance();
        Thing t1 = (Thing) Class.forName(args[1]).newInstance();

        if ((t0 instanceof A) && (t1 instanceof A))
            Do.doAA(t0, t1);
        else if ((t0 instanceof A) && (t1 instanceof B))
            Do.doAB(t0, t1);
        else if ((t0 instanceof B) && (t1 instanceof A))
            Do.doBA(t0, t1);
        else if ((t0 instanceof B) && (t1 instanceof B))
            Do.doBB(t0, t1);
    }
}
```

```
public interface Thing {
    // Your implementations of doIt() should call the appropriate Do.doXX method
    void doIt(A a);
    void doIt(B b);

    void accept(Thing t);
}

public class Main throws Exception {
    public static void main(String[] args) {
        Thing t0 = (Thing) Class.forName(args[0]).newInstance();
        Thing t1 = (Thing) Class.forName(args[1]).newInstance();
    }
}
```

Question 3. Multi-threading (13 points). Construct a Java program that may deadlock. Your program should be a complete, runnable Java program with a `main()` method that, when invoked, shows the deadlock. Also, write a short **explanation** of the sequence of events leading to the deadlock.

Question 4. Locking (20 points). In Java 1.4, synchronization must start and end according to a block structure. Sometimes this is inconvenient. Java 1.5 allows more flexible locking designs by providing implementations of the following interface (actually, it includes two more methods, which we've omitted to keep things simpler):

```
public interface Lock {
    void lock();
    boolean trylock();
    void unlock();
}
```

a. (5 points) Modify the following class to use the `ReentrantLock` implementation of the above interface (which you will write for part b, on the next page) in place of the regular synchronization given below. You don't need to rewrite the class; just mark your changes clearly.

```
public class SharedFileReader {
    private FileReader theFile;
    private int count = 0;

    public SharedFileReader(String fileName) throws FileNotFoundException {
        theFile = new FileReader(fileName);
    }

    public synchronized int getCount() {
        return count;
    }

    public synchronized int read() throws IOException {
        int c = theFile.read();
        count++;
        return c;
    }
}
```

b. (15 points) Below and/or on the next page, write code for the `ReentrantLock` class, which implements the `Lock` interface. You will probably want to use `wait()` and `notifyAll()`.

- Clients call `lock/unlock` to acquire or release the “lock.”
- A thread may acquire the same lock multiple times (locks are reentrant). The lock is released when `unlock()` calls balance `lock()` (and successful `trylock()`) calls.
- The `lock` method blocks until it is possible to acquire the lock.
- If a thread tries to release a lock it does not hold, you should throw `IllegalStateException`.
- A call to `trylock()` checks to see if the current thread can acquire the lock. If it can, it acquires the lock and returns true. If it cannot, rather than blocking, it immediately returns false.
- 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!**

```
public class ReentrantLock implements Lock {
```

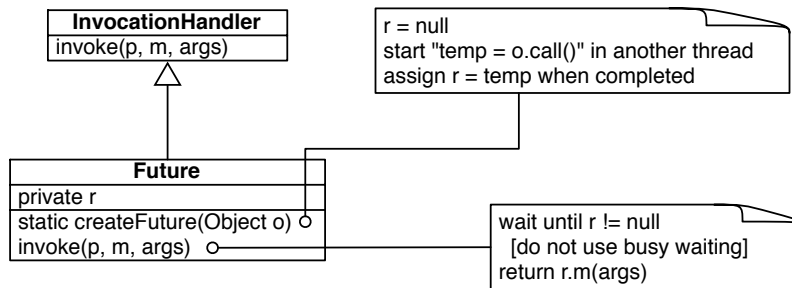
Question 5. Dynamic Proxies and Transparent Futures (25 points). In this problem, you will use dynamic proxies to turn objects into futures. A *future* is computation that is run in a separate thread. Sometime after the future is started, the result of the computation is *demanded*, for example, by trying to invoke a method of the object returned by the computation. This causes the demanding thread to block until the computation is complete. On the next page, implement the `createFuture()` method using dynamic proxies. Here is how your program should work:

- `createFuture()` takes an Object `o` as a parameter.
 - First, `createFuture()` uses reflection to find `o`'s public method named `call` taking no arguments and returning an object. Throw `IllegalArgumentException` if there is no such method. Let `C` be the return type of `call()`.
 - Next, `createFuture()` starts a thread to compute `o.call()`. When ready, the result of `o.call()` will be stored in some private field `r`.
 - Finally, `createFuture()` returns a dynamic proxy `result` implementing all interfaces of class `C`.

This object `result` is a proxy for `r`—all calls to `result.m(args)` should be delegated as calls to `r.m(args)`. The catch is that `r` is being computed in a separate thread, and it may not be ready right away. Thus a client that tries to invoke one of `result`'s methods will block until `r` is ready:

- When `result.m(args)` is called, the call should block until `r` has been computed (or has failed to be computed because of an exception). One of two things happens once `r` is available:
 1. If `r` has been computed, the result of `r.m(args)` is returned. If `r.m(args)` throws an exception, that exception should be thrown (don't forget to unwrap it).
 2. If the computation of `r` failed, i.e., `o.call()` threw an exception, then that same exception (unwrapped) is thrown. (No matter which method `m` has been invoked.)

Here is a diagram illustrating how this design will work (with exceptions left out):



The last page of the exam contains the portion of the dynamic proxy API you'll need to do this problem.

```
public class Future {
```

```
    public static Object createFuture(Object o) {
```

Question 6. More Short Answer (10 points).

a. (4 points) Describe two disadvantages of using reflection instead of regular Java operations. Explain your answers.

b. (3 points) Explain what invoking the method `AccessController.doPrivileged(PrivilegedAction action)` does. In particular, what happens with subsequent calls to `checkPermission()`?

c. (3 points) A *memory leak* is when a program fails to deallocate memory that it no longer needs. Is it possible for a Java program to have a memory leak? Explain.

Cheat Sheet – API for Dynamic Proxies

Class	Method	Description
Object	Class getClass()	Get the Class for this object
Class	ClassLoader getClassLoader()	Return the class loader for this class
	Class[] getInterfaces()	Get the interfaces declared by this class
	Method[] getMethods()	Returns an array containing public methods of this class or interface, including inherited methods.
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.)
	String getName()	Returns the name of the method represented by this object.
	Class[] getParameterTypes()	Returns an array of objects representing the formal parameter types, in order, of this method.
InvocationTargetException	Throwable getCause()	Returns the cause of this exception (the thrown target exception, which may be null).
Proxy	static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h);	Returns an instance of a proxy class for the specified interfaces that dispatches method invocations to the specified handler.
InvocationHandler	Object invoke(Object proxy, Method method, Object[] args) throws Throwable;	This method will be invoked on an invocation handler when a method is invoked on a proxy instance that it is associated with.