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).

<table>
<thead>
<tr>
<th>Question</th>
<th>Score</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>
Question 1. Short Answer (20 points).

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

   Answer: An instance of a regular inner class has an implicit reference to an instance of the enclosing, outer class. This reference is used whenever the inner class refers to an (instance) field or method of the outer class. A static inner class, on the other hand, has no such reference, and hence it may not implicitly use fields and methods of the outer class.

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

   Answer: Invoking \texttt{o.notifyAll()} wakes up every thread waiting on \texttt{o}. In contrast, invoking \texttt{o.notify()} wakes up exactly one thread. If that thread happens to be the wrong one (because the condition it is waiting for is not met), and if that thread goes back to sleep without itself notifying other waiting threads, the program may deadlock. Thus it is much trickier to use \texttt{notify()} correctly, and \texttt{notifyAll()} is usually the right one to pick.

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 \texttt{o} to intercept calls to \texttt{o}'s methods (so that calls to them time out after \texttt{ms} milliseconds).

   • Programs use RMI stubs to invoke a method remotely on another machine.

   Answer: In order, Observer, Bridge (or Proxy), Proxy, Proxy.
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.)

**Answer:** A refactoring is a change that preserves the behavior of the program. Typically the programmer will apply many (small) refactorings to transform his or her program, and they would like to ensure that none of the refactorings alters the program’s behavior. Hence automated testing makes it easy the programmer to rerun all tests between small refactorings and make sure that nothing has changed.

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?

**Answer:** A precondition on a method $m$ is a specification of what must be true before when $m$ is called. A postcondition of a method $m$ states what is true after the function has been executed, assuming that the precondition was met. Conceptually, exceptions are part of the postcondition of a method, since they describe the result of running the method.
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.

```java
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();

      t0.accept(t1);
   }
}

public class A implements Thing {
   public void accept(Thing t) { t.doIt(this); }
   public void doIt(A a) { Do.doAA(a, this); }
   public void doIt(B b) { Do.doBA(b, this); }
}

public class B implements Thing {
   public void accept(Thing t) { t.doIt(this); }
   public void doIt(A a) { Do.doAB(a, this); }
   public void doIt(B b) { Do.doBB(b, this); }
}
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.

Answer: There are many possible answers to this question. Here is one program.

```java
public class Foo {
    public Object lock1 = new Object();
    public Object lock2 = new Object();
    public class Thread1 extends Thread {
        public void run() {
            synchronized (lock1) {
                synchronized (lock2) {
                    System.out.println("Got both locks.");
                }
            }
        }
    }
    public class Thread2 extends Thread {
        public void run() {
            synchronized (lock2) {
                synchronized (lock1) {
                    System.out.println("Got both locks.");
                }
            }
        }
    }
    public static void main(String[] args) {
        (new Thread1()).start();
        (new Thread2()).start();
    }
}
```

This program may deadlock if Thread1 acquires lock1, and then before it can continue, Thread2 is scheduled to acquire lock2. In this case, neither thread can proceed, because each is waiting for the other's lock.

Other valid answers include combinations of wait and notify(All) that block. For example, if a thread blocks in a call to wait and there is no possible call to notifyAll, I considered that a valid 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;
    }
}

Answer:

public class SharedFileReader {
    private FileReader theFile;
    private int count = 0;
    private Lock theLock = new ReentrantLock();

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

    public synchronized int getCount() {
        int theCount;
        theLock.lock();
        theCount = count;
        theLock.unlock();
        return theCount;
    }

    public synchronized int read() throws IOException {
        theLock.lock();
        try {
            int c = theFile.read();
            count++;
            return c;
        }
        finally {
            theLock.unlock();
        }
    }
}
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 {
    private Thread owner = null;
    private int count = 0;

    synchronized public void lock() {
        Thread t = Thread.currentThread();
        while (owner != null && owner != t)
            try { wait(); } catch (InterruptedException e) { }
        owner = t;
        count++;
    }

    synchronized public void unlock() {
        if (owner != Thread.currentThread())
            throw new IllegalStateException();
        count--;
        if (count == 0) {
            owner = null;
            notifyAll();
        }
    }

    synchronized public boolean trylock() {
        Thread t = Thread.currentThread();
        if (owner != t && owner != null)
            return false;
        owner = t;
        count++;
        return true;
    }
}
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).

This 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):

![Diagram of dynamic proxies and futures]

The last page of the exam contains the portion of the dynamic proxy API you’ll need to do this problem.
public class Future implements InvocationHandler {
    private Object result;
    private Throwable exn;
    private boolean complete = false;

    public static Object createFuture(final Object o) {
        Class c = o.getClass();
        Method[] m = c.getMethods();
        int i = 0;
        while (i < m.length) {
            if (m[i].getName().equals("call") && m[i].getParameterTypes().length == 0)
                break;
            i++;
        }
        if (i == m.length)
            throw new IllegalArgumentException();
        final Method theMethod = m[i];
        final Future theFuture = new Future();
        Thread t = new Thread() {
            public void run() {
                try {
                    synchronized (theFuture) {
                        theFuture.result = theMethod.invoke(o, null);
                        theFuture.complete = true;
                        theFuture.notifyAll();
                    }
                }
                catch (InvocationTargetException e) {
                    synchronized (theFuture) {
                        theFuture.exn = e.getCause();
                        theFuture.complete = true;
                        theFuture.notifyAll();
                    }
                }
            }
        };
        t.start();
        return Proxy.newProxyInstance(theMethod.getReturnType().getClassLoader(),
                                        theMethod.getReturnType().getInterfaces(), theFuture);
    }

    public synchronized Object invoke(Object proxy, Method m, Object[] args) throws Throwable {
        while (!complete)
            try { wait(); } catch (InterruptedException e) { }
        if (exn != null)
            throw exn;
        try {
            return m.invoke(result, args);
        }
        catch (InvocationTargetException e) {
            throw e.getCause();
        }
    }
}
Question 6. More Short Answer (10 points).

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

   Answer: There are four main disadvantages:

   1. Readability – The same operations performed using reflection are much more verbose, and therefore much harder to read and understand.
   2. Performance – Accessing fields or invoking methods with reflection is much slower than doing so directly. Since the use of reflection often gets in the way of compiler optimizations, it may be much, much slower.
   3. Code size – Again, since performing the same task with reflection takes many more steps, it requires more code, which may increase the size of your program. (Note that good uses of reflection often reduce code size compared to large if/then/else block alternatives.)
   4. Ease of errors – When invoking methods and accessing fields using reflection, many errors that would be caught be the compiler normally become run-time exceptions, making it easy to make certain mistakes.

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

   Answer: The doPrivileged() method call performs action with all privileges of the caller’s protection domain. If during the execution of action there is a call to checkPermission(), the stack walk to check for the appropriate permissions will halt at the call to doPrivileged(). In this way doPrivileged() allows untrusted code to call methods to perform trusted operations on its behalf.

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.

   Answer: Yes. The Java garbage collector uses various techniques to determine what memory is live (i.e., may be accessed later) at any given point. These techniques are necessarily conservative, meaning that the garbage collector might determine that data is live when, in fact the programmer will no longer use it. See the lecture slides on garbage collection for one example.
## Cheat Sheet – API for Dynamic Proxies

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