1. (Threads)
   Give the implementation of a Java class that supports (non-recursive) read locks and write locks, with only one writer, or any number of readers, allowed to acquire the lock at any point in time:

```
interface ReadWriteMonitor {
  void acquireReadLock() throws InterruptedException;
  void releaseReadLock();

  void acquireWriteLock() throws InterruptedException;
  void releaseWriteLock();
  }
```

   For full credit, if a single thread requests a write lock, it should eventually acquire the write lock, even if other threads are constantly acquiring read locks (i.e., it can't just wait until a moment when no other thread wants a read lock, because such a moment may never come).

   Answer:

```
class MyLock implements ReadWriteMonitor {
  int readLocks;
  boolean writeLocked;
  int writeLocksDesired;
  void synchronized acquireReadLock() throws InterruptedException {
        while (writeLocked || writeLocksDesired != 0) wait();
        readLocks++;
        }
  void synchronized releaseReadLock() {
        readLocks--;
        notifyAll();
        }
  void synchronized acquireWriteLock() throws InterruptedException {
        writeLocksDesired++;
        try {
          while (writeLocked || readLocks != 0 ) wait();
          writeLocked = true;
          }
        finally {
          writeLocksDesired--;
          }
        }
  void synchronized releaseWriteLock() {
        writeLocked = false;
        notifyAll();
        }
  }
```

2. (Thread communication)
Design a class that implements the TodoList interface:

```
interface TodoList {
  void add(Runnable r);
  }
```

Assume you have use of a class QueueImpl, which implements:

```
interface Queue {
  void enqueue(Object r);
  Object dequeue();
  int size();
  }
```

The dequeue function returns null if there is nothing to dequeue. The QueueImpl class is not thread safe (does not use any synchronization). The ToDoList implementation should invoke the run methods of the task added to the todo list, one at a time, in the order they are added. However, the add function should return immediately. Note that no more than one task should be running at a time.

Answer:

```
class MyTodoList implements TodoList, Runnable {
  Queue q;
  synchronized public void add(Runnable r) {
        if (q == null) {
                q = new Queue();
                Thread t = new Thread(this);
                t.setDaemon(true);
                t.start();
                }
        q.enqueue(r);
        notifyAll();
        }
  public void run() {
        while (true) {
        Runnable r;
        synchronized(this) {
                while (q.size() <= 0)
                        try { wait(); }
                        catch (InterruptedException e) {};
                r = q.dequeue();
                };
        r.run();
        }}
  }
```

3. (RMI)
   Say I want to set up a distributed service, with clients and servers communicating via RMI. I want to allow anyone to write their own client application. However, I don't want to have the server uploading client stubs for each of the different clients I communicate with, for several reasons: it requires clients to set up a codebase where the stub can be uploaded from, there are security issues with uploading the code to the server (denial of service attacks aren't prevented by the security model), etc.

   Instead, I want to have just one client stub, no matter how many client applications the server communicates with. That way, the stub code doesn't have to be uploaded. Don't depend on the hack that two different clients with the same class name that implement the same interface can use the same stub (as we did in project 4).

   Sketch a design for allowing servers to send messages to client applications, without requiring any code to be uploaded to the server. There is a good solution that doesn't require any mucking with rmic (the RMI compiler) or any other details of the RMI implementation. Such a solution is (much) preferred. Your solution may require some changes in how people write and use clients: that is OK.

   Answer:
   Define a standard ClientProxy class. The ClientProxy class is a remote object that implements the Client interface, and takes a reference to another (local) object that implements the Client interface. The ClientProxy simply passes all calls through to the local object. Since the ClientProxy is standard and the only remote object, the server can have a copy of the ClientProxy_Stub code already loaded, and doesn't need to load any code from the clients.

4. (Jini)
   In Jini, say you have already identified a lookup service. Without going into excessive detail, what kinds of queries can you ask of a lookup service? What is required to make these queries truly useful for clients from multiple vendors/implementors?

   Answer:
   You can ask for services that implement a particular interface. You can also ask for services that have a particular set of attributes (e.g. Building is AVWilliams, Speed is 10 pages/minute).

   The usefulness of these schemes all depend on different vendors/implementors agreeing on common standards for interfaces and attributes.

5. (RMI and threads)
Consider the following code for threading and synchronization issues. Comment on whether it can exhibit:

- Unsynchronized access to shared data (data races)
- Significant unresponsiveness due to lock contention
- Could deadlock
- Other problems you can identify (not necessarily threading related)

In each case, your answer may depend the interaction of the MessageBroadcast class with other classes. You should write your question as to whether there are any circumstances under which those problems arise.

Would your answers be any different if Listener was not a Remote interface? Describe any differences.

```
import java.util.*;
import java.rmi.RemoteException;

public class MessageBroadcast implements Listener {
  Set listeners = new HashSet();
  public synchronized void addListener(Listener l) {
        listeners.add(l);
        }
  public synchronized void removeListener(Listener l) {
        listeners.remove(l);
        }
  public synchronized void message(String s) throws RemoteException {
        for(Iterator i = listeners.iterator(); i.hasNext(); ) {
                Listener l = (Listener) i.next();
                l.message(s);
                }
        }
  }

interface Listener extends java.rmi.Remote {
  void message(String s) throws RemoteException;
  }
```

If you found any problems, give a new solution that fixes the problems you found and doesn't introduce any new ones. It is OK if your new solution allows Listeners to receive messages out of order, but clearly state if your new solution has this feature.

Java API notes: HashSet's are unsynchronized. You can create a synchronized Set with

```
Set listeners = Collection.synchronizedSet(new HashSet())
```

For both synchronized and unsynchronized sets, if one thread modifies a set while another thread is iterating through an iterator for the set, it is an error. The interface java.util.Set is defined as:

```
public interface Set extends java.util.Collection {
    public abstract boolean add(java.lang.Object);
    public abstract boolean addAll(java.util.Collection);
    public abstract void clear();
    public abstract boolean contains(java.lang.Object);
    public abstract boolean containsAll(java.util.Collection);
    public abstract boolean isEmpty();
    public abstract java.util.Iterator iterator();
    public abstract boolean remove(java.lang.Object);
    public abstract boolean removeAll(java.util.Collection);
    public abstract boolean retainAll(java.util.Collection);
    public abstract int size();
    public abstract java.lang.Object toArray()[];
    public abstract java.lang.Object toArray(java.lang.Object[])[];
    }
```

Answer:
The code is adequately synchronized, if perhaps over synchronized. No data races exist in the code.

However, the message(...) function could lead to the system being unresponsive, because it holds onto a lock while sending messages to listeners. If one client is very slow, it will delay other clients and attempts to add or remove listeners.

There is some potential for deadlock. Say that sending a message to a listener cause a chain of events to leads to attempting to register another listener. If those calls occur via RMI, then the call to register another listener could occur in a thread other than the one that broadcast the message, leading to deadlock.

The message(...) does not handle RemoteExceptions well. If a RemoteException occurs in sending a message to any listener, it doesn't try to contact the other listeners and just immediately propagates the message to it's caller.

If Listener were not a RemoteInterface, then deadlock would be a little harder to provoke. But if another thread held a lock on a Listener, than then sent a message to a MessageBroadcast that listener was subscribed to, then deadlock could result if another thread was sending messages through that MessageBroadcast.

The basic fix to these problems is to not hold a lock while we make calls to listeners. However, if we are iterating through the HashSet while another thread adds or removes listeners, we have a problem. So we make a copy of the set of Listeners we need to send a message to.

```
public void message(final String s) {
  Object [] l;
  synchronized (this) {
        l = listeners.toArray();
        }
  for(int i = 0; i < l.length; i++) {
        final Listener fl = ((Listener)l[i]);
        Thread t = new Thread() {
                public void run() {
                    try {
                        fl.message(s);
                    } catch (RemoteException e) {}
                    }};
        t.start();
        }
  }
```

My solution just ignores RemoteExceptions. Alternatively, it could call removeListener when a RemoteException is thrown talking to a Listener.

Because this code is unsynchronized, it is possible that if two threads invokes the message function, that different clients will receive the messages in different orders, and that a particular client will receive messages in an order different than the order of the message() invocations.

6. (Generic Java)
Given the following GJ code:

```
class Pair<A, B> {
  public A first;
  public B second;
  public Pair(A x, B y) {first = x; second = y}
  public A car() { return first; }
  public B cdr() { return second;}
  public static <D> Pair<D, D> duplicate(D x) {
    return new Pair<D, D>(x, x);
  }
}

class MyUse {
  public static void main(String args[]) {
    Pair<Integer, Float> p1 = Pair(Integer(3), Float(5.5));
    Integer i = car(p1);
    Pair<Integer, Integer> p2 = Pair.duplicate(i);
    Float f = cdr(p1);
    p1 = Pair.duplicate(f);
  }
}
```

Show the translation into core Java of the two classes and show any errors that might be reported if the GJ compiler did the translation.

Answer:

```
class Pair {
  public Object first;
  public Object second;
  public Pair(Object x, Object y) {first = x; second = y}
  public Object car() { return first; }
  public Object cdr() { return second;}
  public static Pair duplicate(Object x) {
    return new Pair(x, x);
  }
}

class MyUse {
  public static void main(String args[]) {
    Pair p1 = Pair(Integer(3), Float(5.5));
    Integer i = (Integer) car(p1);
    Pair p2 = Pair.duplicate(i);
    Float f = (Float) cdr(p1);
    p1 = Pair.duplicate(f);
```

```
  }
}
```

The last line will cause a type error at GJ compile time, because p1 is of type Pair¡Integer, Float¿ and the return type on the right hand side is Pair¡Float, Float¿.