

Practice Final

Final Exam Info

Our regular exam time is Tue June 12th, in Gates b01 3:30-5:30. SITN students may take the exam on campus, or may wait at their site and we will send a copy of the exam to be taken on Tue (or possibly Wed) and faxed back. On campus students who would like to attend the alternate should email Nick with the subject "k alternate exam".

The final exam will focus on short answer questions and code writing questions. The short answer questions will be +1 for a right answer, 0 if left blank, and -1 for a wrong answer. For the coding questions, We will not be picky about details such as syntax, exact method names, or import directives. However, it will be important that you can write structurally correct code for the topics we have studied...

Swing: components, drawing, event listeners

Model-View-Controller structure

Threads: creation, synchronization (synchronized), coordination (wait/notify)

RMI: creation, messages, threads

The exam will be 2 hours long and will be open note, open book (but not open computer). It is unlikely I will emphasize questions where you can just look up the answer. The questions will more likely require you to write code that demonstrates understanding of a few concepts at once.

To study for the exam, you should review all the lecture examples, your own homework solution code, and these problems from previous exams. You can also review the web criteria writeups for the homeworks -- they discuss the key parts of the correct solution. To really study for a code-writing exam, **passive** review of the solution code is not a good strategy. You should get out a blank piece of paper and try to create solution code from scratch for lecture examples, homework problems, etc. -- even if your solution is not perfect, you learn a lot more making a blank-paper attempt.

This handout includes one cs193k exam with solutions, and one without solutions.

(Thanks to Jason Townsend for working up these solutions...)

1) Short Answer (10 points)

Brief answers are fine for these.

a) What is one feature that makes Vector slower than the equivalent new Collection classes?

All Vector methods are synchronized, even if the client is not threaded

b) Why has Thread.stop() been deprecated in Java 2?

Because it's hard to prevent this from interrupting a critical section and leaving something in an inconsistent state.

c) What is the difference between paint() and repaint()?

paint() is the notification that a component should draw. Repaint is a request by the application that a paint should happen in the future.

d) Remember in HW1, you built a FilteredTableModel that worked by keeping a pointer to another "real" table model that actually stored the data. Could the table model from HW3, the TMClient, act as the "real" table model for a FilteredTableModel? Put another way: could you take the filter table from HW1, add it into HW3, and get a remote table displayed with filtering? (yes or no?)

Yes. Since it only needs a class that implements the TableModel interface, and TMClient is sufficient (neat!).

2) Swing GUI (15 points)

Here is a moderately complex but useless frame which exercises some of the core areas of Swing we've used over the quarter. Write a JFrame subclass with the following features...

- It uses a BorderLayout for its outermost layout
- When the frame closes, its application exits.
- The west contains be a single "Frustration" JButton which disables itself when clicked.
- The east contains a single JLabel containing the text "Spring".
- The center contains a 4 buttons, all with the title "Foo". These buttons are aligned in a single vertical column.
- The south contains 4 buttons, also with the title "Foo". These buttons are aligned in a single horizontal row.
- Write one method with enough parameters that it can be used to create both of the above 4 Foo button groupings. Or put another way: don't repeat the code to create the 4 Foo buttons.

- Clicking any of the 8 Foo buttons switches the Spring label back and forth between being enabled and disabled.
- Since the 8 Foo buttons all do the same thing, they should all use a single listener object. There should not be 8 separate listener objects.

Your code...(constructor first)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MyFrame extends JFrame {

    public MyFrame(String name) {
        super(name);
        getContentPane().setLayout(new BorderLayout());

        final JButton frustration = new JButton("Frustration");
        frustration.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                frustration.setEnabled(false);
            }
        });
        getContentPane().add(frustration, BorderLayout.WEST);

        final JLabel spring = new JLabel("Spring");
        getContentPane().add(spring, BorderLayout.EAST);

        ActionListener springToggle = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                spring.setEnabled(!spring.isEnabled());
            }
        };

        JPanel centerPanel = new JPanel();
        centerPanel.setLayout(new BoxLayout(centerPanel, BoxLayout.Y_AXIS));
        makeFourFooButtons(centerPanel, springToggle);
        getContentPane().add(centerPanel, BorderLayout.CENTER);

        JPanel southPanel = new JPanel();
        southPanel.setLayout(new BoxLayout(southPanel, BoxLayout.X_AXIS));
        makeFourFooButtons(southPanel, springToggle);
        getContentPane().add(southPanel, BorderLayout.SOUTH);

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }

    public void makeFourFooButtons(Container panel, ActionListener listener)
```

```

{
    for (int i = 0; i < 4; ++i) {
        JButton foo = new JButton("Foo");
        panel.add(foo);
        foo.addActionListener(listener);
    }
}

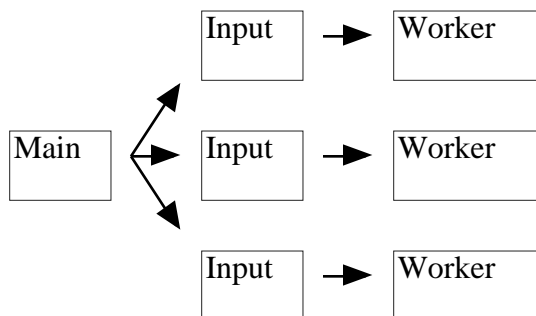
public static void main(String args[]) {
    JFrame frame = new MyFrame("CS193K");
    frame.pack();
    frame.show();
}
}

```

3) Thread Decryption (35 points)

This problem will use several classes and Threads to do decryption in parallel. There are three classes....

- The Main object sets things up, reads lines out of the input file and feeds them to Workers through their Input objects. The input file is made of strings like "w3ena;lkjasdf30924323xzxz" each representing an encryption problem that a Worker is going to try to solve.
- The Input object holds a single string for its Worker. In the homework, the TransactionBlock object could hold an entire array of inputs. In contrast, in this problem, the Input object can only hold one input at a time. Each Worker will have its own Input object.
- Worker thread objects removes an input string from its Input object, does a (slow) decrypt() test on the string, and if the test is successful, prints the string. The decrypt() operation is very slow, so the main thread should have an easy time keeping the Input objects filled with strings.



1) Input

The Input object is the connection between the main thread and each Worker. Each worker has its own Input object. For string storage, each Input should store a single string pointer. If the string pointer is null, the Input is empty, and so has

room to contain an incoming string. Input should support a `get()` method that the Worker uses to get a string out of the Input. `Get()` should remove the string from the Input, making the Input empty. If the Input is empty at the time of the `get()`, then the request must block until there is a string in the Input. The Worker cannot proceed without a string to operate on.

On the other hand, when putting string into the Input objects, the Main thread does not need to block if there is not room in the Input. The attempt to add may or may not succeed, but failure does not necessarily lead to blocking. If there is not room for the string in one Input, the main thread can still try to add the string to one of the other Inputs which may have room. The main thread should block when no Input object has room. The main thread should awaken when at least one of the Input objects has room, and then it can try adding to each of them, knowing that at least one of the attempts must succeed. There must be a mechanism by which any of the Workers can alert the main thread that there is room (somewhere) for another string.

(partial credit) If you cannot devise a way for the Workers to notify the main thread that there is room, you could have the main thread loop constantly trying to add to the Input objects. This is inefficient, but it works.

2) Worker

Worker is a simple Thread subclass. Each Worker should get (at least) a pointer to its Input object at construction time. The Worker should repeatedly retrieve a string from its Input and run the boolean `Crypt.decrypt(string)` method on the string (we're not writing this method, we're just calling it). If `decrypt()` returns true, then print the string out. The special string "end" marks the end of the input, and the Worker should exit normally. (We are not going to check `isInterrupted()` in this problem.)

3) Main

The Main object should also be a subclass of Thread. Its `run()` should orchestrate the whole computation. First it should create and start everything: 4 Inputs and 4 Workers. The "4" in this case should just be a constant — Main should use arrays to support an arbitrary number of Input/Worker pairs. Main should then open the file "crypt.txt" which contains problem strings, one per line. Each string should be fed to one Worker for processing through its Input. (as described above) Main should block when there is no Input object with room for another string. When main knows there is room in at least one Input, it's ok for it potentially to try the add operation on all 4 (trying to add to an Input is cheap). When the file is exhausted, main should feed the string "end" to each Input, and then wait for all the Workers to finish.

Solution: the use of `setSpace()`, `getSpace()` in main is the trickiest part -- solutions that got everything but that still got most of the points.

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
```

```

class Input {

    private String myString;
    private Main myMain;

    public Input(Main theMain) {
        myMain = theMain;
    }

    public synchronized String get() {
        while (myString == null) {
            try {
                wait(); // block of no string available
            } catch (InterruptedException ie) {
                ie.printStackTrace();
            }
        }
        String temp = myString;
        myString = null;
        synchronized(myMain) {
            myMain.setSpace(true); // in case they were waiting
        }
        return temp;
    }

    /**
     * @return true if the string was successfully put, false
    otherwise
     */
    public synchronized boolean tryPut(String toPut) {
        if (myString == null) {
            myString = toPut;
            notify(); // in case our worker is waiting for
    this input
            return true;
        } else {
            return false;
        }
    }
}

class Worker extends Thread {

    private Input myInput;

    public Worker(Input in) {
        myInput = in;
    }

    public void run() {
        String str = myInput.get();
        while (!str.equals("end")) {
            if (Crypt.decrypt(str)) {

```

```

        System.out.println(str);
    }
}

public class Main extends Thread {

    private Input[] inputArray;
    private Worker[] workerArray;
    private static final int NUM_WORKERS = 4;

    private boolean space = true;    // space available for writing

    // Notify if setting that there is space available
    public synchronized void setSpace(boolean value) {
        space = value;
        if (space) notify();
    }

    // Wait until there's space available
    public synchronized void getSpace() {
        if(!space) {
            try {
                wait();
            }
            catch (InterruptedException) (ignored) {}
        }
    }

    public Main() {
        inputArray = new Input[NUM_WORKERS];
        workerArray = new Worker[NUM_WORKERS];
        for (int i = 0; i < NUM_WORKERS; ++i) {
            inputArray[i] = new Input(this);
            workerArray[i] = new Worker(inputArray[i]);
        }
    }

    public void run() {
        try {
            BufferedReader inputFile =
                new BufferedReader(new FileReader("crypt.txt"));
            int i;
            for (i = 0; i < NUM_WORKERS; ++i) {
                workerArray[i].start();
            }
            String aLine = inputFile.readLine();
            while (aLine != null) {
                boolean success = false;
                while (!success) {
                    setSpace(false);    // assume there's no space
                    for (i = 0; i < NUM_WORKERS; ++i) {
                        if (inputArray[i].tryPut(aLine)) {
                            success = true;
                        }
                    }
                }
            }
        }
    }
}

```

```

        break;
    }
    }
    if (!success) {
        getSpace(); // check/wait for space
    }
}
aLine = inputFile.readLine();
}

for (i = 0; i < NUM_WORKERS; ++i) {
    workerArray[i].join();
}
} catch (Exception e) {
    e.printStackTrace();
}
}

public static void main(String args[]) {
    Main theMain = new Main();
    theMain.run();
}
}

class Crypt {
    public static boolean decrypt(String theString) {
        // some lengthy operation
        return true;
    }
}
}

```

4) RMI (40 points)

As usual, the RMI problem will separate a computation into "client" and "server" sides. The basic idea is...

The DBServer exists on the server side, storing a database of a million strings in a single Vector. The DBServer responds to a search(String target) message. The goal of search() is to search through the database on the server side and return to the caller copies of the relatively few strings in the database that contain the target string. The search() message is part of the DBRemote interface that DBServer exposes to remote clients.

DBClient is a simple JFrame subclass with a text field, a search button, an "answer" text area, and a DBRemote connection to the DBServer. When the search button is clicked, the DBClient sends a search() message, and when the answer comes back, it is put in the answer text area.

There is one complication. search() could be written to return a Vector of the matching strings. This is simple, but has the disadvantage that the client needs to

wait for the entire `search()` to complete before getting back the answer. Our approach will be to send an `XObjectRemote` as a second argument to the `search()`. The `search()` can return quickly (with no answers yet). When it gets the `search()` message, the server can start up a separate thread to do the search, and that thread can use the `XObjectRemote` to communicate back the string answers as they are found, one at a time. The communication of each string can also be done in a separate thread - there's no reason to make the searching wait for the transmission of each string.

To minimize what you need to write, we will concentrate on `search()` itself, and ignore most of the rest of the routine RMI code. Your code should catch `RemoteException` where necessary, but can then ignore the exception silently.

XObjectRemote

Here's the code for `XObjectRemote` -- you should use these without modification. Your `DBServer` and `DBClient` can use the `XObjectRemote` however they like, so long as they are consistent with each other — it's a private dialog.

```
public interface XObjectRemote extends java.rmi.Remote {
    public void send(Object message) throws RemoteException;
}

public class XObjectServer extends UnicastRemoteObject
    implements XObjectRemote {
    XObjectListener listener;

    public XObjectServer(XObjectListener listener) throws RemoteException {
        super();
        this.listener = listener;
    }

    public void send(Object message) throws RemoteException {
        listener.send(message);
    }
}
```

a) DBRemote

Define the `DBRemote` interface to specify the prototype for the `search()` message that the client can send to the server. The client cannot interrupt or withdraw its `search()` requests.

b) DBServer

The only code you need to write is `search()` and any helper methods it needs. We will assume that the `DBServer` loads its million strings at construction time. We will also assume that all the `Naming.rebind()` stuff has been done. Use `String.indexOf(target)` to test if the target is in each string in the database -- it returns -1 if the target is not found.

c) DBFrame

Write the code for the DBFrame constructor which should set up all the components. DBFrame should contain a text field and search button to initiate the search and a JTextArea to show the results. Use getText() to get the target string out of the JTextField. Leave the JTextArea with its default constructed size, and use its convenient append(String) message to append the answer strings as they come in. Beyond the DBFrame constructor, write the code to initiate the search() and get back the results. We will assume that the naming lookup and other setup is written somewhere else.

```
import java.rmi.*;

public interface DBRemote extends java.rmi.Remote {
    // Ask the db to search for the given string
    // and answer back on the given XOR object
    public void search(String searchString, XObjectRemote returnPath) throws
    RemoteException;
}
```

b)

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

public class DBServer extends UnicastRemoteObject
    implements DBRemote {

    Vector strings;

    public void search(final String searchString, final XObjectRemote
returnPath) {
        if (searchString == null) return;
        Thread searchThread = new Thread() {
            public void run() {
                String tryString;
                for (int i = 0; i < strings.size(); ++i) {
                    tryString = (String)strings.elementAt(i);
                    if (tryString != null && tryString.indexOf(searchString) != -1) {
                        // found a match
                        try {
                            returnPath.send(tryString);
                        } catch (RemoteException re) {
                            re.printStackTrace();
                        }
                    }
                }
            }
        };
        searchThread.start();
    }
}
```

```

// not asked for in the question
public DBServer() throws RemoteException {
    super();
    // do something to load strings
}
}

c)
import java.awt.*;
import java.awt.event.*;
import java.rmi.*;
import javax.swing.*;

// GUI side of the db/remote question
// We listen to the XObjectServer object we send to the db

public class DBFrame extends JFrame implements XObjectListener {
    DBRemote dbr; // assume this gets set somehow

    JTextArea textArea;
    JTextField textField;

    public DBFrame(String name) {
        super(name);
        getContentPane().setLayout(new BorderLayout());

        textField = new JTextField();
        getContentPane().add(textField, BorderLayout.NORTH);

        textArea = new JTextArea();
        getContentPane().add(textArea, BorderLayout.CENTER);

        JButton searchButton = new JButton("Search");
        searchButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try {
                    dbr.search(textField.getText(), new
XObjectServer(DBFrame.this));
                } catch (RemoteException re) {
                    re.printStackTrace();
                }
            }
        });
    }

    // notification on the XObject
    // note: not on Swing thread
    public void send(Object message) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                textArea.append((String)message);
            }
        });
    }
}

```

```

    }
}

```

d) Interruption

(there is no code writing for this part) Suppose you wanted to have an "Interrupt" button on the DBFrame which can interrupt the search() thread which is running on the server. Write a one paragraph description of how you could build this using XObjectRemote-XObjectServer (XOR-XOS). For your proposal, describe the steps that happen starting with the initiation of the search, the click of the interrupt button on the client side, leading to the interrupt() message being sent to the right Thread object on the server side. Explain which object is keeping the pointer to the correct Thread object to interrupt.

Solution: Create a little server side "stop" object on the server that keeps a pointer to the searching thread object. Pass back to the client a remote copy of the stop object. The client can send a message to the stop object, so that it can do an interrupt() on the real server side thread.

1) Short Answer (20 points)

These are worth 2..4 points if correct, 1 point if left blank, and 0 if answered incorrectly — if you don't know the answer, leave it blank (or cross out 100% of your answer).

a. What does the update region represent (one sentence)? (3)

b. What does this code snippet print out? (2)

```

Thread t1 = new Thread() {
    public void run() {
        foo();
    }

    public void foo() {
        if (Thread.currentThread()==this)
            System.out.println("yes");
        else System.out.println("no");
    }
};
t1.foo();

```

c. What are all the possible outputs of the following code? (4) (The exception code has been removed for clarity. Assume there are no exceptions.)

```
Object lock = new Object();
Thread t1 = new Thread() {
    public void run() {
        System.out.println("a");
        lock.wait();
        System.out.println("b");
    }
};

Thread t2 = new Thread() {
    public void run() {
        System.out.println("c");
        lock.notify();
    }
};

t1.start();
t2.start();
```

d. What is a significant capability of RMI that CORBA does not have (one sentence)? (3)

e. Suppose there's a "Boing" button in a JFrame. Write code so that when the button is clicked, 30 seconds later the button changes to be disabled (setEnabled(false)). Use a separate thread so the GUI does not lock up during the 30 seconds. (4)

```
final JButton button = new JButton("Boing");
container.add(button);

button.addActionListener(
// YOUR CODE HERE
```

f. Re-write the following code to be significantly faster. The algorithm does not change, but the Java code that implements it can be significantly sped up. The "foo" object contains ints "a" and "b" and an int array "data". The algorithm adds up the numbers in the array in the range [a..b). (4)

```

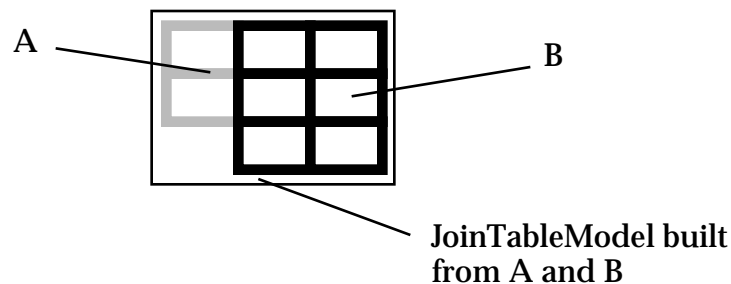
int i;
int sum = 0;
for (i=foo.a; i<foo.b; i++) {
    sum += foo.data[i];
}

```

2) MVC (25 points)

For this problem, you will build a `JoinTableModel` (JTM) that takes two table models and presents their data as one big table model. Instead of writing the whole class, you will just write a few key pieces of code.

Each JTM takes two table models, A and B, in its constructor and presents their data as a big table as if they were next to each other. This drawing shows a JTM built with (2 x 1) A and (3 x 2) B. The resulting JTM is (3 x 3). Col 0 in the JTM takes its data from Col 0 in A. Col 1 in the JTM takes its data from Col 0 in B. The JTM should present "missing" elements, such as (2, 0) in the drawing, as null.



We'll use the following assumptions...

- As usual, the JTM should not store any of its own table element data. It should message the underlying A and B to get the "real" data as necessary.
- A and B will not change shape after the JTM is created — the number of rows and columns in A and B will not change.
- For simplicity, you may assume that `getValueAt()` messages sent to A and B will return null for out of bounds requests.
- For simplicity, the only "update" case we will deal with will be a change of a single cell at a time in A or B.
- For simplicity, JTM will present the data in A and B, but will not edit that data. The JTM will return false for `isEditable()`.

a) Basic Setup

Complete the code for the class header, ivar declarations, and constructor as needed for the other parts of your solution...

```
public class JoinTableModel extends AbstractTableModel

{

private TableModel a;
private TableModel b;

public JoinTableModel(TableModel aModel, TableModel bModel) {
    a = aModel;
    b = bModel;
}
```

b) Data

Define the two methods `getRowCount()` and `getValueAt()`.

```
public int getRowCount() {

public Object getValueAt(int row, int col) {
```

c) Updates

For simplicity, the only update case the JTM needs to deal with is when a single cell in A or B changes. Write the necessary listener code below. You may need to add to your part (a) answer a little to set up the listener structure.

For reference, here are the docs for the `tableChanged()` message...

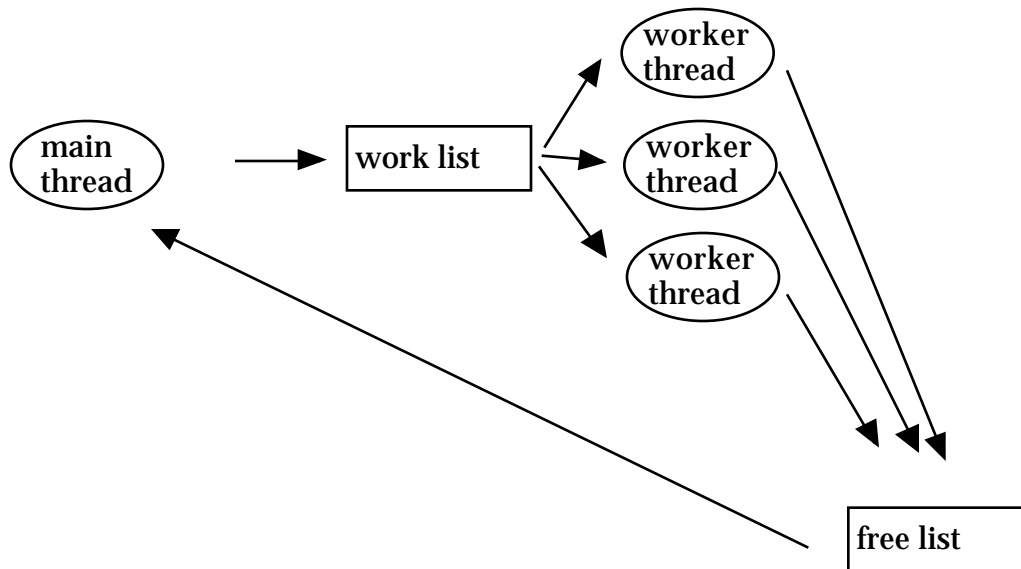
```
public void tableChanged(TableModelEvent e);
// TableModelEvent responds to these messages...
int getColumn() // changed col
int getFirstRow() // changed row
TableModel getSource() // the TableModel changing
```

3) Threads (30 points)

For this problem, you will build a little thread system similar to the homework 2b.

The main thread runs a loop where it gets a "job" object representing some work to be done, sets up its computation, and puts it in a "work" list. Worker threads pull jobs off the work list and send the compute() message to each job to do its computation. Then, the worker puts the job back on a "free" list for re-use. When the main thread needs a job object, it pulls it off the free list instead of allocating it with a call to new. The main thread blocks when there are no jobs available in the free list. Likewise, the worker threads block when there are no jobs available in the work list.

This drawing shows the path of the job objects in the system...



Here are two classes you will use **without modification**...

Job

```

// Represents a computation to be done
class Job {
    public Job next; // JobList uses this field to make the list

    // Setup the job for a computation
    public void setup(int value) { <some code> }

    // Perform the computation
    public void compute() { <some code> }
}
  
```

JobList

Stores jobs in a linked list. Supports add() and get() operations. The JobList methods are not thread safe — **you will need to work around this problem in your code without modifying the given Job or JobList code.**


```
// Stores a list of Jobs. Not thread safe.
class JobList {
    private Job head = null;

    // Add a job to the list
    public void add(Job job) {
        job.next = head;
        head = job;
    }

    // Remove a job from the list and return it.
    // Returns null if no job is available.
    public Job get() {
        if (head == null) return(null);
        else {
            Job result = head;
            head = head.next;
            return(result);
        }
    }
}
```

Simplifications

We'll make the following simplifying assumptions...

No exceptions — completely ignore exceptions in the code. Act as if `wait()`, `sleep()`, etc. do not throw exceptions.

notify() — it's ok to send `notify` even if there's a chance that no thread is waiting. For this problem, both `notify()` and `notifyAll()` will work, so don't worry about the distinction between the two.

Infinite loop — in parts (a) and (b) the program does not halt correctly since the worker threads are in an infinite loop pulling jobs off the work list. We will fix this in part (c).

a) Main Thread

The first part of the main thread is done for you. The given code puts 10 job objects on the free list — that will be the only supply of job objects for the whole run. The given code also starts up three Worker threads (defined in part b). Your job is to fill in the main loop which gets a job object, sets it up, and puts it on the work list...

```
class Main extends Thread {
    JobList free = new JobList();
    JobList work = new JobList();

    public void run() {
        // Put 10 jobs in the free list
        for (int i=0; i<10; i++) {
            free.add(new Job());
        }

        // start 3 workers
```

```

for (int i=0; i<3; i++) {
    new Worker(free, work).start();
}
// YOUR CODE IN HERE...
// main loop:
// -get a job object
// -set it up with the value i
// -put it on the work list
for (int i=0; i<1000; i++) {

```

b) Worker

The basic Worker code is done for you. Add the necessary code inside the while loop...

```

class Worker extends Thread {
    private JobList free;
    private JobList work;

    public Worker(JobList free, JobList work) {
        this.free = free;
        this.work = work;
    }

    public void run() {
        Job job;

        // YOUR CODE IN HERE...
        // -acquire a job
        // -compute()
        // -put on free list
        while(true) {

```

c) Orderly Halting

There is no code writing for this problem. As written above, the code never halts since the workers are in an infinite loop pulling jobs of the work list. We need a way for the Main thread to communicate to the workers that there are no more jobs. In the homework, we did this by feeding a "special" job to each worker to mark the end. That approach will not work in this case because the data in the job is private — the worker cannot see it. Propose a **different way** that the main thread can signal the workers to exit their run() after all the jobs have been computed. Your solution can modify the Main and Worker classes. Describe your scheme in three sentences starting with the following phrase:

After the main thread has put its 1000 jobs on the work list, the main thread

4) RMI (25 points)

For this problem you will build a little RMI system that includes two XObjectServers. Here's the XObject code for reference...

```

// XObjectRemote.java

```

```

public interface XObjectRemote extends java.rmi.Remote {
    public void send(Object message) throws RemoteException;
}

// XObjectServer.java
public class XObjectServer extends UnicastRemoteObject
    implements XObjectRemote {
    XObjectListener listener;

    public XObjectServer(XObjectListener listener) throws RemoteException {
        super();
        this.listener = listener;
    }

    public void send(Object message) throws RemoteException {
        listener.send(message);
    }
}

// XObjectListener.java
public interface XObjectListener {
    public void send(Object message);
}

```

Here's the setup for the problem: There is a **BoingServer** and a **BoingClient** -- called the "server" and "client" in this discussion. As usual, the **BoingServer** implements an interface called **BoingRemote**. The server has an **XObjectServer** object. The client has its own **XObjectServer** object.

Here is your challenge...

1. Add a message(s) to **BoingRemote** so that your setup code can create the following situation: The client obtains an **XObjectRemote** stub for the **XObjectServer** in the server. Likewise, the server obtains an **XObjectRemote** stub for the **XObjectServer** in the client.
2. When the server receives a message on its **XObjectServer**, it should turn around and send that message right back to the client using the **XObjectRemote** obtained in step(1) above.
3. Once everything is set up, the client should send the string "hello" on the **XObjectRemote()** obtained in step (1). A moment later, the client should get the "hello" back on its **XObjectServer** (boing!). The client should print out whatever it receives on its **XObjectServer**.

The routine code is already written — you need to fill in the key areas. You may omit all code having to do with exceptions.. In the starter code, the variable "serverXO" is associated with **XObjectServer** on the server side. The variable

"clientXO" is associated with the XObjectServer on the client side. You may use another naming convention if you prefer.

a) BoingRemote

```
// BoingRemote.java
public interface BoingRemote extends java.rmi.Remote {
```

```
    public static final String SERVICE = "boing";
}
```

b) BoingServer

```
public class BoingServer extends UnicastRemoteObject
    implements BoingRemote {
    // routine server setup
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        String name = "//localhost:" + args[0] +
            "/" + BoingRemote.SERVICE;

        BoingRemote impl = new BoingServer();
        Naming.rebind(name, impl);
    }

    XObjectRemote clientXO;
    XObjectServer serverXO;

    // YOUR METHODS ETC. HERE
```

c) BoingClient

```
// BoingClient.java
public class BoingClient {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        String name = "/" + args[0] + "/" + BoingRemote.SERVICE;

        // Get the stub for the server object
        BoingRemote server = (BoingRemote) Naming.lookup(name);

        XObjectServer clientXO;
        XObjectRemote serverXO;

        // YOUR CODE HERE...
```

