Guidelines

Put your name on each page before starting the exam. Write your answers directly on the exam sheets, using the back of the page as necessary. If you finish with more than 15 minutes left in the class, then bring your exam to the front when you are finished and leave the class as quietly as possible. Otherwise, please stay in your seat until the end.

If you have a question, raise your hand and I will come to you. Note, that I am unlikely to answer general questions however. 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.

NOTE: There are quite a few questions to answer. Get started right away and budget your time wisely.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
1. Short answers (16 points). Give very short (1 to 2 sentences for each issue) answers to the following questions. **Longer responses to these questions will not be read.**

   (a) 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.

   (b) How is the value of the codebase property used by a Java JVM?

   (c) In class we discussed several ways to deal with state-dependent actions. One way is called guarding and another is called retrying. What is a state-dependent action? Briefly compare and contrast these two approaches.

   (d) When writing an RMI program why would you choose to make a variable Serializable? Why would you have it implement a Remote interface?
2. State-dependent Actions (14 points). Consider the following program. The retryUntilSuccess() method attempts to create a Resource. If it fails it should continue to retry after specific, increasingly longer intervals. Given the code below, it is possible that the delayTime can become larger than it should be. (a) Explain how this could happen? (b) Rewrite the catch block in retryUntilSuccess() so that the retry behavior works as desired.

```java
class ClientUsingResource {
    String server = "machineX";
    int portnumber = 4000;

    Resource retryUntilSuccess() {
        long delayTime = 1000 + (long)(Math.random() * 1000);
        for (;;) {
            try {
                return new Resource(); // throws Exception on failure
            } catch (Exception ex) {
                Thread.sleep(delayTime);
                delayTime = delayTime * 3/2 + 1; // increase 50%
            }
        }
    }

    static public void main (String [] args) {
        ClientUsingResource c = new ClientUsingResource();
        c.retryUntilSuccess();
    }
}
```
Resource retryUntilSuccess() {
    long delayTime = 1000 + (long)(Math.random() * 1000);
    long timeleft = 0, start = 0, elapsed = 0;
    for (;;) {
        try {
            return new Resource(); // throws Exception on failure
        } catch (Exception ex) {
            try {
                start = System.currentTimeMillis();
                timeleft = delayTime;
                while (timeleft > 0) {
                    Thread.sleep(timeleft);
                    elapsed = System.currentTimeMillis() - start;
                    timeleft = timeleft - elapsed;
                }
            } catch (InterruptedException e) {
                elapsed = System.currentTimeMillis() - start;
                timeleft = timeleft - elapsed;
            }
        }
        delayTime = delayTime * 3/2 + 1; // increase 50%
        timeleft = delayTime;
    }
}
3. Multi-threading (10 points). Construct a short Java program (and not the one in the next problem!) that is guaranteed to 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.

```java
class WillDeadLock {
    synchronized void order1 (WillDeadLock other) {
        other.order2(this);
    }
    synchronized void order2 (WillDeadLock other) {
    }
}
public static void main (String [] args) {
    final WillDeadLock o1 = new WillDeadLock();
    final WillDeadLock o2 = new WillDeadLock();
    (new Thread ( new Runnable () { public void run () { o1.order1(o2);}})).start();
    (new Thread ( new Runnable () { public void run () { o2.order1(o1);}})).start();
}
```
4. Deadlock (10 points). The following code defines two classes, Taxi and Dispatcher. Suppose an application has \( n \) Taxis and 1 Dispatcher. Provide a wait graph showing how this code can deadlock.

```java
class Taxi {
    private Point location, destination;
    private final Dispatcher dispatcher;

    public Taxi(Dispatcher dispatcher) {this.dispatcher = dispatcher;}

    public synchronized Point getLocation() {return location;}

    public synchronized void setLocation(Point location) {
        this.location = location;
        if (location.equals(destination)) dispatcher.notifyAvailable(this);
    }
}

class Dispatcher {
    private final Set<Taxi> taxis;
    private final Set<Taxi> availableTaxis;

    public Dispatcher() {
        taxis = new HashSet<Taxi>();
        availableTaxis = new HashSet<Taxi>();
    }

    public synchronized void notifyAvailable(Taxi taxi) {
        availableTaxis.add(taxi);
    }

    public synchronized Image getImage() {
        Image image = new Image();
        for (Taxi t : taxis)
            image.draw(t.getLocation());
        return image;
    }
}
```

5. Multi-Threading (20 points). In class we talked about semaphores. For our purposes a semaphore allows \( n \) threads into a critical section of code. The following implementation of a semaphore is inadequate because it allows barging. (a) What is barging? (b) Show a trace in which barging occurs.

```java
public class CountingSemaphore {
    private int value = 0;

    public CountingSemaphore(int initial) {
        if (initial > 0) value = initial;
    }

    public synchronized void P() throws InterruptedException {
        while (value == 0) wait();
        value--;
    }

    public synchronized void V() {
        if (value == 0) notify();
        value++;
    }
}
```

(a) Barging occurs when a non-waiting thread enters a critical section before another thread that was already waiting.

(b) 
\[
\text{value} = 1 \\
t1 \text{ calls } P() \\
t2 \text{ calls } P() \text{ and waits} \\
t1 \text{ calls } V() - \text{calls } notify \text{ and sets } value \text{ to } 1 \\
t1 \text{ calls } P() - \text{barges ahead of } t2 \text{ which is still waiting}
\]
6. Java RMI (15 points).

(a) Given the following Java Remote interface, write a class called ExpenseConverterImpl that implements it. Note that convert() is a method to convert expenses in Currency A to an equivalent in Currency B. Assume a conversion rate of 1.5.

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ExpenseConverter extends Remote {
    int convert(int amount) throws RemoteException;
}
```

```java
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.server;

public class ExpenseConverterImpl extends UnicastRemoteObject implements ExpenseConverter {
    public ExpenseConverterImpl() throws RemoteException {
    }
    public int convert(int amount) throws RemoteException {
        return amount * 3/2;
    }
}
```
(b) Given the following Java RMI client code, which depends on the interface defined above, write the corresponding server code.

```java
import java.rmi.*;
import java.net.*;

public class CurrencyConverterClient {
    public static void main(String[] args) {
        try {
            CurrencyConverter c = (CurrencyConverter) Naming.lookup("rmi://hostX/CurrencyService");
            System.out.println(c.convert(Integer.parseInt(args[0])));
        } catch (Exception e) {} 
    }
}
```

```java
import java.io.*;
import java.rmi.*;
import java.rmi.registry.*;

public class CurrencyConverterServer {
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        String name = "rmi://localhost/CurrencyService";
        try {
            java.rmi.registry.LocateRegistry.createRegistry(1099);
            CurrencyConverter cc = new CurrencyConverterImpl();
            Naming.rebind(name, cc);
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```
7. Test Coverage (15 points). Consider the code below. In class we discussed three different kinds of coverage criteria: statement, branch and condition. Give a set of test cases (values for x, y and z) that achieves: (a) 100% statement coverage, but less than 100% branch and condition coverage, (b) 100% branch coverage but less than 100% condition coverage, and (c) 100% condition coverage.

class CoverageTest {
    
    static public void main (String [] args) {
        int x = Integer.parseInt(args[0]);
        int y = Integer.parseInt(args[1]);
        int z = Integer.parseInt(args[2]);

        if (z==2 || x > 1) {
            System.out.println("first statement");
            z = z / x;
        } else {
            System.out.println("second statement");
            x = z - 1;
            if (x > 1 && y < 0) {
                System.out.println("third statement");
                z = z + z;
            }
            System.out.println("fourth statement");
            z = z^2;
        }
    }
}