CS 102 Intro to Computer Science II **Fall 2001**

Exam II

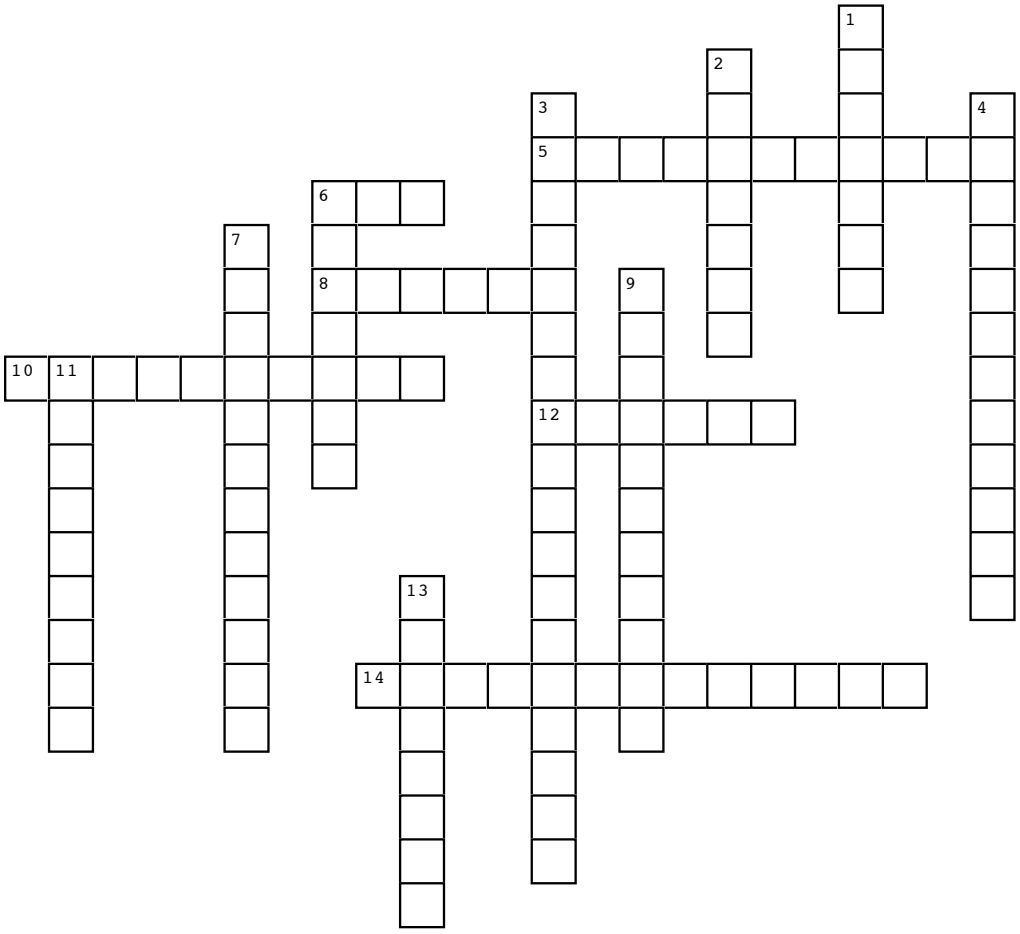
Given: 13 December 2001 Due: 12:30 PM

Name:
Lab Section:

- This exam is open book and open notes.
- Please check that you have all pages, numbered 1 through 19. Write your name on each piece of paper if you are concerned that the pages might become separated.
- Write your answers concisely and legibly *directly on this exam*. Do not use extra sheets of paper.
- Do your own work. No discussion or collaboration with other students is permitted.
- If a question is unclear to you, please raise your hand and somebody will come to help you.
- The exam is divided into parts as described below. By each section heading, an estimate of the time required to complete that section is provided to help you pace yourself. If you get stuck on a question, don't spend too much time on it. Go on to the next question and the answer may occur to you later.
- Partial credit will be given where appropriate. If you see how to approach a problem but don't see the final answer, be sure to at least write down your approach.

Section	**Score**	**Possible**
1. Definitions		10
2. Java Event Model and Inner Classes		10
3. Exceptions		10
4. Parsing		10
5. Concurrency I		15
7. Protection		10
7. Concurrency II		20
8. Network Programming		15
TOTAL		100

1. Definitions (10 minutes – 10 points)



ACROSS

5) C++ has multiple _____ while Java has only single _____

6) acronym for screen-based entry and display

8) provides bidirectional streams for network programs

10) mechanism for discovering and invoking a class's methods at runtime

12) An independent locus of execution, offers run() method

14) can exist when threads perform unsychronized updates to a shared variable

DOWN

1) package-level access

2) extracting and processing tokens from a text stream

3) computational model with a fixed number of states and labelled transitions

4) Java type, offers accept() to create a socket

6) One of Java's inventors

7) keyword in Java, only one thread at a time has access

9) only the "+" operator has this in Java; it is abundant in C++

11) Mechanism for responding to unexpected situations

13) processes A and B each have exlusive access to what the other one needs

2. Java Event Model and Inner Classes (10 minutes – 10 points)

In Othello, you were given the following fragment of code:

```
for (int i=0; i < 8; ++i) {
  for (int j=0; j < 8; ++j) {
    board[i][j] = new BoardSquare(i*8 + j, /* encode location */
                                  blackPiece, whitePiece);

    board[i][j].addMouseListener(new BSListener(this, board[i][j]));
  }
}
```

The doubly-nested loop constructs an 8×8 gameboard, with each square represented by a `BoardSquare` object. A "mouse listener" is associated with each square using the following `BSListener` class:

```
class BSListener extends MouseAdapter {
  private final Client client;
  private final BoardSquare bs;

  BSListener(Client client, BoardSquare bs) {
    this.client = client;
    this.bs = bs;
  }

  public void mouseReleased(MouseEvent e) {
    client.clickedSquare(bs);
  }
}
```

Let's consider replacing the `BSListener` class with an anonymous, inner class, whose behavior should be the same as before. In other words, the fragment of code is replaced by the following:

```
for (int i=0; i < 8; ++i) {
  for (int j=0; j < 8; ++j) {
    board[i][j] = new BoardSquare(i*8 + j, /* encode location */
                          blackPiece, whitePiece);

    board[i][j].addMouseListener(
      new MouseAdapter() { // fill in below









                          }
                                 );
  }
}
```

(a) (5 points) A direct substitution of values from the first code fragment results in references to the local variables `i` and `j` in the inner class, but this will not work—why?

(b) (5 points) Recalling that a `MouseEvent` has the method `getComponent()` to return the object on which the event was generated, fill in the above code to complete the anonymous, inner class.

3. Exceptions (10 minutes – 10 points)

```
public class IntException extends Exception {
   private int num;
   public IntException(int num) { this.num = num; }
   public int getNum() { return num; }
}
```

In this problem, you will write a recursive factorial method that *uses no return statements*. Instead, the answer is computed by throwing exceptions. Once your solution is done, the following code

```
try {
   factorial(5);
}
catch(IntException e) {
   System.out.println("Answer is " + e.getNum());
}
```

prints out "`Answer is 120`".

Continued on next page. . .

Fill in the following code to accomplish this task. Your solution must be recursive, not iterative, and you must not use `return`.

```
public static void factorial(int n) throws IntException {

}
```

4. Parsing (10 minutes – 10 points)

Suppose that the state of an 8×8 Othello board is represented as a string s of 64 tokens drawn from the following alphabet:

X indicates that a square is occupied by a black piece.

O indicates that a square is occupied by a white piece.

b indicates that a square is blank—unoccupied by any piece.

Tokens are separated by one or more spaces, and the tokens are presented left-to-right, top-to-bottom, as one would read a book.

Below, write code that uses `StringTokenizer` to process the string s and make the appropriate call to each `board[i][j]` square, using the methods `setBlack()`, `setWhite()`, and `setEmpty()` as in Lab 4.

5. Concurrency I (15 minutes – 15 points)

Consider the implementation of a class that should give out a stream of integers in a thread-safe manner. This class is part of the `seq` package that is used in Problem 7.

```
package seq;

public final class Stamper {

  private int stamp;

  public Stamper() {
    stamp = 0;
  }

  private void setStamp(int val) { stamp = val; }
  private int  getStamp() { return stamp; }

  public int next() {
    int curVal = getStamp();
    int nextVal = curVal + 1;
    setStamp(nextVal);
    return getStamp();
  }

}
```

For each statement below, explain in detail why the statement is true or false; provide specific examples to demonstrate your claims.

(a) (5 points) The `Stamper` class contains one or more race conditions.

(b) (5 points) All methods shown, except the constructor should have the `synchronized` attribute attached to them.

(c) (5 points) If all methods except the constructor have the `synchronized` attribute attached to them, then use of the `Stamper` class by two different threads could result in deadlock.

6. Protection (10 minutes – 10 points)

Note: You may find this question easier to answer after doing Problem 7, but you should probably read through the code before proceeding on.

Consider the construction of a `seq` package, which consists of the `Stamper` class given in Problem 5 as well as the classes given in this problem. The classes are shown below, but the methods' protection is unspecified. Fill in the appropriate modifiers (including protection, final, synchronization, etc) using the word "default" where no protection should be specified (so we can tell the difference between no response and default protection). Some usage notes:

- The `Resource` class will be subclassed, but methods within `Resource` should not be redefined.
- Subclasses of `Resource` will likely be outside the `seq` package.
- Access rights to methods should be inferred only from the code provided in this problem, from what you know about Java objects in general, and from the code shown in Problem 5; no other access rights should be inferred. Your solution will be graded based on how appropriately you constrain access.

(a) (5 points)

```
package seq;

public class SThread extends Thread {

  private int resourceLevel;

  ___________________ SThread() {
    super();
    resourceLevel = 0;
  }

  ___________________ int getResourceLevel() {
     return resourceLevel;
  }


  ___________________ boolean okToAcquire(int newLevel) {
    return newLevel >= getResourceLevel();
  }

  ___________________ void setResourceLevel(int newLevel) {
    if (!okToAcquire(newLevel))
      throw new Error(this +
                      " tried to allocate a resource at level " +
            newLevel + " Bad Thread!!");
    else
      resourceLevel = newLevel;
  }

  ___________________ String toString() {
    return "Seq thread, current resource level=" +
      resourceLevel + " " +
      super.toString();
  }
}
```

(b) (5 points)

```
package seq;

public class Resource {

  private static Stamper stamper = new Stamper();
  private int myLevel;

  ____________________  Resource() {
    myLevel = stamper.next();
  }

  ____________________  SThread verifyIsSThread() {
    if (!(Thread.currentThread() instanceof SThread))
      throw new Error("Only an SThread can manipulate reousrces");
    return (SThread) Thread.currentThread();
  }

  ____________________  boolean okToLock() {
    SThread s = verifyIsSThread();
    return s.okToAcquire(myLevel);
  }

  ____________________  void acquire() {
    SThread s = verifyIsSThread();
    s.setResourceLevel(myLevel);
  }
}
```

7. Concurrency II (20 minutes – 20 points)

Consider the following two classes: a `Demo` class that spawns two threads that use a `Bumpable` object.

```
public class Demo extends seq.SThread {

  private Bumpable a, b;

  public void run() {
    a = new Bumpable(10); b = new Bumpable(20);

    (new SThread() {
         public void run() {
           System.out.println("Thread 1 " + a.bumpVal(b));
         }
      }).start();

    (new SThread() {
         public void run() {
           System.out.println("Thread 2 " + b.bumpVal(a));
         }
      }).start();
  }
  public static void main(String[] args) {
    (new Demo()).start();
  }
}

public class Bumpable {

    private int total;
    public Bumpable(int n) { super(); total = n; }

    public synchronized int bumpVal(Bumpable b) {
       total += b.getVal();
       return total;
    }

    private synchronized int getVal() {
       return total;
    }

}
```

(a) The classes as presented above can result in deadlock (trust me: I actually ran them, and the deadlock is evident). Explain in detail how deadlock occurs for these classes.

(b) We next explore a mechanism that could catch the fault in the above program, and throw an exception instead of deadlocking. The idea is to avoid deadlock by ensuring that a set of resources is always acquired in a consistent order. We implement this in a Java package called `seq`, which consists of the `Stamper` class shown in Problem 5 as well as the classes `SThread` and `Resource` given in Problem 6, whose behavior is summarized as follows:

- The `SThread` class is a simple extension of `Thread` that can record the highest level of resources obtained so far by that thread.
- To acquire an instance of `Resource` (or it subclasses), the `acquire()` method is called. The code provided will throw an exception if the requesting thread is not an `SThread` or if it is acquring this resource out of order.
- Every time a `Resource` object is instantiated (typically by `super()` in a subclass), it receives a unique, ascending sequence number using a `Stamper` object.
- Developers fit into this picture by subclassing `Resource` outside the `seq` package.

Below, reimplement `Bumpable` so that it extends `Resource` and is deadlock-free thanks to the `seq` package. Your code must not contain any race conditions—be sure to put the `synchronized` keyword on the appropriate methods.

8. Network Programming (15 minutes – 15 points)

The Othello server for Lab 4 was essentially an intermediary in an otherwise peer-to-peer application. In particular, the server for that lab only accepts two connections and is then tied up for the duration of a game—it receives a move, judges the move's validity, updates and broadcasts the gameboard.

The demand for Othello is on the rise! We now want to service a wider community, and allow multiple games to take place concurrently. You are asked to design this new server, according to the following.

- Your new server must be called `OBroker`.
- You must reuse the `OServer` class just as it was given to you for Lab 4. Recall that its constructor is:

  ```
  public OServer(int port)
  ```

  which takes in the port at which this `OServer` should wait for exactly two clients to connect to play a game.
- The `OBroker` class must continually accept connections, matching each consecutive pair to play a game. That game should be conducted by instantiating an appropriate `OServer` object. The `OBroker` object *must not wait* for any game to finish—it must continue to match clients into games while games are being played.
- You may assume that ports 5000 and up are available for your use on the server.
- The client code will change slightly, but you have control of those changes. You'll be asked about this below.
- Beware race conditions!

(a) (5 points) Below, write the code for `OBroker` class

(b) (5 points) Below, describe any changes you require of the Client behavior from that specified for Lab 4.

(c) (2 points) If an adversary wanted to foil your `OBroker` and `OServer` system, how could he or she accomplish this?

(d) (3 points) What steps could you take to resolve any problems you listed in part 8c.