

# Final Exam

CMSC 433  
Programming Language Technologies and Paradigms  
Fall 2003

December 16, 2003

## 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. Bring your exam to the front when you are finished. Please be as quiet as possible.

If you have a question, raise your hand and I will come to you. However, to minimize interruptions, you may only do this once for the entire exam. Therefore, wait until you are sure you don't have any more questions before raising your hand. Errors on the exam will be posted on the board as they are discovered. 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.

Question	Points	Score
1	30	
2	30	
3	30	
4	10	
Total	100	

1. Short Answers (5 points each, 30 total). Give very short (1 to 2 sentences) answers to the following questions. **Longer responses will not be read.**

- (a) Ousterhaut argues that programmers should use event-based, rather than thread-based programming approaches, much more frequently than they do now. One reason is that it's too hard to write correct threaded programs. Give 2 specific and distinct examples of why this is true.

**Answer:**

*Any 2 of: synchronization, deadlock, hard to debug, threads break abstraction, or callbacks can deadlock.*

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

**Answer:**

*The codebase indicates where to find class files for references that leave this JVM.*

- (c) Java locks serve 3 functions. List and briefly explain each one. **Answer:**

*Atomicity. Allows operation to be done without interference from other threads. Visibility. Signals when values will be updated across threads Ordering. Allows programmers to ensure that an operation A occurs before operation B*

- (d) One principle of design patterns is “*program to an interface, not to an implementation*”. Specifically, how is this realized in the Abstract Factory design pattern.

**Answer:**

*The Abstract Factory pattern defines two kinds of interfaces: one kind is interfaces for each product (e.g, buttons, menus, dialog boxes) and the other is an interface for factories.*

*The product interface is implemented by each kind of concrete product.*

*The factory interface is implemented by concrete factories that ensure that only products of a certain type are created.*

*Clients are expected to use the interfaces in their code.*

- (e) In class we talked about several ways to deal with state dependent actions. What are state-dependent actions? Describe 2 different ways to deal with state-dependent actions.

**Answer:**

*A state-dependent action is an action that can be taken only when the program is in a particular state.*

*Ways to deal with state-dependent actions include*

- i. balking*
- ii. trying*
- iii. retrying*
- iv. guarding*
- v. timing out*
- vi. planning*

- (f) Non-blocking I/O. In our discussion of scalable server architectures we compared architectures based on both blocking and non-blocking I/O. The non-blocking architectures relied on selectable java channels. What is non-blocking I/O? What is a selector? And, conceptually, how do these two concepts interact to improve performance of large-scale servers when compared to an architecture based on blocking I/O and thread-per-message message handling.

**Answer:**

*Non-blocking I/O methods return immediately rather than waiting for data when an I/O stream is not ready.*

*Selectors monitor a set of channels for a specific set of registered events.*

*These two classes improve performance because*

- 1) Selectors can efficiently monitor many channels in a single thread*
- 2) Selectors block only until at least one channel is ready to be used. This avoids busy waiting and maximizes availability.*

2. Java Concurrency (30 points). Consider the following program.

```
class Q {

    int n;    boolean valueset = false;

    synchronized int get() {
        while (!valueset) {
            try {wait();}
            catch (InterruptedException e) {System.out.println(e);}
        }
        System.out.println("Got: " + n); valueset = false; notify(); return n;
    }
    synchronized void put(int n) {
        while (valueset) {
            try {wait();}
            catch (InterruptedException e) {System.out.println(e);}
        }
        this.n = n; valueset = true; System.out.println("Put: " + n); notify();
    }
}

class Producer implements Runnable {
    Q q;

    Producer(Q q) {this.q = q; new Thread(this).start();}
    public void run() {
        int i = 0; while (true) {q.put(i++);}
    }
}

class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {this.q = q; new Thread(this).start();}
    public void run() {while (true) {q.get();}
    }
}

class PCFixed {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q); // Thread p1
        new Producer(q); // Thread p2
        new Consumer(q); // Thread c1
    }
}
```

Although this program is intended to run forever, on one run this program produced exactly the following output.

```
Put: 0
Got: 0
Put: 0
Got: 0
Put: 1
```

- (a) Assuming that Thread p1 was the first Thread to put an item on the queue, provide the **smallest** program trace you can that is consistent with the above output. Needless long traces will be severely penalized.

In describing your program trace, create a table like the one below, showing the steps executed by each thread. A step will be defined as an entry to Q.put() (written EP), an exit from Q.put() (written XP), an entry to Q.get() (written EG), an exit from Q.get() (written XG), an entry to wait() (written EW), an exit from wait() (written XW), acquiring a lock on q (written AL), and releasing a lock on q (written RL). (The steps are interleaved, so only one thread should be taking a step on each line in the table.)

Notes: Assume that thread execution is interleaved and that context switching occurs only when the current thread is blocked. Threads will enter synchronized methods before attempting to acquire a lock and will release the lock before exiting.

step	P1	P2	C1
1			
2			
..			

- (b) Briefly describe how you might avoid this problem?

**Answer:**

```

EP Thread[Thread-1,5,main]
AL Thread[Thread-1,5,main]
  Put: 0
RL Thread[Thread-1,5,main]
XP Thread[Thread-1,5,main]
EP Thread[Thread-1,5,main]

EG Thread[Thread-3,5,main]
AL Thread[Thread-3,5,main]
  Got: 0
RL Thread[Thread-3,5,main]
XG Thread[Thread-3,5,main]
EG Thread[Thread-3,5,main]

EP Thread[Thread-2,5,main]
AL Thread[Thread-2,5,main]
  Put: 0
RL Thread[Thread-2,5,main]
XP Thread[Thread-2,5,main]
EP Thread[Thread-2,5,main]

AL Thread[Thread-3,5,main]
  Got: 0
RL Thread[Thread-3,5,main]
XG Thread[Thread-3,5,main]
EG Thread[Thread-3,5,main]

AL Thread[Thread-3,5,main]
EW Thread[Thread-3,5,main]
RL Thread[Thread-3,5,main]

AL Thread[Thread-2,5,main]
  Put: 1
RL Thread[Thread-2,5,main]
XP Thread[Thread-2,5,main]
EP Thread[Thread-2,5,main]
AL Thread[Thread-2,5,main]
EW Thread[Thread-2,5,main]
RL Thread[Thread-2,5,main]

AL Thread[Thread-1,5,main]
EW Thread[Thread-1,5,main]
RL Thread[Thread-1,5,main]

```

You could avoid this problem by using `notifyAll()` rather than `notify()`.

3. Java Concurrency (30 points) Some executions of the following program result in deadlock. Read the code carefully to determine how deadlock could occur. Then answer the questions that follow.

```
public class CanResultInDeadlock {
    public static void main(String args[]) {
        Counter a = new Counter(2);
        Counter b = new Counter(2);
        a.setNextCounter(b);
        b.setNextCounter(a);
        Thread firstThread = new CountdownThread(a);
        Thread secondThread = new CountdownThread(b);
        firstThread.start();
        secondThread.start();
    }
}

class Counter {
    int startValue;        // value to start counting down from
    int countDown;        // current value in the count down
    Counter nextCounter;  // the counter to be reset upon reaching zero

    Counter(int startValue) {
        this.startValue = startValue;
        reset();
    }

    void setNextCounter(Counter next) { nextCounter = next; }

    synchronized void reset() { countDown = startValue; }

    synchronized boolean isZero() { return (countDown == 0); }

    synchronized void decrement() {
        if (countDown > 0) {
            countDown--;
            System.out.println("Countdown: " + countDown + " " + this);
            if (isZero())
                nextCounter.reset();
        }
    }
}

class CountdownThread extends Thread {
    Counter counter;

    CountdownThread(Counter counter) { this.counter = counter; }

    public void run() {
        while (true)
            counter.decrement();
    }
}
```

- (a) Show how deadlock is possible in the above program by providing the interleaving of steps for one example of a deadlocking execution. In describing your program trace, create a table like the one below, showing the steps executed by each thread. A step will be defined as an entry to counter.decrement() (written ED), an exit from counter.decrement() (written XD), an entry to counter.reset()(written ER), an exit from counter.reset()(written XR), acquiring a lock on counter (written AL), and releasing a lock on counter (written RL). (The steps are interleaved, so only one thread should be taking a step on each line in the table.)

step	firstThread	secondThread
..		

- (b) Draw a wait graph that illustrates the deadlock situation resulting from your execution. By each arrow in the graph, write the line number from the table above that resulted in that arrow.

**Answer:**

```
ER Thread[main,5,main] AL Thread[main,5,main] RL Thread[main,5,main] XR Thread[main,5,main]
ER Thread[main,5,main] AL Thread[main,5,main] RL Thread[main,5,main] XR Thread[main,5,main]
ED Thread[Thread-1,5,main] AL Thread[Thread-1,5,main] Countdown: 1 Counter@13f5d07
RL Thread[Thread-1,5,main] XD Thread[Thread-1,5,main] ED Thread[Thread-1,5,main]
AL Thread[Thread-1,5,main] Countdown: 0 Counter@13f5d07 ED Thread[Thread-2,5,main]
AL Thread[Thread-2,5,main] Countdown: 1 Counter@f4a24a RL Thread[Thread-2,5,main]
XD Thread[Thread-2,5,main] ED Thread[Thread-2,5,main] AL Thread[Thread-2,5,main]
Countdown: 0 Counter@f4a24a
```

4. (Junit 10 points). Examine the following java code. Then fill in the missing Junit test code in the ShoppingCartTest class. Be as succinct as possible. Overly long responses will be severely penalized.

```
public class ProductNotFoundException extends Exception {
    public ProductNotFoundException() {super();}
}

public class Product {
    private String _title;    private double _price;

    public Product(String title, double price) {
        _title = title;    _price = price;
    }

    public String getTitle() {return _title;}

    public double getPrice() {return _price;}

    public boolean equals(Object o) {
        if (o instanceof Product) {
            Product p = (Product)o;
            return p.getTitle().equals(_title);
        }
        return false;
    }
}

public class ShoppingCart {
    private ArrayList _items;

    public ShoppingCart() {_items = new ArrayList();}

    public double getBalance() {
        Iterator i = _items.iterator();
        double balance = 0.00;
        while (i.hasNext()) {
            Product p = (Product)i.next();
            balance = balance + p.getPrice();
        }
        return balance;
    }

    public void addItem(Product p) {_items.add(p);}

    public void removeItem(Product p) throws ProductNotFoundException {
        if (!_items.remove(p)) {throw new ProductNotFoundException();}
    }

    public int getItemCount() {return _items.size();}

    public void empty() {_items = new ArrayList();}

    public boolean isEmpty() {return (_items.size() == 0);}
}
```

```

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class ShoppingCartTest extends TestCase {
// DEFINE TEST VARIABLES HERE

    public ShoppingCartTest(String name) {super(name);}

    protected void setUp() {
// CREATE ALL TEST OBJECTS HERE

    }
    protected void tearDown() {
// TEAR DOWN HERE

    }

    public void testProductAdd() {
// TEST THAT 1 NEW BOOK IS ADDED TO SHOPPINGCART PROPERLY

    }

    public void testEmpty() {
// TEST THAT A SHOPPINGCART IS EMPTY

    }

    public void testProductRemove() throws ProductNotFoundException {
// TEST THAT 1 BOOK IS REMOVED FROM SHOPPINGCART PROPERLY

    }

    public void testProductNotFound() {
// TEST THAT REMOVING A BOOK NOT IN THE SHOPPINGCART IS HANDLED PROPERLY

    }
}

```

```

public static Test suite() {
    TestSuite suite = new TestSuite(ShoppingCartTest.class);
    return suite;
}

public static void main(String args[]) {
    junit.textui.TestRunner.run(suite());
}
}

```

**Answer:**

```

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class ShoppingCartTest extends TestCase {

    private ShoppingCart _bookCart;
    private Product _defaultBook1;
    private Product _defaultbook2;
    public ShoppingCartTest(String name) {
        super(name);
    }

    protected void setUp() {
        _bookCart = new ShoppingCart();
        _defaultBook1 = new Product("Extreme Programming", 23.95);
        _bookCart.addItem(_defaultBook);
        _defaultbook2 = new Product("Refactoring", 53.95);
    }

    protected void tearDown() {
        _bookCart = null;
    }

    public void testProductAdd() {

        _bookCart.addItem(defaultBook2);
        double expectedBalance = _defaultBook.getPrice() + defaultBook2.getPrice();
        assertEquals(expectedBalance, _bookCart.getBalance(), 0.0);
        assertEquals(2, _bookCart.getItemCount());
    }

    public void testEmpty() {
        _bookCart.empty();
        assertTrue(_bookCart.isEmpty());
    }

    public void testProductRemove() throws ProductNotFoundException {
        _bookCart.removeItem(_defaultBook);
        assertEquals(0, _bookCart.getItemCount());
        assertEquals(0.0, _bookCart.getBalance(), 0.0);
    }

    public void testProductNotFound() {
        try {

```

```
        _bookCart.removeItem(defaultBook2);
        fail("Should raise a ProductNotFoundException");
    } catch(ProductNotFoundException success) {
    }
}

public static Test suite() {
    TestSuite suite = new TestSuite(ShoppingCartTest.class);
    return suite;
}

public static void main(String args[]) {
    junit.textui.TestRunner.run(suite());
}
}
```