

## Exam I

Given: 11 March 2003

Due: end of exam period

Name:

Lab Section:

- This exam is open book and open notes.
- Please check that you have all pages, numbered 1 through 13. Write your name on each piece of paper, in case the pages 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 waste 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.

<b>Section</b>	<b>Score</b>	<b>Possible</b>
1. Definitions		15
2. Inner classes		15
3. Design		15
4. Hash functions		15
5. Race conditions		20
6. Exclusion		20
TOTAL		100

1. Definitions (5 minutes – 10 points)

## 2. Inner classes and concurrency (5 minutes – 15 points)

For each of the following, state whether the statement is true or false, and explain why.

- (a) (3 points) An inner class cannot access its outer class's local variables unless they are `final`.

true       false

Java does not grant access to local variables outside of their declaring method. The values of local variables can be transmitted into an inner class, but that class has no way to change their value. Thus, Java insists that they be `final`.

- (b) (3 points) An inner class cannot access its outer class's instance variables unless they are `final` and are not `private`.

true       false

The inner class is packaged as a separate outer class. Accessors and mutators are inserted to give the "inner" class access to the outer class's instance variables. They need not be `final` nor `private`.

- (c) (3 points) The only way to make loads and stores of 64-bit values (`long` and `double`) atomic is to declare them `volatile`.

true       false

Ordinary synchronization would also force the values to be written and read atomically. Thus, another way to make these loads and stores atomic is to lock some shadow, `Object` object to cover every load and store of the value. It's not convenient, so `volatile` is the method of choice.

- (d) (3 points) It is impossible to obtain a lock on an instance of an anonymous inner class.

true       false

```
Object o = new Whatever() {
    /* an inner class */
}
synchronized(o)
```

Consider:

But Note, Graders: I misworded the question, so another possibility is that is not possible, since the class type is unknown. So if somebody says `false` and gives this reason, that's fine with me

- (e) (3 points) Locks need never be obtained on an immutable class, if that class references no `static` variables.

true       false

The class is thread-safe on construction and with no mutators, there is no way any thread could write the instance variables in the class.

### 3. Design and Style (10 minutes – 15 points)

Below I have defined an object that is intended to represent *rational* numbers—numbers that can be expressed as  $num/denom$  where  $num$  and  $denom$  are integers. If I submitted the following code to your CS 102 TAs, it should come back with the point deductions shown in comments below.

```
public class Rational {
    public int num, denom;    // -3 pts, confinement failure
    protected int num, denom // or they could use private

    public Rational(int num, int denom) {
        this.num = num;      // -4 pts, failure to use reduction
        this.denom = denom;  //

        set(num, denom);
    }

    protected void set(int num, int denom) {

        if (denom == 0) throw new Error("Divide by zero");

        this.denom = denom;  // -4 pts, failure to detect exception
        this.num = num;
    }

    public void reciprocate() { // -4 pts, failure to use reduction
        int oldDenom = this.denom;
        this.denom = num;
        this.num = oldDenom;

        set(denom, num);
    }

    public double toDouble() { return ((double) num / denom); }
    public String toString() { return num + "/" + denom; }
}
```

Coincidentally, I left space in the object definition so that you can make changes to the code to satisfy the TAs. By making the appropriate changes above, you earn that many points for this problem.

#### 4. Maps, Sets, Hash Functions (10 minutes – 15 points)

Suppose we want to collect `Rational` objects in `Map` or `Set` implementations, in particular those that use hash functions to locate a `Rational`.

- (a) (5 points) Given the default implementation of `hashCode()` and `equals(Object)`, what harm can come of the following?

```
Set s = new HashSet();
s.add(new Rational(1,2));
s.add(new Rational(1,2));
```

The two `Rational` objects will each be viewed as distinct, because default `hashCode` is based on an object's address in memory. The `Set` which should have just once instance of the fraction would then have two instances.

- (b) (10 points) Below, fill in implementations for `Rational`'s methods that support hash-function-based collection objects.

```
public int hashCode() {

    // Give them credit for any reasonable implementation
    // including: return 0

    Best is:

    return num + denom;    // or some commutative function of num and denom
                          // why? So it can still be found after reciprocal

}

public boolean equals(Object o) {

    Rational r = (Rational) o;    // you can assume this works

    return (this.num == r.num && this.denom == r.denom); // or similar

}
```

## 5. Race Conditions (15 minutes – 20 points)

The code shown below uses the `Rational` object defined in Problem 3.

```
Rational half = new Rational(1, 2);

// Start of Stuff
half.reciprocate(); // thing 1
half.reciprocate(); // thing 2
// End of Stuff

System.out.println("Value is " + half);
```

- (a) (4 points) For a program containing just one thread, what output do you expect the above code to produce?

Value is 1/2

- (b) (6 points) Below, use anonymous inner classes to rewrite the two lines of code shown between the `Stuff` comments, so that *each* line of code runs in a thread separate from the thread that instantiated the object. That is, one thread should do `thing 1`, another thread should do `thing 2`, and neither of those threads should be the thread that did `new Rational(1, 2)`.

Your code should be as elegant as possible. Be sure that you cause the threads to run!

```
// Note: half needs to be final
new Thread() {
    public void run() {
        half.reciprocate(); // thing 1
    }
}.start();

new Thread() {
    public void run() {
        half.reciprocate(); // thing 2
    }
}.start();
```

- (c) (5 points) Suppose you ran your multithreaded program, and the output was as follows:

```
Value is 1/1
```

Describe *precisely* the sequence of events that resulted in that output.

Either in the `set` method or in the inline code, one thread got as far as setting `denom = num` but then was interrupted. At that point, the code that spawned the two threads proceeded to print the object. Since `num` and `denom` temporarily have the same value, that's why the out was as shown

- (d) (5 points) On the next page is another copy of the `Rational` object. Mark it up to show how you would modify it to avoid the problem present in the multithreaded program. Your solution here will be graded on its elegance (as few changes as possible), its correctness, and its liveness properties.

They must use `synchronized` to ensure that the update is done atomically. This could be done by specifying `synchronized` on the `reciprocal` method, but a more general approach would put it on the `set` method if they used reduction to call `set` from `reciprocal`. Just make sure that they have a lock on the object when the numerator and denominator are swapped.

## 6. Exclusion: Multiple Objects (15 minutes – 20 points)

```
public class MoreRational extends Rational {  
  
    // This class extends your thread-safe version of Rational  
    //  
    public MoreRational(int num, int denom) { super(num, denom); }  
  
    public void mpy(Rational other) {  
        set(this.num * other.num, this.denom * other.denom);  
    }  
}
```

- (a) (5 points) Even with your thread-safe changes to `Rational`, the `mpy` method above contains a race condition. Sketch an example that illustrates the race condition.

```
Initially, r1 = new Rational(1,2),  
          r2 = new Rational(1,2)
```

```
Thread 1:                Thread 2:  
  
    r1.mpy(r2);          r2.reciprocal();
```

```
Result could be r1 = 1/2  Instead of either 1/4 or 1/1  
This could happen because Thread 2 is halfway through doing  
the reciprocal operation, and num = denom = 1.  
At that moment, Thread 2 carries out its mpy operation, thinking  
r2 = 1/1  
and obtaining r1 = 1/2
```

- (b) (5 points) Would adding synchronized accessors `getNum()` and `getDenom()` help to eliminate the race condition? Why or why not?

No, because both values need to be obtained “at once”—atomically.

(c) (5 points) Below, rewrite `mpy` to eliminate the race condition by locking multiple objects. Your answer must

- Be as live as possible
- Be totally safe
- Not allow deadlock to happen

For the purposes of this problem on this exam, you are *not* allowed to instantiate any objects inside `mpy`.

```
public void mpy(Rational other) {
```

Their answer needs to have a lock on both objects.

Making `mpy` synchronized DOES NOT work, because if two objects try to `mpy` each other, deadlock will result.

Also, they can't use a helper method that affects the object, because the result is always supposed to apply to this object. (A static helper would be OK)

So they have to leave `mpy` unsynchronized, but obtain a lock on this object and the other object, in the right order:

```
if (System.identityHashCode(this) < System.identityHashCode(other)) {
    synchronized(this) {
        synchronized(other) {
            // do it
        }
    }
}
else {
    synchronized(other) {
        synchronized(this) {
            // do it
        }
    }
}
```

}

- (d) (5 points) The need for obtaining multiple locks can be avoided by using copying instead of obtaining locks. Below, provide Java code changes to `Rational` and/or `MoreRational` to support the copying and avoid locking multiple objects. You needn't copy code below that is given above. Be sure to show the code for `mpy`. The `Rational` class must remain mutable.

Something like:

```
public synchronized Rational copy() { // or use clone
    return new Rational(num, denom);
}
```

and then in `mpy` do:

```
public void mpy(Rational other) { // it's NOT ok to sync !!
    Rational c = other.copy();    // because copy gets a lock too
    synchronized(this) {        // now sync!
        this.num = this.num * c.num;
        this.denom = this.denom * c.denom;
    }
}
```