

Final Exam

CMSC 433
Programming Language Technologies and Paradigms
Fall 2002

December 21, 2002

Guidelines

The exam time is from **8:00 am to 10:00 am**, totalling 2 hours.

This exam has 11 pages; count them to make sure you have all of them. Put your name 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 early, you may bring your exam to the front, but 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.

Use good test-taking strategies. Look over the entire test to see which problems seem easier or harder to you, look at how much each problem is worth, and then decide the order in which to do the problems.

Good luck, and have a great break!

Question	Points	Score
1	20	
2	20	
3	15	
4	15	
5	15	
6	15	
Total	100	

1. (Short Answer, 20 points)

- (a) (2 points) Name a violation of safety, and name one violation of liveness. Alternatively, explain the difference between the two.

Answer:

A violation of safety is a race condition. A violation of liveness is deadlock. Safety is “nothing bad happens” while liveness is “something good eventually happens.”

- (b) (4 points) Fill out the following table:

Answer:

<i>If an object is</i>		<i>Then it</i>	
<i>unshared</i>	<i>immutable</i>	<i>could require synchronization</i>	<i>never needs synchronization</i>
×	×		×
×			×
	×		×
		×	

- (c) (4 points) Two policies for state-dependent actions are *balking* and *guarding* (or *waiting*). Can both be used in both single-threaded and multi-threaded programs? Explain why in two or three sentences.

Answer:

Balking occurs when a method returns immediately, with an exception, when its precondition is not met. Guarding occurs when a method waits until the precondition is met before proceeding. Guarding doesn't make sense for single-threaded programs, since it assumes another thread will come along to change the condition. Balking is sensible for both.

- (d) (4 points) How are the *Singleton* and *Type-safe Enum* design patterns related?

Answer:

Singleton ensures only one object is ever created, while Type-safe Enum generalizes this idea: only n objects are ever created.

- (e) (4 points) Recall from the homework that a thread can be in one of four states: created, runnable, blocked, or terminated. Name two reasons that a thread can be in the *blocked* state.

Answer:

It can be blocked due to sleeping, waiting to acquire a lock, waiting on a lock's wait-set, waiting on I/O, and possibly others.

2. (Concurrency (coding), 20 points)

In Project 3, you wrote a single-threaded webserver. The central dispatch point for the server was the `BasicServer` class. A sample implementation of this class is shown below.

```
public class BasicServer {
    HttpRequestProcessor processor;
    ServerSocket socket;
    public BasicServer(int port, HttpRequestProcessor processor) throws IOException {
        this.processor = processor;
        socket = new ServerSocket(port);
    }
    private void process() throws IOException {
        Socket conn = socket.accept();
        try {
            HttpRequest request = HttpRequest.parseRequest(conn.getInputStream());
            HttpReply reply = new HttpReply();
            StandardHttpRequestEvent event = new StandardHttpRequestEvent(request, reply, conn);
            processor.process(event);
            event.send(conn.getOutputStream());
        } finally { conn.close(); }
    }
    public void go () {
        while(true) {
            try { process(); } catch (IOException e) { System.err.println(e); }
        }
    }
}
```

- (a) (15 points) Write a new class called `ThreadedServer` whose `go` method accepts connections and *starts a new thread* to process each one. You do not need to recopy any `BasicServer` code you reuse. Rather, clearly label code fragments above, and then use the labels in your new classes. Write your answer on the next page.
- (b) (5 points) In addition to making this change, are there any other changes you would have to make to your project for the code to work properly with `ThreadedServer` rather than `BasicServer`? Describe the general flavor of all the changes, using one or two sentences, and then describe one concrete change to your code that you would make.

Answer:

You need to properly synchronize the rest of the program. For example, `HttpCommandProcessor` stores a table of the commands it supports. Accesses to this table would have to be synchronized, since they could now occur in simultaneous threads.

Put your answer for part (a) here. **Answer:**

```
public class ThreadedServer {
    // same constructor and fields as BasicServer
    HttpRequestProcessor processor;
    ServerSocket socket;
    public ThreadedServer(int port, HttpRequestProcessor processor)
        throws IOException {
        this.processor = processor;
        socket = new ServerSocket(port);
    }
    private void process() throws IOException {
        final Socket conn = socket.accept();
        new Thread() {
            public void run() {
                try {
                    try {
                        HttpRequest request = HttpRequest.parseRequest(conn.getInputStream());
                        HttpReply reply = new HttpReply();
                        StandardHttpRequestEvent event =
                            new StandardHttpRequestEvent(request, reply, conn);
                        processor.process(event);
                        event.send(conn.getOutputStream());
                    }
                    finally { conn.close(); }
                } catch (IOException e) { System.err.println(e); }
            }
        }.start();
    }
    // same go() as BasicServer
    public void go () {
        while(true) {
            try { process(); } catch (IOException e) { System.err.println(e); }
        }
    }
}
```

3. (Concurrency, 15 points) The code snippets below *may* allow for concurrency errors, such as race conditions and deadlock. If so, describe the problem in general, and where it occurs in the code. You may wish to include an execution trace that illustrates the problem. For the first two cases, assume that multiple threads may be manipulating the same objects at the same time. The last case shows a trimmed-down version of the Elevator thread from project 4. Here, Person threads could call the `currentFloor` method at any time.

(a) (5 points)

```
1: public class Cell {
2:     private Object _value;
3:     public Cell(Object v) {
4:         _value = v;
5:     }
6:     public synchronized Object get() { return _value; }
7:     public synchronized void set(Object v) { _value = v; }
8:     public synchronized swap(Cell p) {
9:         synchronized (p) {
10:            p.set(get());
11:            set(p.get());
12:        }
13:    }
14: }
```

Answer:

This program could deadlock at line 9. Here's a trace with two threads t_1, t_2 and two Cell objects c_1, c_2 :

t_1 calls $c_1.swap(c_2)$:

t_1 acquires the lock on c_1 , and then context switches

t_2 calls $c_2.swap(c_1)$:

t_2 acquires the lock on c_2

t_2 attempts to acquire lock on c_1 , but cannot, so blocks

t_1 attempts to acquire lock on c_2 , but cannot, so blocks

deadlock

(b) (5 points)

```
1: public class MyInteger {
2:     public static final Object lock = new Object();
3:     private int _value;
4:     public void add(MyInteger n) {
5:         synchronized (lock) {
6:             _value += n._value;
7:         }
8:     }
9: }
10: public class MyNatural {
11:     private int _value;
12:     public void add(MyNatural n) {
13:         synchronized (MyInteger.lock) {
14:             _value += n._value;
15:         }
16:     }
17: }
```

Answer:

There is nothing wrong with this program, though it might be a bit slow since only a single global lock is used.

(c) (5 points)

```
1: public class Elevator extends Thread {
2:     private int floor;
3:     private Direction direction;
4:     private Building building;
5:     public synchronized int currentFloor() {
6:         return floor;
7:     }
8:     public void run() {
9:         while (true) {
10:            if (floor == building.NUM_FLOORS-1)
11:                direction = Direction.down;
12:            else if (floor == 0)
13:                direction = Direction.up;
14:            if (direction == Direction.up)
15:                floor = floor + 1;
16:            else floor = floor - 1;
17:        }
18:    }
19: }
```

Answer:

Access to floor is not properly synchronized, and could result in a visibility violation. In particular, because the changes to floor in run are not synchronized, another thread calling currentFloor may not see the most up-to-date version.

4. (Design Patterns (coding), 15 points)

Some functions are expensive to compute. If we need the result of a function twice, it makes sense to *cache* it, rather than compute the same answer many times. Write a *proxy* for objects that implement the `Callable` interface (shown below) called `CachedCallable`, following the proxy design pattern. This class should keep a `HashMap` that stores old results, indexed by their argument. The constructor for `CachedCallable` should take the `Callable` that it will proxy as its argument.

```
public interface Callable {
    Object execute(Object argument)
}
```

Answer:

```
import java.util.*;
public class CachedCallable implements Callable {
    private Callable c;
    private HashMap map = new HashMap();
    public CachedCallable(Callable c) {
        this.c = c;
    }
    Object execute(Object argument) {
        Object cachedResult = map.get(argument);
        if (cachedResult != null) return cachedResult;
        cachedResult = c.execute(argument);
        map.put(argument, cachedResult);
        return cachedResult;
    }
}
```

5. (RMI and Concurrency (coding), 15 points)

In Project 5, you wrote a peer-to-peer network for the purpose of sharing `Service` objects. Each `Portal` maintains a local map in which it stores `Service` instances, and these may be either local or remote references (i.e., stubs). The simple `Main` class we provided has a loop for executing services as indicated by the user:

```
Portal p = ...
String data = ...
BufferedReader input = ...
while (true) {
    String command = input.readLine();
    Service svc = p.getService (command);
    if (svc == null)
        p.forward (new FindServiceMessage (p.getAddress(), command));
    else {
        try {
            data = svc.transformString (data);
        } catch (RemoteException e) {
            System.out.println (e);
        }
    }
}
```

In this approach, if the service is not present in the local map then `p.getService` returns `null`. This policy requires the user to retype the command in the hope that the service will be present later. We could change `getService` to take an additional `boolean` argument `doWait`. When `getService` is called, it will try to get the service from the local map, but if not present and `doWait` is true, it will send a `FindServiceMessage` as above, and will wait until the service is added to the map. Once this happens, it will extract the service from the map and return it to the user.

Implement this change to `getService` using the template on the next page. Do not use busywaiting!

Answer:

```
public class Portal implements LocalPortal, RemotePortal {
    HashMap serviceMap = Collections.synchronizedMap(new HashMap());

    public Service getService(String serviceName, bool doWait) {
        Service service = (Service) serviceMap.get (serviceName);
        if (service != null) return service;
        forward (new FindServiceMessage(getAddress(), serviceName);
        synchronized (serviceMap) {
            while ((service = serviceMap.get(serviceName)) == null)
                try { serviceMap.wait(); } catch (InterruptedException e) { }
        }
        return service;
    }

    public void addService (String serviceName, Service service) {
        synchronized (serviceMap) {
            serviceMap.put (serviceName, service);
            serviceMap.notifyAll();
        }
    }
}
```

6. (Special Topics, 15 points)

- (a) (5 points) Describe one application of Java *reflection*. Describe why it is a reasonable application; e.g., justify that it could not be done any other way.

Answer:

One application of reflection is serialization. Reflection is necessary because the values of the fields of a given object must be extracted, for all possible objects. Subclassing and inheritance won't help you here.

- (b) (5 points) Swing GUI programs are *event-driven*: each time you create a Component, you register a handler (or *listener*) to process events that are related to that component. For example, you would register a handler to run each time a particular button is clicked. What design pattern does this implement? Why?

Answer:

This implements the observer pattern. In particular, the component acts as the subject, and the handler acts as the observer. Whenever an event occurs to the subject, the handler must be notified, so it can take action.

- (c) (5 points) In three sentences or less, describe the coolest thing you learned in this course.

Answer:

How to be a better programmer!