

Name:

Account:

CMSC 433 Exam 2: April 25, 2002

1. Design Patterns (20 points)

The code below uses the Visitor pattern to perform a traversal over a tree consisting of nodes containing integers, longs, floats, and doubles. The particular NodeVisitor class shown, SumVisitor, separately sums the values of the integer/long nodes and the float/double nodes. Your tasks are to:

- Write a new visitor class, PrintVisitor, that does a traversal over a tree of Nodes, printing each node value to System.out with a space after each value printed. Also append a single letter indicating the type of the value to each number printed (**i** for integer, **l** for long, **f** for float, **d** for double), before the space.
- Add a new Node type, ShortNode, for values of type short. shorts are compatible with integers and longs, so ShortNode must have a longValue() method. Also write the SumVisitor method required for a ShortNode.

```
interface NodeVisitor {
    void Visit(IntNode n);
    void Visit(LongNode n);
    void Visit(FloatNode n);
    void Visit(DoubleNode n);
}

class SumVisitor implements NodeVisitor {
    long lTotal = 0; double dTotal = 0.0;

    public void Visit(IntNode n) { lTotal += n.longValue(); }
    public void Visit(LongNode n) { lTotal += n.longValue(); }
    public void Visit(FloatNode n) { dTotal += n.doubleValue(); }
    public void Visit(DoubleNode n) { dTotal += n.doubleValue(); }
}

class Node { /* a class for a tree of numbers */
    LinkedList children;
    void Accept(NodeVisitor v) {}
}
```

```

class IntNode extends Node {
    Integer value;
    void Accept(NodeVisitor v) {
        ListIterator iter; Node n;
        for (iter = children.listIterator(); iter.hasNext(); ) {
            n = (Node) iter.next(); n.Accept(v);
        }
        v.Visit(this);
    }
    long longValue() { return value.longValue(); }
}

```

```

class LongNode extends Node {
    long value;
    void Accept(NodeVisitor v) {
        ListIterator iter; Node n;
        for (iter = children.listIterator(); iter.hasNext(); ) {
            n = (Node) iter.next(); n.Accept(v);
        }
        v.Visit(this);
    }
    long longValue() { return value; }
}

```

```

class FloatNode extends Node {
    Float value;
    void Accept(NodeVisitor v) {
        ListIterator iter; Node n;
        for (iter = children.listIterator(); iter.hasNext(); ) {
            n = (Node) iter.next(); n.Accept(v);
        }
        v.Visit(this);
    }
    double doubleValue() { return value.doubleValue(); }
}

```

```

class DoubleNode extends Node {
    double value;
    void Accept(NodeVisitor v) {
        ListIterator iter; Node n;
        for (iter = children.listIterator(); iter.hasNext(); ) {
            n = (Node) iter.next(); n.Accept(v);
        }
        v.Visit(this);
    }
    double doubleValue() { return value; }
}

```

```

public class VisMain {
    public static void main(String args[]) {
        Node root;
        // create tree here, starting at root

        NodeVisitor visitor = new SumVisitor();
        root.Accept(visitor);
    }
}

```

Answer:

```

class PrintVisitor implements NodeVisitor {

    void Visit(IntNode n) { System.out.println(n.value.intValue() + "i "); }
    void Visit(LongNode n) { System.out.println(n.value + "l "); }
    void Visit(FloatNode n) { System.out.println(n.value.floatValue() + "f "); }
    void Visit(DoubleNode n) { System.out.println(n.value + "d "); }
}

class ShortNode extends Node {
    Short value;

    void Accept(NodeVisitor v) {
        ListIterator iter; Node n;
        for (iter = children.listIterator(); iter.hasNext(); ) {
            n = (Node) iter.next();
            n.Accept(v);
        }
        v.Visit(this);
    }

    long longValue() { return value.longValue(); }
}

void Visit(ShortNode n) { lTotal += n.longValue(); }

```

Name:

Account:

2. Distributed Computing & RMI (20 points)

Suppose we want to design a client-server system, such as a web browser and web server, that uses RMI to communicate between client objects and server objects, with clients initiating all calls. Describe the steps required in both the client and server objects to establish an environment that enables a client to make any remote call that the server makes available. Make sure to describe the information that the client must possess in order to establish that environment. What server methods can the client call?

Answer:

The server must start up an RMI registry (or register with another registry), and then register itself with `Naming.bind()` or `Naming.rebind()`,

The client needs to do a `Naming.lookup()` into the registry to get a remote reference to the server object. Once it has that reference, it can do calls into the server object.

The client needs the machine name (IP address) and port number on which the server is running an RMI registry (or where the server registered itself in another RMI registry), and a string that represents the server object.

The client can call any methods that are specified in a Remote interface that the server class implements.

Name:

Account:

3. Threads (25 points)

The code below models a burger restaurant. It has three classes: Restaurant, Burger, and Chef.

The Restaurant class has an input tray in which orders are placed and an output tray where cooked burgers are placed. It has two internal threads: one for taking orders and placing them on the input tray and one for getting burgers off the output tray and giving them to clients.

The Burger class represents both the order for the burger (before it is cooked) and the actual burger (after it is cooked). The burger class also knows how to cook itself and how to place itself in the output tray.

The Chef class represents chefs. Each chef has a differently sized grill that can simultaneously accommodate a fixed number of burgers. That number is stored in the variable *capacity*. Each chef will take orders from the input tray if they exist and if the chef has space on the grill. If the chef can't take more orders for either reason it blocks until that situation changes. (No spin locks!!)

Write the code needed to make these three classes interact properly. Pay attention to issues of synchronization. Make sure the classes block when they can't make progress. You are to add code, but you can't change any code that's already written.

```
class Burger {
    // Burger represents both the order and the actual burger
    long cookTime = 5000l; Restaurant myRestaurant;
    Burger (Restaurant r) { myRestaurant = r; }

    void cookMe (final Chef myChef) {
        // Finish writing code to do the following
        // 1. create a new thread. 2. Go to sleep for cookTime.
        // 3. Let cook know you're finished. 4. put burger in output tray.
        (new Thread () {
            public void run () {
                try {Thread.sleep(cookTime);}
                catch (InterruptedException e) {}

                // FILL IN THE REST HERE
            }
        }).start();
    }
}

class Chef extends Thread {
    int capacity; Restaurant myRestaurant;
    Chef (int cap, Restaurant r) {
        capacity = cap; myRestaurant = r;
    }
    // Write code to do the following
    // 1. if you have space and an order is waiting get it
```

```

// from input tray. 2. cook burger.
public void run () {
    // FILL IN HERE
}

public class Restaurant {
    List in; List out;

    public static void main (String [] args) {
        final Restaurant r = new Restaurant();
        r.in = Collections.synchronizedList(new LinkedList()); //input tray
        r.out = Collections.synchronizedList(new LinkedList()); // output tray

        // start up the chefs
        Chef c1 = new Chef(10,r), c2 = new Chef (15,r), c3 = new Chef (7,r);
        c1.start(); c2.start(); c3.start();

        // This thread takes orders and puts them in input tray
        (new Thread() {
            public void run() {
                while (true) {
                    Random rand = new Random();
                    try { Thread.sleep(Math.abs(rand.nextLong()) % 1000); }
                    catch (InterruptedException e) {}
                    r.in.add (new Burger(r));
                    synchronized (r.in) { r.in.notifyAll(); }
                } // end while
            } // end run()
        }).start();

        // This thread takes burgers from output tray and gives them to clients
        (new Thread () {
            public void run() {
                while (true) {
                    while (r.out.size() == 0) {
                        try { synchronized (r.out) { r.out.wait(); } }
                        catch (InterruptedException e){}
                    }
                    // Get the burger - this is where the client would be called
                    (Burger) r.out.remove(0);
                } // end while
            } // end run()
        }).start();
    } // end main()
} //end Restaurant

```

Answer:

```
// class Burger
void cookMe (final Chef myChef) {
    // write code to do the following
    // 1. create a new thread. 2. Go to sleep for cookTime.
    // 3. Let cook know you're finished. 4. put burger in output tray.

    (new Thread () {
        public void run () {
            try {Thread.sleep(cookTime);}
            catch (InterruptedException e) {}

            // FILL IN THE REST HERE
            synchronized (myChef) {
                myChef.capacity++; myChef.notify();
            }
            myRestaurant.out.add(Burger.this);
            synchronized (myRestaurant.out) {
                myRestaurant.out.notifyAll();
            }
        }
    }).start();
}

// class Chef
public void run () {
    // FILL IN HERE
    while (true) {
        synchronized (this) {
            while (capacity <= 0) {
                try {wait();}
                catch (InterruptedException e) {}
            }
        }
        synchronized (myRestaurant.in) {
            while (myRestaurant.in.size() <= 0) {
                try { myRestaurant.in.wait(); }
                catch (InterruptedException e) {}
            }
            try {((Burger) myRestaurant.in.remove(0)).cookMe(this);}
            catch (IndexOutOfBoundsException e) {}
        }
        synchronized(this) { capacity--; }
    }
}
```

Name:

Account:

4. Design Patterns (25 points)

Rewrite the Sieve class from projects 3 and 5 using the State pattern. Each state object should implement the State interface given below.

You will need to write the constructor and run methods for the Sieve class, and the classes for the three states. State1 means that the Sieve hasn't seen its first prime number. State2 means the Sieve has seen its first prime, but either hasn't seen its second prime yet or that it has seen the second prime, but hasn't created a new Sieve yet. State 3 means the Sieve has seen its first two prime numbers and has created a new Sieve. Each individual state object should be implemented using the Singleton pattern.

You may add to the code given below. You shouldn't need to add any new variables or use any methods not defined below. You may not change anything that's already written.

State2 would normally need to create a new Sieve object, change the filter streams, etc. When you get to that part just write the phrase "MAKE NEW SIEVE HERE."

```
public interface State {
    // do appropriate processing and return next State
    State process (Sieve sieve, Integer value);
}

public class Sieve implements Runnable {
    private State state;
    int firstPrime;
    Filter myFilter;

    Sieve () {
        // WRITE THIS ONE
    }

    Integer readNextInt () {
        // returns the next Integer or null if
        // input buffer is closed
    }

    void changeOutputStream (java.io.Writer newOutputStream) {
        // flushes output buffer
        // changes output stream to newOutputStream,
        // handles synchronization
    }

    void closeOutBuff () { myFilter.getOutputStream().close(); }

    void writeInt (Integer outValue) {
        myFilter.getOutputStream().write(outValue.toString().toCharArray());
    }
}
```

```

public void run() {
    // WRITE THIS ONE
}

public void setFilter(Filter inFilter) { myFilter = inFilter; }
}

// State classes go here

```

Answer:

```

// class Sieve
Sieve () {
    // WRITE THIS ONE
    state = State1.instance;
}

public void run() {
    // WRITE THIS ONE
    Integer i;
    while ((i = readNextInt()) != null) {
        state = state.process(this, i);
    }
    closeOutBuff();
}

class State1 implements State {
    private State1() { }
    final static State instance = new State1();
    public State process (Sieve sieve, Integer value) {
        sieve.firstPrime = value.intValue();
        sieve.writeInt(value);
        return State2.instance;
    }
}

```

```

class State2 implements State {
    private State2() { }
    final static State instance = new State2();
    public State process (Sieve sieve, Integer value) {
        if (value.intValue() % sieve.firstPrime != 0) {
            sieve.writeInt(value);
            PipedReader newInPipe = new PipedReader();
            PipedWriter newOutPipe = new PipedWriter(newInPipe);
            // MAKE NEW SIEVE HERE
            Sieve newSieve = new Sieve ();
            FilterImpl newFilter = new FilterImpl(
                newInPipe,
                sieve.myFilter.getOutputConnection(),
                sieve.myFilter.getInputBuffer().getMaxBufferSize(),
                sieve.myFilter.getOutputBuffer().getMaxBufferSize(),
                newSieve);
            newSieve.setFilter(newFilter);
            sieve.changeOutputStream(newOutPipe);
            newFilter.startFilter();
            // END OF MAKING NEW SIEVE
            return State3.instance;
        }
        else {
            return State2.instance;
        }
    }
}

class State3 implements State {
    private State3() { }
    final static State instance = new State3();
    public State process (Sieve sieve, Integer value) {
        if (value.intValue() % sieve.firstPrime != 0) {
            sieve.writeInt(value);
        }
        return State3.instance;
    }
}

```

Name:

Account:

5. (Short answers) 10 points

- (a) (5 points) Describe one advantage and one disadvantage of using RMI instead of sockets for performing a distributed computation.

Answer:

Advantage - Using RMI avoids the need to figure out how to encode and decode a request and response as a bitstream. Many other things are easier using RMI, such as passing a remote reference to another service and serialization. However, these can be done with Sockets.

Disadvantage - The primary disadvantage of RMI for communication is that it is specific to Java, and is inefficient for transferring (serializing) large amounts of data (e.g., a large array)

- (b) (5 points) Give two reasons why you would want to make a class implement Serializable instead of Remote.

Answer:

- i. If the class has only immutable fields, then there is no reason to require a remote access back to the original object, since copies of the fields are completely equivalent to the original values.
- ii. For efficiency, so that method calls would be local instead of remote when the object is passed around. That won't work if the program wants the original object to be changed by a remote call.