22 Concurrency

In the last few weeks, we have been looking at interactive applications with a common structure. For instance, in the case of the Observer Pattern lecture, the MVC Pattern lecture, and Space Game, all code was structured around an event loop (the command line) that triggers other parts of the application to do some work — the work gets down, and then control comes back to the loop so that more input from the user can be processed.

There's a serious problem with that code structure: if some of the processing takes a long time, the user cannot do anything about it, since control has not returned to the command loop.

GUI applications of yore used to be structured like that: an application was a big event loop, the OS or windowing system would post events in a queue, the loop would take the next event from the queue and call the right "event handler", before looking at the next event in the queue. If an event handler was slow, took too long, or just looped forever, the application was utterly unresponsive. Sadly, this still happens in some badly designed applications.

The cure is *concurrency* — the idea that an application is not made up of just one "thread of control", but rather multiple threads of control, executing at the same time, and independently from each other.

This is very powerful, but also quite easy to get wrong, and comes with its own set of difficulties.

We consider two approaches to concurrency in a language such as Java — these approaches are not completely orthogonal, but they do illustrate different concepts.

I base what's below on the code I presented in class, available on the course web page.

22.1 OS-Based Concurrency

All current operating systems are concurrent. After all, they run multiple applications at the same time. These are called "processes" at the level of the operating system. So one way to use concurrency is to run different programs, and have them communicate through some operating-system-provided facility. Such facilities are generally called IPC (inter-process communication), and one such example is sockets.

I won't go into too much details about sockets, since there is a lot of information about them. Suffices to say they are a generally way for processes across different machines to communicate.

Java provides a socket library, but to simplify the presentation I provide a small abstraction layer for sockets.

There are two kinds of sockets: server sockets, and channel (or communication) sockets. Say two processes want to communicate. One has to ping the other, so to speak. The one asking the other for communication is called the *client*, while the one receiving a request for communication is called the *server*. A server needs to be ready to communicate. To do so, it creates a *server socket*, which has an address on the machine (called a port), and sits waiting for a client to come knocking. A client wanting to communicate with the server needs to create a channel socket, stating which machine to find the server on, and the port at which the server is listening. Once the connection request is made, the server can accept that connection, and itself obtain a channel socket representing the other end of the communication channel with the client's own channel socket. The client and the server can then use that channel socket to read and write data using the same facilities for reading and writing from, say, the terminal.

Here is the signature for channel sockets, or `ChanSocket`:

```
public static ChanSocket create (String addr, int port);
public BufferedReader in () {
public PrintWriter out () {
public void close () {
```

You create a channel socket specifying the address of the machine to connect to (as an IP address or a machine name or whatever your operating system recognizes as a machine address on the network), and a port on that machine on which to find a server listening. Once a channel socket has been created, methods `in()` and `out()` provide input and output streams that can be read and written using methods `readLine()` and `println()`, just like any other input/output stream. Method `close()` terminates a channel socket and disconnects it from the server.

Here is the signature for server sockets, or `ServSocket`:

```
public static ServSocket create (int port);
public ChanSocket accept ();
public void close ();
```

You create a server socket by specifying the port to listen on. Once a server socket has been created, you can listen on it using `accept()`. That method blocks until a client tries to connect to the port, at which point the connection is completed and the server obtains a `ChanSocket` that is connected with the client's channel socket with which it contacted the server. Method `close()` closes a server socket, disconnecting it from the port.

To see how we can use the above, here are two simple applications implementing a server that expects two integers on a socket and sends back their sums on the socket. A client is in charge of sending those two integers and receiving the integer back.

First, let's present the client.

```
import java.io.*;
import java.net.*;

public class AddClient {

   private static void error (String s) {
      System.err.println(s);
      System.exit(−1);
   }

   private static Option<String> getInput () {

      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
      String response = "";
      try {
         response = br.readLine();
         if (response==null) {
            return Option.none();
         }
      } catch (IOException ioe) {
         error("IO error reading from terminal");
      }
      return Option.some(response);
   }

   public static void main (String[] argv) throws IOException {
      ChanSocket addSocket = null;

      addSocket = ChanSocket.create("127.0.0.1",8044);

      System.out.println("Connected to server");

      while (true) {
         System.out.print("First number: ");
         Option<String> input1 = getInput();
         if (input1.isNone())
            break;
         addSocket.out().println(input1.valOf());
         System.out.print("Second number: ");
         Option<String> input2 = getInput();
```

```
            if (input2.isNone())
                break;
            addSocket.out().println(input2.valOf());
            String response = addSocket.in().readLine();
            System.out.println("result= " + response);
        }

        addSocket.close();
    }
}
```

The `main()` method does all the work: it attempts to create a channel socket to a server on the same machine (that's what address `127.0.0.1` usually means), at port 8044. If it finds one, then it repeatedly gets two integers from the user, sends them to the server (via `addSocket.out().println()`), and reads back the response (via `addSocket.in().readLine()`).

The server is equally simple.

```
import java.net.*;
import java.io.*;

public class AddServer {

    private static void error (String s) {
        System.err.println(s);
        System.exit(-1);
    }

    private static int stringToInt (String s) {
        try {
            return Integer.parseInt(s);
        } catch (NumberFormatException nfe) {
            return 0;
        }
    }

    public static void main (String[] argv) throws IOException {
        ServSocket serverSocket = ServSocket.create(8044);

        System.out.println("Listening on port 8044");

        ChanSocket clientSocket = serverSocket.accept();
```

```
        System.out.println("Accepting connection");

        while (true) {
            String input1 = clientSocket.in().readLine();
            String input2 = clientSocket.in().readLine();
            if (input1==null || input2==null)
                break;
            int i1 = stringToInt(input1);
            int i2 = stringToInt(input2);
            System.out.println("Computing sum of " + i1 + " and " + i2);
            clientSocket.out().println(i1+i2);
        }

        clientSocket.close();
        serverSocket.close();
    }
}
```

Again, all the action is in `main()`: the server creates a server socket to listen on port 8044, and once it accepts a connection from a client, it reads two integers from the resulting channel socket, adds them, and sends the result back on the socket.

Note that sockets communicate by exchanging strings, so if you want to send other kinds of data, you need to convert to and from strings, which is clunky, but sometimes the best you can do.

You can try the above code on a machine by first running the server, which will wait for a client to connect, then running the client. If you do this in Eclipse, note that you need two Eclipses going for this to work. If you do this from the command-line, you either need two command-lines, or know how to run processes in the background.

What else? Sockets are a *message-passing model*: sockets work by sending messages between processes. Message passing is a very nice model in concurrency, it avoids a lot of the problems that we will next see when looking at concurrency in Java.

Problems with sockets for concurrency include (1) they're not very flexible, and (2) they're *very* heavy — they require lots of resources from the OS. A plus, however, is that the client and the server can in fact be implemented in different languages. It is an easy thing to write the above server in C, say, and communicate with that C server using sockets, for instance using the above Java client.

22.2 Language-Based Concurrency

Most programming languages nowadays include a way to write concurrent programs directly in the language, using a facility such as threads. (Concurrent programs are sometimes called multi-threaded, although there are concurrent programs that are not based on the notion of a thread the way we usually understand it.)

In Java, a thread is a piece of code executing independently from other pieces of code. When your program starts, it actually runs on a particular thread called the main thread. You can create and run code on new threadsas follows: (1) create a thread by doing something like:

```
Thread t = new Thread(codeForThread);
```

then (2) start the thread by calling

```
t.start();
```

This will start executing the code in `codeForThread` in its own thread, independently from the rest of the program. In particular, the thread that called `t.start()` will happily continue executing the next statement after the `t.start()` one, while the new thread executes.

And what is this `codeForThread`? It is an object implementing the `Runnable` interface — the `Runnable` interface specifies that the object must have a `void run ()` method. When a thread is started, what happens is that the `run()` method in the object that you used to construct the thread in the first place is what gets called and executes independently from the rest of the code.

And that's it. What about communication between threads? Well, communication between threads is done by *shared memory*. Intuitively, every thread in your program has access to the memory, which is a shared resource, and one thread can write in a memory cell where another thread can read the value. In other words, there can be sharing between threads, which is a nice side-effect of having mutation and sharing in the language, but also makes reasoning about multi-threaded programs *exceedingly* difficult.

Let's look at a simple example before looking at what can go wrong. First, let's rewrite the simple addition server above as a multi-threaded Java program. Here's the main file:

```
public class Main {

   //Display a message, preceded by the name of the current thread
   static void threadMessage (String msg) {
      String threadName = Thread.currentThread().getName();
      System.out.println("[" + threadName + ": " + msg + "]");
   }
```

```
    public static void main (String[] argv) throws InterruptedException {

        Channel<Pair<Integer,Integer>> valuesCh = Channel.create();
        Channel<Integer> resultCh = Channel.create();

        Thread addClientT, addServerT;
        threadMessage("starting add client thread");
        addClientT = new Thread(AddClient.create(valuesCh,resultCh),"client");
        addClientT.start();

        threadMessage("starting server thread");
        addServerT = new Thread(AddServer.create(valuesCh,resultCh),"server");
        addServerT.start();

        threadMessage("waiting for client thread to finish");
        addClientT.join();
        threadMessage("terminating server thread");
        addServerT.interrupt();
        addServerT.join();
    }
}
```

Without going into too much details, the `main()` function creates a thread for the client (`addClientT`) and a thread for the server (`addServerT`), starts them, then waits for them to finish. It also creates *channels* between those threads, one to send values from the client thread to the server thread, and one to send values back from the server thread to the client thread. Channels are not built-in, but here's a simple (not very good, but illustrative of the main points) version.

```
public class Channel<E> {

    private Option<E> content;

    private Channel () {
        content = Option.none();
    }

    public static <T> Channel<T> create () {
        return new Channel<T>();
    }

    // Put a value on the channel if none is present
```

```
    //
    public boolean put (E val) {
            if (content.isNone()) {
                content = Option.some(val);
                return true;
            }
            return false;
        }

    // Put a value on the channel, block until the put can be done
    //
    public void putOrBlock (E val) throws InterruptedException {
        while (!put(val)) {
            Thread.sleep(100);
        }
    }

    // Get a value (possibly none) from the channel
    //
    public Option<E> get () {
            if (content.isNone())
                return content;
            Option<E> result = content;
            content = Option.none();
            return result;
        }

    // Block until there is a value on the channel, then get it
    //
    public E getOrBlock () throws InterruptedException {
        Option<E> response;
        response = get();
        while (response.isNone()) {
            Thread.sleep(100);
            response = get();
        }
        return response.valOf();
    }
}
```

A channel is parametrized by a type of values it can hold, and the main methods it provides is a `put()` method for putting a value on the channel (which only does so if there isn't

already a value on the channel — it returns true or false depending on whether it was able to put the value on the channel), and a `get()` method to get a value from the channel, which if there is one removes it from the channel.

To simplify the code below, there are `blocking` versions of `put()` and `get()`, where `putOrBlock()` attempts to put a value on the channel — basically waiting until the channel is freed (say, by having another thread `get()` its value) before putting the value on the channel and returning — and where `getOrBlock()` attempts to get a a value from the channel, waiting until there is a value present before returning it.

With such channels in hand, we can implement the code for the client thread and the code for the server thread. Note that in `Main`, both the client thread and the server thread gets the channels `valuesCh` and `resultCh` — these channels are shared between the threads.

First, the client code.

```
import java.io.*;

public class AddClient implements Runnable {

   private Channel<Pair<Integer,Integer>> valuesCh;
   private Channel<Integer> resultCh;

   private AddClient (Channel<Pair<Integer,Integer>> c1,
                      Channel<Integer> c2) {
      valuesCh = c1;
      resultCh = c2;
   }

   public static AddClient create (Channel<Pair<Integer,Integer>> c1,
                                   Channel<Integer> c2) {
      return new AddClient(c1,c2);
   }

   private static int stringToInt (String s) {
      try {
         return Integer.parseInt(s);
      } catch (NumberFormatException nfe) {
         return 0;
      }
   }

   private static Option<String> getInput () {
```

```
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    String response = "";

    try {
        response = br.readLine();
        if (response==null) {
            return Option.none();
        }
    } catch (IOException ioe) {
        System.err.println("IO error reading from terminal");
        System.exit(-1);
    }
    return Option.some(response);
}

public void run () {

    try {
        Main.threadMessage("started");

        while (true) {

            System.out.print("First number: ");
            Option<String> input1 = getInput();
            if (input1.isNone())
                break;
            int i1 = stringToInt(input1.valOf());

            System.out.print("Second number: ");
            Option<String> input2 = getInput();
            if (input2.isNone())
                break;
            int i2 = stringToInt(input2.valOf());

            valuesCh.putOrBlock(Pair.create(i1,i2));
            Main.threadMessage("awaiting response from server");
            Integer response = resultCh.getOrBlock();
            System.out.println("result= " + response);
        }
        Main.threadMessage("finished");
    } catch (InterruptedException e) {
        Main.threadMessage("interrupted");
```

```
            return;
        }
    }
}
```

The class implements `Runnable`, meaning that it has a `run()` method that gets called when a thread is created to run `AddClient`. Looking at that `run()` method, we see that it gets two numbers from the user, and puts that pair of numbers[1] in the `valuesCh` channel — waiting until putting those values succeeds before proceeding. It then waits for a response to appear on `resultCh` using `getOrBlock()`, and when one does it displays it before repeating through the whole process.

What about the server code?

```
import java.io.*;

public class AddServer implements Runnable {

    private Channel<Pair<Integer,Integer>> valuesCh;
    private Channel<Integer> resultCh;

    private AddServer (Channel<Pair<Integer,Integer>> c1,
                       Channel<Integer> c2) {
        valuesCh = c1;
        resultCh = c2;
    }

    public static AddServer create (Channel<Pair<Integer,Integer>> c1,
                                    Channel<Integer> c2) {
        return new AddServer(c1,c2);
    }

    public void run () {

        try {

            Main.threadMessage("started");

            while (true) {
                Main.threadMessage("awaiting input from client thread");
```

[1]This uses a class `Pair<U,V>` representing pairs, which we saw in previous lectures, and I'll let you look at in that code yourself.

```
            Pair<Integer,Integer> input = valuesCh.getOrBlock();
            int i1 = input.first();
            int i2 = input.second();
            Main.threadMessage("computing sum of " + i1 + " and " + i2);

            resultCh.putOrBlock(i1+i2);
          }
        } catch (InterruptedException e) {
          Main.threadMessage("interrupted");
          return;
        }
    }
}
```

It is quite similar, and in fact a bit simpler. If you look at method `run()` which holds the code that the thread will execute, it waits for a pair of integers to appear on `valuesCh`, computes the sum and puts it on the `resultCh` channel, waiting for that channel to be free before proceeding by repeating the whole thing again.

This works, but there's a big problem with the above. In the presence of two clients wanting to communicate with the server, there is a *race condition* that develops when attempting to read the `resultCh` channel. Consider the following scenario. You have two threads `t1` and `t2` running an instance of `AddClient`, that is, waiting for integers from the user, sending them to the server before getting the sum back. Now those threads run concurrently, and one may be faster than the other, in ways that are essentially out of anyone's control. Suppose `t1` is faster in sending its pair of integers i_1 and j_1 to the server. The server accepts them, freeing the `valuesCh` channel. Meanwhile, `t2` sends its own pair of integers, i_2 and j_2 to the server. The channel `valuesCh` is free, so the values are put there. The server, meanwhile, computes the sum $i_1 + j_1$ and puts that answer on `resultCh` before reading the next pair of integers from the `valuesCh` channel. Suppose that `t2` is faster than `t1` in reaching the point where it tries to read from `resultCh`. It reads the result, which it $i_1 + j_1$, and wrongly takes that as the result of its own query, which should have been $i_2 + j_2$. Because of the concurrency, `t1` and `t2` were in a race to get at the result posted on `resultCh`, and once in a while the wrong thread will get the answer. Race conditions are exceedingly difficult to debug, and they are very easy to write in concurrent code with shared memory.

To show you that this is not just an academic problem, let me modify the above example to showcase this problem. First, let's modify the `AddClient` thread to generate numbers at random rather than obtain them from the user.

```
import java.util.*;
import java.io.*;
```

```
public class AddClient implements Runnable {

   private Channel<Pair<Integer,Integer>> valuesCh;
   private Channel<Integer> resultCh;

   private int delay;

   private AddClient (int s, Channel<Pair<Integer,Integer>> c1,
                      Channel<Integer> c2) {
      delay = s;
      valuesCh = c1;
      resultCh = c2;
   }

   public static AddClient create (int s, Channel<Pair<Integer,Integer>> c1,
                                   Channel<Integer> c2) {
      return new AddClient(s,c1,c2);
   }

   private static Random rnd = new Random();

   public static int random(int bound) {
      return rnd.nextInt(bound);
   }

   public void run () {

      try {

         Main.threadMessage("start");

         while (true) {
            int i1 = random(100);
            int i2 = random(100);

            Main.threadMessage("asking for " + i1 + " + " + i2);
            valuesCh.putOrBlock(Pair.create(i1,i2));

            Thread.sleep(delay);

            Integer response = resultCh.getOrBlock();
```

```
            Main.threadMessage("Received response = " + response);
            System.out.print(i1 + " + " + i2 + " = " + response);
            if (i1+i2 == response.intValue())
               System.out.println("");
            else
               System.out.println("                                ===>  WRONG!");
         }
         //      Main.threadMessage("finished");
      } catch (InterruptedException e) {
         Main.threadMessage("interrupted");
         return;
      }
   }
}
```

Aside from the randomness, the code is as before, with one difference: when you create an instance of `AddClient`, you specify a "slowness" factor — a delay that gets inserted between the point where the client sends its request to the server, and when the response is looked for. (This abstract away form the code maybe performing some other work, such as a long computation, or some input and output to the network or to disk.) After response is obtained from the server thread, we do a quick check to see if we got the right answer. Our hope is that `WRONG!` never appears in the output of this program.

The `AddServer` class is as before, and the `Main` class now creates a couple of client threads with different slowness factors:

```
public class Main {

   //Display a message, preceded by the name of the current thread
   static void threadMessage (String msg) {
      String threadName = Thread.currentThread().getName();
      System.out.println("[" + threadName + ": " + msg + "]");
   }

   public static void main (String[] argv) throws InterruptedException {

      Channel<Pair<Integer,Integer>> valuesCh = Channel.create();
      Channel<Integer> resultCh = Channel.create();

      Thread addClient1, addClient2, addServer;
      addClient1 = new Thread(AddClient.create(50,valuesCh,resultCh), "client1");
      addClient1.start();
```

```
        addClient2 = new Thread(AddClient.create(10,valuesCh,resultCh), "client2");
        addClient2.start();

        addServer = new Thread(AddServer.create(valuesCh,resultCh),"server");
        addServer.start();

        Thread.sleep(60000);
        addClient1.interrupt();
        addClient2.interrupt();
        addServer.interrupt();
        addClient1.join();
        addClient2.join();
        addServer.join();
    }
}
```

Let's run this code, and see what happens:

```
[client1: start]
[server: started]
[client2: start]
[client1: asking for 84 + 97]
[client2: asking for 50 + 18]
[server: computing 50 + 18 = 68]
[client2: Received response = 68]
50 + 18 = 68
[client2: asking for 73 + 59]
[server: computing 73 + 59 = 132]
[client2: Received response = 132]
73 + 59 = 132
[client2: asking for 87 + 33]
[server: computing 87 + 33 = 120]
[client2: Received response = 120]
87 + 33 = 120
[client2: asking for 43 + 79]
[server: computing 43 + 79 = 122]
[client2: Received response = 122]
43 + 79 = 122
[client2: asking for 81 + 23]
[server: computing 81 + 23 = 104]
[client1: Received response = 104]
```

```
84 + 97 = 104                          ===>  WRONG!
[client1: asking for 42 + 66]
[server: computing 42 + 66 = 108]
[client1: Received response = 108]
42 + 66 = 108
[client1: asking for 75 + 48]
[server: computing 75 + 48 = 123]
[client1: Received response = 123]
```

Oops. At least one sum ended up wrong. Look at the interleaving of the client and server statements to see the order in which the statements were executed, and see if you can spot where the race condition was resolved the wrong way.

So how can we get rid of the race condition? In general, there are different approaches that can be taken, mostly because there are many different causes for race conditions.

In the above example, one problem is that the result channel `resultCh` is shared amongst all the client threads — the race condition occurs when different clients race to read the result channel, and sometimes getting the wrong answer because they did not look at the result in time. So one way to solve the problem is to have every client use a dedicated result channel. Another way is to simply have the client, when it sends a pair of integers to the server, to also send a channel on which the server can send the result and that only that client knows how to read.

Here's the resulting code, that I invite you to look at. It uses a class `Triple<T,U,V>` to represent triples of values of different types, a slight generalization of `Pair<U,V>`.

```
public class Main {

   //Display a message, preceded by the name of the current thread
   static void threadMessage (String msg) {
      String threadName = Thread.currentThread().getName();
      System.out.println("[" + threadName + ": " + msg + "]");
   }

   public static void main (String[] argv) throws InterruptedException {
      Channel<Triple<Integer,Integer,Channel<Integer>>> valuesCh =
         Channel.create();

      Thread addClient1, addClient2, addServer;
      addClient1 = new Thread(AddClient.create(50,valuesCh), "client1");
      addClient1.start();

      addClient2 = new Thread(AddClient.create(10,valuesCh), "client2");
      addClient2.start();
```

```
        addServer = new Thread(AddServer.create(valuesCh),"server");
        addServer.start();


        Thread.sleep(60000);
        addClient1.interrupt();
        addClient2.interrupt();
        addServer.interrupt();
        addClient1.join();
        addClient2.join();
        addServer.join();
    }
}
```

```
import java.util.*;
import java.io.*;

public class AddClient implements Runnable {

    private Channel<Triple<Integer,Integer,Channel<Integer>>> valuesCh;

    private int delay;

    private AddClient (int s,
                       Channel<Triple<Integer,Integer,Channel<Integer>>> c) {
        delay = s;
        valuesCh = c;
    }

    public static AddClient create
            (int s, Channel<Triple<Integer,Integer,Channel<Integer>>> c) {
        return new AddClient(s,c);
    }

    private static Random rnd = new Random();

    public static int random(int bound) {
        return rnd.nextInt(bound);
    }

    public void run () {
```

```
        try {
            Main.threadMessage("start");

            while (true) {
                int i1 = random(100);
                int i2 = random(100);

                Channel<Integer> resultCh = Channel.create();

                Main.threadMessage("asking for " + i1 + " + " + i2);
                valuesCh.putOrBlock(Triple.create(i1,i2,resultCh));

                Thread.sleep(delay);

                Integer response = resultCh.getOrBlock();
                Main.threadMessage("Received response = " + response);
                System.out.print(i1+" + "+i2+" = " + response);
                if (i1+i2 == response.intValue())
                    System.out.println("");
                else
                    System.out.println("                              ====>  WRONG!");

            }
            //      Main.threadMessage("finished");
        } catch (InterruptedException e) {
            Main.threadMessage("interrupted");
            return;
        }
    }
}
```

```
import java.io.*;

public class AddServer implements Runnable {

    private Channel<Triple<Integer,Integer,Channel<Integer>>> valuesCh;

    private AddServer (Channel<Triple<Integer,Integer,Channel<Integer>>> c) {
        valuesCh = c;
    }

    public static AddServer create
```

```
                (Channel<Triple<Integer,Integer,Channel<Integer>>> c) {
        return new AddServer(c);
    }

    public void run () {
        try {
            Main.threadMessage("started");

            while (true) {

                Triple<Integer,Integer,Channel<Integer>> input =
                    valuesCh.getOrBlock();

                int i1 = input.first();
                int i2 = input.second();
                Channel<Integer> resultCh = input.third();
                Main.threadMessage("computing "+i1+" + "+i2+" = "+(i1+i2));

                resultCh.putOrBlock(i1+i2);
            }
            // Main.threadMessage("finished");
        } catch (InterruptedException e) {
            Main.threadMessage("interrupted");
            return;
        }
    }
}
```

Actually, now that I type this up, I'm realizing that I don't actually need to create a new `resultCh` at every iteration through the loop — it suffices for there to be a different `resultCh` per client thread. It still needs to be passed to the server thread every time a query is sent to the server, but we could simply create a `resultCh` before the loop, and use that `resultCh` for every query that the thread sends to the server. I'm not going to do that here, but you should try it out yourself.

There's a lot — *a lot* — more about concurrent (even multi-threaded) programming that I haven't covered here. In particular, the notion of atomicity, which may also be a problem with the code I give above, although it is simple enough that perhaps that is not an issue. But in general, it is. Atomicity (or lack thereof) leads to a form of race condition that is much worse that the one above. I will let you read about it yourself, along with reading about locks, and critical regions, and synchronized methods. I have put some pointers on the course web page.