The University of Melbourne
Department of Computer Science and Software Engineering
433-254 Software Design
Second Semester, 2003
Tutorial 11
Week 12

1. What are design patterns? Develop and discuss a singleton design pattern with a suitable example of your own.

   Sample Answer:
   A *pattern* can be defined a solution for a problem in a context. Thus, a *design pattern* can be defined as: *a common solution for a common problem faced during the design of software systems*. Design patterns are abstract (not concrete), thus, they may have many implementations.

   Singleton Design Pattern:
   - Intent: Ensure a class only has one instance, and provide a global point of access to it.
   - Idea: Have an encapsulated static variable holding the single instance of a class. Provide a static get-operation that creates the single instance once and returns it from then on.
   - Example: While Java has many Singleton-like classes, java.lang.Runtime is a by the book example of a singleton. Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime method.

```
public class Runtime {
   private static Runtime currentRuntime = new Runtime();

    /**
    * Returns the runtime object associated with the current
    * Java application.
    * Most of the methods of class <code>Runtime</code> are instance
    * methods and must be invoked with respect to the current runtime object.
    *
    * @return  the <code>Runtime</code> object associated with the current
    *          Java application.
    */
   public static Runtime getRuntime() {
       return currentRuntime;
   }

   /** Don't let anyone else instantiate this class */
   private Runtime() {}
```

   `'private static Runtime currentRuntime = new Runtime();'` makes the class itself responsible for keeping track of its sole instance.", the key word private means that only this class may access the field currentRuntime. The keyword static denotes that only one instance of particular field may exist.

   The first method in the class goes on to implement "there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point." Both of these idioms are further reinforced by the privatization of the Runtime constructor, as the author notes in her comments.

2. Discuss the process of creation of server and client sockets with Exceptions handled explicitly with a suitable example.

   Sample Answer:

When programming a client, you must follow these four steps:

- Open a socket.
- Open an input and output stream to the socket.
- Read from and write to the socket according to the server's protocol.
- Clean up.

These steps are pretty much the same for all clients. The only step that varies is step three, since it depends on the server you are talking to.

How do I open a socket?

If you are programming a client, then you would open a socket like this:

```
Socket MyClient;
MyClient = new Socket("Machine name", PortNumber);
```

Where Machine name is the machine you are trying to open a connection to, and PortNumber is the port (a number) on which the server you are trying to connect to is running. When selecting a port number, you should note that port numbers between 0 and 1,023 are reserved for privileged users (that is, super user or root). These port numbers are reserved for standard services, such as email, FTP, and HTTP. When selecting a port number for your server, select one that is greater than 1,023!

In the example above, we didn't make use of exception handling; however, it is a good idea to handle exceptions. The above can be rewritten as:

```
Socket MyClient;
try {
           MyClient = new Socket("Machine name", PortNumber);
}
catch (IOException e) {
    System.out.println(e);
}
```

If you are programming a server, then this is how you open a socket:

```
ServerSocket MyService;
try {
   MyServerice = new ServerSocket(PortNumber);
        }
        catch (IOException e) {
           System.out.println(e);
        }
```

When implementing a server you also need to create a socket object from the `ServerSocket` in order to listen for and accept connections from clients.

```
Socket clientSocket = null;
try {
   serviceSocket = MyService.accept();
        }
```

```
catch (IOException e) {
   System.out.println(e);
}
```

How do I create an input stream?

On the client side, you can use the `DataInputStream` class to create an input stream to receive response from the server:

```
DataInputStream input;
try {
   input = new DataInputStream(MyClient.getInputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

The class `DataInputStream` allows you to read lines of text and Java primitive data types in a portable way. It has methods such as `read`, `readChar`, `readInt`, `readDouble`, and `readLine`,. Use whichever function you think suits your needs depending on the type of data that you receive from the server.

On the server side, you can use `DataInputStream` to receive input from the client:

```
DataInputStream input;
try {
      input = new DataInputStream(serviceSocket.getInputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

How do I create an output stream?

On the client side, you can create an output stream to send information to the server socket using the class `PrintStream` or `DataOutputStream` of java.io:

```
PrintStream output;
try {
   output = new PrintStream(MyClient.getOutputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

The class PrintStream has methods for displaying textual representation of Java primitive data types. Its Write and println methods are important here. Also, you may want to use the `DataOutputStream`:

```
DataOutputStream output;
try {
   output = new DataOutputStream(MyClient.getOutputStream());
}
```

```
catch (IOException e) {
   System.out.println(e);
}
```

The class `DataOutputStream` allows you to write Java primitive data types; many of its methods write a single Java primitive type to the output stream. The method `writeBytes` is a useful one.

On the server side, you can use the class `PrintStream` to send information to the client.

```
PrintStream output;
try {
   output = new PrintStream(serviceSocket.getOutputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

Note: You can use the class `DataOutputStream` as mentioned above.

How do I close sockets?

You should always close the output and input stream before you close the socket.

On the client side:

```
try {
           output.close();
           input.close();
   MyClient.close();
}
catch (IOException e) {
   System.out.println(e);
}
```

On the server side:

```
try {
   output.close();
   input.close();
   serviceSocket.close();
   MyService.close();
}
catch (IOException e) {
   System.out.println(e);
}
```

Examples
Here are two applications: a simple SMTP (simple mail transfer protocol) client, and a simple echo server.

1. SMTP client

Let's write an SMTP (simple mail transfer protocol) client -- one so simple that we have all the data encapsulated within the program. You may change the code around to suit your needs. An interesting modification would be to change it so that you accept the data from the command-line argument and also get the input (the body of the message) from standard input. Try to modify it so that it behaves the same as the mail program that comes with Unix.

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

public class smtpClient {
    public static void main(String[] args) {

// declaration section:
// smtpClient: our client socket
// os: output stream
// is: input stream

        Socket smtpSocket = null;
        DataOutputStream os = null;
        DataInputStream is = null;

// Initialization section:
// Try to open a socket on port 25
// Try to open input and output streams

        try {
            smtpSocket = new Socket("hostname", 25);
            os = new DataOutputStream(smtpSocket.getOutputStream());
            is = new DataInputStream(smtpSocket.getInputStream());
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: hostname");
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to:
hostname");
        }

// If everything has been initialized then we want to write some data
// to the socket we have opened a connection to on port 25

if (smtpSocket != null && os != null && is != null) {
            try {

// The capital string before each colon has a special meaning to SMTP
// you may want to read the SMTP specification, RFC1822/3

os.writeBytes("HELO\n");
                os.writeBytes("MAIL From: k3is@fundy.csd.unbsj.ca\n");
                os.writeBytes("RCPT To: k3is@fundy.csd.unbsj.ca\n");
                os.writeBytes("DATA\n");
                os.writeBytes("From: k3is@fundy.csd.unbsj.ca\n");
                os.writeBytes("Subject: testing\n");
                os.writeBytes("Hi there\n"); // message body
                os.writeBytes("\n.\n");
os.writeBytes("QUIT");

// keep on reading from/to the socket till we receive the "Ok" from
SMTP,
// once we received that then we want to break.
```

```
                String responseLine;
                while ((responseLine = is.readLine()) != null) {
                    System.out.println("Server: " + responseLine);
                    if (responseLine.indexOf("Ok") != -1) {
                      break;
                    }
                }

// clean up:
// close the output stream
// close the input stream
// close the socket

os.close();
                is.close();
                smtpSocket.close();
            } catch (UnknownHostException e) {
                System.err.println("Trying to connect to unknown host: "
+ e);
            } catch (IOException e) {
                System.err.println("IOException:  " + e);
            }
        }
    }
}
```

2. Echo server

Now let's write a server. The echo server receives text from the client and then sends that exact text back to the client. This is just about the simplest server you can write. Note that this server handles only one client. Try to modify it to handle multiple clients using threads.

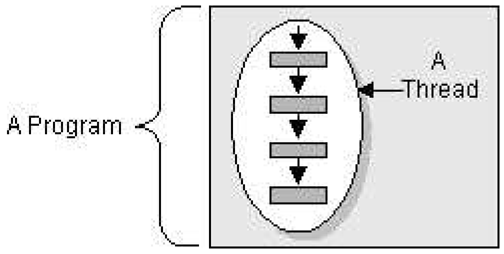
```
import java.io.*;
import java.net.*;

public class echo3 {
    public static void main(String args[]) {

// declaration section:
// declare a server socket and a client socket for the server
// declare an input and an output stream

        ServerSocket echoServer = null;
        String line;
        DataInputStream is;
        PrintStream os;
        Socket clientSocket = null;

// Try to open a server socket on port 9999
// Note that we can't choose a port less than 1023 if we are not
// privileged users (root)

        try {
           echoServer = new ServerSocket(9999);
        }
        catch (IOException e) {
           System.out.println(e);
        }

// Create a socket object from the ServerSocket to listen and accept
// connections.
// Open input and output streams

try {
           clientSocket = echoServer.accept();
           is = new DataInputStream(clientSocket.getInputStream());
           os = new PrintStream(clientSocket.getOutputStream());

// As long as we receive data, echo that data back to the client.

           while (true) {
             line = is.readLine();
             os.println(line);
           }
        }
catch (IOException e) {
           System.out.println(e);
        }
    }
}
```

3. What are threads? Discuss some new applications of threads (apart from those explained in the lecture).

Sample Answer:
To improve a program's performance, developers typically use threads. A thread--sometimes called an *execution context* or a *lightweight process*--is a single sequential flow of control within a program. You use threads to isolate tasks.

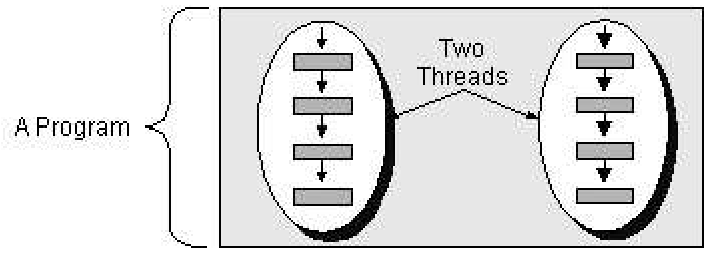
Well you are familiar with writing sequential programs. You've written a program that displays "Hello World!" or sorts a list of names or computes a list of prime numbers. These are sequential programs. That is, each has a beginning, an execution sequence, and an end. At any given time during the runtime of the program, there is a single point of execution.

A thread is similar to the sequential programs. A single thread also has a beginning, a sequence, and an end and at any given time during the runtime of the thread, there is a single point of execution. However, a thread itself is not a program; it cannot run on its own. Rather, it runs within a program. The following figure shows this relationship.



Definition: A thread is a single sequential flow of control within a program.

There is nothing new in the concept of a single thread. The real hoopla surrounding threads is not about a single sequential thread. Rather, it's about the use of multiple threads in a single program, running at the same time and performing different tasks. This is illustrated by the following figure:

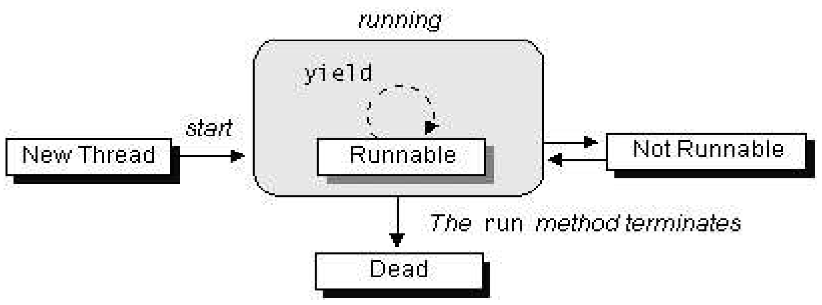


The HotJava Web browser is an example of a multithreaded application. Within the HotJava browser you can scroll a page while it's downloading an applet or image, play animation and sound concurrently, print a page in the background while you download a new page, or watch three sorting algorithms race to the finish. You are used to life operating in a concurrent fashion...so why not your browser?

Some texts use the name *lightweight process* instead of thread. A thread is similar to a real process in that a thread and a running program are both a single sequential flow of control. However, a thread is considered lightweight because it runs within the context of a full-blown program and takes advantage of the resources allocated for that program and the program's environment.

As a sequential flow of control, a thread must carve out some of its own resources within a running program. (It must have its own execution stack and program counter for example.) The code running within the thread works only within that context. Thus, some other texts use *execution context* as a synonym for thread.

The following diagram shows the states that a Java thread can be in during its life. It also illustrates which method calls cause a transition to another state. This figure is not a complete finite state diagram, but rather



an overview of the more interesting and common facets of a thread's life.

4. Discuss two methods of implementing the running behavior of Java Threads with suitable examples.

Sample Answer:

Subclassing Thread and Overriding run

The first way to customize what a thread does when it is running is to subclass Thread (itself a Runnable object) and override its empty run method so that it does something. Let's look at the SimpleThread class, the first of two classes in this example, which does just that:

```
public class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}
```
The first method in the SimpleThread class is a constructor that takes a String as its only argument. This constructor is implemented by calling a superclass constructor and is interesting to us only because it sets the Thread's name, which is used later in the program.

The next method in the SimpleThread class is the run method. The run method is the heart of any Thread and where the action of the Thread takes place. The run method of the SimpleThread class contains a for loop that iterates ten times. In each iteration the method displays the iteration number and the name of the Thread, then sleeps for a random interval of up to 1 second. After the loop has finished, the run method prints DONE! along with the name of the thread. That's it for the SimpleThread class.

The TwoThreadsDemo class provides a main method that creates two SimpleThread threads: one is named "Jamaica" and the other is named "Fiji". (If you can't decide on where to go for vacation you can use this program to help you decide--go to the island whose thread prints "DONE!" first.)

```
public class TwoThreadsDemo {
    public static void main (String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}
```
The main method also starts each thread immediately following its construction by calling the start method. Compile and run the above program and watch your vacation fate unfold. You should see output similar to the following:
```
0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
3 Fiji
3 Jamaica
4 Jamaica
4 Fiji
5 Jamaica
5 Fiji
6 Fiji
6 Jamaica
7 Jamaica
7 Fiji
8 Fiji
9 Fiji
8 Jamaica
DONE! Fiji
9 Jamaica
DONE! Jamaica
```
(Looks like I'm going to Fiji!!) Notice how the output from each thread is intermingled with the output from the other. This is because both SimpleThread threads are running concurrently. Thus, both run methods are running at the same time and each thread is displaying its output at the same time as the other.

Try This: Change the main program so that it creates a third thread with the name "Bora Bora". Compile and run the program again. Does this change the island of choice for your vacation?

Now, let's look at another example, the Clock applet, that uses the other technique for providing a run method to a Thread.

Implementing the Runnable Interface

The Clock applet shown below displays the current time and updates its display every second. The Clock applet uses a different technique than SimpleThread for providing the run method for its thread. Instead of subclassing Thread, Clock implements the Runnable interface (and therefore implements the run method defined in it). Clock then creates a thread and provides itself as an argument to the Thread's constructor. When created in this way, the Thread gets its run method from the object passed into the constructor. The code that accomplishes this is shown in bold here:

```
import java.awt.Graphics;
import java.util.*;
import java.text.DateFormat;
import java.applet.Applet;

public class Clock extends Applet implements Runnable {
    private Thread clockThread = null;
    public void start() {
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock");
            clockThread.start();
        }
    }
    public void run() {
        Thread myThread = Thread.currentThread();
        while (clockThread == myThread) {
            repaint();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e){
            // the VM doesn't want us to sleep anymore,
            // so get back to work
            }
        }
    }
    public void paint(Graphics g) {
        // get the time and convert it to a date
        Calendar cal = Calendar.getInstance();
        Date date = cal.getTime();
        // format it and display it
        DateFormat dateFormatter = DateFormat.getTimeInstance();
        g.drawString(dateFormatter.format(date), 5, 10);
    }
    // overrides Applet's stop method, not Thread's
    public void stop() {
        clockThread = null;
    }
}
```

The Clock applet's run method loops until the browser asks it to stop. In each iteration of the loop, the clock repaints its display. The paint method figures out what time it is, formats it in a localized way, and displays it.

Deciding to Use the Runnable Interface

You have now seen two ways to provide the run method for a Java thread:

1. Subclass the Thread class defined in the java.lang package and override the run method.

2. Provide a class that implements the Runnable interface (also defined in the java.lang package) and therefore implements the run method. In this case, a Runnable object provides the run method to the thread.

There are good reasons for choosing either of these options over the other. However, for most cases, including that of the Clock applet, the following rule of thumb will guide you to the best option.

Rule of Thumb: If your class must subclass some other class (the most common example being Applet), you should use Runnable as described in option #2.

To run in a Java-enabled browser, the Clock class has to be a subclass of the Applet class. Also, the Clock applet needs a thread so that it can continuously update its display without taking over the process in which it is running. (Some browsers might create a new thread for each applet so as to prevent a misbehaved applet from taking over the main browser thread. However, you should not count on this when writing your applets; your applets should create their own threads when doing computer-intensive work.) But since the Java language does not support multiple class inheritance, the Clock class cannot be a subclass of both Thread and Applet. Thus the Clock class must use the Runnable interface to provide its threaded behavior.