Network Computing Lecture 5

Professor Louise E. Moser

Spring 2012

Java Threads and Java Sockets

- A **thread** is a **sequential** flow of control within a program, sometimes called an **execution context** or a **lightweight process**
- Several threads can run concurrently. They will actually be interleaved, but the appearance is that they run in parallel
- Each thread can be programmed as a sequential program, but care is required when several threads use the same data
- For network programming, the server needs to use threads to be able to serve multiple clients efficiently

Threads in Java

- The *Thread* class implements a generic thread that does nothing
  - The implementation of its *run()* method is empty
- The *run()* method defines the behavior of a thread
- Its code implements the thread's running behavior
- Threads that do nothing are not particularly useful
- There are two ways of providing a *run()* method for a thread
  - You can subclass the *Thread* class and override the *run()* method
  - You can implement the *Runnable* interface of the *Thread* class for a runnable object

Subclassing the Thread Class

public class ***SimpleThread extends Thread*** *{*

- The example *SimpleThread* class on the next slide **extends** *Thread*
- It has a *SimpleThread* constructor
  - The constructor takes a *String* argument
  - The constructor calls the superclass constructor *Thread*, which sets the thread's name to the string
- The only method in the *SimpleThread* class is the *run()* method
  - The *run()* method contains a *for* loop that iterates 10 times
  - In each iteration, the *run()* method displays the iteration number and the name of the thread, and then sleeps for a random amount of time up to 1 second
  - After the loop has finished, the *run()* method prints *DONE!* along with the name of the thread
- The *main()* method creates two copies of *SimpleThread* using *new* and starts them executing by invoking their *start()* method

Simple Thread Example

```
public class SimpleThread extends Thread {
    public SimpleThread(String str) { // Constructor
        super(str); // Initialize the name of the thread
    }
    public void run() { // Override the run method
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((int)(Math.random() * 1000));
            }
            catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}
public class TwoThreadsTest {
    public static void main (String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}
```

Interleaving Example

- An example of interleaving
  the operations of the two threads
  Fiji and Jamaica

0 Fiji
1 Fiji
0 Jamaica
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

Implementing the Runnable Interface

public class Clock ***extends JApplet***
implements Runnable *{*

- The example *Clock* class on slides 9-10 **extends** *JApplet*
- It cannot also extend *Thread;* instead, it **implements** the *Runnable* interface
- *Clock* declares a private *Thread* variable, *clockThread*
- *Clock* implements the *run()* method of the *Runnable* interface

Implementing the Runnable Interface

- *Clock* implements the methods, *start()* and *stop()*
- When *start()* is called, it
  - Creates a new *Thread* variable, *clockThread*
  - Binds that thread to *this* class, *Clock*

```
clockThread = new Thread(this, "Clock");
```

- It then invokes *clockThread.start()* to start the thread
- Because the new thread is bound to the *Clock* class, the *start()* method of *clockThread* invokes the *run()* method of *Clock* to run the thread

Clock Example

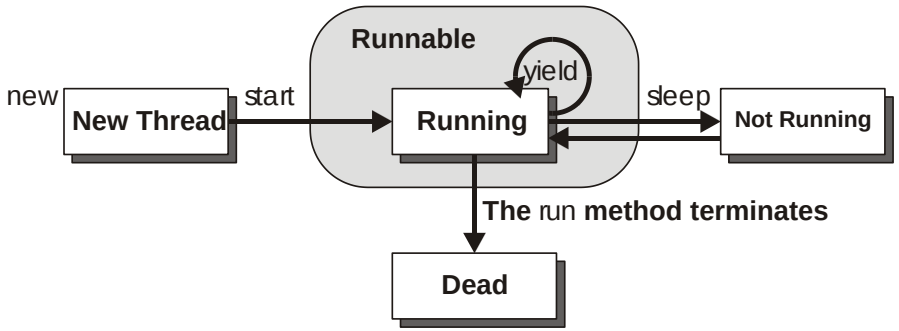
```
import java.awt.Graphics;
import java.util.*;
import java.text.DateFormat;

public class Clock  extends JApplet  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 {
            myThread.sleep(1000);
         }
         catch (InterruptedException e){ }
   }   }
```

Clock Example (cont)

```
    public void paint(Graphics g) {
        Calendar cal = Calendar.getInstance();
        Date date = cal.getTime();
        DateFormat dateFormatter = DateFormat.getTimeInstance();
        g.drawString(dateFormatter.format(date), 5, 10);
    }
    public void stop() {
        clockThread = null;
    }
}
```

The Lifecycle of a Thread

- The *new()* operation creates the thread
- The *start()* method sets the thread's *run()* method running
- There may be several runnable threads. The JVM determines which one is actually running. The *yield* operation allows a long running thread let other threads run
- The thread dies when the *run()* method terminates
- A thread can become temporarily not running when
  - It invokes its *sleep()* method
  - It calls a *wait()* method to wait on a specific condition
  - It is blocked on I/O



Synchronizing Threads

- Most useful threads interact with other threads
  - If two or more threads both use the same data, we cannot allow them to access the data whenever they want
- There are **critical sections** of program code such that
  - When one thread is in the critical section, manipulating data, other threads must be prevented from entering the critical section and mucking up the data
- Java provides a means of synchronizing threads, called **monitors**

Producer-Consumer Example

- There are two threads, ***Producer*** and ***Consumer***
  - ***Producer*** generates data and puts it in a shared object, the *cubbyhole* monitor
  - ***Consumer*** extracts data from the *cubbyhole* monitor and prints it
- The essential code is:

```
public void run() {  // part of Producer
  for (int i = 0; i < 10; i++) {
      cubbyhole.put(i);
...
}}
public void run() {  // part of Consumer
  int value = 0;
  for (int i = 0; i < 10; i++) {
      value = cubbyhole.get();
      System.out.println("Consumer #" + this.number + " got: " + value);
}}
```

- There is no synchronization in the code of the *Producer* or *Consumer*
  - All of the synchronization is in the code of the *cubbyhole* monitor

Synchronized Methods

- The ***cubbyhole*** monitor declares two methods ***get()*** and ***put()***
  - These are declared to be **synchronized**, so that if a thread is in one of the methods, no other thread can enter the other method

```
public synchronized int get() { // Simplified!
   available = false;
   return contents;
}

public synchronized void put(int value) { // Simplified!
   contents = value;
   available = true;
}
```

- But what if the consumer invokes *get()* before the producer invokes *put()* to put data into the *cubbyhole*?

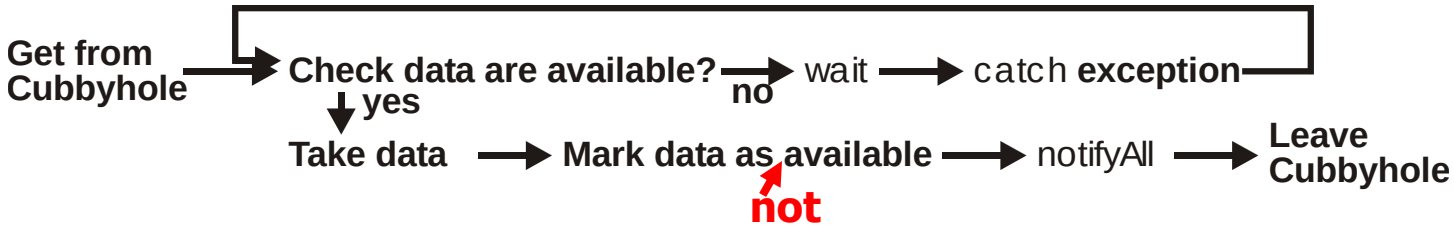
Synchronized Methods (cont)

```
public synchronized int get() { // Enter synchronized method get()
  while (available == false) {  // Test if data are available
    try {
      wait();                   //  If not, consumer waits
    }
    catch (InterruptedException e) { } // Wakeup
  }                                    // Go back to test again
  available = false;                   // Take data, set available to false
  notifyAll();                         // Notify producer if it is waiting
  return contents;                     // Exit from get()
}
```

- The *put()* method is similar

Waiting outside the Monitor

- The ***wait()*** method allows a thread to wait (outside the monitor) if data are not yet available for it
  - A waiting thread does not block other threads from entering the monitor
- The ***notifyAll()*** method allows a thread, when it has set the data, to wake up other threads that are waiting
- The waiting thread wakes up by catching ***InterruptedException***
  - It should then check to make certain that the data are indeed available



Producer

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" + this.number + " put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            }
            catch (InterruptedException e) { }
}   }  }
```

Consumer

```
public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.number + " got: " + value);
} } }
```

CubbyHole

```
public class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        while (available == false) {
            try {
                wait();
            }
            catch (InterruptedException e) { }
        }
        available = false;
        notifyAll();
        return contents;
    }
```

CubbyHole (cont)

```
    public synchronized void put(int value) {
        while (available == true) {
            try {
                wait();
            }
            catch (InterruptedException e) { }
        }
        contents = value;
        available = true;
        notifyAll();
    }
}
```

Using Threads to Serve Multiple Clients

- ***MultiJabberServer*** is an echo server that can serve several clients simultaneously
- It consists of two parts:
  - *ServeOneJabber* is a **thread** that serves a **single client** and echoes messages back to it
  - *MultiJabberServer* is a **server** that waits on its public server port for new clients to connect to it
    - For each such client, it obtains a new *socket* for the connection to the client and spawns a new *ServeOneJabber* thread to serve the new client

The main() Method of MultiJabberServer

- The ***main()*** method opens a ***ServerSocket s*** on port 8080
- It then enters an infinite loop waiting for clients to connect

```
while(true) {
  Socket socket = s.accept();
  try {
    new ServeOneJabber(socket);
  }
  catch(IOException e) {
```

- Note that *s.accept()* blocks until a client opens a connection
- Then, *s.accept()* returns the **socket** allocated to that client
- Then, the server creates a new *ServeOneJabber* **thread** and initializes it with the new socket

MultiJabber Server

```
public class MultiJabberServer {
  static final int PORT = 8080;
  public static void main(String[] args)
      throws IOException {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Server Started");
    try {
      while(true) {
        // Blocks until a connection occurs
        Socket socket = s.accept();
        try {
          new ServeOneJabber(socket);
        }
        catch(IOException e) {
          // If it fails, close the socket;
          // otherwise, the thread will close it
          socket.close();
      } }
    } finally {
      s.close();
} } }
```

Declaring the ServeOneJabber Thread

- The *ServerOneJabber* thread extends the *Thread* class and overwrites its *run()* method

```
class ServeOneJabber extends Thread {
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;
```

- Note the *BufferedReader* for the input and the *PrintWriter* for the output

Initializing the ServeOneJabber Thread

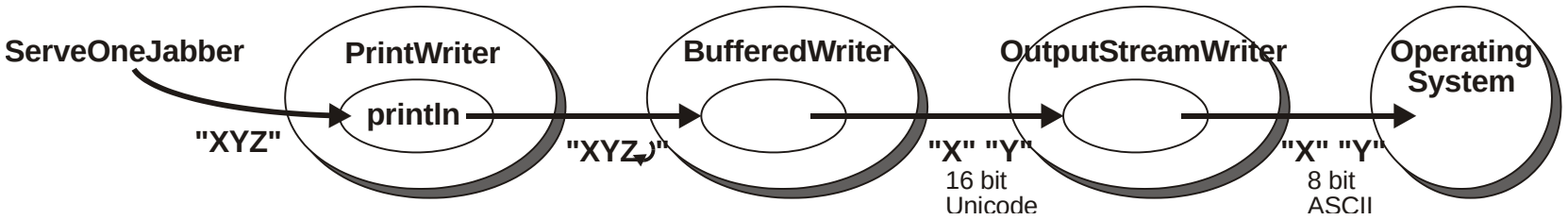
```
public ServeOneJabber(Socket s) // Constructor
    throws IOException {
     socket = s;
     in = new BufferedReader(
                  new InputStreamReader(socket.getInputStream()));
    out = new PrintWriter(
                  new BufferedWriter(
                       new OutputStreamWriter(socket.getOutputStream())), true);
    start(); // Calls the run() method
}
```

- *s* is used to initialize the *socket* connecting to the client, and to open input and output streams to the client
- The *true* parameter of *PrintWriter* tells it to flush the output buffers for each *println* operation
- *start()* launches the *run()* method

Initializing ServeOneJabber (cont)

out=new PrintWriter(
new BufferedWriter(
new OutputStreamWriter(...)

- When *PrintWriter* produces its output, it sends the output to *BufferedWriter*
- When *BufferedWriter* produces its output, it sends the output to *OutputStreamWriter*
- When *OutputStreamWriter* produces its output, it sends the output to *out*



The run() Method of ServeOneJabber

```
public void run() {
  try {
    while (true) {
      String str = in.readLine();
      if (str.equals("END")) break;
      ....
      out.println(str);
    }
  }
 catch ...
```

- In an infinite loop, the thread reads strings from the input stream and sends them to the output stream, until it finds an "*END*" string

ServeOneJabber

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

class ServeOneJabber extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;

  public ServeOneJabber(Socket s) // Constructor
      throws IOException {
    socket = s;
    in = new BufferedReader(
              new InputStreamReader(socket.getInputStream()));
    // Enable auto-flush:
    out = new PrintWriter(
                new BufferedWriter(
                      new OutputStreamWriter(socket.getOutputStream())), true);
    // If any of the above calls throw an exception, the caller is responsible for
    // closing the socket; otherwise, the thread will close it.
    start(); // Calls run()
  }
```

ServeOneJabber (cont)

```
public void run() {
   try {
     while (true) {
       String str = in.readLine();
       if (str.equals("END")) break;
       System.out.println("Echoing: " + str);
       out.println(str);
     }
     System.out.println("closing...");
   }
   catch (IOException e) {
   }
  finally {
     try {
       socket.close();
     }
     catch(IOException e) { }
} } }
```