A Short Introduction to Multi-threading in Java for CS421 Projects

Alper Rifat Uluçınar
CS421 Course Assistant

Bilkent University
Computer Engineering Dep.

2011-12-21

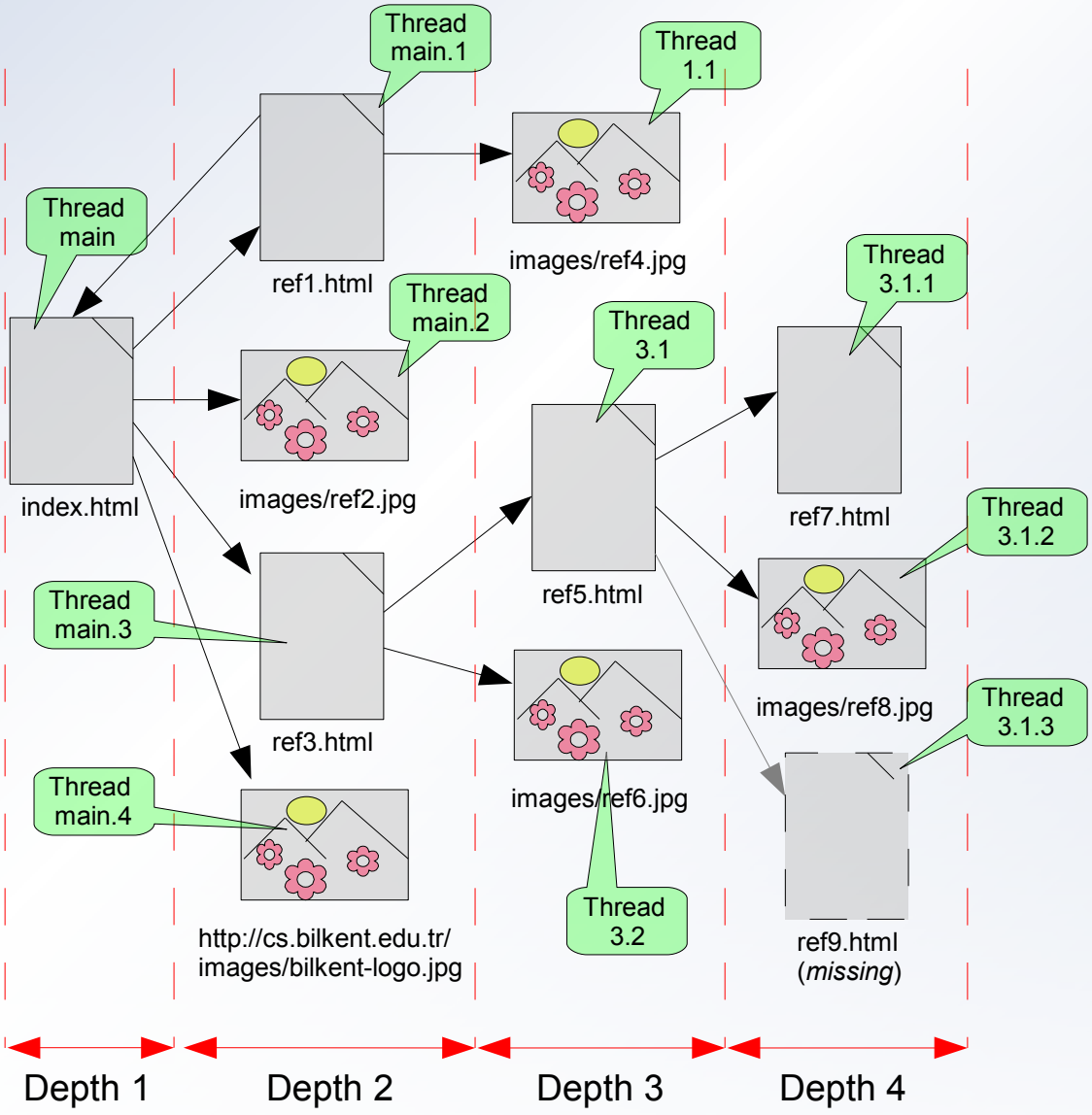
Tutorial Outline

- Programming Assignment #2 Discussion
- Concurrent Programming
- Processes & Threads
- Networked Server Example
- Java Threads
- Networked Server Example – Revisited
- Thread Interference
- Thread Synchronization

Discussion on the 2nd Programming Assignment

- Multi-threaded HTTP *crawler* in Java
  i. Download the root URL and store the file on disk in the working dir of your program
     - The root URL will be **the first command-line arg**
  ii. Parse the downloaded HTML file & find references to other resources
  iii. For each referenced resource, start a new thread to download, parse (if necessary) & store on disk that resource
- Your program should crawl up to a specified depth
  - Max depth will be **the second command-line arg**
  - Root HTML file is at depth 1
- `java Crawler <root_HTML_file_URL> <max_depth>`

Discussion on the 2nd Programming Assignment - Example Crawling Scenario

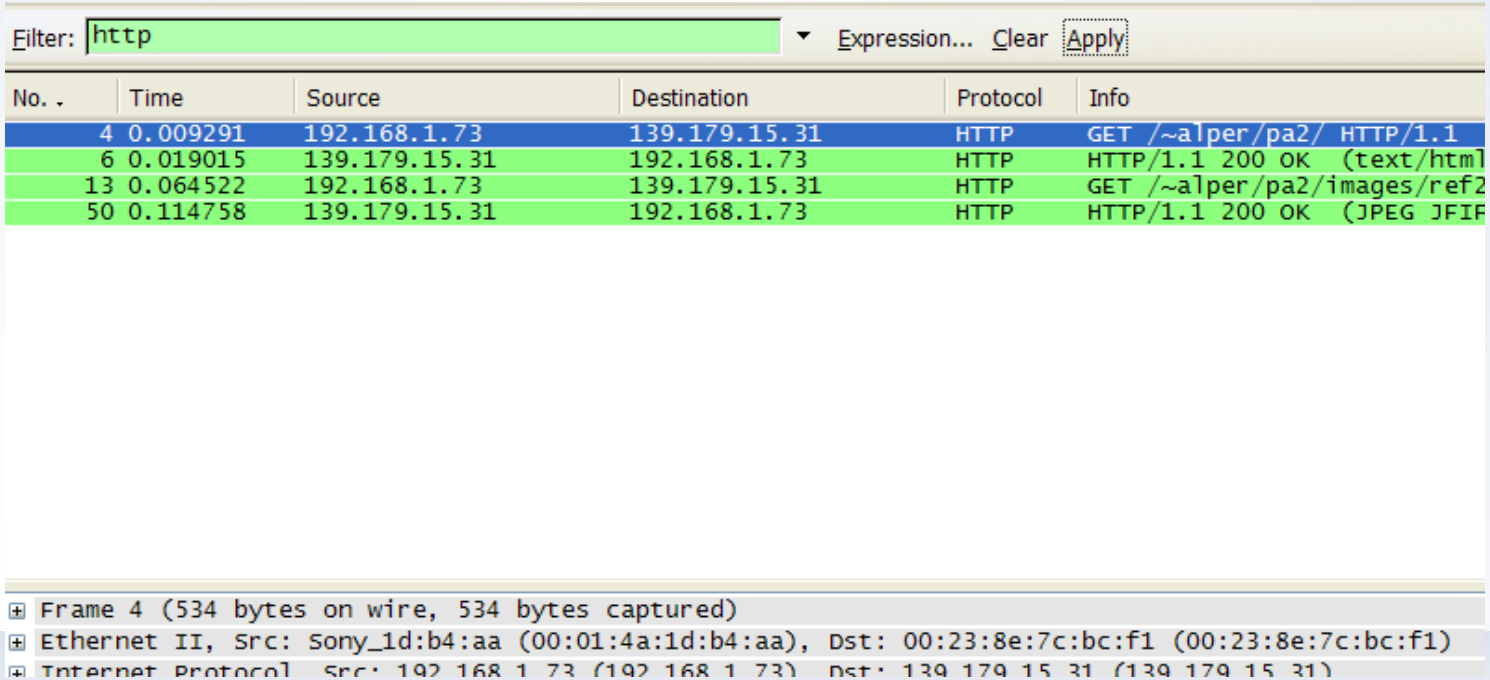


Discussion on the 2nd Programming Assignment - About URLs

- *U*niform *R*esource *L*ocator
  - RFC 1738 (T. Berners-Lee, L. Masinter, M. McCahill, December 1994), RFC 2396, RFC 2732
  - **Where** a resource precisely is?
  - **How** to retrieve it?
- http://java.sun.com:80/reference/docs/index.html
  - *Protocol:* http
  - *Host:* java.sun.com
  - *Port:* 80
  - *Path:* /reference/docs/index.html
  - *File Name:* index.html
- No need to specify default port # of the protocol [80 for HTTP]
  - http://java.sun.com/reference/docs/index.html

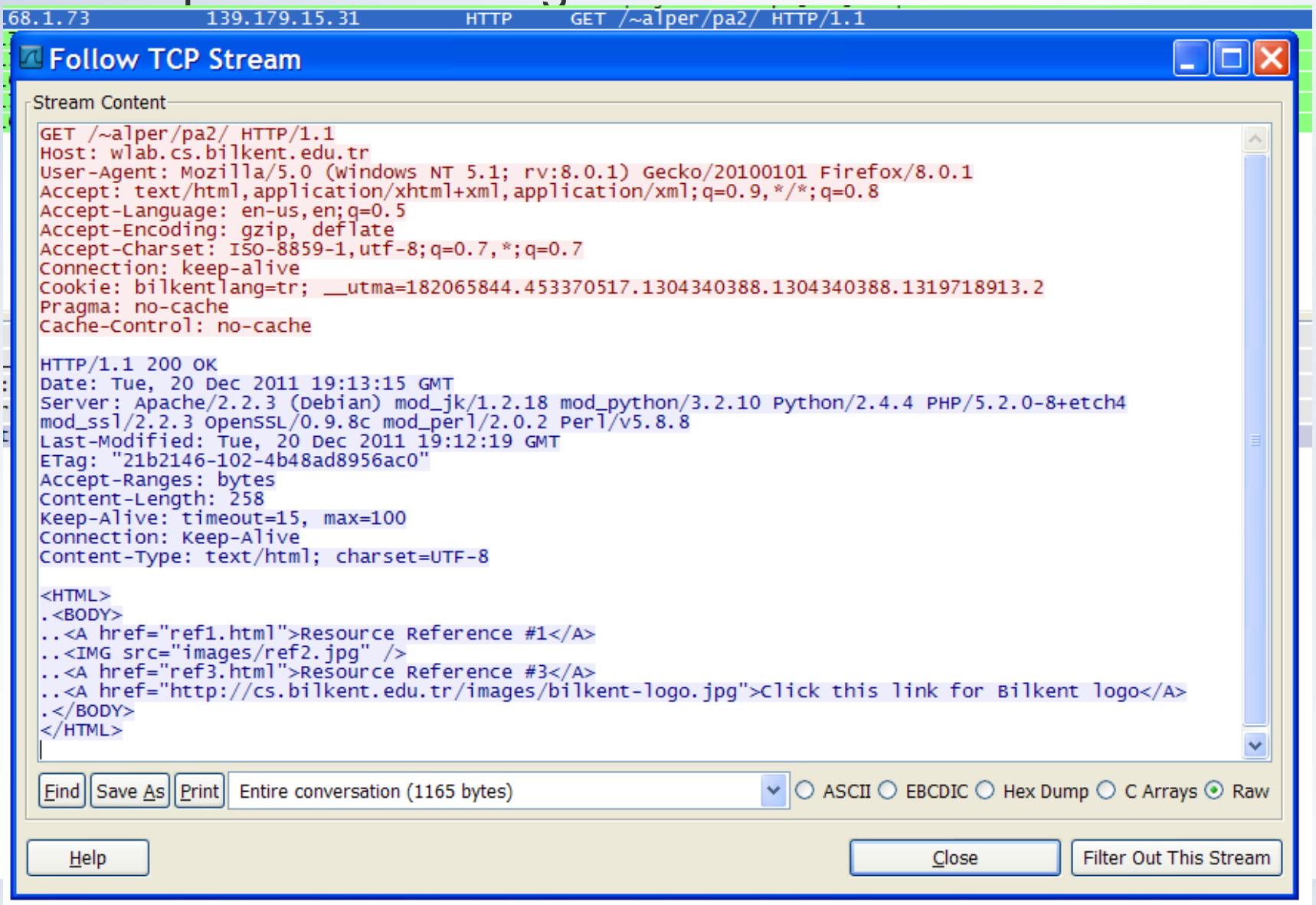
Analyzing an HTTP Request

- See the protocol working with Wireshark
  - Start capturing with Wireshark on the correct network interface
  - Request
    http://wlab.cs.bilkent.edu.tr/~alper/pa2/index.html
    with your browser
  - Filter Wireshark packet view with the string `http`



Analyzing an HTTP Request

- See the protocol working with Wireshark



Making an HTTP Request with Java

- Request headers

```
⊞ Frame 15 (534 bytes on wire, 534 bytes captured)
⊞ Ethernet II, Src: Sony_1d:b4:aa (00:01:4a:1d:b4:aa), Dst: 00:23:8e:7c:bc:f1 (00:23:8e:7c:bc:f1)
⊞ Internet Protocol, Src: 192.168.1.73 (192.168.1.73), Dst: 139.179.15.31 (139.179.15.31)
⊞ Transmission Control Protocol, Src Port: unisql-java (1979), Dst Port: http (80), Seq: 1, Ack: 1, Len: 480
⊟ Hypertext Transfer Protocol
  ⊞ GET /~alper/pa2/ HTTP/1.1\r\n
    Host: wlab.cs.bilkent.edu.tr\r\n
    User-Agent: Mozilla/5.0 (Windows NT 5.1; rv:8.0.1) Gecko/20100101 Firefox/8.0.1\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
    Accept-Language: en-us,en;q=0.5\r\n
    Accept-Encoding: gzip, deflate\r\n
    Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
    Connection: keep-alive\r\n
    Cookie: bilkentlang=tr; __utma=182065844.453370517.1304340388.1304340388.1319718913.2\r\n
    Pragma: no-cache\r\n
    Cache-Control: no-cache\r\n
    \r\n
```

- Not all headers are mandatory
  - First line and the **Host** header field suffices for HTTP/1.1

Making an HTTP Request with Java

- How to request?

```
15          String host = "wlab.cs.bilkent.edu.tr";
16          int port = 80;
17          Socket clientSock = new Socket( host, port );
18
19          /* Making request */
20          BufferedWriter writer = new BufferedWriter(
21                                      new OutputStreamWriter(
22                                          clientSock.getOutputStream() ) );
23
24          writer.write( "GET /~alper/pa2/index.html HTTP/1.1\r\n" );
25          writer.write( "Host: wlab.cs.bilkent.edu.tr\r\n" );
26          writer.write( "Connection: close\r\n" );
27          writer.write( "\r\n" );
28
29          writer.flush();
```

Downloading a Text Resource via HTTP

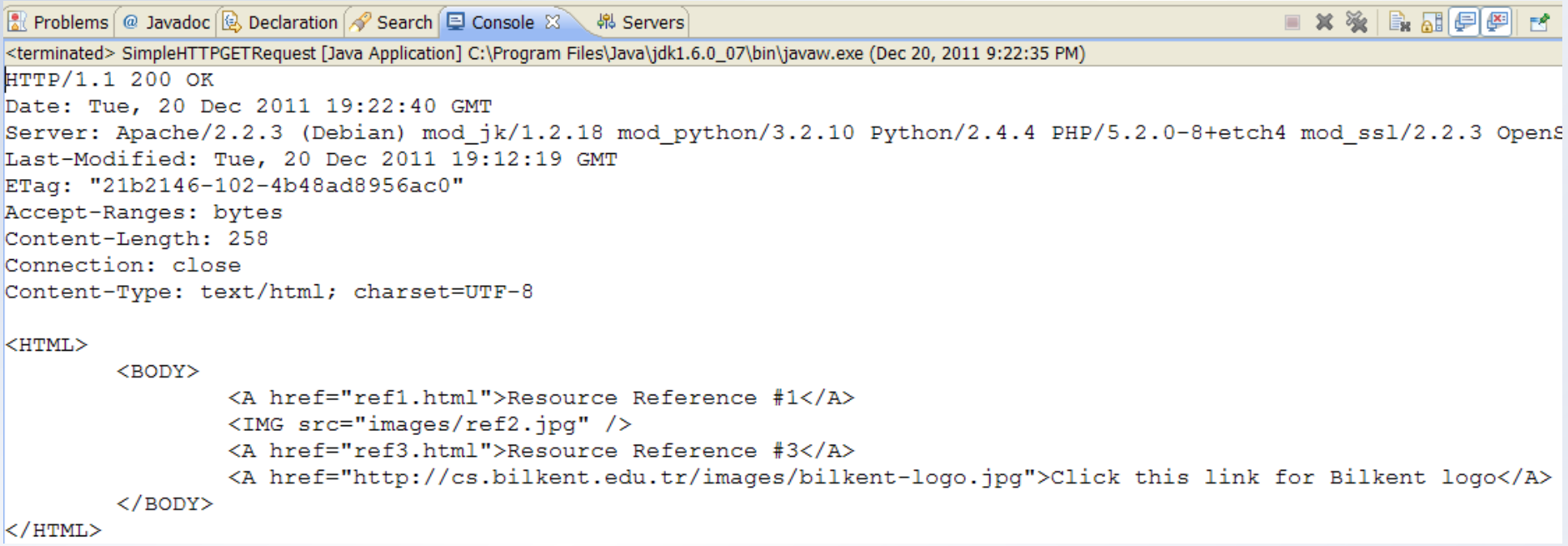
- How to read a text resource?

```
30
31          /* Reading text resource */
32          BufferedReader reader = new BufferedReader(
33                                      new InputStreamReader(
34                                          clientSock.getInputStream() ) );
35          String line = reader.readLine();
36
37          while( line != null )
38          {
39              System.out.println( line );
40              line = reader.readLine();
41          }
42
43          clientSock.close();
```

- Don't forget to close your sockets, streams, etc.!!!
- Code relies on the **Connection: close** request header!!!

Downloading a Text Resource via HTTP

- Sample Output



- Don't forget to check the returned HTTP status code!
- Please handle at least 404 (not found)...

Concurrent Programming

- Multiple tasks to be performed in *parallel*
  - Read your e-mails while listening to your MP3 play list
  - Or in the context of a single application:
    - An HTTP server can serve multiple clients in parallel (simultaneously)
    - Your web browser fetches & renders the embedded images in an HTML file in parallel
- Two basic units of execution in concurrent software
  - Processes
  - Threads

Processes & Threads

- Process
  - Executable program loaded in memory
  - Owns a private set of run-time resources
    - Address space
    - IPC resources (pipes, shared memory segments, etc.)
    - Open disk files
    - Sockets
    - …
  - Processes may communicate with each other via IPC resources

Processes & Threads

- Thread
  - Sometimes called lightweight processes
    - In general, fewer OS resources are needed for a new thread than for a new process
  - Threads exist within a process
    - Every process has at least one (i.e. *main thread*)
    - Threads share the parent process's resources
      - Address space, open files, sockets, etc.
    - => Threads of the same process can communicate more efficiently (via shared address space) but this convenience comes with a price!
    - Programmer needs to take care of race conditions, deadlocks, etc. (problems common to concurrent programming)
- JRE (and Java API) support concurrent programming

A Motivating Networked Server Example

- Fibonacci numbers
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
  - $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$
- Fibonacci Server
  - Implemented over TCP
  - Listens to port 1234
  - A client connects to Fibonacci server and requests the Fibonacci number at a specific index
  - Server responds with the Fibonacci number at that index

Client Server

3 --->

<--- 2

A Motivating Networked Server Example – Single Threaded Fibonacci Server

```
ServerSocket sSock = new ServerSocket( 1234, 1 );

while( this.serverState == FibonacciServer.state.STATE_RUNNING )
{
     Socket sock = sSock.accept();
     BufferedReader reader = new BufferedReader(new
                                InputStreamReader(sock.getInputStream(),"ASCII"));
     int order = Integer.parseInt(reader.readLine());
     PrintWriter writer = new PrintWriter( new
                                OutputStreamWriter(sock.getOutputStream(),"ASCII" ));

     if(order >= 0)
          writer.println(this.fibonacciAt(order));

     writer.close();
     reader.close();
     sock.close();
}

sSock.close();
```

A Motivating Networked Server Example – Single Threaded Fibonacci Server

- `sSock.accept()` call blocks until an incoming connection request is received
- After a client is connected, the server is busy with servicing the client (reading the index, calculating the Fibonacci number, writing the response, …)
- All these tasks are handled by a single thread *(main thread)*
- What happens when a **second** client tries to connect?
- What happens when a **third** client tries to connect [ServerSocket's backlog parameter is 1]?

A Motivating Networked Server Example – Single Threaded Fibonacci Server

- While the single (main) thread of the server is busy servicing the first client:
  - The connection request of the second client is put into the “connection indication queue”
  - Since the backlog parameter passed to ServerSocket's constructor is 1, the third client's connection request is rejected
- Single threaded Fibonacci server can
  - Service at most one client at a time
  - While a client is being serviced, new connection requests will be put into a queue to be further processed
  - If the queue is full, further connection requests will be rejected

Java Threads

- How to code Fibonacci server so that it can handle multiple requests in parallel (concurrently)
- One solution: Use threads!
  - Accept a connection request
  - Instantiate a new thread that will handle the client's request
    - That will read the index from the client
    - That will calculate the Fibonacci number at that index
    - That will respond with the Fibonacci number to the client
    - That will then terminate

Java Threads

- How to make use of threads in Java?
  - Through `java.lang.Thread` API...
  - Instantiate a new `Thread` object and provide the code that will run in that thread
- 2 ways of doing this:
  - Provide a `java.lang.Runnable` object to `Thread` constructor:

```
public class MyJob implements Runnable
{
      public void run()
      {
            … // work for thread
      }
}

Thread t = new Thread(new MyJob()); // instantiate thread
t.start();                          // begin executing thread
…                                   // thread executing in parallel
```

Java Threads

- Subclass `Thread`:
  - The `Thread` class itself implements `Runnable` interface though its run method performs nothing...

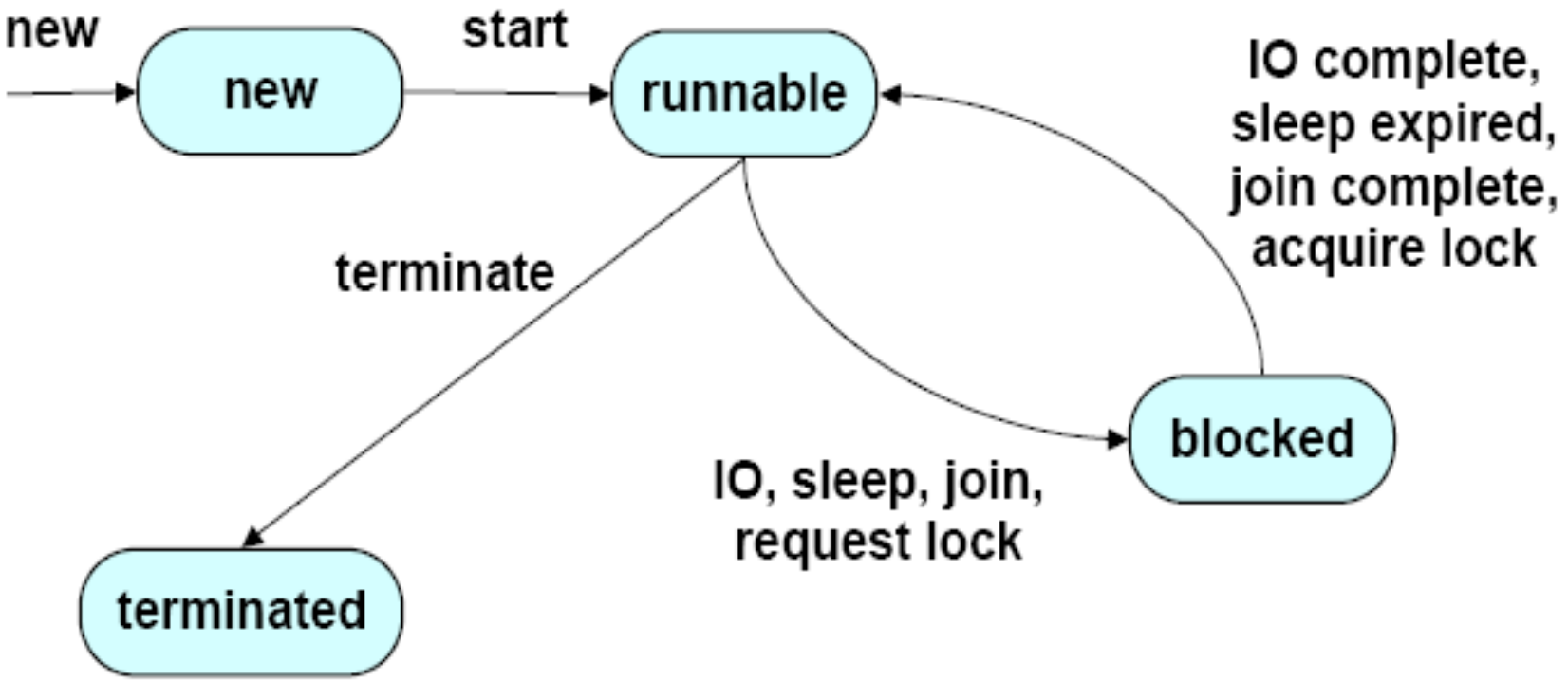
```
public class MyThread extends Thread
{
      public void run()
      {
            … // work for thread
      }
}

MyThread t = new MyThread();      // instantiate thread
t.start();                        // start thread
…                 // thread executing in parallel
```

- Please note that both examples call `Thread.start()` to start the execution of the new thread

Thread State Diagram

- State diagram



A Motivating Networked Server Example – Multi-threaded Fibonacci Server

```
ServerSocket sSock = new ServerSocket(1234, 1);

while(this.serverState == FibonacciServer.state.STATE_RUNNING)
{
     Socket sock = sSock.accept();
     ServerJob job = new ServerJob(sock);
     Thread t = new Thread(job);

     t.start();
}

sSock.close();
```

A Motivating Networked Server Example – Multi-threaded Fibonacci Server

```
public class ServerJob implements Runnable {
     private Socket sock;
     public ServerJob( Socket sock ) {
          this.sock = sock;
     }
     public void run() {
          BufferedReader reader = new BufferedReader(new
                                  InputStreamReader(sock.getInputStream(),"ASCII"));
          int order = Integer.parseInt(reader.readLine());
          PrintWriter writer = new PrintWriter( new
                                  OutputStreamWriter(sock.getOutputStream(),"ASCII" ));

          if(order >= 0)
               writer.println(this.fibonacciAt(order));

          writer.close();
          reader.close();
          this.sock.close();
     }
}
```

A Motivating Networked Server Example – Multi-threaded Fibonacci Server

- Each client is served in a separate thread
  - Main thread instantiates a new `Thread` object for each request
  - Main thread then calls the `start()` method to begin the execution of the new thread
  - The new thread begins execution of the `run()` method of `ServerJob` class
  - The new thread reads the index, calculates the Fibonacci number and writes the response
  - After servicing the client, the thread terminates (by returning from the `run()` method)

Thread Interference

- Happens when 2 operations, running in different threads, but acting on the same data, *interleave*
- For such operations to interleave, they should consist of multiple steps => The sequences of steps overlap
- Interference example: Assume threads **A** and **B** concurrently want to increment the variable **c** by **1**. The initial value of c is 0:
  - **Thread A**: Retrieve c [initial value is 0]
  - **Thread B**: Retrieve c [initial value is 0]
  - **Thread A**: Add 1 to retrieved value [result is 1]
  - **Thread B**: Add 1 to retrieved value [result is 1]
  - **Thread B**: Store the new value in c [c is now 1]
  - **Thread A**: Store the new value in c [c is now 1]

Thread Interference

- Since the variable c was intended to be incremented by 1 twice (by threads A and B), the expected final value of c is 2.
- However because of the *race condition,* the final value of c turned out to be 1. It could also have been 2 according to the *interleaving* order of the threads.
- Let's observe thread interference on real Java threads

Thread Interference – A reliable and an unreliable bank

- A single bank account with an initial balance of 1000000 tl
- 100 clients working in parallel on this same account
- Each client performs a total of 200 random withdraw and deposit operations on the account
  - A client first determines a random deposit value
  - He then performs the deposit operation
  - Immediately after the deposit operation, he withdraws the same amount
- So the balance of the account is expected NOT TO change after the total of 100x200 = 20000 deposit and withdraw operations
- Let's observe what happens if the clients (threads) aren't *synchronized*

Thread Synchronization

- To prevent thread interference and memory consistency errors, Java provides 2 basic synchronization mechanisms
  - Synchronized *methods*
  - Synchronized *statements*
- The programmer is responsible for taking care of synchronization requirements among multiple threads
- Failure to do so might result in run-time errors that are rather difficult to spot
  - On one platform, the error(s) might never show up but on a different platform (with multiple cores for instance), it might always occur

Thread Synchronization – Synchronized Methods

- To make a method synchronized, add the `synchronized` keyword to the method declaration:

```
public synchronized void
withdraw( int amount )
{
    this.balance -= amount;
}

public synchronized void
deposit( int amount )
{
    this.balance += amount;
}

public synchronized int getBalance()
{
    return this.balance;
}
```

Thread Synchronization – Synchronized Methods

- Making a method synchronized has two effects:
  - It's not possible for two invocations (by two concurrent threads) of synchronized methods on the same object to *interleave*
    - A synchronized instance method's execution cannot be interleaved by the execution of another synchronized instance method on the same object
  - Since no other synchronized methods can be executed by other threads while a thread is executing a synchronized method of an object, when the thread returns from a synchronized method, all other subsequent synchronized method invocations (by the other threads) are guaranteed to see a consistent object state

Thread Synchronization – Synchronized Statements

- Another mechanism for providing synchronized blocks of code is to use synchronized statements:

```
private Account anAccount = new Account(1000000);
...

/* Synchronized code section starts here... */
synchronized(anAccount)
{
   anAccount.deposit(100);
}
/* Synchronized code section ends here... */

// code that goes here is NOT synchronized
.
.
.
```

Thread Synchronization – Synchronized Statements

- Synchronized statements provide a fine-grained mechanism for synchronization
  - A whole method body is not synchronized
  - Instead, only the body of the synchronized statement is synchronized
  - Fine-grained synchronization can improve concurrency

Thread Interference – A reliable and an unreliable bank

- Since the same account object is shared & **updated** by multiple concurrent clients (threads), read/write accesses to it has to be synchronized
  - So that thread interference won't happen if the execution of the threads interleave
- Now, let's see whether properly synchronized code solves the interference problem...

References

- The Java Tutorials, online at http://java.sun.com/docs/books/tutorial/
- Java Platform SE 6 API Specification, online at http://java.sun.com/javase/6/docs/api/
- Openoffice template used to prepare this presentation is Ooo2, available online from http://technology.chtsai.org/impress/
- www.cs.umd.edu/class/spring2006/cmsc132/Slides/lec34.pdf