A Short Tutorial for Implementing CS421 Projects in Java

Alper Rifat Uluçınar
CS421 Course Assistant

Bilkent University
Computer Engineering Dep.

2010-11-02

Tutorial Outline

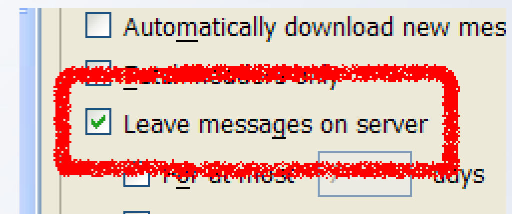
- Assignment Overview
- Analyzing POP3 & SMTP with Wireshark
- Related POP3 Commands
- Structure of a Simple POP3 Client in Java
- Related SMTP Commands
- Command-line Arguments
- Java Exceptions
- Choose an IDE
- Debugging with Eclipse
- Java Sockets
- References

Programming Assignment #1

- What you are asked for in short?
  - To implement a POP3 & SMTP client in Java:
    - No GUI... => Command-Line Interface (CLI)
    - Basic functionality:
      - For POP3: Authentication (**USER/PASS**), Listing the messages in inbox (**LIST**), Retrieve a message (**RETR**), Terminate the session (**QUIT**), Refresh inbox listing
      - For SMTP: Compose a new message and have it delivered (**HELO**, **MAIL FROM**, **RCPT TO**, **DATA**)
    - Socket API is allowed only
    - No 3rd party libraries
  - Prepare a README file
  - Submit your project as a zip archive

Analyzing a POP3 Session With Wireshark

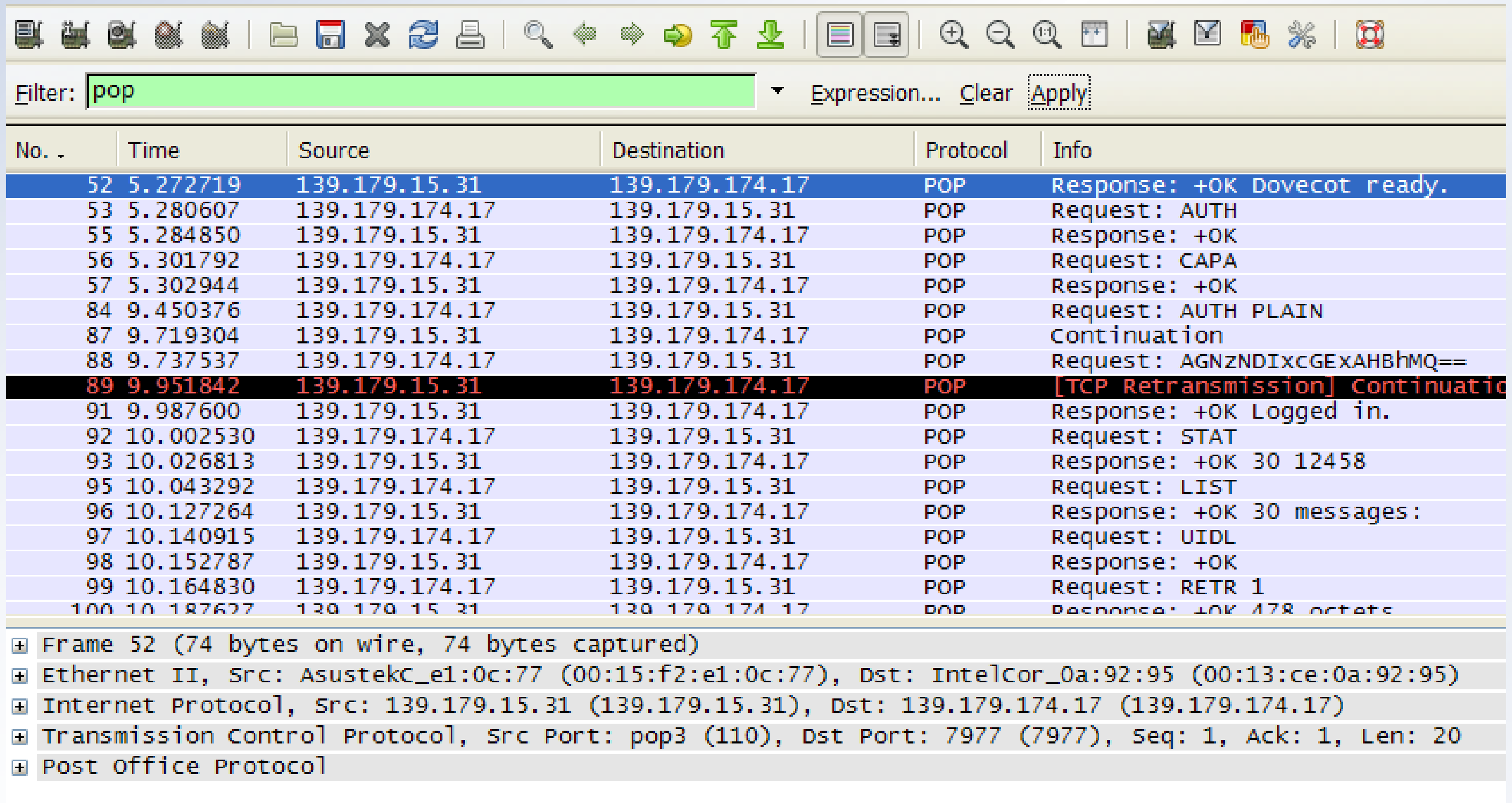
- Configure a POP3 client (Thunderbird, Outlook Express, etc.) with the following information:
  - **POP3** server: wlab.cs.bilkent.edu.tr
  - Port: 110
  - E-mail account: cs421pa1@test.com
  - Account user name: cs421pa1
  - Password: pa1
  - No transport layer security
  - Make sure **to leave messages on server unconditionally**



- Start capturing with Wireshark on the correct network interface (no need for promiscuous mode)
- Fetch e-mail with your client
- Stop capturing of network packets

Analyzing a POP3 Session With Wireshark

- Filter Wireshark packet view with the filter string `pop`



Analyzing a POP3 Session With Wireshark

- Right click on a TCP segment belonging to the POP3 session and on the context menu that appears, click “Follow TCP stream”

Filter: pop Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Info
52	5.272719	139.179.15.31		POP	Response: +OK Dovecot ready.
53	5.280607	139.179.174.17		POP	Request: AUTH
55	5.284850	139.179.15.31		POP	Response: +OK
56	5.301792	139.179.174.17		POP	Request: CAPA
57	5.302944	139.179.15.31		POP	Response: +OK
84	9.450376	139.179.174.17		POP	Request: AUTH PLAIN
87	9.719304	139.179.15.31		POP	Continuation
88	9.737537	139.179.174.17		POP	Request: AGNzNDIxcGExAHBhMQ==
89	9.951842	139.179.15.31		POP	[TCP Retransmission] Continua
91	9.987600	139.179.15.31		POP	Response: +OK Logged in.
92	10.002530	139.179.174.17		POP	Request: STAT
93	10.026813	139.179.15.31		POP	Response: +OK 30 12458
95	10.043292	139.179.174.17		POP	Request: LIST
96	10.127264	139.179.15.31		POP	Response: +OK 30 messages:
97	10.140915	139.179.174.17		POP	Request: UIDL
98	10.152787	139.179.15.31		POP	Response: +OK
99	10.164830	139.179.174.17		POP	Request: RETR 1
100	10.187627	139.179.15.31		POP	Response: +OK 478 octets

- Mark Packet (toggle)
- Set Time Reference (toggle)
- Apply as Filter ▸
- Prepare a Filter ▸
- Conversation Filter ▸
- Colorize Conversation ▸
- SCTP ▸
- Follow TCP Stream
- Follow UDP Stream
- Follow SSL Stream
- Copy ▸
- Export Selected Packet Bytes

Analyzing a POP3 Session With Wireshark

- See the protocol working with Wireshark

179.174.17 POP Response: +OK Dovecot ready.

irdmi2 > pop3 [ACK] Seq=1 Ack=21 Win=65516 Len=0

Follow TCP Stream

Stream Content

```
+OK Dovecot ready.
USER cs421pa1
+OK
PASS pa1
+OK Logged in.
LIST
+OK 10 messages:
1 425
2 425
3 425
4 425
5 425
6 425
7 425
8 425
9 425
10 425
.
QUIT
+OK Logging out.
```

Find | Save As | Print | Entire conversation (188 bytes) | ASCII | EBCDIC | Hex Dump | C Arrays | Raw

Help | Close | Filter Out This Stream

Analyzing an SMTP Session With Wireshark

- Configure an SMTP client (Thunderbird, Outlook Express, etc.) with the following information:
  - SMTP server: wlab.cs.bilkent.edu.tr
  - Port: 25
  - No authentication or no transport layer security (TLS, SSL or similar) !!!
    - Bad, bad, bad...
- Start capturing with Wireshark on the correct network interface
- Compose a test e-mail and have it sent
- Filter Wireshark packet view with the filter string `smtp`

Analyzing an SMTP Session With Wireshark

- See the protocol working with Wireshark

9.174.17	139.179.15.31	TCP	8017 > smtp [ACK] Seq=1 Ack=1 Win=65536 Len=0
9.15.31	139.179.174.17	SMTP	Response: 220 wlab.cs.bilkent.edu.tr ESMTP Exim 4.63 Wed, 02 Nov 2011

Follow TCP Stream

Stream Content

```
220 wlab.cs.bilkent.edu.tr ESMTP Exim 4.63 Wed, 02 Nov 2011 14:26:01 +0200
HELO bilkent.edu.tr
250 wlab.cs.bilkent.edu.tr Hello bilkent.edu.tr [139.179.174.17]
MAIL FROM: alper@bilkent.edu.tr
250 OK
RCPT TO: cs421pa1@test.com
250 Accepted
DATA
354 Enter message, ending with "." on a line by itself
This is a test message.
.
250 OK id=1RLZtz-0006Vw-A5
QUIT
221 wlab.cs.bilkent.edu.tr closing connection
```

Find | Save As | Print | Entire conversation (417 bytes) | ASCII | EBCDIC | Hex Dump | C Arrays | Raw

Help | Close | Filter Out This Stream

POP3 Commands Overview [RFC 1939]:

- When you open a TCP connection to the POP3 server, it should welcome you with a greeting message:

  +OK Dovecot ready.

- POP3 session passes through a number of states:
  - Authorization State
  - Transaction State
  - Update State
- The client should complete the **Authorization State** by sending the credentials passed as command-line arguments to it:

  USER cs421pa1

  +OK

  PASS pa1

  +OK Logged in.

POP3 Commands Overview [RFC 1939]:

- What if the client supplies wrong credentials:

  +OK Dovecot ready.

  USER cs421pa1

  +OK

  PASS idontknowit

  -ERR Authentication failed.

- Your Java client should report any errors encountered during the session
- After the client successfully supplies the credentials and if the server can acquire the resources, the session enters the **Transaction State**.

POP3 Commands Overview [RFC 1939]:

- Not all commands valid in every state:
  - **LIST**, **RETR** are only valid in the **Transaction State:**

```
+OK Dovecot ready.
LIST
-ERR Unknown command.
```

- Once the session is in the **Transaction State**, the client may issue a **LIST** command:

```
LIST
+OK 5 messages:
1 478
2 404
3 366
4 430
5 529
```

POP3 Commands Overview [RFC 1939]:

- Or a **RETR** command:

RETR 5

+OK 529 octets

Return-path: <alper@wlab.cs.bilkent.edu.tr>

Envelope-to: cs421pa1@test.com

Delivery-date: Wed, 02 Nov 2011 00:12:11 +0200

Received: from localhost ([127.0.0.1] helo=wlab.cs.bilkent.edu.tr)

by wlab.cs.bilkent.edu.tr with smtp (Exim 4.63)

(envelope-from <alper@wlab.cs.bilkent.edu.tr>)

id 1RLMYr-0005qB-Ai

for cs421pa1@test.com; Wed, 02 Nov 2011 00:12:11 +0200

Message-Id: <E1RLMYr-0005qB-Ai@wlab.cs.bilkent.edu.tr>

From: alper@wlab.cs.bilkent.edu.tr

Date: Wed, 02 Nov 2011 00:12:11 +0200

This is a test message.

.

POP3 Commands Overview [RFC 1939]:

- The session enters the **Update State** once the client issues a **QUIT** command:

  QUIT

  +OK Logging out.

- **QUIT** command is also valid in the **Authorization State**!
  - Which implies while grading your projects we may just request session termination with the **QUIT** command without listing, retrieving or composing messages!!!

Sample POP3 Client in Java

- Open a TCP connection to the specified POP3 server on port 110
- Initialize Java objects to read from & write to the TCP stream

```
Socket clientSock = new Socket( pop3Server, 110 );
BufferedReader reader = new BufferedReader(
                                new InputStreamReader(
                                    clientSock.getInputStream() ) );
DataOutputStream out = new DataOutputStream(
                            clientSock.getOutputStream() );
```

- Read & process the server's greeting

```
// Read the greeting message from server
String response = reader.readLine();

// process the greeting message...
```

Sample POP3 Client in Java

- Pass through the Authentication state

```
out.writeBytes( "USER " + userName + "\r\n" );
response = reader.readLine();

// process response...

out.writeBytes( "PASS " + password + "\r\n" );
response = reader.readLine();

// process response...
```

Sample POP3 Client in Java

- Issue a **LIST** command and read & process each line in a loop till a period is encountered

```
out.writeBytes( "LIST\r\n" );
response = reader.readLine();

// read & process each line
```

- Issue the **QUIT** command and close the socket

```
out.writeBytes( "QUIT\r\n" );
response = reader.readLine();

// process response...

clientSock.close();
```

SMTP Commands Overview [RFC 821]:

- When you open a TCP connection to the SMTP server, it should welcome you with a connection greeting reply:

  220 wlab.cs.bilkent.edu.tr ESMTP Exim 4.63 Wed, 02 Nov 2011 00:27:36 +0200

- The sender-SMTP identifies itself to the receiver-SMTP with the **HELO** command:

  HELO wlab2.cs.bilkent.edu.tr

  250 wlab.cs.bilkent.edu.tr Hello alper at wlab2.cs.bilkent.edu.tr [139.179.21.115]

- The command may be interpreted as: Hello, I am **wlab2**.cs.bilkent.edu.tr
- The recipient-SMTP also introduces itself in the response message

SMTP Commands Overview [RFC 821]:

- The sender-SMTP then initiates a mail transaction with the **MAIL** command:

  MAIL FROM: alper@wlab2.cs.bilkent.edu.tr

  250 OK

- During the mail transaction, mail data is delivered to the receiver-SMTP server, which in turn may deliver the mail to one or more mailboxes or pass it on to another server
- The sender-SMTP identifies an individual recipient of the mail data with the **RCPT** command:

  RCPT TO: cs421pa1@test.com

  250 Accepted

SMTP Commands Overview [RFC 821]:

- What if a recipient is rejected?

  RCPT TO: whoisthat@test.com

  550 Unrouteable address

- Your Java client should report any errors encountered during the session
- Then the sender-SMTP sends mail data with the **DATA** command:

  DATA

  354 Enter message, ending with "." on a line by itself

  This is a test message.

  .

  250 OK id=1RLNFX-0005s7-2b

SMTP Commands Overview [RFC 821]:

- Finally the sender-SMTP requests the receiver-SMTP to close the transmission channel with the **QUIT** command:

  QUIT

  221 wlab.cs.bilkent.edu.tr closing connection

- Related reply codes (please refer to RFC 821):

  220: <domain> Service ready

  221: <domain> Service closing transmission channel

  250: Requested mail action okay, completed

  354: Start mail input; end with <CRLF>.<CRLF>

Command-line Arguments to Java Programs

- Your program should accept 4 command-line arguments:

  java EmailClient **<userName> <password> <POP3ServerAddress> <SMTPServerAddress>**

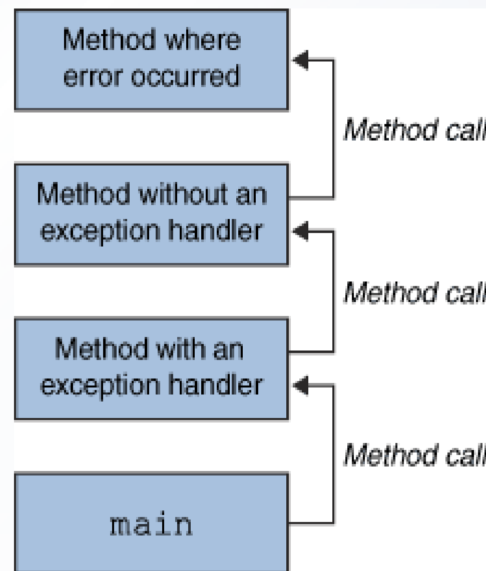
- Your **main** method, with signature static void main(String args[]), accepts a String[] parameter:
  - args[0] is the **<userName>** parameter
  - args[1] is the **<password>** parameter
  - args[2] is the **<POP3ServerAddress>** parameter
  - args[3] is the **<SMTPServerAddress>** parameter
- You should use args.length to check the number of parameters passed to your program & print appropriate error messages if there is an error in the number of arguments

Command-line Arguments to Java Programs

```
 5 public class CommandLineArgs
 6 {
 7     /**
 8      * @param args
 9      */
10     public static void main( String[] args )
11     {
12         if( args.length == 0 )
13             System.err.println( "No cmd-line arguments were supplied..." );
14 
15         for( int i = 0; i < args.length; i++ )
16             System.out.println( args[ i ] );
17     }
18 }
19 
```

Java Exceptions

- Java uses *exceptions* to handle errors and other exceptional events
- When something exceptional occurs (such as a division by zero or a dropped TCP connection), a `Throwable` object is *thrown*
  - The JRE looks for a *handler* for the exception in the call stack



Java Exceptions - "*Catch* or *Specify*" Requirement

- "*Catch* or *Specify*" requirement
  - Code in which **certain** exceptions might be thrown must:
    - Either specify that it may throw those exceptions
    - Or handle the exception with a "try-catch" block
- Three kinds of exceptions
    - Errors (all `Error`s and its subclasses)
    - Runtime exceptions (`RuntimeException`s and its subclasses)
    - Checked exceptions (all `Throwable`s except `Error`s and `RuntimeException`s and their subclasses)
- Checked exceptions are subject to the "*Catch* or *Specify*" requirement

Java Exceptions – Common Pitfalls

- Empty `catch` blocks

```
InetAddress localAddr = InetAddress.getLocalHost();
DatagramSocket udpSocket = new DatagramSocket( 9090, localAddr );
byte[] recvBuff = new byte[ 512 ];
DatagramPacket udpSegment = new DatagramPacket( recvBuff, recvBuff.length );

while( true )
{
	try
	{
		udpSocket.receive( udpSegment );
		process( recvBuff, udpSegment.getLength() );
	}
	catch( IOException ioe ) {}
}
```

- What if `udpSocket.receive` or `process` throws an `IOException`?

Java Exceptions – Common Pitfalls

- Do not bypass “catch or specify” requirements with empty catch blocks
  - Handle them appropriately!
  - Reporting them may save you time
  - Ignorance is the root of all evil!

```
while( true )
{
    try
    {
        udpSocket.receive( udpSegment );
        process( recvBuff, udpSegment.getLength() );
    }
    catch( IOException ioe )
    {
        ioe.printStackTrace();
    }
}
```

Java Exceptions – Common Pitfalls

- A stack trace example

```
java.io.IOException: Received segment is of invalid size
    at UDPServer.process(UDPServer.java:32)
    at UDPServer.main(UDPServer.java:20)
```

- So, from the information available in the stack trace, we know that:
  - `UDPServer.main` called `UDPServer.process` at line 20
  - In `UDPServer.process`, at line 32, an `IOException` was thrown with the message “`Received segment is of invalid size`”
- Check line 32 of UDPServer.java. You can trace back till you understand the problem...

Choose an IDE

- IDE stands for **I**ntegrated **D**evelopment **E**nvironment
- Why use an IDE?
  - Source code editor
  - Compiler/Interpreter
  - Debugger
  - Type browser
  - Class hierarchy browser
  - A lot more...
  - **All integrated in a single environment**
- Good engineers make use of appropriate tools
- Will save you time
  - Aid you in learning new APIs
  - Help you spotting problems faster

Choose an IDE

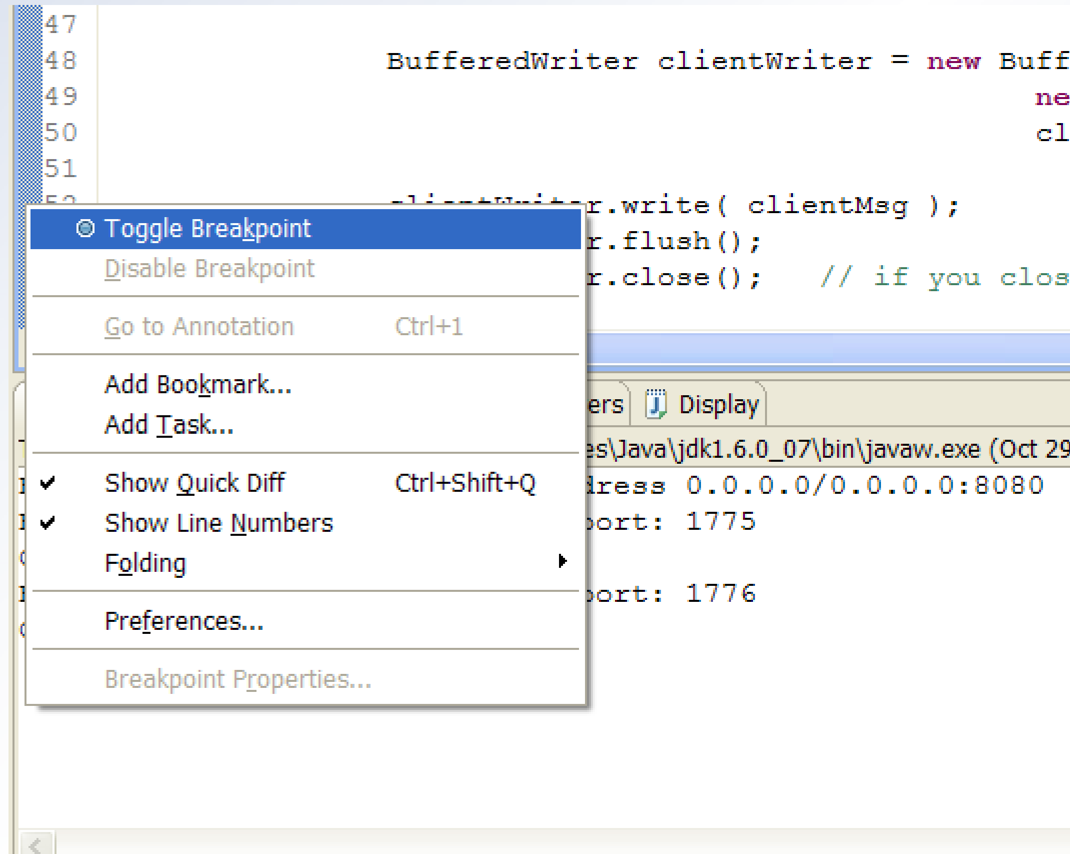
- Some (free) alternatives for Java
  - Eclipse [http://www.eclipse.org]
  - Jcreator® LE from Xinox Software [http://www.jcreator.com]
  - Netbeans [http://www.netbeans.org]

Debugging with Eclipse

- Eclipse hosts a powerful debugger
  - So that you may spot runtime problems faster
- You may...
  - suspend execution with *[un]conditional breakpoints*
  - *display* values of variables
  - *alter* values of variables
  - display *call stacks* of individual threads
  - debug *multithreaded applications*

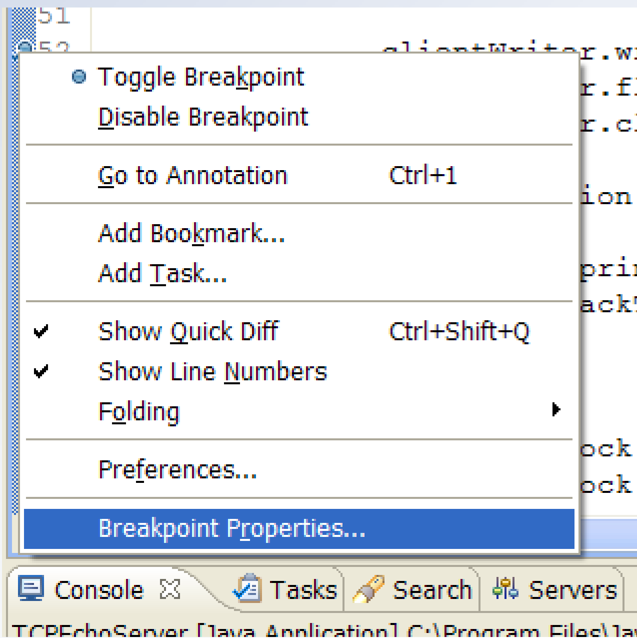
Debugging with Eclipse

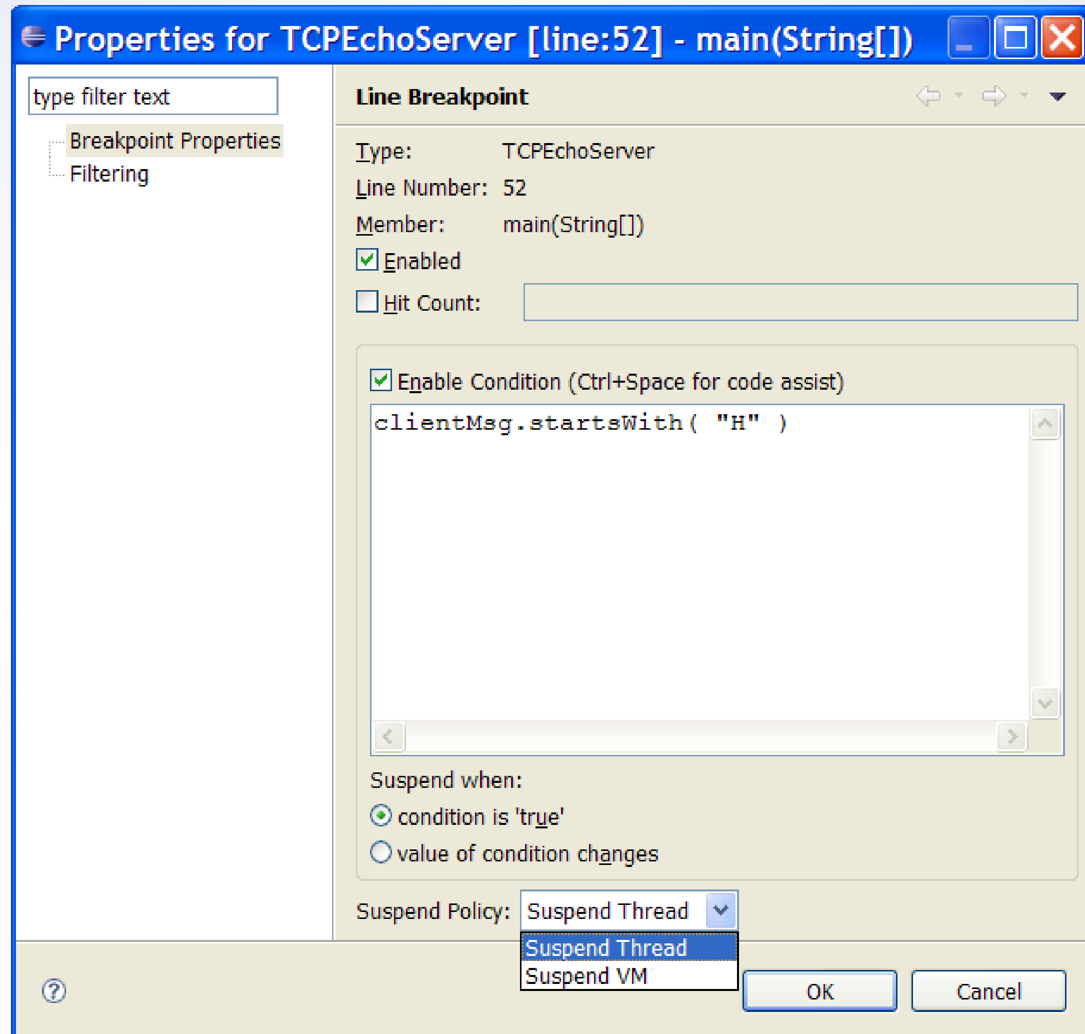
- Adding a breakpoint from the context menu:



Debugging with Eclipse

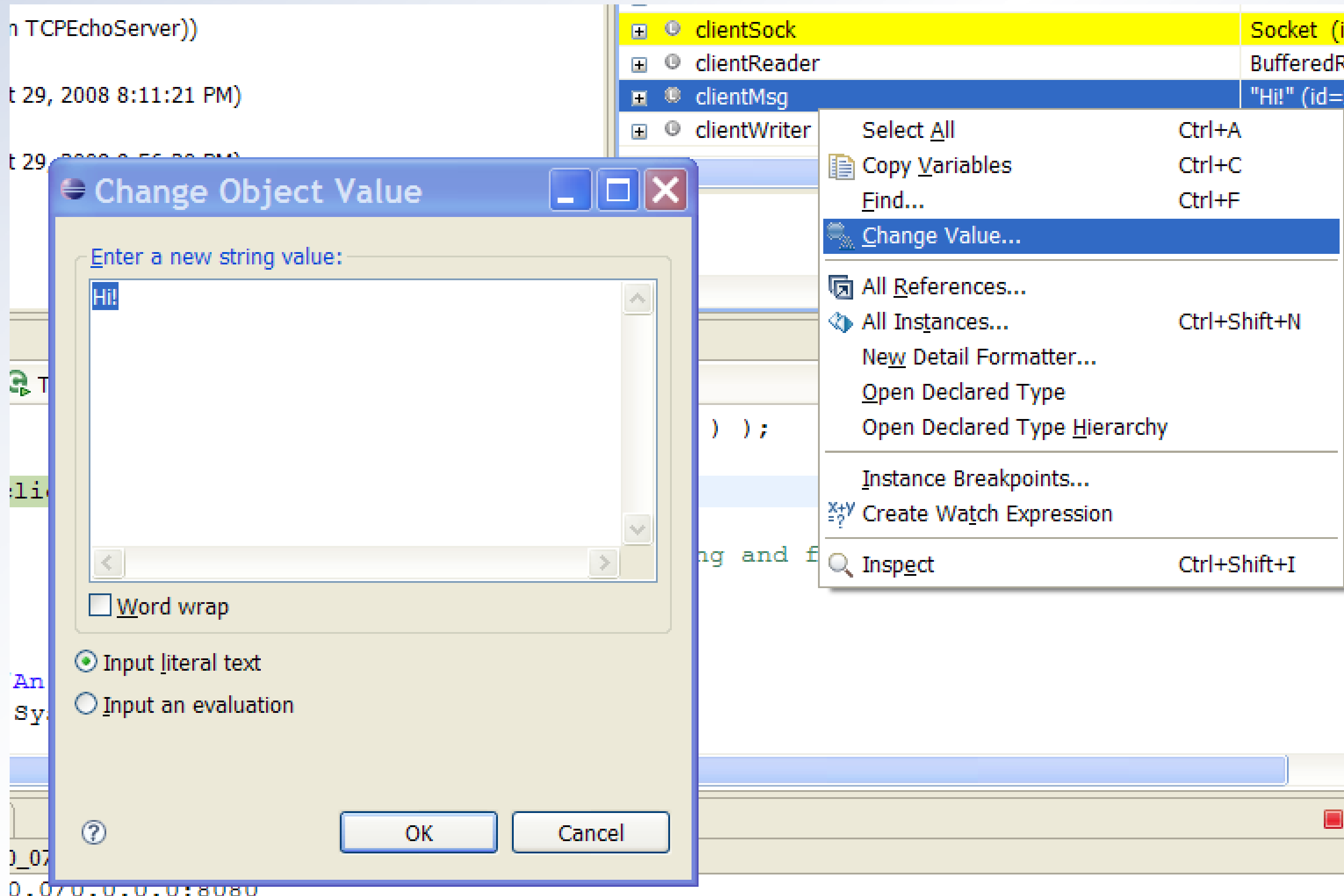
- Adding a condition to an existing breakpoint from the context menu:





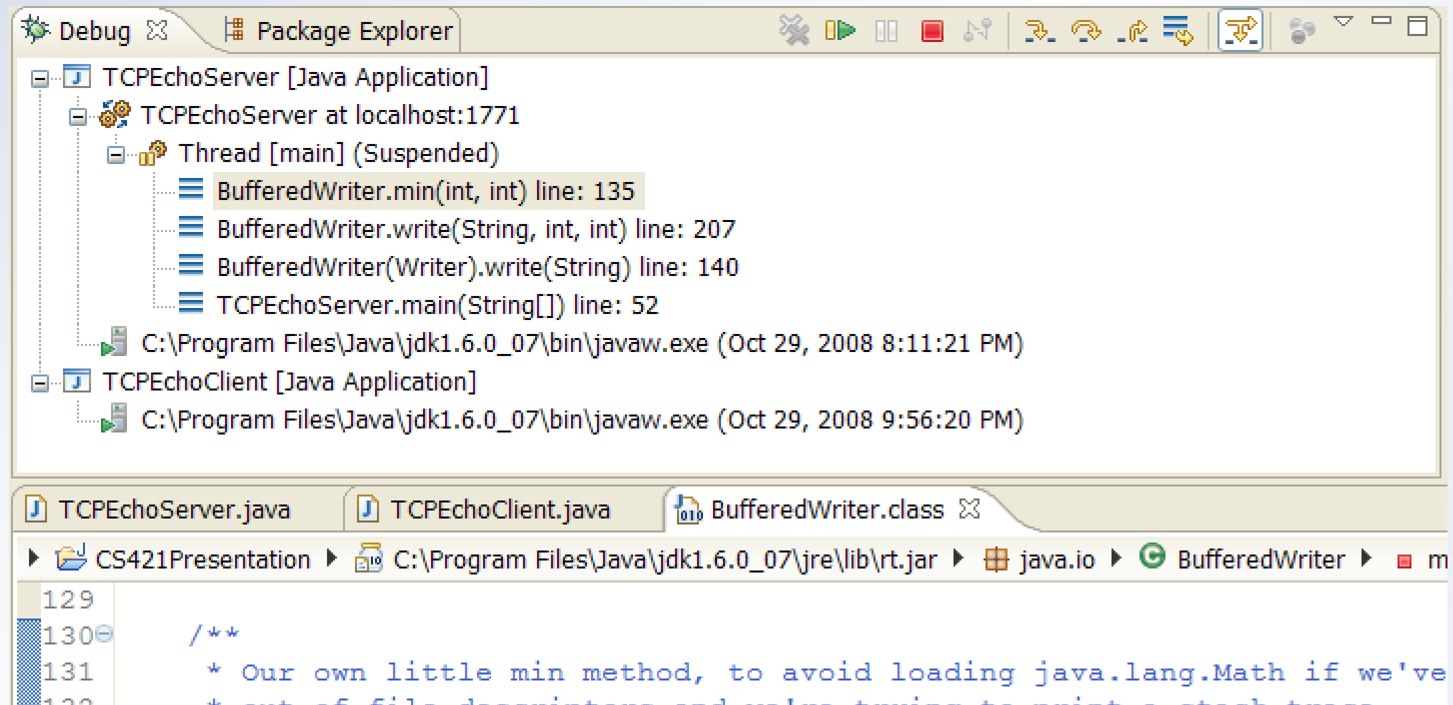
Debugging with Eclipse

- Modifying a variable:



Debugging with Eclipse

- Sample call stack view:



Java Sockets

- The socket API in Java is available through the classes in the `java.net` package
  - `URL`, `URLConnection`, `Socket`, `ServerSocket` use TCP
  - `DatagramPacket`, `DatagramSocket`, `MulticastSocket` use UDP
- `URL`, `URLConnection` and its subclasses (i.e. `HttpURLConnection`, `JarURLConnection`, etc.) implement application level services
  - Hence, for some projects, you may not be allowed to make use of them. Watch the restrictions!
- To implement your very own applications (though they may be reimplementations for the CS421 course :), use transport layer sockets API

Java Sockets - ServerSocket

- Implements server sockets
  - A server socket waits for requests to come in over the network
  - After some operation based on the request is performed, a response is generally returned to the client
- Public constructors
  - `ServerSocket()`
  - `ServerSocket( int port )`
  - `ServerSocket( int port, int backlog )`
  - `ServerSocket( int port, int backlog, InetAddress bindAddr )`

Java Sockets - ServerSocket

- Constructor `ServerSocket()`
  - Instantiates an unbound server socket
  - You have to bind it with the `bind` call
- Constructor `ServerSocket( int port )`
  - Instantiates a server socket bound to the specified port
  - A port of `0` binds it on any free port
  - Backlog set to `50`
- Constructor `ServerSocket( int port, int backlog )`
  - Same as above but the maximum queue length for incoming connection indications (a request to connect) is set to `backlog` parameter

Java Sockets - ServerSocket

- Constructor `ServerSocket( int port, int backlog, InetAddress bindAddr )`
  - Same as `ServerSocket( int port, int backlog )` but the server socket will only accept connection requests on the specified bind address.
  - If the host has multiple IP addresses and the server socket should accept connections on only one of them, use this constructor
  - If `bindAddr` is `null`, the server socket will accept connections on all addresses.

Java Sockets – InetAddress

- You may find out the local addresses of your host with the following code snippet:

```
InetAddress localHost;
InetAddress localAddresses[];

try
{
      // returns the local host
      localHost = InetAddress.getLocalHost();
      // get the address list for the localhost
      localAddresses = InetAddress.getAllByName( localHost.getHostName() );
}
catch( UnknownHostException uhe )     // what if TCP/IP not installed?
{
      System.out.println( "Local host has no IP addresses!" );
      uhe.printStackTrace();

      return;    // cannot continue...
}

for( int i = 0; i < localAddresses.length; i++ )
{
      // print the fully qualified domain name for the IP address (best effort
         method)
      System.out.print( localAddresses[ i ].getCanonicalHostName() );
      System.out.print( ": " );
      // print the IP address string in textual presentation
      System.out.println( localAddresses[ i ].getHostAddress() );
}
```

Java Sockets - InetAddress

- Sample output from the snippet on the previous slide:

```
gezgin: 192.168.1.2
gezgin: 139.179.21.240
```

- `InetAddress.getCanonicalHostName` does a best effort lookup and according to the underlying system configuration, a FQDN may not always be available...

Java Sockets – Backlog (server-side)

- A short server-side code snippet to explain the backlog parameter:

```
// backlog=1, port=8080
ServerSocket serverSock = new ServerSocket( 8080, 1 );

// listen for and accept a single connection
serverSock.accept();

// keeping the socket bound, halt execution
synchronized( BacklogTestServer.class )
{
    System.out.println( "calling wait..." );
    BacklogTestServer.class.wait();
}
```

Java Sockets – Backlog (client-side)

- A short client-side code snippet to explain the backlog parameter:

```
int count = 0;   // count the number of successful connection attempts

while( true )
{
    try
    {
        // try to connect the socket to port 8080 on localhost
        new Socket( InetAddress.getLocalHost(), 8080 );
    }
    catch( IOException ioe )
    {
        ioe.printStackTrace();

        if( ioe instanceof ConnectException )
            System.out.println( "Has reached connection queue limit." );

        return; // we have reached the connection limit
    }

    System.out.println( "Number of connected clients: " + ++count );
}
```

Java Sockets – Backlog

- Sample output for backlog=1:

```
Number of connected clients: 1
Number of connected clients: 2
java.net.ConnectException: Connection refused: connect
at java.net.PlainSocketImpl.socketConnect(Native Method)
at java.net.PlainSocketImpl.doConnect(PlainSocketImpl.java:333)
at java.net.PlainSocketImpl.connectToAddress(PlainSocketImpl.java:195)
at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:182)
at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:366)
at java.net.Socket.connect(Socket.java:519)
at java.net.Socket.connect(Socket.java:469)
at java.net.Socket.<init>(Socket.java:366)
at java.net.Socket.<init>(Socket.java:209)
at BacklogTestClient.main(BacklogTestClient.java:19)
Has reached connection queue limit. Giving up...
```

Java Sockets - Socket

- Implements client sockets, an endpoint for communication between two machines
- Public constructors
  - `Socket()`
  - `Socket(InetAddress addr,int port)`
  - `Socket(InetAddress addr,int port,`
    `InetAddress localAddr,int localPort)`
  - `Socket(String host,int port)`
  - `Socket(String host,int port,`
    `InetAddress localAddr,int localPort)`
  - `Socket(Proxy proxy)`

Java Sockets - Socket

- Constructor `Socket()`
  - Instantiates a new *unconnected* socket
  - Connect the socket with a call to `Socket.connect`
- Constructor `Socket(InetAddress addr,int port)`
  - Instantiates a new socket **and** *tries to* connect it to the specified port number at the specified IP address
  - No need to call `Socket.connect` as the constructor already calls it [see the track trace in backlog discussion]
- Constructor `Socket(InetAddress addr,int port, InetAddress localAddr,int localPort)`
  - Same as above but *binds* the socket to the specified port number and the specified **local** address

Java Sockets - Socket

- Constructor `Socket(String host,int port)`
  - Similar to `Socket(InetAddress addr,int port)`. Instantiates a new socket **and** *tries to* connect it to the specified port number at the specified **named host**
  - May additionally throw an `UnknownHostException`. (What if the supplied host name cannot be resolved into an IP address?)
- Constructor

  `Socket(String host,int port, InetAddress localAddr,int localPort)`
  - Similar to `Socket(InetAddress addr,int port,InetAddress localAddr,int localPort)`. Instead of giving remote host's address, we supply its name (so we may expect an `UnknownHostException`).

Java Sockets - Socket

- Constructor `Socket(Proxy proxy)`
  - Instantiates an unconnected socket (like `Socket()`), specifying the type of the socket proxy to use.
  - You may use this constructor to instantiate a new socket that will connect through a SOCKS proxy server for example...
  - Not directly relevant with web cache servers :)
    - A web cache proxy lives in the *application layer*
    - A SOCKS proxy (or any type of proxy this constructor accepts) should be living below the *application layer*

Java Sockets – Where to Bind?

- It's wise to leave the bind port and the bind address of a **client socket** to the socket implementation *unless* you have a specific requirement
  - You won't worry about “port in use exceptions” or about choosing the right local address...
- You may use the `Socket(String host,int port)` constructor (or a similar one that does not accept bind parameters) and later learn where the socket is bound
  - Via calls to `Socket.getLocalAddress()` and `Socket.getLocalPort()` once the socket is bound
  - If a socket is not connected, it may not have been bound (for instance just after a new socket has been instantiated via constructor `Socket()` before it's connected)

Java Sockets – Sending Data

- Here is a code snippet to send data from a socket:

```
Socket sock = new Socket(serverAddr, 8080);
BufferedWriter writer = new BufferedWriter(
new OutputStreamWriter(sock.getOutputStream()));

writer.write("Hi!\n");
writer.flush();
```

- If you send data through a buffered stream, don't forget to **flush it**!
  - If you don't, your data may stay buffered in a local buffer, waiting for more data to be transmitted
  - The TCP client/server examples in your book **do not use** a buffered output stream (instead use a `DataOutputStream` which is not buffered)

Java Sockets – Receiving Data

- Here is a code snippet to read data from a socket:

```
BufferedReader reader = new BufferedReader(new
InputStreamReader(sock.getInputStream()));
String serverMsg = reader.readLine();

System.out.println( "Got reply: " + serverMsg );
```

- For the above snippet to work as expected:
  - Either the remote end-point should send a string terminated by a newline (or containing a newline)
  - Or the remote end-point should close its associated output stream (or associated socket) making sure all output data is flushed

Java Sockets – DatagramSocket

- Represents a socket for sending and receiving UDP segments (datagram packets in Java jargon)
- Each segment sent or received is individually addressed and routed. Multiple packets sent from one host to another may:
  - Take different routes
  - Arrive in any order or not arrive at all
- Public constructors
  - `DatagramSocket()`
  - `DatagramSocket( int port )`
  - `DatagramSocket( int localPort, InetAddress localAddr )`
  - `DatagramSocket( SocketAddress bindAddr )`

Java Sockets – DatagramSocket

- Constructor `DatagramSocket()`
  - Constructs a datagram socket and binds it to any available port
- Constructor `DatagramSocket( int port )`
  - Constructs a new datagram socket and binds it to the specified port
- Constructor `DatagramSocket( int localPort, InetAddress localAddr )`
  - Constructs a new datagram socket and binds it to the specified port at the specified **local** address
- Constructor `DatagramSocket( SocketAddress bindAddr )`
  - Similar to `DatagramSocket( int localPort, InetAddress localAddr )`

Java Sockets – DatagramPacket

- Represents a datagram packet (UDP segment)
- Construction
  - All constructors take a buffer (`byte[]`) parameter which holds (or will hold) sent (or received) data and another parameter that specifies the buffer's length
  - Some constructors take the remote socket's address and port number (either as a `SocketAddress` object or a couple of an `InetAddress` object and a port number)

Java Sockets – How to Send a UDP Segment

- The following code snippets demonstrate two ways of sending UDP segments:

```
DatagramSocket udpSock = new DatagramSocket();
byte msg[] = getMsg();
InetAddress remoteAddr=InetAddress.getByName("remoteHostname");
int remotePort=9090;
DatagramPacket segment=new DatagramPacket(msg,
msg.length,remoteAddr,remotePort);

udpSock.send(segment);
```

or:

```
.
.
.
DatagramPacket segment = new DatagramPacket(msg,msg.length);

udpSock.connect(remoteAddr, remotePort);
udpSock.send(segment);
```

Java Sockets – DatagramSocket.connect

- UDP provides a *connectionless* service
  - `DatagramSocket.connect` does **not** establish a connection
  - Restricts the endpoint to/from which packets can be sent/recevied
- The following will cause an `IllegalArgumentException` to be thrown (an unchecked exception):

```
DatagramPacket segment = new DatagramPacket(msg,msg.length);

udpSock.connect(remoteAddr, remotePort);
udpSock.send(segment);

int anotherPort = 9090;
DatagramPacket segment2 = new DatagramPacket(msg,msg.length,
InetAddress.getByName("anotherRemoteHostname"),anotherPort);

udpSock.send(segment2);
```

Java Sockets – DatagramSocket.connect

- Here is a sample stack trace you may see in such a case:

```
Exception in thread "main"
java.lang.IllegalArgumentException: connected address and
packet address differ
at java.net.DatagramSocket.send(DatagramSocket.java:603)
at UDPClient.main(UDPClient.java:22)
```

- In summary:
  - `Socket.connect` establishes a *real* TCP connection. TCP is connection-oriented.
  - `DatagramSocket.connect` merely poses a restriction on the remote end-point.

Java Sockets – How to Receive a UDP Segment

- The following code snippet demonstrates receiving UDP segments:

```
InetAddress localAddr = InetAddress.getLocalHost();
DatagramSocket udpSocket = new DatagramSocket( 9090, localAddr );
byte[] recvBuff = new byte[ 512 ];
DatagramPacket udpSegment = new DatagramPacket( recvBuff,
recvBuff.length );

udpSocket.receive( udpSegment );
```

- Since the receive buffer supplied (`recvBuff`) is 512 bytes long, longer segments will be **truncated**!!!
- You may learn the number of bytes received as follows:

```
udpSegment.getLength();
```

- The actual bytes received reside in `recvBuff`

Java Sockets – Blocking & Non-blocking I/O

- All the socket APIs introduced so far are blocking...
  - The calling thread is blocked until the operation (connect, read, write, etc.) completes or fails (probably with a checked exception)
  - Such a thread is said to be *I/O blocked*
- Consider your main thread of the **server** process has accepted a connection request and is now *blocked* in a read call (`InputStream.read`):
  - The communication link between the server and the client from which the server is trying to read is **congested** so that it will take lots of time to complete the read request...
  - What if another client tries to connect to get service?
  - What if the link is not congested but rather the client from which the server is trying to read is a malicious client?

Java Sockets – Blocking & Non-blocking I/O

- => Do not block the thread that accepts connection requests...
  - But we are operating in the blocking I/O mode...
  - Then have a dedicated (*new*) **thread** service each client. So you will have a multithreaded server.
  - Or have a dedicated server **process** service each client.
  - Use *non-blocking I/O*. This mode is available in Java since Java 1.4 (with `java.nio` package).
- In some projects, we may ask you to implement multithreaded applications
  - So that your server applications (or peers) can perform various tasks in parallel

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/
- RFC 2616, Hypertext Transfer Protocol – HTTP/1.1
- Openoffice template used to prepare this presentation is Ooo2, available online from http://technology.chtsai.org/impress/
- RFC 821, 2821
- RFC 1939