



***A Short Tutorial for  
Implementing CS421 Projects  
in Java***

2010-10-26

**Alper Rifat Uluçınar**  
CS421 Course Assistant  
Bilkent University  
Computer Engineering Dep.

# Tutorial Outline

- URLs & the HTTP Protocol – An Overview
- HTTP client implementation hints for Java
- HTML forms
- multipart/form-data MIME type & HTTP POST messages
- Java Exceptions
- Choose an IDE
- Debugging with Eclipse
- Java Sockets
- References

# About URLs

- *Uniform Resource Locator*
  - 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

# About URLs

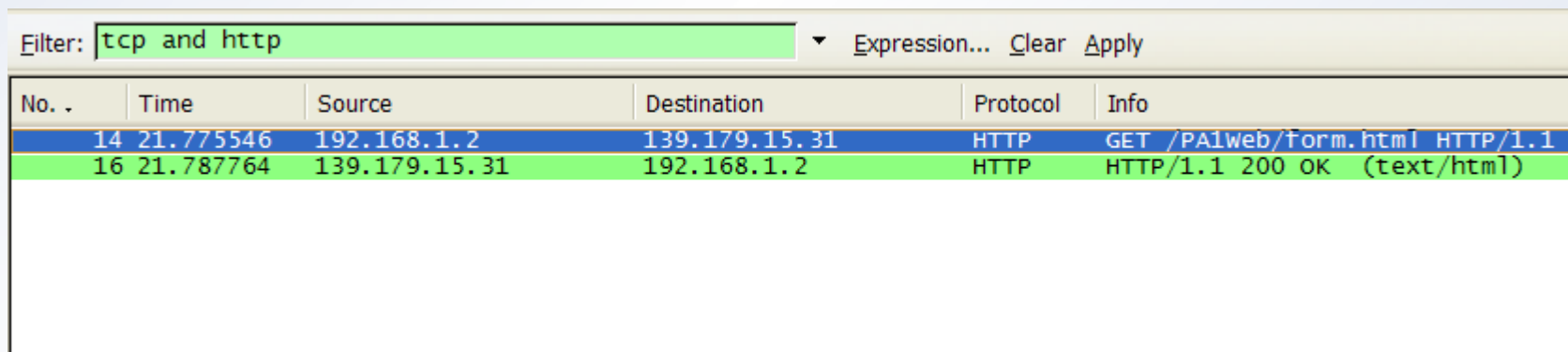
- No need to specify default port # of the protocol
  - `http://java.sun.com/reference/docs/index.html`  
Default port is 80 for HTTP  
Equivalent to  
`http://java.sun.com:80/reference/docs/index.html`
  - `https://webmail.bilkent.edu.tr`  
Default port is 443 for HTTPS  
Equivalent to `https://webmail.bilkent.edu.tr:443`

# Analyzing an HTTP Request with Wireshark

- Start capturing with Wireshark on the correct network interface

Request <http://wlab.cs.bilkent.edu.tr/PA1Web/form.html> with your browser

- Filter Wireshark packet view with the filter string `tcp and http`

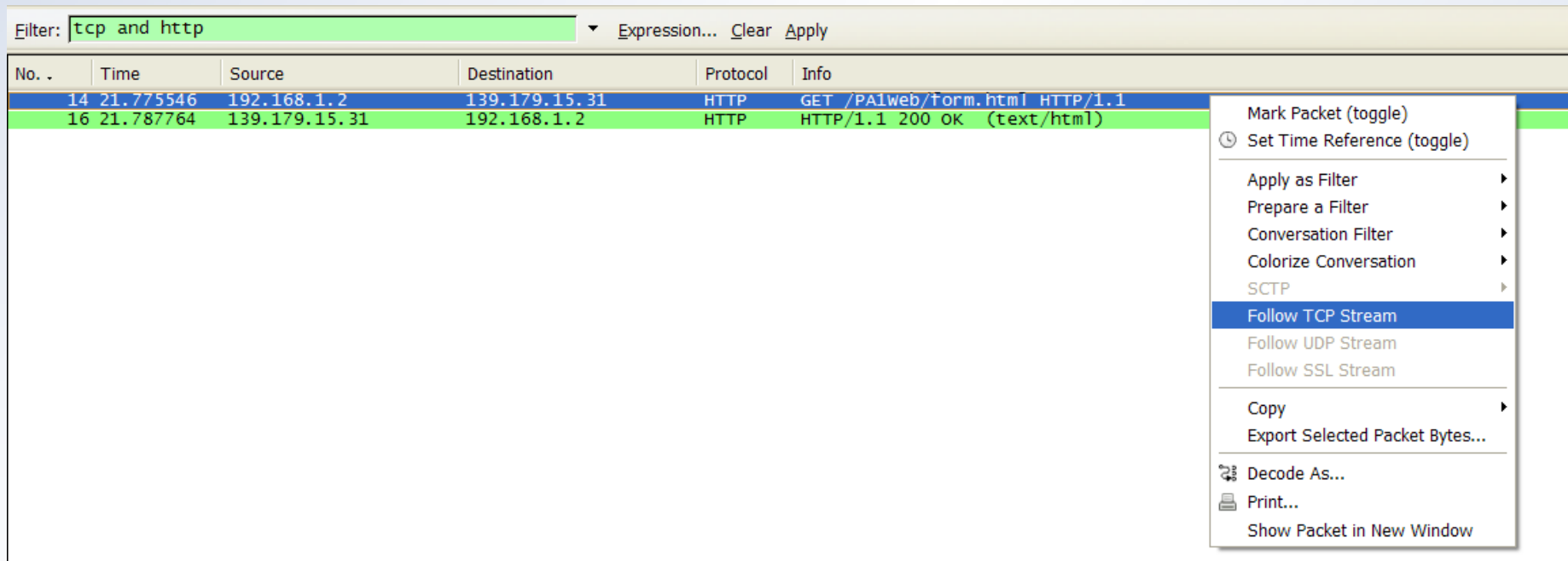


The screenshot shows the Wireshark interface with a filter bar at the top containing the text "tcp and http". Below the filter bar is a table of captured packets. The table has columns for "No.", "Time", "Source", "Destination", "Protocol", and "Info". Two packets are visible: packet 14, which is a GET request from 192.168.1.2 to 139.179.15.31, and packet 16, which is an HTTP 200 OK response from 139.179.15.31 to 192.168.1.2.

No. .	Time	Source	Destination	Protocol	Info
14	21.775546	192.168.1.2	139.179.15.31	HTTP	GET /PA1web/form.html HTTP/1.1
16	21.787764	139.179.15.31	192.168.1.2	HTTP	HTTP/1.1 200 OK (text/html)

# Analyzing an HTTP Request with Wireshark

- Right click on a TCP segment belonging to the HTTP request you made and on the context menu that appears, click “Follow TCP stream”



The screenshot shows the Wireshark interface with a filter set to 'tcp and http'. The packet list pane displays two packets:

No.	Time	Source	Destination	Protocol	Info
14	21.775546	192.168.1.2	139.179.15.31	HTTP	GET /PA1web/form.html HTTP/1.1
16	21.787764	139.179.15.31	192.168.1.2	HTTP	HTTP/1.1 200 OK (text/html)

A context menu is open over the second packet (No. 16), with the 'Follow TCP Stream' option highlighted. The menu items are:

- 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...
- Decode As...
- Print...
- Show Packet in New Window

# Analyzing an HTTP Request with Wireshark

- See the protocol working with Wireshark

The image shows the Wireshark network protocol analyzer interface. The main pane displays a list of captured packets. Packet 14 is selected, showing an HTTP GET request for /PA1web/form.html. A 'Follow TCP Stream' dialog box is open, displaying the stream content for this packet. The stream content shows the request and the server's response (200 OK) with various headers and a partial HTML body.

No. .	Time	Source	Destination	Protocol	Info
11	21.755560	192.168.1.2	139.179.15.31	TCP	samd > http [SYN] Seq=0 win=65535 Len=0 MSS=1460 WS=1
12	21.764990	139.179.15.31	192.168.1.2	TCP	http > samd [SYN, ACK] Seq=0 Ack=1 win=5840 Len=0 MSS=14
13	21.765166	192.168.1.2	139.179.15.31	TCP	samd > http [ACK] Seq=1 Ack=1 win=65536 Len=0
14	21.775546	192.168.1.2	139.179.15.31	HTTP	GET /PA1web/form.html HTTP/1.1
15	21.785063	139.179.15.31	192.168.1.2	TCP	http > samd [ACK] Seq=1 Ack=84 win=5888 Len=0
16	21.787764	139.179.15.31	192.168.1.2	HTTP	HTTP/1.1 200 OK (text/html)
17	21.788311	139.179.15.31	192.168.1.2	TCP	http > samd [FIN, ACK] Seq=971 Ack=84 win=5888 Len=0
18	21.788343				
19	21.840977				
20	21.849607				

**Follow TCP Stream**

Stream Content

```
GET /PA1web/form.html HTTP/1.1
Host: wlab.cs.bilkent.edu.tr
Connection: close

HTTP/1.1 200 OK
Date: Mon, 25 Oct 2010 06:40:57 GMT
Server: Apache/2.2.3 (Debian) mod_jk/1.2.18 mod_python/3.2.10 Python/2.4.4 PHP/5.2.0-8+etch4
mod_ssl/2.2.3 openssl/0.9.8c mod_perl/2.0.2 Perl/v5.8.8
Content-Length: 680
Connection: close
Content-Type: text/html; charset=ISO-8859-1

<HTML>
.<BODY>
..<FORM ACTION='dosubmit?part1=0m' METHOD='post' ENCTYPE='multipart/form-data'>
...
...<INPUT TYPE='TEXT' NAME='name_1_Lr' />
...
...<INPUT TYPE='TEXT' NAME='name_2_Kd' />
...
...<INPUT TYPE='TEXT' NAME='name_3_MV' />
...
...<INPUT TYPE='TEXT' NAME='name_4_A%2F' />
...
...<INPUT TYPE='TEXT' NAME='name_5_13' />
...
...<INPUT TYPE='TEXT' NAME='name_6_ej' />
```

Find Save As Print Entire conversation (1053 bytes) [Dropdown] [Radio] ASCII [Radio] EBCDIC [Radio] Hex Dump [Radio] C Arrays [Radio] Raw [Radio] Raw

Help [Close] Filter Out This Stream

# HTTP Protocol [RFC 2616] – Satellite's Camera View

- Message types
  - Requests from client to server
  - Responses from server to client
- Red text on the previous slide is a request message
- Blue text on the same slide is the response message

# HTTP Protocol [RFC 2616] – Satellite's Camera View

- Simplified HTTP request message:
  - **Method SP Request-URI SP HTTP-versionCRLF**  
**\*((General-header | Request-header | Entity-  
header)CRLF)**  
**CRLF**  
**[Message-body]**
- Example method strings: *GET, HEAD*
- Example request uri: */PA1Web/form.html*
- Example version strings: *HTTP/1.0, HTTP/1.1*
- SP is the space character (ASCII code 0x20)
- CRLF is `\r\n` in Java

# HTTP Protocol [RFC 2616] – Satellite's Camera View

- Minimalist HTTP 1.0 request message example:

```
GET /PA1Web/form.html HTTP/1.0
```

- Minimalist HTTP 1.1 request message example:

```
GET /PA1Web/form.html HTTP/1.1
```

```
Host: wlab.cs.bilkent.edu.tr
```

- Request line and the header fields are all terminated with CRLF (`\r\n`)
- An empty line (a line with nothing preceding CRLF) marks the end of the header fields
- `Host` header field is mandatory for HTTP/1.1
- We assume here that we do not use an HTTP proxy

# HTTP Protocol [RFC 2616] – Satellite's Camera View

- Simplified HTTP response message:
  - **HTTP-version SP Status-code SP Reason-phraseCRLF**  
**\*((General-header | Response-header | Entity-header)CRLF)**  
**CRLF**  
**[Message-body]**
- Example version string: HTTP/1.1
- Example status code strings: 200, 301, 302, 404, 500
- Example reason phrases: *OK, Moved Permanently, Found, Not Found, Internal Server Error*

# HTTP Protocol [RFC 2616] – Satellite's Camera View

- HTTP response message example:

```
HTTP/1.1 200 OK
Date: Mon, 25 Oct 2010 06:40:57 GMT
Server: Apache/2.2.3 (Debian) mod_jk/1.2.18 mod_python/
 3.2.10 Python/2.4.4 PHP/5.2.0-8+etch4 mod_ssl/2.2.3
  OpenSSL/0.9.8c mod_perl/2.0.2 Perl/v5.8.8
Content-Length: 680
Connection: close
Content-Type: text/html; charset=ISO-8859-1

<HTML>
  <BODY>
    <FORM ACTION='doSubmit?part1=0m'
      METHOD='post' ENCTYPE='multipart/form-
        data'
    ...
  </HTML>
```

# HTTP Protocol [RFC 2616] – Satellite's Camera View

- Another response message example:

```
HTTP/1.1 500 Internal Server Error
Date: Mon, 25 Oct 2010 07:22:51 GMT
Server: Apache/2.2.3 (Debian) mod_jk/1.2.18 mod_python/
 3.2.10 Python/2.4.4 PHP/5.2.0-8+etch4 mod_ssl/2.2.3
  OpenSSL/0.9.8c mod_perl/2.0.2 Perl/v5.8.8
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html;charset=ISO-8859-1

<HTML>
  <BODY>
  ...
  </BODY>
</HTML>
```

# HTTP Protocol [RFC 2616] – Satellite's Camera View

- Status line and the headers are all terminated with CRLF
- An empty line (a line with nothing preceding CRLF) marks the end of the header fields and the beginning of the optional entity body
- 500 status code indicates an “Internal Server Error”
  - Server was unable to fulfill the client's request
  - Cause of the “error” can be anything
    - I/O error on the server
    - Failed transaction
    - Misconfiguration
    - ...

# HTTP Protocol [RFC 2616] – Satellite's Camera View

- For this assignment, you will get a 500 response from the server:
  - If you submit a form downloaded more than 60s ago
  - If you do not submit the form names and the “part1” parameter correctly
  - If you upload a file that's larger than 50KB
  - If something else goes wrong at the server side
- Please handle HTTP 500 responses in your program
  - The server *might* have sent you details about the error in the body of the response
  - Some browsers display the body of the 500 response message but some do not
  - Please display a diagnostic message containing the server's response body

# Making an HTTP Request with Java

- HTTP request as seen in Wireshark

```
Frame 5 (647 bytes on wire, 647 bytes captured)
Ethernet II, Src: Sony_1d:b4:aa (00:01:4a:1d:b4:aa), Dst: USRoboti_e8:9f:a8 (00:c0:49:e8:9f:a8)
Internet Protocol, Src: 192.168.1.2 (192.168.1.2), Dst: 139.179.15.31 (139.179.15.31)
Transmission Control Protocol, Src Port: cl-db-request (4136), Dst Port: http (80), Seq: 1, Ack: 1, Len: 593
Hypertext Transfer Protocol
GET /PA1web/form.html HTTP/1.1\r\n
Host: wlab.cs.bilkent.edu.tr\r\n
User-Agent: Mozilla/5.0 (windows; U; windows NT 5.1; en-US; rv:1.9.0.19) Gecko/2010031422 Firefox/3.0.19 (.NET CLR 3.5.30729)\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
Keep-Alive: 300\r\n
Connection: keep-alive\r\n
Cookie: JSESSIONID=FDf1E63582BC88EE04F0A73268A0A051; bilkentlang=tr; __utma=182065844.1501143790.1254831537.1263912544.1264586115.6\r\n
Pragma: no-cache\r\n
Cache-Control: no-cache\r\n
\r\n
```

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

# Making an HTTP Request with Java

- How to make an HTTP GET 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 /PA1Web/form.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();
```

# Making an HTTP Request with Java

- How to make a range 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/pal/test1.txt HTTP/1.1\r\n" );
25     writer.write( "Host: wlab.cs.bilkent.edu.tr\r\n" );
26     writer.write( "Range: bytes=0-9\r\n" );
27     writer.write( "Connection: close\r\n" );
28     writer.write( "\r\n" );
29
30     writer.flush();
```

# Downloading a Text Resource via HTTP

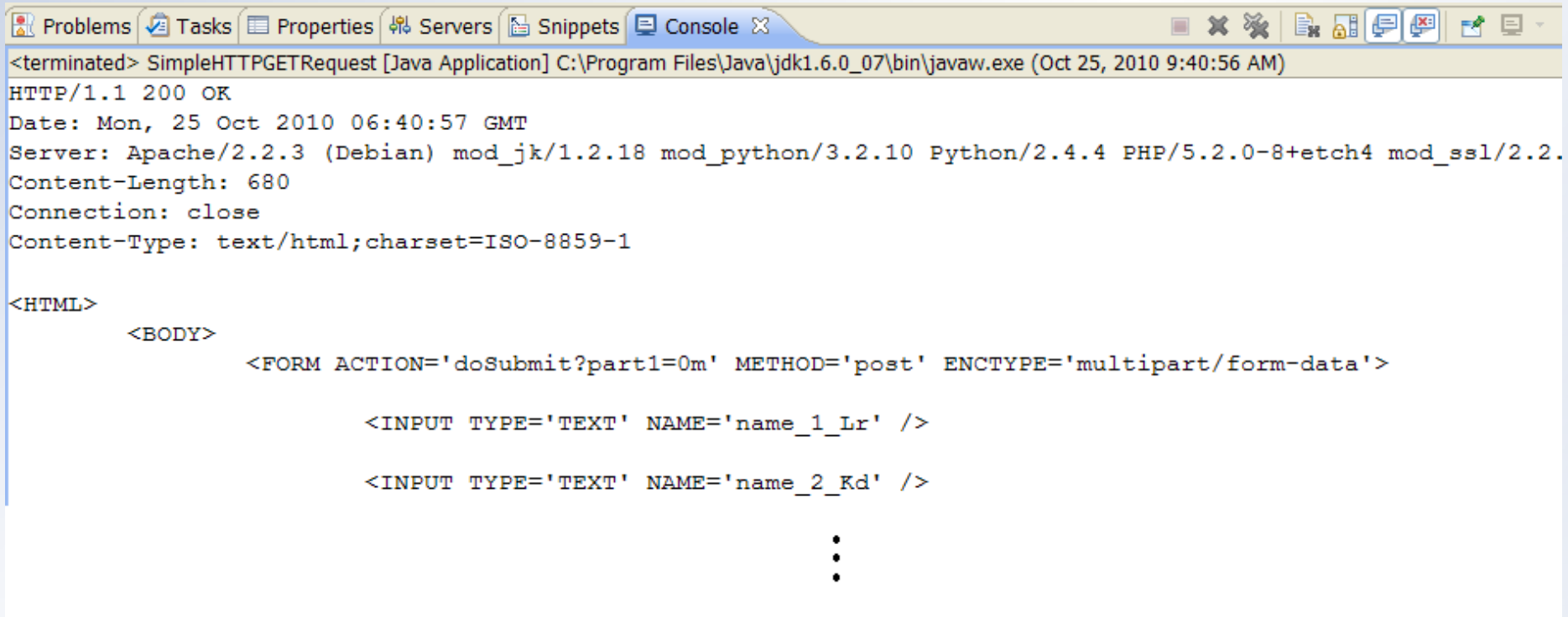
- How to read a text resource?

```
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



```
<terminated> SimpleHTTPGETRequest [Java Application] C:\Program Files\Java\jdk1.6.0_07\bin\javaw.exe (Oct 25, 2010 9:40:56 AM)
HTTP/1.1 200 OK
Date: Mon, 25 Oct 2010 06:40:57 GMT
Server: Apache/2.2.3 (Debian) mod_jk/1.2.18 mod_python/3.2.10 Python/2.4.4 PHP/5.2.0-8+etch4 mod_ssl/2.2.
Content-Length: 680
Connection: close
Content-Type: text/html;charset=ISO-8859-1

<HTML>
  <BODY>
    <FORM ACTION='doSubmit?part1=0m' METHOD='post' ENCTYPE='multipart/form-data'>
      <INPUT TYPE='TEXT' NAME='name_1_Lr' />
      <INPUT TYPE='TEXT' NAME='name_2_Kd' />
      :
    </FORM>
  </BODY>
</HTML>
```

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

# HTML Forms & HTTP POST – Relation with the Assignment

- In this assignment, you have to implement the HTTP POST method:
  - With a message body of type **multipart/form-data**
  - The body will contain data for an **HTML form**
- HTML forms allow clients to submit data to a server
- Workflow for the assignment:
  - Ask the URL of the HTML form
  - Download the HTML form to be submitted – automatically follow any redirections
  - Have the user fill in the form
  - Submit the form data

# HTML Forms

- Form example

```
<HTML><BODY>  
<FORM ACTION='doSubmit?part1=pB8' METHOD='post'  
ENCTYPE='multipart/form-data'>  
    <INPUT TYPE='TEXT' NAME='name_1_bmX' />  
    <INPUT TYPE='TEXT' NAME='name_2_WRk' />  
    <INPUT TYPE='TEXT' NAME='name_3_2zJ' />  
    <INPUT TYPE='TEXT' NAME='name_4_gto' />  
    <INPUT TYPE='TEXT' NAME='name_5_7CY' />  
    <INPUT TYPE='FILE' NAME='file_V7Qg%3D%3D' />  
    <INPUT TYPE='SUBMIT' VALUE='Submit' />  
</FORM>  
</BODY></HTML>
```

- This is *HyperText Markup Language* – **HTML**
- 5 INPUT elements of TYPE **TEXT**, 1 INPUT element of TYPE **FILE**

# HTML Forms

- FORM element
  - ACTION attribute: Tells the client the Request URI for the POST method. The form data should be submitted to this URI
  - May be an **absolute** or **relative** URI
  - `doSubmit?part1=pB8` is a relative URI
    - Relative to the URI of the form itself
  - If the URL of the form is  
`http://wlab.cs.bilkent.edu.tr/PA1Web/form.html`  
then `doSubmit?part1=pB8` resolves into:  
`http://wlab.cs.bilkent.edu.tr/PA1Web/doSubmit?part1=pB8`

# HTML Forms

- If the ACTION attribute specified `/doSubmit` instead of `doSubmit` (please pay attention to the leading /):  
then it would resolve into:

`http://wlab.cs.bilkent.edu.tr/doSubmit`

- If the ACTION attribute specified `http://www.google.com/doSubmit` (absolute URI) instead of `doSubmit`:

then it would resolve into:

`http://www.google.com/doSubmit`

- `<FORM`  
    `ACTION="http://www.google.com/doSubmit`  
    `" METHOD=...`

# HTML Forms

- FORM element
  - METHOD attribute: The HTTP method to use. For this assignment, only **post** is allowed here
  - ENCTYPE attribute: The content encoding to be used for the request body. For this assignment, only **multipart/form-data** is allowed here.
- INPUT element
  - TYPE attribute: TEXT, FILE, SUBMIT options are allowed for this assignment
  - NAME attribute: Server checks the names!!!
  - VALUE attribute: May specify default values
    - `<INPUT TYPE='TEXT' VALUE='Alper' NAME='student_1' />`

# HTML Forms - How to Render a Form for This Assignment

- Form example:

```
<FORM ACTION='doSubmit' METHOD='post'  
  ENCTYPE='multipart/form-data'>
```

```
  Student id #1 <INPUT TYPE='TEXT'  
    NAME='student_1' />
```

```
  Student id #2 (Leave blank if you have done it  
    on your own) <INPUT TYPE='TEXT'  
    NAME='student_2' />
```

```
  Submission File <INPUT TYPE='FILE'  
    NAME='submission_file' />
```

```
<INPUT TYPE='SUBMIT' VALUE="Submit" />
```

```
</FORM>
```

# HTML Forms - How to Render a Form for This Assignment

- Rendering example:

```
Please input form URL:
```

```
http://wlab.cs.bilkent.edu.tr/PA1Web/form.html
```

```
HTML file containing the form has been downloaded...
```

```
Rendering form:
```

```
student_1 [TEXT]: 12345678
```

```
student_2 [TEXT]: 87654321
```

```
submission_file [FILE]: D:\temp\upload-test.bin
```

```
[SUBMIT] (Submit):
```

```
Form data submitted...
```

```
Server status: 200
```

- Rendering rule: NAME [TYPE] (VALUE): ...

## multipart/form-data MIME Type

- Initially defined for “Form-based File Upload in HTML” (RFC 1867). Generalized in RFC 2388
- Allows the client to submit form data
  - Your HTTP client shall submit the form data
    - Via HTTP POST **method**
    - With a multipart/form-data **body**
- **multipart/form-data** defines a MIME type suitable for representing form data
  - A **multipart/form-data** message may consist of **multiple parts** :)
  - Each part can contain **form-data**

# multipart/form-data MIME Type

- Sample multipart/form-data message:

```
--BOUNDARY_STR
Content-Disposition: form-data; name="student_1_name"

Ali
--BOUNDARY_STR
Content-Disposition: form-data; name="student_2_name"

Veli
--BOUNDARY_STR
Content-Disposition: form-data; name="f_up"; filename="hw.bin"
Content-Type: application/octet-stream

...
--BOUNDARY_STR--
```

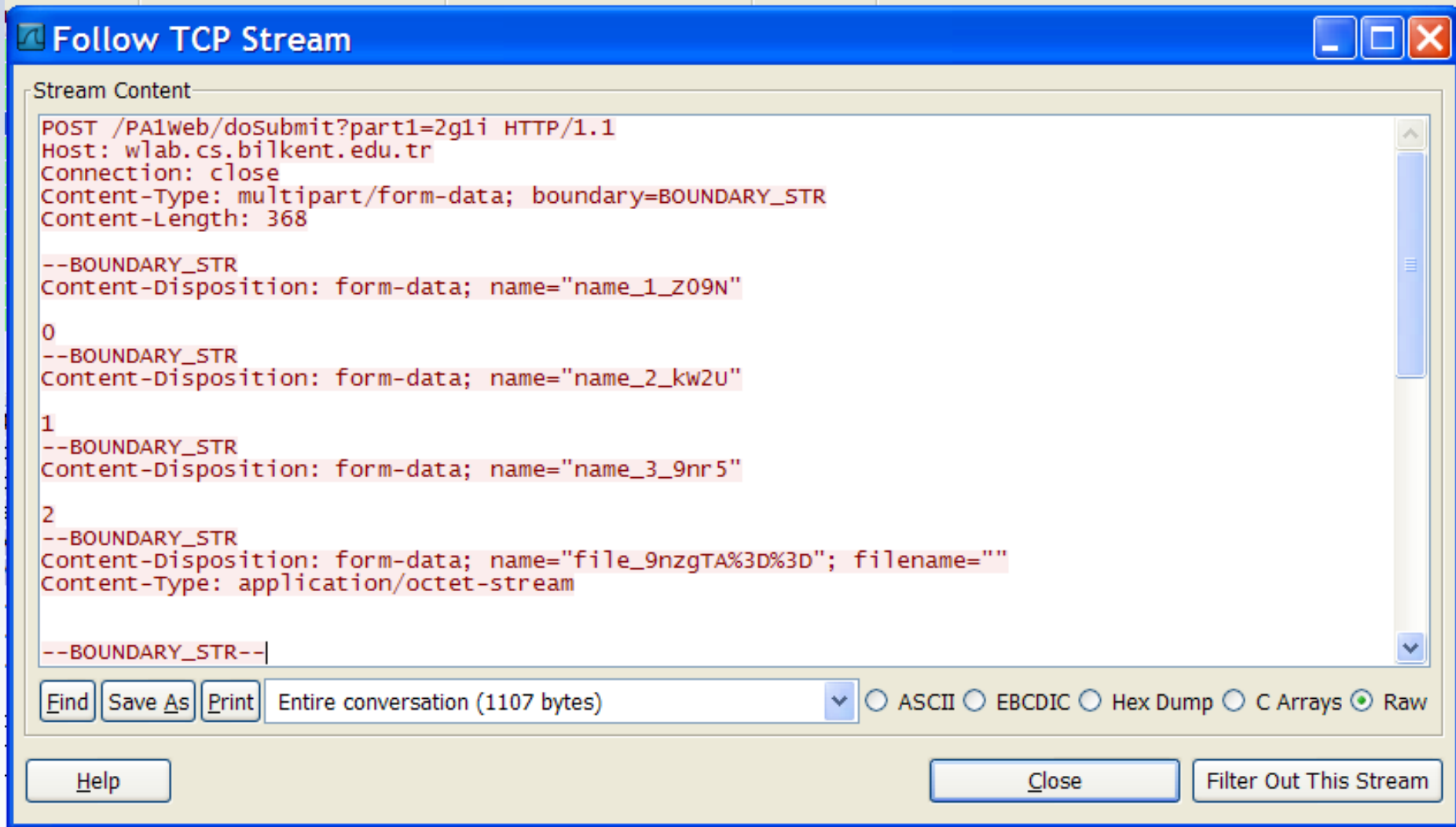
# multipart/form-data MIME Type

- Each part is *bounded* with boundary strings
  - In the previous example, the boundary string selected WAS `BOUNDARY_STR`
  - Not a good boundary string: What if a form field contains `--BOUNDARY_STR` as its data?
  - A typical (and relatively safe) boundary might be:  
`-----26444301151915`
  - You have to generate the boundary string in your program & tell which boundary string you will use to the server in the header part (with the Content-Type header field):
    - `Content-Type: multipart/form-data;  
boundary=BOUNDARY_STR`

# multipart/form-data MIME Type

- Presentational information for MIME messages is conveyed with the **Content-Disposition** header field:
  - RFC 2183
  - Will the (multiple) parts of a MIME message be presented to the user as a single document?
    - `Content-Disposition: inline ...`
  - Will the (multiple) parts of a MIME message be presented to the user as one main document with a list of **separate** attachments?
    - `Content-Disposition: attachment ...`
  - Following the disposition type, disposition parameters are specified... Please refer to the RFC.
- For this assignment: disposition type is `form-data` and don't forget to specify the `name` disposition parameter...

# POST Request with multipart/form-data body



The screenshot shows a window titled "Follow TCP Stream" with a scrollable text area containing the following text:

```
POST /PA1web/doSubmit?part1=2g1i HTTP/1.1
Host: wlab.cs.bilkent.edu.tr
Connection: close
Content-Type: multipart/form-data; boundary=BOUNDARY_STR
Content-Length: 368

--BOUNDARY_STR
Content-Disposition: form-data; name="name_1_Z09N"

0
--BOUNDARY_STR
Content-Disposition: form-data; name="name_2_kw2U"

1
--BOUNDARY_STR
Content-Disposition: form-data; name="name_3_9nr5"

2
--BOUNDARY_STR
Content-Disposition: form-data; name="file_9nzgTA%3D%3D"; filename=""
Content-Type: application/octet-stream

--BOUNDARY_STR--
```

At the bottom of the window, there are several controls:

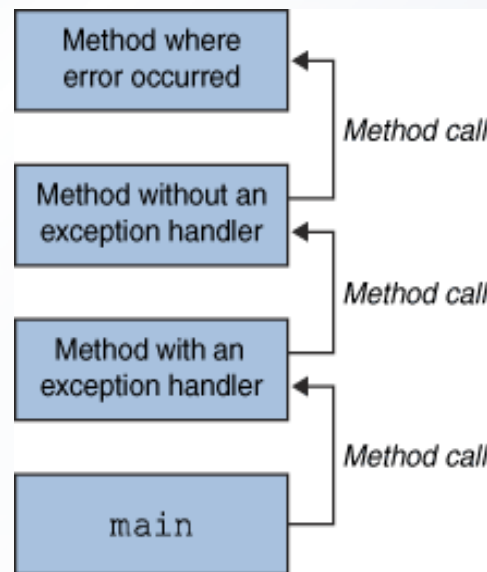
- Buttons: Find, Save As, Print
- Text field: Entire conversation (1107 bytes)
- Radio buttons: ASCII, EBCDIC, Hex Dump, C Arrays, Raw (selected)
- Buttons: Help, Close, Filter Out This Stream

## POST Request with multipart/form-data body

- Please pay attention to the **Content-Length** header field
  - Specifies the length of the body of the message [RFC 2616 Section 14.13]
  - In the previous message, length of the multipart/form-data MIME message (368 bytes)
- The final part in the previous example was for an INPUT element of TYPE FILE
  - The user submitted an *empty* file
  - Please do not assume any encoding for the contents of the file!
  - => Please do not use Writer classes (which impose a text encoding on the data being written) while sending the file's content

# 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 `Errors` and its subclasses)
  - Runtime exceptions (`RuntimeExceptions` and its subclasses)
  - Checked exceptions (all `Throwables` except `Errors` and `RuntimeExceptions` 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

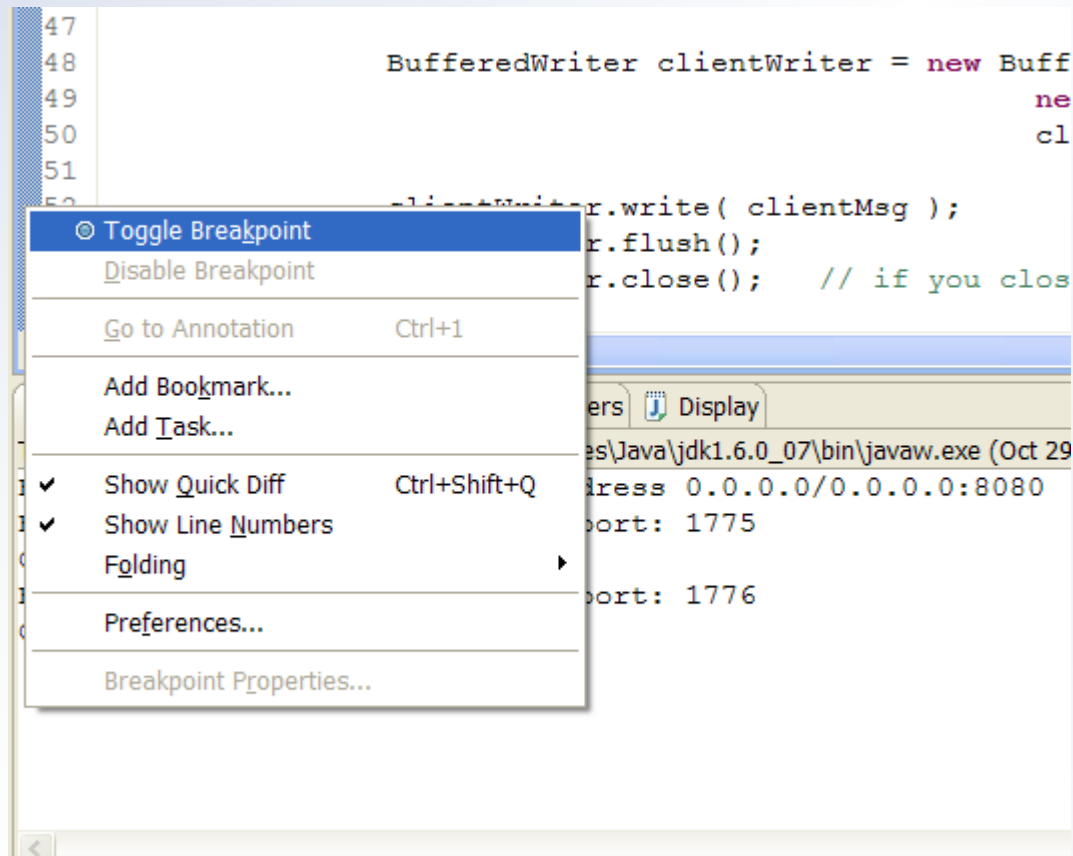
- Some (free) alternatives for Java
  - Eclipse [<http://www.eclipse.org>]
  - Jcreator<sup>®</sup> LE from Xinox Software [<http://www.jcreator.com>]
  - Netbeans [<http://www.netbeans.org>]

# 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*

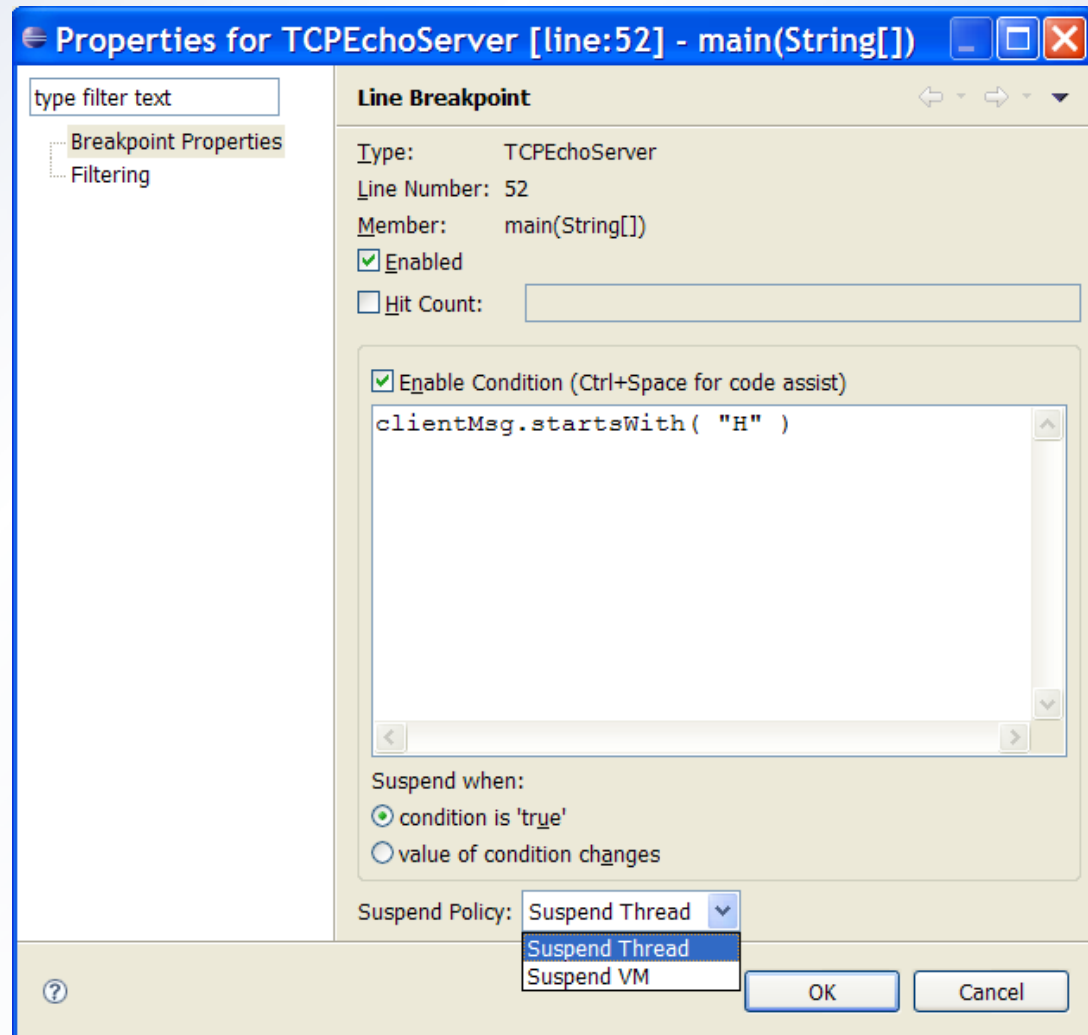
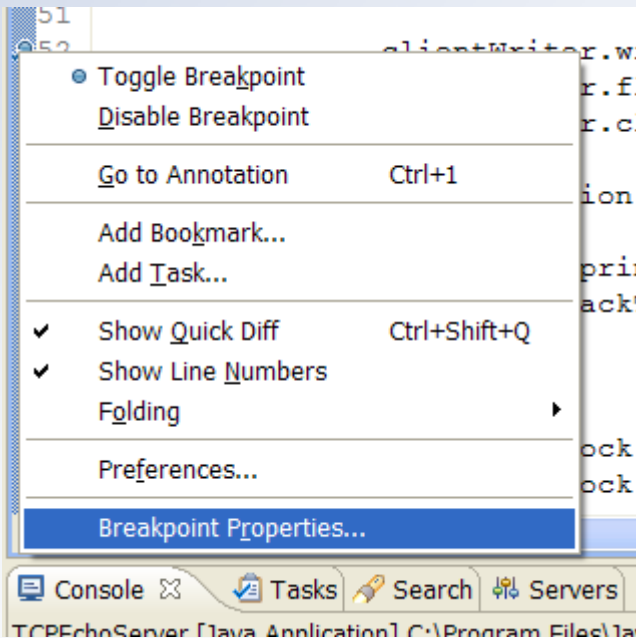
# Debugging with Eclipse

- Adding a breakpoint from the context menu:



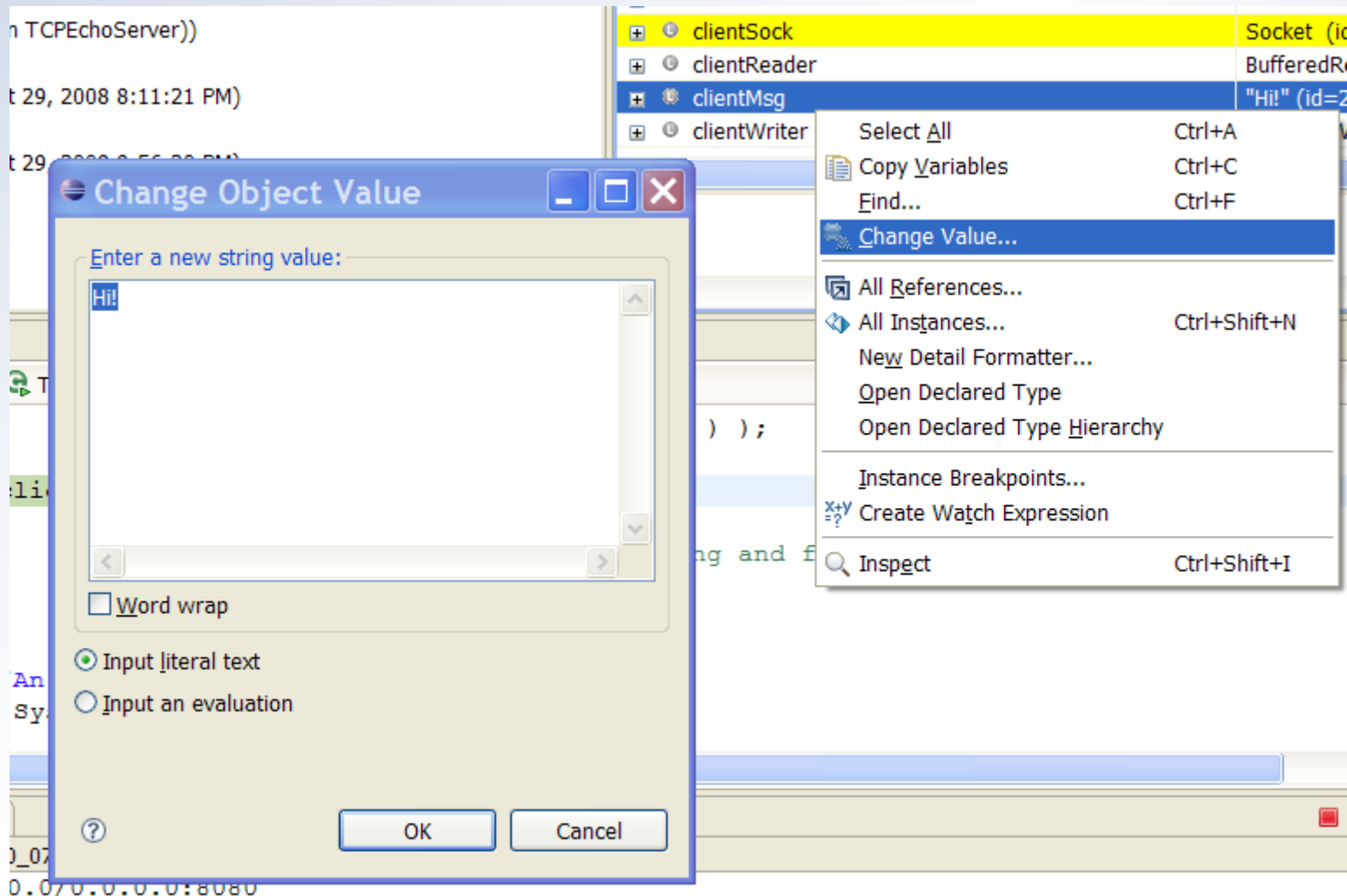
# Debugging with Eclipse

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



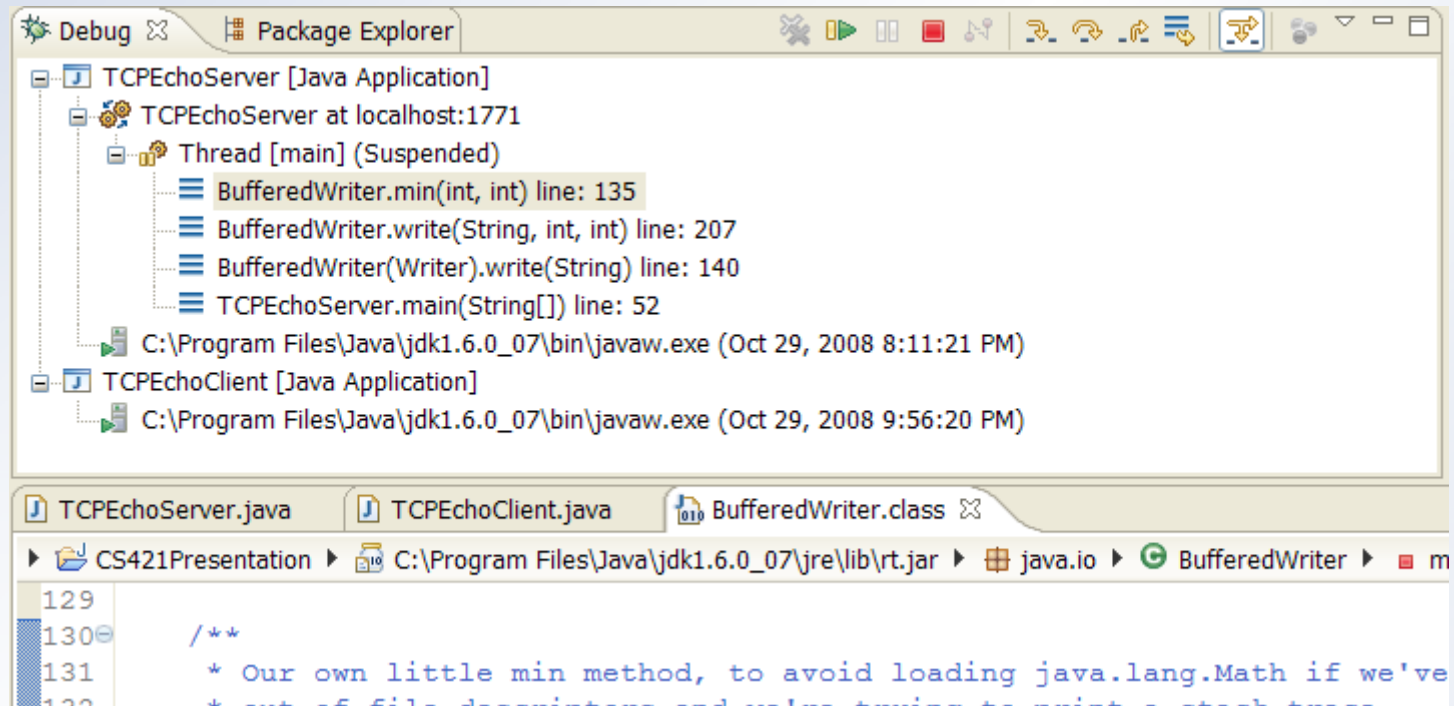
# Debugging with Eclipse

- 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/received
- 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 UDPCClient.main(UDPCClient.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/>