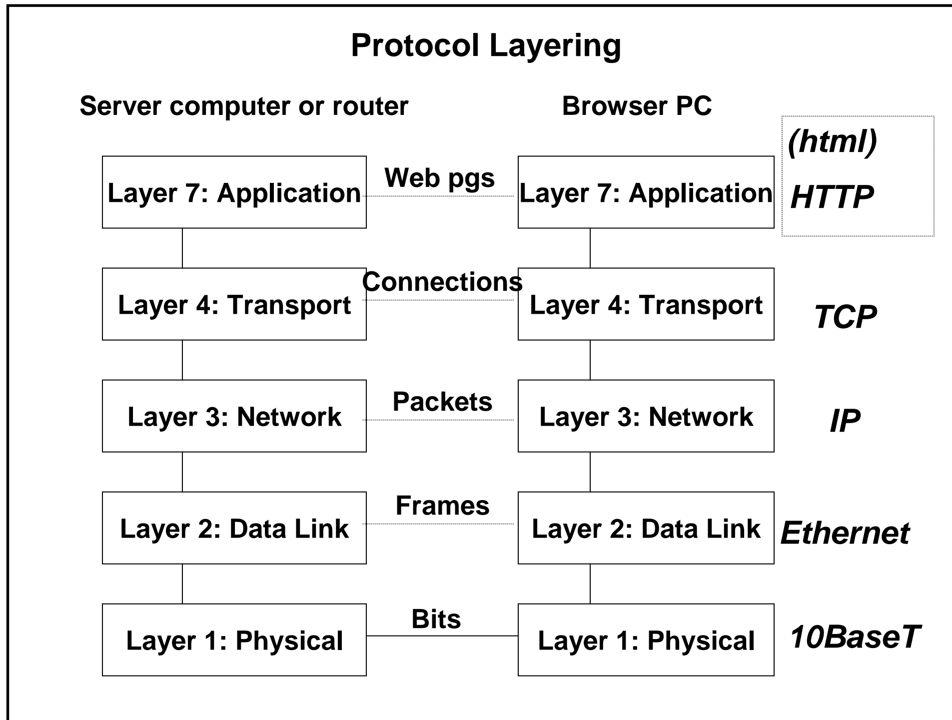
1.00 - Lecture 37

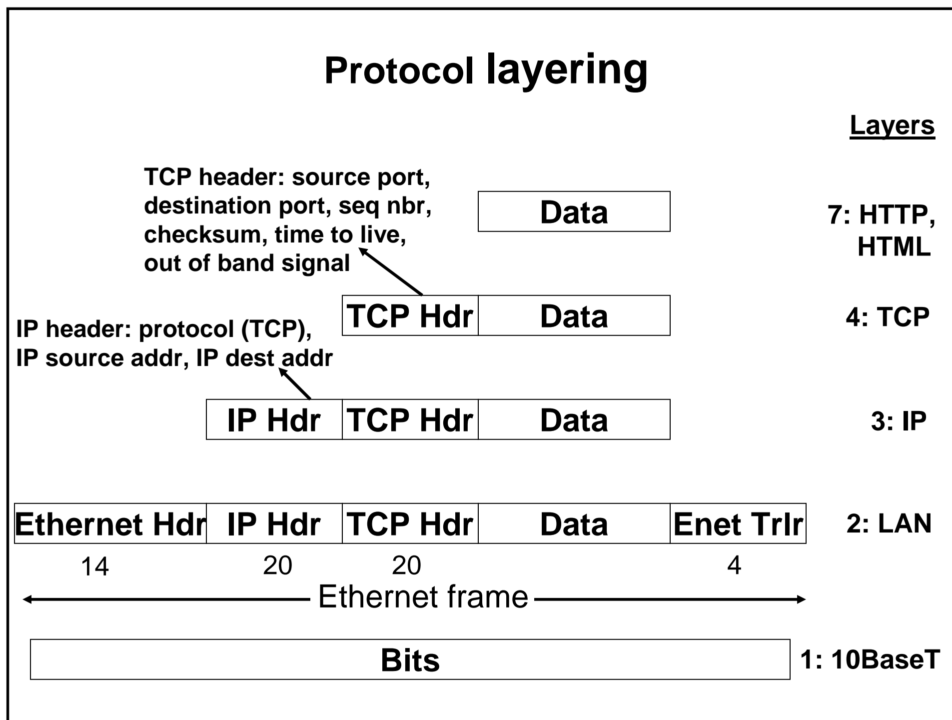
December 13, 2005

Java and the Web

Internet and TCP/IP

- Internet is "just" a set of loosely interconnected networks
  - A set of local area networks connected via wide area networks
  - Network segments interconnect via routers:
    - Dedicated computers that manage packets of data
  - TCP/IP is the universal data protocol on the network
  - Actual format, content is left to higher-level protocols, like the Web
- TCP/IP connections
  - Client is typically a data consumer that sends short requests
    - On Web, client is a browser
  - Server is typically a data provider that sends long responses
    - Listen for requests and transmit desired data, static or dynamic
    - On Web, servers talk a protocol called HTTP on port 80
  - TCP/IP connection is active only long enough to exchange data
    - Avoid overhead of many communication channels, but lose state


Protocol Layering
Server computer or router
Browser PC
Layer 7: Application
Web pgs
Layer 7: Application
(html)
HTTP
Layer 4: Transport
Connections
Layer 4: Transport
TCP
Layer 3: Network
Packets
Layer 3: Network
IP
Layer 2: Data Link
Frames
Layer 2: Data Link
Ethernet
Layer 1: Physical
Bits
Layer 1: Physical
10BaseT

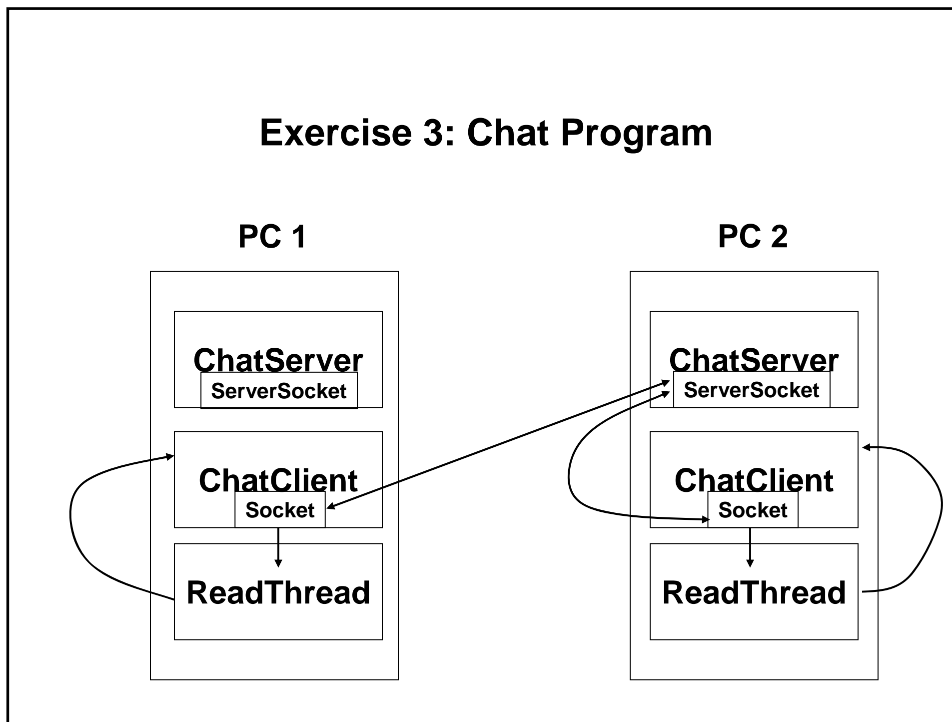

Protocol layering
Layers
TCP header: source port, destination port, seq nbr, checksum, time to live, out of band signal
Data
7: HTTP, HTML
TCP Hdr
Data
4: TCP
IP header: protocol (TCP), IP source addr, IP dest addr
IP Hdr
TCP Hdr
Data
3: IP
Ethernet Hdr
IP Hdr
TCP Hdr
Data
Enet Trlr
2: LAN
14
20
20
4
Ethernet frame
Bits
1: 10BaseT

Exercise: TCP/IP Connection

```
// Download and run TcpipTest. What is its output?
// At what layer (1, 2, 3, 4, or 7) is this program operating?
import java.net.*;
import java.io.*;

public class TcpipTest {
   public static void main(String[] args) {
     try {     // Socket is tcp/ip connection: ip address, port
        Socket s= new Socket("time-a.nist.gov", 13);
        InputStreamReader is= new
                        InputStreamReader(s.getInputStream());
        BufferedReader b= new BufferedReader(is); // Same as file!
        String currentLine = "";
        while ((currentLine = b.readLine()) != null)
          System.out.println(currentLine);
        b.close();
     } catch (IOException e) {
        System.out.println(e);
     }
   }
}  // Socket programs usually use threads, due to delays, losses…
```

Exercise 2: IP Addresses

```
// Download and run AddressTest to see IP addresses
import java.net.*;
import javax.swing.*;

public class AddressTest {
   public static void main(String[] args) {
     try {
        InetAddress local= InetAddress.getLocalHost();
        System.out.println("Local  host: "+ local);

        String input= JOptionPane.showInputDialog
               ("Enter host (e.g., web.mit.edu): ");
        InetAddress other= InetAddress.getByName(input);
        System.out.println("Other host: "+ other);

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



Case Study: An Internet Chat Program

- **We will look at the design of some programs that enable people to:**
  - **find each other on the Internet**
  - **Initiate and participate in text-based chat sessions**
- **We'll start with creating a Chat Location Server – this will be a registry of people who are open to contact via our chat program. Think of this as a "registry".**

Chat Location Server: Design Issues

- To keep things simple, we'll keep a Map of users (identified by a name they choose), their IP addresses, and the time when they registered.
- We're skipping over issues of security (such as verifying people are who they claim to be). In a real system, you might have passwords that people have to provide in order to register.
- Since people move around (on wireless networks, for example, their IP address can change. We'll design our locator service so that registration is only good for a fixed time period, after which it must be renewed.

Using Threads

- Our program will have a main thread that will listen on a "well known" port for registration or lookup requests.
- We'll have a thread that periodically reviews who is registered and eliminates those older than some fixed time.
- Once request to register is received, a separate thread will be created for each request.
- If a response to an request is needed, we will use a separate thread for response.

Chatter class

```
import java.util.Date;

    class Chatter {
        String username;
        Date timeRegistered;
        String address;

        Chatter(String u, String a) {
            username = u;
            address = a;
            timeRegistered = new Date();
        }
    }
```

Main, p.1

```
public class ChatLocatorService {
    public static final int CHATLOCATORPORT = 1301;
    public static final int CHATRESPONSEPORT = 1302;
    public static final int NSERVE = 30;
    public static final long DURATION = 1000 * 60 * 60 * 4;
    private  DateFormat formatter =
        DateFormat.getDateTimeInstance( DateFormat.SHORT,
        DateFormat.MEDIUM );
    private static HashMap<String, Chatter> userList =
         new HashMap<String, Chatter>();

    public static void main(String[] args) {
        new ChatLocatorService().startListening();
    }
```

Main, p.2

```
protected  void startListening() {
        ServerSocket soc = null;
        new ExpirationThread().start();
        try {
            soc = new
                    ServerSocket(ChatLocatorService.CHATLOCATORPORT,
                    ChatLocatorService.NSERVE);
             System.out.println("Server Socket created");
        } catch (IOException e) {
             System.out.println("IO error creating ServerSocket");
             System.exit(1);
        }
```

Main, p.3

```
while (true) {
            try {
                Socket chatSocket = soc.accept();
                System.out.println("New service request accepted");
                GetInputThread rThread = new GetInputThread(this,
chatSocket);
                rThread.start();
            } catch (IOException e) {
                 System.out.println("IO error creating Socket");
            }
        }
}
```

Main, p. 4

```
synchronized void outputList()
{
        Set<String> keySet = userList.keySet();
        System.out.println("Current registered chatters");
        for (String s : keySet) {
                Chatter c = userList.get(s);
                System.out.println("User: " + c.username + " IP:" +
                        c.address +  " time: " +
                        formatter.format(c.timeRegistered));
        }
}
```

Main, p. 5

```
synchronized void registerChatter(String user, String ipAddress) {
        if(userList.get(user) != null) {
                userList.remove(user);
                System.out.println("User :" + user + " removed");
        }
        userList.put(user, new Chatter(user, ipAddress));
        System.out.println("User :" + user + " added");
        outputList();
}

synchronized void unregisterChatter(String user) {
        userList.remove(user);
}

synchronized Chatter lookupChatter(String user) {
        return userList.get(user);
}
```

Main, p. 6

```
    synchronized void sendChatterAddress(String user, String destination)
    {
        GetOutputThread oThread = new
                GetOutputThread(userList.get(user), destination);
        oThread.start();
    }

}
```

ExpirationThread, p. 1

```
    class ExpirationThread extends Thread {
    public ExpirationThread() {
        super();
    }

public void run() {
        while(true) {
                try {
                    sleep(ChatLocatorService.DURATION);
                    updateUserList();
                }
                catch(InterruptedException e) {
                    System.out.println("ExpirationThread interrupted");
                }
        }
    }
```

ExpirationThread, p. 2

```
    synchronized void updateUserList() {
            Set<String> keySet = userList.keySet();
            Date now = new Date();
            System.out.println("Update of user list started");
            for (String s : keySet) {
               Chatter c = userList.get(s);
               if(now.getTime() - c.timeRegistered.getTime() > DURATION)
               {
                       userList.remove(s);
                       System.out.println("Registration of user: " + c.username
                               + " expired");
               }
            }
            System.out.println("Update of user list completed");
         }
    }
}
```

GetInputThread, p. 1

```
import java.io.IOException;
import java.net.*;
import java.util.Scanner;

public class GetInputThread extends Thread {
    private ChatLocatorService locator; // class providing lookup services
    private Socket input; // stream to read from

    public GetInputThread(ChatLocatorService c, Socket s) {
            super();
            input = s;
            locator = c;
    }
```

GetInputThread, p. 2

```
public void run() {
        try {
                Scanner scan = new Scanner(input.getInputStream());
                String command = scan.next();
                if (command.equals("REGISTER")) {
                        String userName = scan.next();
                        String ipAddress = scan.next();
                        locator.registerChatter(userName, ipAddress);   }
                else if (command.equals("UNREGISTER")) {
                        String userName = scan.next();
                        locator.unregisterChatter(userName); }
                else if (command.equals("LOOKUP")) {
                        String userName = scan.next();
                        String requestIP = scan.next();
                        locator.sendChatterAddress(userName, requestIP); }
                scan.close();
                input.close();
        } catch (IOException e) {
                System.out.println("IO Error in ChatLocatorService");
} } }
```

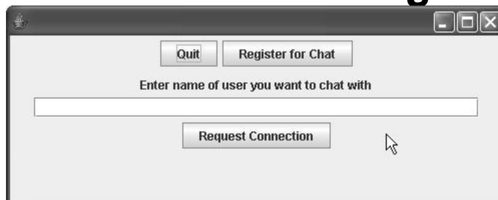
GetOutputThread, p. 1

```
import java.io.IOException;
import java.net.Socket;
import java.io.OutputStreamWriter;

public class GetOutputThread extends Thread {

    private Chatter chatter; // session to forward characters
    private String destination; // IP address to send info to

    public GetOutputThread(Chatter c, String dest) {
            super();
            destination = dest;
            chatter = c;
    }
```

GetOutputThread, p. 2

```
public void run() {
    try {
        Socket output = new Socket(destination,
         ChatLocatorService.CHATRESPONSEPORT );
        OutputStreamWriter out = new
            OutputStreamWriter(output.getOutputStream());
        if(chatter == null)
            out.write("NULL");
        else {
            out.write("USER ");
            out.write(chatter.username);
            out.write(" ");
            out.write(chatter.address);
        }
        out.close( );
        output.close( );
    } catch (IOException e) {
        System.out.println("IO Error in GetOutputThread");
    }
} }
```

Chat Program Interface



HTTP protocol

- **Four phases:**
  - **<u>Open tcp/ip connection:</u> Based on URL**
  - **<u>Http request</u>: Browser opens connection to server and sends:**
    - **Request method, (and request data at bottom if POST or PUT request)**
    - **URL,**
    - **HTTP version number**
    - **Header information (informational, optional), terminated with blank line**
  - **<u>Http response</u>: Server processes request and sends:**
    - **HTTP protocol version and status code**
    - **Header information, terminated by blank line**
    - **Text (data)**
  - **<u>Close tcp/ip connection</u>**

HTTP request examples

Typical browser request: telnet web.mit.edu 80, then type:

GET /about-mit.html HTTP/1.1

Host: web.mit.edu ***(required)***

Accept: text/html, text/plain, image/jpeg, image/gif, */* ***(optional)***

(blank line)

Typical server response:

HTTP/1.1 200 OK

Server: Apache/1.3.3 Ben-SSL/1.28 (Unix)

Content-Type: text/html

Content-Length: 8300

(blank line)

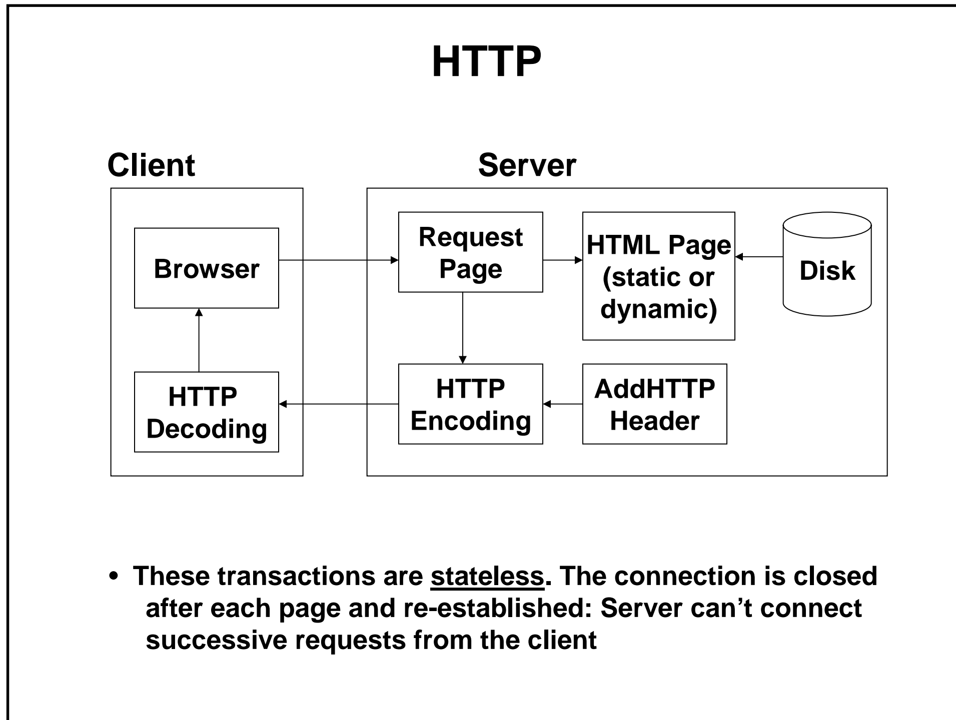
<HTML>

<HEAD><TITLE>About MIT</TITLE></HEAD>

<BODY>The mission of MIT…</BODY>

</HTML>

HTTP



- These transactions are stateless. The connection is closed after each page and re-established: Server can't connect successive requests from the client

Hypertext markup language (html) example

<HTML>
<HEAD>
<TITLE> Welcome to the Aircraft Parts System </TITLE>
</HEAD>
<BODY>
<H1> Welcome to the Aircraft Parts System </H1>
This system handles orders for aircraft and balloon parts. We handle aircraft parts, and parts for all balloon manufacturers. We comply with the latest US regulations.
<P>
The use of this system is subject to <A HREF= MITRule.html> MIT rules and regulations. </A>
</BODY>
</HTML>

(Aircraft1.html)

HTML

- Tags (e.g. <TAG>) never display but direct the browser
- Often in pairs (e.g. <TAG> and </TAG>) to delimit section
- Some tags have attributes (e.g. <A HREF=abc.html> </A>)
- HTML document begins with <HTML>, ends with </HTML>
  - Two sections within document: HEAD and BODY
  - Head has identifying information not displayed
  - Body is displayed, with formatting:
    - Paragraph <P>
    - Header levels 1 through 6 <H1> through <H6>
    - Line break or carriage return
    - Anchor <A>, placed around text or graphics; used for hyperlinks

HTTP and HTML

- HTTP
  - Is only direct form of interaction between browser and server
  - Was an extremely perceptive extension of email, ftp protocols by Tim Berners-Lee to enable Web browsers
  - Request-response paradigm
  - Connection made for each request/response pair
  - Most popular protocol on Internet
- HTML
  - Text description language, based on tags
  - High level description rather than specific formatting
  - HTML is static, which is not sufficient for Web applications
  - Many extensions and changes (scripting, Java) for dynamic content

HTML forms

- Used as front ends to server programs
  - Java Server Pages, servlets, .NET framework (Microsoft)
- Forms are user interface components to collect data from user and transmit it to the server application program
  - Forms are placed on Web pages that can also have other elements
  - Java applets can be used but rarely are
  - All of these run on the browser and are user interface components

How HTML forms transmit data

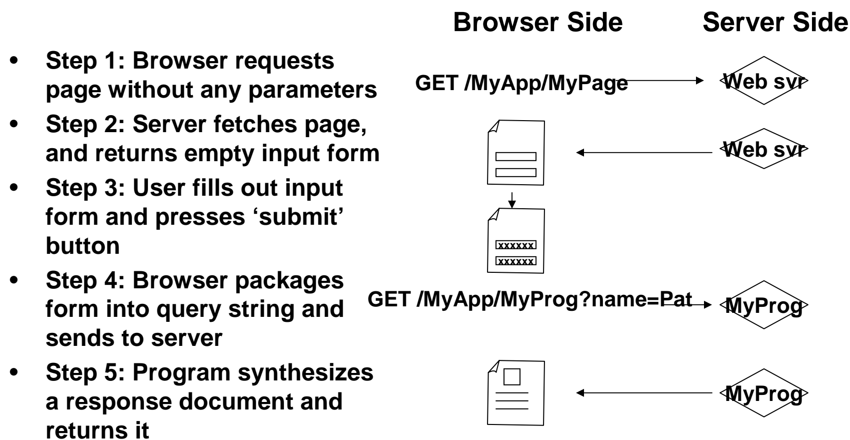
- Forms allow a series of components to be placed on the page
  - Each component has a name and a value
  - Entire form is associated with the URL of a server side program that will process the input data
  - Form data is sent when user presses 'Submit' button (component)
  - Data is sent to URL as string of form:
    - `Name1=Value1&Name2=Value2&…NameN=ValueN`
  - If data is sent with HTTP GET command, it is appended to end of GET string after a ?:
    - `GET /Index.html?Name1=Value1…`
  - If data is sent with HTTP PUT command, it is sent after the blank line as a string
  - Server programs (Java Server Pages, servlet) have methods to extract the data from the string and use it in the program

HTML 4.0 tags for forms

Tag	Type	Definition
<FORM>		Start a form
<INPUT TYPE=	text	Single line of text entry
	password	Single line password entry
	file	File to upload, with ‘Browse’ button
	checkbox	Checkbox
	radio	Radio button (option box)
	image	Image acting as button
	hidden	Track user, store predefined input, etc.
	submit	Submit button for form
	reset	Button to restore default values
<SELECT>		List box or combo box
<OPTION>		Item in scrolling list or popup menu
<TEXTAREA>		Start multiple-line text entry field

Browser-server interaction

- **Step 1: Browser requests page without any parameters**
- **Step 2: Server fetches page, and returns empty input form**
- **Step 3: User fills out input form and presses ‘submit’ button**
- **Step 4: Browser packages form into query string and sends to server**
- **Step 5: Program synthesizes a response document and returns it**



Java on servers

- Java servlets. Implement the following steps:
  - Read data sent by browser
    - Usually entered by user, but could be from applet or JavaScript
  - Look up other information from the HTTP request
    - Browser capabilities, cookies, host, etc. from HTTP headers
  - Generate results
    - Access database, execute program (possibly via CORBA/COM), etc.
  - Format the results as an HTML or other document
    - Servlets can generate GIF, gzip, many MIME types
  - Set HTTP response parameters in headers
  - Send document back to browser
- Java Virtual Machine (JVM) runs the servlets on the server
- These evolved from applets (limited use) into servlets (extremely useful, secure, standard)

Servlet example

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Hello100 extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>Hello 1.00</TITLE></HEAD>\n" +
                "<BODY>\n" +
                "<H1>Hello 1.00</H1>\n" +
                "</BODY></HTML>");}}
```

Java Server Pages (JSP)

- Allow us to mix static HTML with dynamically generated content from servlets
  - Many Web pages are primarily static
  - Changeable parts are in only a few locations on the page
- A servlet requires us to program the entire page
- JSP allows the Web designer to write static HTML and leave stubs for dynamic content
  - Java programmers can then write the stubs
- ASP.NET is very similar (Microsoft)

JSP example

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML> <HEAD>
<TITLE>JSP Example</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H2>JSP Expressions</H2>
<UL>
  <LI>Current time: <%= new java.util.Date() %>
  <LI>Your hostname: <%= request.getRemoteHost() %>
  <LI>Your session ID: <%= session.getId() %>
  <LI>The <CODE>testParam</CODE> form parameter:
      <%= request.getParameter("testParam") %>
</UL>
</BODY>
</HTML>
```

JSP Example 2

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD> <TITLE>Color Test</TITLE> </HEAD>
<%
  String bgColor = request.getParameter("bgColor");
  boolean hasColor;
  if (bgColor != null) {
      hasColor = true; }
    else {
      hasColor = false;
      bgColor = "WHITE";}
%>
<BODY BGCOLOR="<%= bgColor %>">
<H2 ALIGN="CENTER">Color Testing</H2>
<%
if (hasColor) {
  out.println(bgColor + " used"); }
else {
  out.println("Default color (WHITE) used";
}
%>
</BODY>
</HTML>
```