Recommended reading:

Core SERVLETS and JAVASERVER by Marty Hall, Sun Microsystems Press/Prentice Hall PTR Book, ISBN 0-13-089340-4

For on-line copy:

http://pdf.coreservlets.com

Other resources:

http://courses.coreservlets.com/Course-Materials/index.html

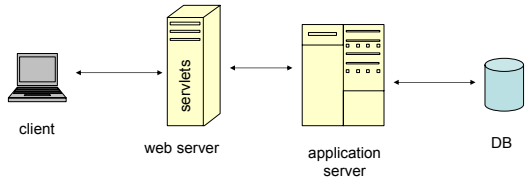
This handouts are adapted from Marty Hall's notes.

A Servlet's Job

- A servlet can be regarded as a Java program residing on a web server.
- Servlets are used to generate web pages dynamically.
- Read explicit data sent by client (form data)
- Read implicit data sent by client (request headers)
- Generate the results
- Send the explicit data back to client (HTML)
- Send the implicit data to client (status codes and response headers)

Response from the web server

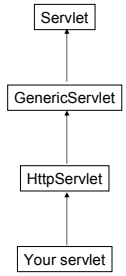
- The response from a web server consists of
  - a status line
    - HTTP/1.1 200 OK
  - Some response headers
    - Content-Type: text/html
    - Date: Mon, 17 Feb 2003 23:11:38 GMT
  - The document.
    - <HTML><BODY>Hello World </BODY></HTML>



Why Build Web Pages Dynamically?

- The Web page is based on data submitted by the user
  - results page from search engines and DB queries
- The Web page is derived from data that changes frequently
  - a weather report or stock price

Servlet Classes



Commonly used servlet methods

- Class HttpServlet includes some methods for handling the corresponding HTTP request methods.
  - doGet() , doPost(), doHead() etc.
- In your servlets, you need to overwrite these methods to make the servlets behave according to your requirements.

First Servlet

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

public class HelloWorld extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.println("Hello World!");
  }
}
```

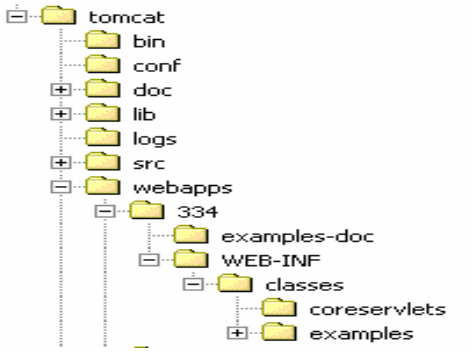

Creating, Deploying and Executing Servlet/JSP in Tomcat3.2

- Tomcat3.2 is the official reference implementation of the servlet2.2 and JSP1.1 specifications.
- It can be used as a small stand-alone server for testing servlets and JSP pages.
- Tomcat is free from http://jakarta.apache.org/.
- There are many versions of Tomcat. The version that I use is Tomcat3.2.

Setup Environment on Windows

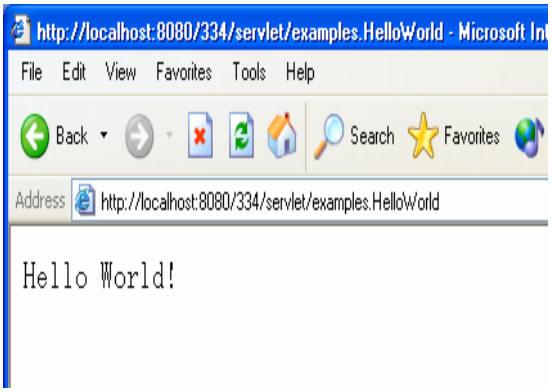
- After installing Tomcat, you need to set up some environment variables.
- TOMCAT_HOME should be set to the directory where Tomcat is installed.
  - set TOMCAT_HOME=d:\tomcat
- CLASSPATH should also be extended to include the jar file which contains the servlet package.
  - set CLASSPATH=d:\tomcat\lib\servlet.jar;%CLASSPATH%

Tomcat 3.2 Directory Structure



Compiling and Invoking HelloWorld

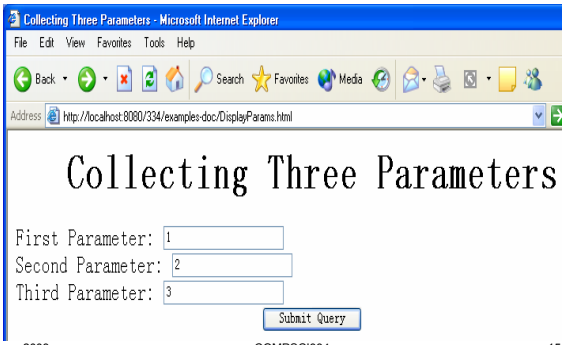
- Place source code HelloWorld.java in directory examples.
- Compile HelloWorld.java.
- Start Tomcat
  - Tomcat listens to http request on port 8080
- Send request
  - http://localhost:8080/334/servlet/examples.HelloWorld



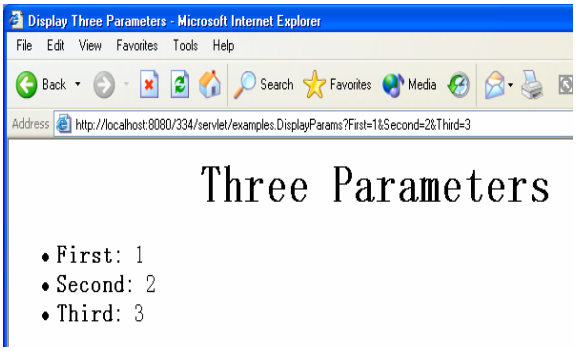
Packaging Servlets

- Move the files to a subdirectory that matches the intended package name
  - For example, HelloWorld.java under examples
- Insert a package statement in the class file
  - package examples;
- Include package name in URL
  - http://localhost:8080/334/servlet/examples.HelloWorld

Handling the Client Request




```
<HTML>
<HEAD>
 <TITLE>Collecting Three Parameters</TITLE>
</HEAD>
<H1 ALIGN="CENTER">Collecting Three Parameters</H1>

<FORM ACTION="/334/servlet/examples.DisplayParams" METHOD="PUT">
 First Parameter:  <INPUT TYPE="TEXT" NAME="First"><BR>
 Second Parameter: <INPUT TYPE="TEXT" NAME="Second"><BR>
 Third Parameter:  <INPUT TYPE="TEXT" NAME="Third"><BR>
 <CENTER>
  <INPUT TYPE="SUBMIT">
 </CENTER>
</FORM>

</BODY>
</HTML>
```

Response in HTML

```
<TITLE>Display Three Parameters</TITLE>
<H1 ALIGN=CENTER> Three Parameters </H1>
<UL>
  <LI><B>First</B>: 1
  <LI><B>Second</B>: 2
  <LI><B>Third</B>: 3
</UL>
</BODY></HTML>
```

```
package examples;

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

public class DisplayParams extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
```



```
out.println("<TITLE>Display Three Parameters</TITLE>\n" +
            "<H1 ALIGN=CENTER> Three Parameters </H1>\n" +
            "<UL>\n" +
            "  <LI><B> First </B>: "
            + request.getParameter("First") + "\n" +
            "  <LI><B> Second </B>: "
            + request.getParameter("Second") + "\n" +
            "  <LI><B> Third </B>: "
            + request.getParameter("Third") + "\n" +
            "</UL>\n" +
            "</BODY></HTML>");
  }
}
```



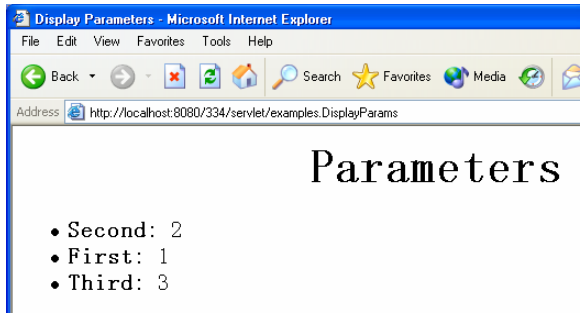
```
<HTML>
<HEAD>
  <TITLE>Collecting Three Parameters</TITLE>
</HEAD>
<H1 ALIGN="CENTER">Collecting Three Parameters</H1>

<FORM ACTION="/334/servlet/examples.DisplayParams" METHOD="POST">
  First Parameter:  <INPUT TYPE="TEXT" NAME="First"><BR>
  Second Parameter: <INPUT TYPE="TEXT" NAME="Second"><BR>
  Third Parameter:  <INPUT TYPE="TEXT" NAME="Third"><BR>
  <CENTER>
    <INPUT TYPE="SUBMIT">
  </CENTER>
</FORM>

</BODY>
</HTML>
```



```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
      doGet(request, response);
  }
```

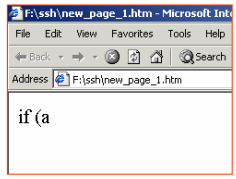


```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<TITLE>Display Parameters</TITLE>" +
                "<H1 ALIGN=CENTER> Parameters </H1>\n" +
                "<UL>\n");
    Enumeration paramNames =
request.getParameterNames();
    while (paramNames.hasMoreElements()) {
        String paramName =
(String)paramNames.nextElement();
        out.println("<LI><B>"+paramName+"</B>:“
                    +request.getParameter(paramName));
    }
    out.println("</UL>\n" + "</BODY></HTML>");
}
```

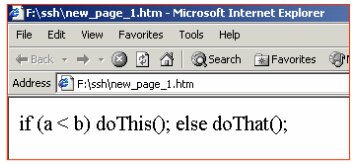



Filtering Strings for HTML-Specific Characters

- HTML files contain some special characters, e.g. <, >, &, etc., which are used as directives in HTML.
- If we want to display a character which is the same as a special HTML character, to make sure that resulting page can be displayed correctly, we need to substitute the character as a sequence of other characters.
  - < → &lt, > → &gt, " → &quot, & → &amp




```
<html>
<body>
if (a<b)
     doThis();
else
     doThat();
</body>
</html>
```




```
<html>
<body>
if (a &lt b)
        doThis();
else
        doThat();
</body>
</html>
```



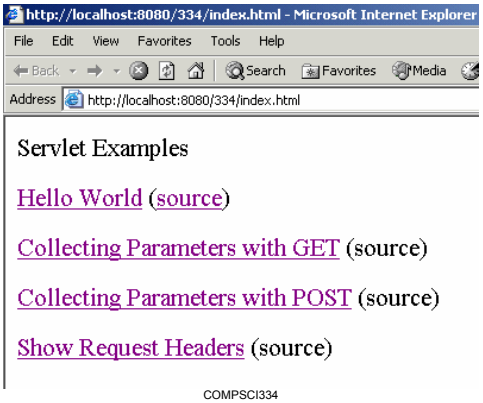
```
public static String filter(String input) {
  StringBuffer filtered = new StringBuffer(input.length());
  char c;
  for(int i=0; i<input.length(); i++) {
    c = input.charAt(i);
    if (c == '<') {
      filtered.append("&lt;");
    } else if (c == '>') {
      filtered.append("&gt;");
    } else if (c == '"') {
      filtered.append("&quot;");
    } else if (c == '&') {
      filtered.append("&amp;");
    } else {
      filtered.append(c);
    }
  }
  return(filtered.toString());
}
```


Reading Request Headers

```
Accept: image/gif, image/jpg, */*
Accept-Encoding: gzip
Host: localhost:8080
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0;
Windows NT 5.0)
```


Some Useful Methods

- General
  - getHeader
  - getHeaderNames
- Related info
  - getMethod, getRequestURI, getProtocol



Showing Request Headers

Request Method: GET
Request URI: /334/servlet/examples.ShowRequestHeaders
Request Protocol: HTTP/1.1

Header Name	Header Value
User-Agent	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
Accept	image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-powerpoint, application/vnd.ms-excel, application/msword, application/x-shockwave-flash, */*
Host	localhost:8080
Accept-Encoding	gzip, deflate
Accept-Language	en-nz
Referer	http://localhost:8080/334/index.html
Connection	Keep-Alive

```
public class ShowRequestHeaders extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<BODY><HTML><H1 ALIGN=CENTER> "+
                "Showing Request Headers</H1>\n" +
                "<B>Request Method: </B>" +
                request.getMethod() + "<BR>\n" +
                "<B>Request URI: </B>" +
                request.getRequestURI() + "<BR>\n" +
                "<B>Request Protocol: </B>" +
                request.getProtocol() + "<BR><BR>\n" +
                "<TABLE BORDER=1 ALIGN=CENTER>\n" +
                "<TR>\n" +
                "<TH>Header Name<TH>Header Value");
```



```
    Enumeration headerNames = request.getHeaderNames();
    while(headerNames.hasMoreElements()) {
      String headerName = (String)headerNames.nextElement();
      out.println("<TR><TD>" + headerName);
      out.println("    <TD>" + request.getHeader(headerName));
    }
    out.println("</TABLE>\n</BODY></HTML>");
  }
```


Sending Compressed Pages

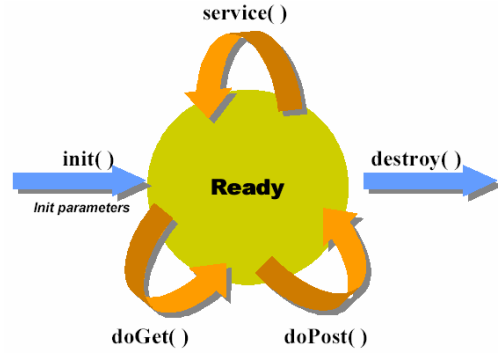
- Gzipped files are compressed files. They can reduce the download time of long text pages.
- Recent browsers automatically uncompress the files whose “Content-Encoding” header marked as “gzip”.
- To send a gzipped page, the servlet must use Java’s GZIPOutputStream to generate the page.


```
public class EncodedPage extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    String encodings = request.getHeader("Accept-Encoding");
    PrintWriter out;
    if ((encodings != null) &&
        (encodings.indexOf("gzip") != -1)) {
      OutputStream out1 = response.getOutputStream();
      out = new PrintWriter(new GZIPOutputStream(out1), false);
      response.setHeader("Content-Encoding", "gzip");
    } else {
      out = response.getWriter();
    }
```



```
      out.println("<HTML><BODY>" );
      String line = "Blah, blah, blah, blah, blah. ";
      for(int i=0; i<10000; i++) {
        out.println(line);
      }
      out.println("</BODY></HTML>");
      out.close();
    }
  }
```

Servlet Life Cycle



- **Init()**
  - Executed **once** when the servlet is first loaded.
  - Not called for each request.
  - Perform any set-up in this method.
- **Service()**
  - Do not override this method!
- **doGet(), doPost(), doXxx()**
  - Handles GET, POST, etc. requests.
  - Override these to provide desired behavior.
- **Destroy()**
  - Called when server deletes servlet instance.
  - Perform any clean-up, and save data to be read by the next init().

Restricting Access to Web Pages

- Step 1
  - Check whether there is Authorization header in the request.
    - Authorization=Basic bWFydHk6bWFydHlwdw==
  - If not, go to Step 2.
  - If so, skip over word "basic" and reverse the base64 encoding of the remaining part.
    - This results in a string of the form username:password.
    - Check the username and password against some stored information.
    - If it matches, return the page.
    - If not, go to Step 2.

- Step 2
  - Return a 401 (Unauthorized) response code and a header of the following form:
    - WWW-Authenticate: BASIC realm="some-name"
  - The header instructs browser to pop up a dialog box telling the user to enter a name and password, then to reconnect with that username and password embedded in a single base64 string inside the Authorization header.


```
public class ProtectedPage extends HttpServlet {
  private Properties passwords;

 public void init(ServletConfig config)
     throws ServletException {
     passwords = new Properties();
          passwords.put("abc001", "micky");
          passwords.put("abc002", "goofy");
 }
```



```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  String authorization = request.getHeader("Authorization");
  if (authorization == null) {
    askForPassword(response);
  } else {
                  System.out.println("Authorization="+authorization);
    String userInfo = authorization.substring(6).trim();
    BASE64Decoder decoder = new BASE64Decoder();
    String nameAndPassword =
      new String(decoder.decodeBuffer(userInfo));
    int index = nameAndPassword.indexOf(":");
    String user = nameAndPassword.substring(0, index);
    String password = nameAndPassword.substring(index+1);
    String realPassword = passwords.getProperty(user);
```

```
if ((realPassword != null) &&
      (realPassword.equals(password))) {
    out.println("<HTML><BODY>"+
            "<H1 ALIGN=CENTER>Welcome to the Protected Page"+
              "</H1>\n </BODY></HTML>");
   } else {
    askForPassword(response);
   }
  }
}
```



```
private void askForPassword(HttpServletResponse response) {
  response.setStatus(response.SC_UNAUTHORIZED); // Ie 401
  response.setHeader("WWW-Authenticate",
                "BASIC realm=\"privileged-few\"");
 }


 public void doPost(HttpServletRequest request,
                HttpServletResponse response)
   throws ServletException, IOException {
  doGet(request, response);
 }
}
```


Redirect pages


```
<HTML>
<HEAD>
 <TITLE>Searching the Web</TITLE>
</HEAD>

<H1 ALIGN="CENTER">Searching the Web</H1>

<FORM ACTION="/334/servlet/examples.SearchEngines">
 <CENTER>
  Search String:
  <INPUT TYPE="TEXT" NAME="searchString"><BR>
 </CENTER>
</FORM>

</BODY>
</HTML>
```



```
public class SearchEngines extends HttpServlet {
 public void doGet(HttpServletRequest request,
             HttpServletResponse response)
   throws ServletException, IOException {
  String searchString = request.getParameter("searchString");
  if ((searchString == null) ||
     (searchString.length() == 0)) {
   response.sendError(response.SC_BAD_REQUEST,
                "Missing search string.");
   return;
  }
  searchString = URLEncoder.encode(searchString);
  response.sendRedirect("http://www.google.co.nz/search?q="
                +searchString);
 }
```


Handling Cookies

- Cookies are textual information that a web server sends to a browser.
  - Cookies are used by the web server to record information about the clients.
- When a browser sends a request to a site, it sends the cookies received from site to the web server.

```
public class SearchEnginesFrontEnd extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    Cookie[] cookies = request.getCookies();
          String searchString = getCookieValue(cookies, "searchString");
          if (searchString==null) searchString="";
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
```



```
out.println
     ("<HTML><BODY>\n" +
      "<H1 ALIGN=\"CENTER\">Searching the Web</H1>\n" +
      "<FORM
ACTION=\"/334/servlet/examples.CustomizedSearchEngines\">\n"
      +  "<CENTER>\n" +
      "Search String:\n" +
      "<INPUT TYPE=\"TEXT\" NAME=\"searchString\"
VALUE=\""+searchString+"\"><BR>\n" +
      "<INPUT TYPE=\"SUBMIT\" VALUE=\"Search\">\n" +
      "</CENTER>\n" +
      "</FORM>\n" +
      "</BODY>\n" +
      "</HTML>\n");
  }
```



```
public String getCookieValue(Cookie[] cookies, String cookieName) {
   if (cookies != null) {
      for(int i=0; i<cookies.length; i++) {
          Cookie cookie = cookies[i];
          if (cookieName.equals(cookie.getName()))
             return(cookie.getValue());
      }
   }
   return(null);
 }
}
```


Session Tracking

- Why session tracking?
  - HTTP protocol is stateless.
  - When doing on-line shopping, a customer might browse many pages and make many orders on different pages.
  - When a customer checks out, we need to find out the orders the customer made while in the shop.

Techniques for session tracking

- Using cookies
  - Send a cookie to the client when the client first visit the site.
  - Whenever the client comes back the same site, the cookie is sent back by the client's browser.
  - Problem
    - Client might disallow cookie

- URL-Rewriting
  - Client appends some extra data that identifies the session on the end of each URL
    - http://host/path/file.html?session=xyz
  - Server associates that identifier with data it has stored about that session
  - Advantage
    - Works even if cookies are disabled or unsupported
  - Disadvantage
    - Must encode all URLs to include the session information that refer to your own site

The Session Tracking API

- Session objects live on the server
- Automatically associated with client via cookies or URL-rewriting
  - Use request.getSession(true) to get either existing or new session
  - Behind the scenes, the system looks at cookie or URL extra info and sees if it matches the key to some previously stored session object. If so, it returns that object. If not, it creates a new one, assigns a cookie or URL info as its key, and returns that new session object.

- The session object is like a hash table
- setAttribute(String name, Object o)
- getAttribute(String name)
- getAttributeNames()


```
public class ShowSession extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
   response.setContentType("text/html");
   PrintWriter out = response.getWriter();
   HttpSession session = request.getSession(true);
   String heading;
   Integer accessCount = (Integer)session.getAttribute("accessCount");
   if (accessCount == null) {
     accessCount = new Integer(0);
     heading = "Welcome, Newcomer";
   } else {
     heading = "Welcome Back";
     accessCount = new Integer(accessCount.intValue() + 1);
   }
   session.setAttribute("accessCount", accessCount);
```



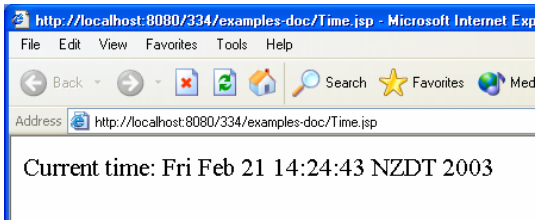
```
out.println("<HTML><BODY>\n" +
            "<H1 ALIGN=\"CENTER\">" + heading + "</H1>\n" +
            "<H2>Information on Your Session:</H2>\n" +
            "<TABLE BORDER=1 ALIGN=\"CENTER\">\n" +
            "<TR >\n" +
            "  <TH>Info Type <TH>Value\n" +
            "<TR>\n" +
            "  <TD>ID\n <TD>" + session.getId() + "\n" +
            "<TR>\n "+
            "  <TD>Creation Time\n "+
            "  <TD>" + new Date(session.getCreationTime()) + "\n" +
```

```
        "<TR>\n" +
        "  <TD>Time of Last Access\n" +
        "  <TD>" + new Date(session.getLastAccessedTime()) + "\n" +
        "<TR>\n" +
        "  <TD>Number of Previous Accesses\n" +
        "  <TD>" + accessCount + "\n" +
        "</TABLE>\n" +
        "</BODY></HTML>");
    }
```


The Need for JSP

- The main drawback of using servlet is that all the HTML text must be generated by println statement.
  - Web page writers must know programming.
- The idea behind JSP
  - A page contains a mixture of Java and HTML code
  - Use regular HTML for most of page
  - Mark Java code with special tags
  - Entire JSP page gets translated into a servlet (once), and servlet is what actually gets invoked (for each request)


```
<HTML>
<BODY>
Current time: <%= new java.util.Date() %>
</BODY>
</HTML>
```



- JSP makes it easier to create and maintain HTML, while still providing full access to servlet code
- JSP pages get translated into servlets the first time the pages are requested
  - Servlet code gets executed. The original JSP page is totally ignored.
  - Page translation does not occur for each request.

Basic Syntax

- HTML Text
  - <H1>Blah</H1>
  - Passed through to client. Really turned into servlet code that looks like
    - out.print("<H1>Blah</H1>");
- JSP Comments
  - <%-- Comment --%>
  - Not sent to client
  - To get <% in output, use <\%

- Expressions
  - <%= Java Expression %>
  - Expression evaluated, converted to String, and placed into HTML page at the place it occurred in JSP page
  - <%= new java.util.Date() %>

JSP/Servlet Correspondence

- **Original JSP**

```
<H1>A Random Number</H1>
<%= Math.random() %>
```

- **Possible resulting servlet code**

```
public void _jspService(HttpServletRequest request,
                        HttpServletResponse response)
    throws ServletException, IOException {
  response.setContentType("text/html");
  HttpSession session = request.getSession(true);
  JspWriter out = response.getWriter();
  out.println("<H1>A Random Number</H1>");
  out.println(Math.random());
  ...
}
```

Predefined Variables

- request
  - The HttpServletRequest
- response
  - The HttpServletResponse
- out
  - The JspWriter used to send output to the client
- session
  - The HttpSession associated with the request

JSP Scriptlets

- A collections of Java code enclosed with <% %>
  - <% Java code %>
  - <%

```
String queryData = request.getQueryString();
out.println("Attached GET data: " + queryData);
```

%>

- Scriplets need not be complete Java expressions
- Complete expressions are usually clearer and easier to maintain.

Example

```
<% if (Math.random() < 0.5) { %>
Have a <B>nice</B> day!
<% } else { %>
Have a <B>lousy</B> day!
<% } %>
```

Representative result

```
if (Math.random() < 0.5) {
  out.println("Have a <B>nice</B> day!");
} else {
  out.println("Have a <B>lousy</B> day!");
}
```


JSP Declarations

```
<HTML>
<BODY>
<H1>JSP Declarations</H1>

<%! private int accessCount = 0; %>
<H2>Accesses to page since server reboot:
<%= ++accessCount %></H2>

</BODY>
</HTML>
```

The scope of the declaration is in a JSP file.

JSP page Directive:

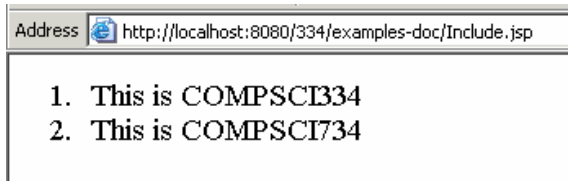
- By default, the servlet imports java.lang.*, javax.servlet.*, javax.servlet.jsp.*, javax.servlet.http.* packages.
  - Different servers might also import various other Java packages.
- To import other packages, the following forms are used:
  - <%@ page import="package.class" %>
  - <%@ page import="package.class1,...,package.classN" %>


```
<HTML>
<BODY>
<%@ page import="java.math.BigInteger" %>
<%! BigInteger bi = new java.math.BigInteger("12345678901234567890"); %>
Number is <%= bi.toString() %>
</BODY>
</HTML>
```


Including Files in JSP Documents

- To reuse JSP, HTML, or plain text content
- Including Files at Request Time
  - To permit updates to the included content without changing the main JSP page(s)
- Including Files at Page Translation Time
  - To reuse JSP content in multiple pages

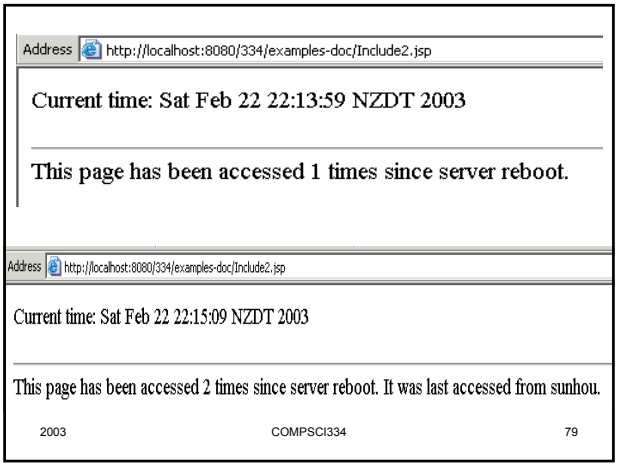

```
<HTML>
<BODY>
<OL>
<LI><jsp:include page="COMPSCI334.html" flush="true" />
<LI><jsp:include page="COMPSCI734.html" flush="true" />
</OL>
</BODY>
</HTML>
```




```
<%!
private int accessCount = 0;
private String accessHost = null;
%>
<P>
<HR>
This page has been accessed <%= ++accessCount %>
times since server reboot.
<%      if (accessHost != null) { %>
                It was last accessed from
                <%= accessHost %>.
<%      }
        accessHost = request.getRemoteHost(); %>
```



```
<HTML>
<BODY>
<%@ page import="java.util.Date" %>
Current time: <%= new java.util.Date() %>
<%@ include file="AccessCount.jsp" %>
</BODY>
</HTML>
```



Using JavaBeans with JSP

- In JSP pages, we should avoid using complicated programming logic.

What Are Beans?

- Java classes that follow certain conventions
  - Must have a zero-argument (empty) constructor
    - You can satisfy this requirement either by explicitly defining such a constructor or by omitting all constructors
  - Should have no public instance variables (fields)
  - Class variables should be accessed through methods called get*Xxx* and set*Xxx*
    - If class has method getTitle that returns a String, class is said to have a String *property* named title
    - Boolean properties use is*Xxx* instead of get*Xxx*


```
package examples;
public class StringBean {
  private String message = "No message specified";
  public String getMessage() {
    return(message);
  }
  public void setMessage(String message) {
    this.message = message;
  }
}
```


Basic Bean Use in JSP

- Format
  - <jsp:useBean id="*name*" class="*package.Class*" />
  - If an existing bean is to be accessed, it is retrieved. If a bean needs to be created, it is instantiated without explicit Java programming.
- Example
  - <jsp:useBean id="sb" class="examples.StringBean " />
  - Same as <% examples.StringBean sb = new examples.StringBean (); %>

Accessing Bean Properties

- Format
  - <jsp:getProperty name="*name*" property="*property*" />
- Example
  - <jsp:getProperty name="sb" property="message" />
  - Same as <%= sb.getMessage() %>

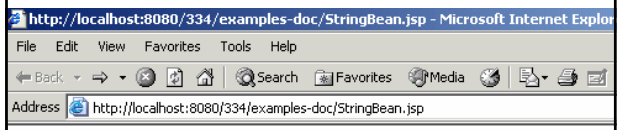
Setting Bean Properties:

- Format
  - <jsp:setProperty name="*name*" property="*property*" value="*value*" />
- Example
  - <jsp:setProperty name="sb" property="message" value="welcome" />
  - Same as <% sb.setMessage("welcome"); %>


```
<HTML>
<BODY>
<jsp:useBean id="stringBean" class="examples.StringBean" />

<OL>
<LI>Initial value (getProperty):
   <I><jsp:getProperty name="stringBean"
                       property="message" /></I>

<LI>Initial value (JSP expression):
   <I><%= stringBean.getMessage() %></I>
```



```
<LI><jsp:setProperty name="stringBean"
                     property="message"
                     value="Welcome 1" />
   Value after setting property with setProperty:
   <I><jsp:getProperty name="stringBean"
                       property="message" /></I>

<LI><% stringBean.setMessage("Welcome 2"); %>
   Value after setting property with scriptlet:
   <I><%= stringBean.getMessage() %></I>
</OL>

</BODY>
</HTML>
```

1. **Initial value (getProperty):** *No message specified*
2. **Initial value (JSP expression):** *No message specified*
3. **Value after setting property with setProperty:** *Welcome 1*
4. **Value after setting property with scriptlet:** *Welcome 2*

Associating Individual Properties with Input Parameters

- Format
  - <jsp:setProperty name="Bean's ID" property="Bean's property" param="form's parameter name" />
- Example
  - <jsp:setProperty name="stringBean" property="message" param="input" />


```
<HTML>
<BODY>
<FORM ACTION="property.jsp">
Enter message here: <INPUT TYPE="text" NAME="input"> <BR>
<INPUT TYPE="SUBMIT" VALUE="Submit">
</FORM>
</BODY>
</HTML>
```

```
<HTML>
<BODY>
<jsp:useBean id="stringBean" class="examples.StringBean" />
<jsp:setProperty name="stringBean" property="message" param="input" />
New value in Bean: <jsp:getProperty name="stringBean" property="message" />
</BODY>
</HTML>
```

Address http://localhost:8080/334/examples-doc/property.jsp?input=abcd

New value in Bean: abcd

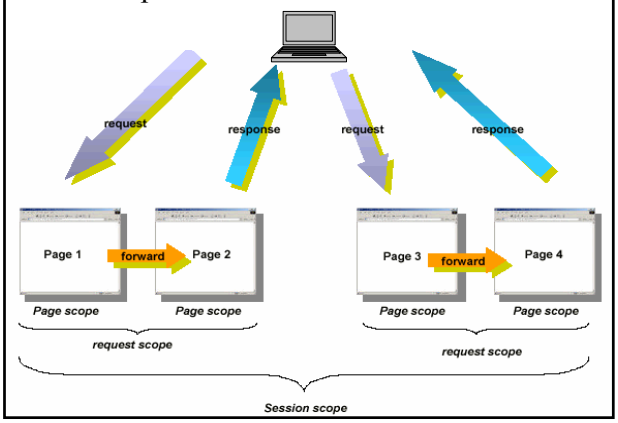
Associating All Properties with Input Parameters

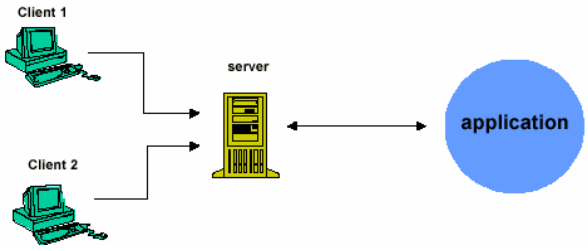
- If the names of the parameters in a form are the same as the names of the properties of a Bean, you can assign the values of the parameters directly to the corresponding properties in the Bean.
- Format
  - <jsp:setProperty name="Bean's ID" property="*" />


```
<HTML>
<BODY>
<FORM ACTION="property2.jsp">
Enter message here: <INPUT TYPE="text" NAME="message"> <BR>
<INPUT TYPE="SUBMIT" VALUE="Submit">
</FORM>
</BODY>
</HTML>
```

```
<HTML>
<BODY>
<jsp:useBean id="stringBean" class="examples.StringBean" />
<jsp:setProperty name="stringBean" property="*" />
New value in Bean: <jsp:getProperty name="stringBean" property="message" />
</BODY>
</HTML>
```


The Scope of JavaBean





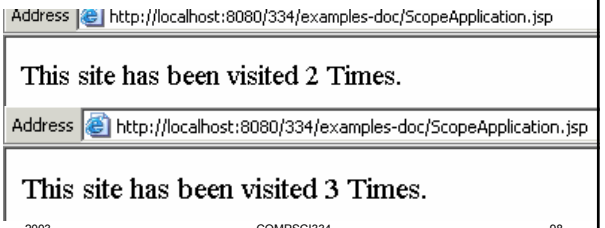

```
<HTML>
<BODY>
<jsp:useBean id="acb" class="examples.AccessCountBean" scope="page" />
This site has been visited
<jsp:getProperty name="acb" property="accessCount" />
Times.
<jsp:setProperty name="acb" property="accessCountIncrement" value="1" />
</BODY>
</HTML>
```

Address http://localhost:8080/334/examples-doc/ScopePage.jsp

This site has been visited 1 Times.

```
<HTML>
<BODY>
<jsp:useBean id="acb" class="examples.AccessCountBean" scope="session" />
This site has been visited
<jsp:getProperty name="acb" property="accessCount" />
Times.
<jsp:setProperty name="acb" property="accessCountIncrement" value="1" />
</BODY>
</HTML>
```


```
<HTML>
<BODY>
<jsp:useBean id="acb" class="examples.AccessCountBean" scope="application" />
This site has been visited
<jsp:getProperty name="acb" property="accessCount" />
Times.
<jsp:setProperty name="acb" property="accessCountIncrement" value="1" />
</BODY>
</HTML>
```



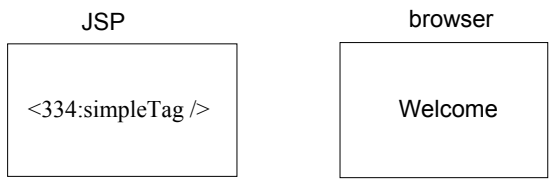
Creating Custom JSP Tag Libraries

- Improve the flexibility and power of JSP pages
- Separate JSP page content from behavior:
  - Content authors author content.
  - Code developers create tags.

Components That Make Up a Tag Library

- The Tag Handler Class
  - Java code that says how to actually translate tag into code
  - Usually extends javax.servlet.jsp.tagext.TagSupport or javax.servlet.jsp.tagext.BodyTagSupport
  - Goes in same directories as servlet class files
    - On Tomcat, the tag class files must be in a package.

- The Tag Library Descriptor File
  - XML file describing tag name, attributes, and implementing tag handler class
  - Normally stored with the JSP file
- The JSP File
  - Imports a tag library (referencing URL of descriptor file)
  - Defines tag prefix, e.g. 334:simplePrime
  - Uses tags



```
package examples.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class SimpleTag extends TagSupport {
  public int doStartTag() {
    try {
      JspWriter out = pageContext.getOut();
      out.print("Welcome");
    } catch(IOException ioe) {
      System.out.println("Error generating prime: " + ioe);
    }
    return(SKIP_BODY);
  }
}
```


Defining a Simple Tag Library Descriptor

- Start with XML header and DOCTYPE
- Top-level element is taglib
- Each tag defined by tag element with:
  - name, whose body defines the base tag name.
    - <name>simpleTag</name>
  - tagclass, which gives the fully qualified class name of the tag handler.
    - <tagclass>examples.tags.SimpleTag</tagclass>
  - bodycontent, which gives hints to development environments. Optional.
    - <bodycontent>empty</bodycontent>
  - info, which gives a short description.
    - <info>Outputs a welcome message.</info>


```
<tag>
  <name>simpleTag</name>
  <tagclass>examples.tags.SimpleTag</tagclass>
  <bodycontent>empty</bodycontent>
  <info>Outputs a welcome message.</info>
</tag>
```


Accessing Custom Tags From JSP Files

- Import the tag library
  - Specify location of TLD file
  - Define a tag prefix (namespace)
  - <%@ taglib uri="334-taglib.tld" prefix="334" %>
- Use the tags
  - <prefix:tagName />
  - Tag name comes from TLD file
  - Prefix comes from taglib directive
  - <334:simpleTag />


```
<HTML>
<BODY>
<H1>Simple Tag Example</H1>
<%@ taglib uri="334-taglib.tld" prefix="334" %>
<334:simpleTag />
</BODY>
</HTML>
```

Assigning Attributes to Tags

- Allowing tags like

```
<prefix:name
    attribute1="value1"
    attribute2="value2"
    ...
    attributeN="valueN"
/>
```

Attributes: The Tag Handler Class

- Use of an attribute called attribute1 simply results in a call to a method called setAttribute1
  - Attribute value is supplied to method as a String
  - Use a class variable to hold the value of the attribute
- To support <prefix:tagName attribute1="x" />, add the following to tag handler class:

```
public void setAttribute1(String value1) {
    doSomethingWith(value1);
}
```



```
package examples.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
public class WelcomeTag extends TagSupport {
  private int repeat = 1;
  private String colour = "black";
  public void setColour(String colour) {
        this.colour = colour;
  }
  public void setRepeat(String repeat) {
        try {
          this.repeat = Integer.parseInt(repeat);
        } catch(NumberFormatException nfe) {
          this.repeat = 2;
        }
  }
```



```
<334:welcomeTag />

With no attributes: <font color="black">Welcome </font><BR>
```

```
<334:welcomeTag colour="red" />

With "colour": <font color="red">Welcome </font><BR>
```

```
<334:welcomeTag colour="blue" repeat="3" />

With both attributes: <font color="blue">Welcome </font><font
color="blue">Welcome </font><font color="blue">Welcome
</font><BR>
```



```
public int doStartTag() {
  try {
   JspWriter out = pageContext.getOut();
        for (int i=0; i<repeat; i++)
          out.print("<font color=\""+colour+"\">"+"Welcome "+"</font>");
        out.println("<BR>");
  } catch(IOException ioe) {
   System.out.println("Error generating prime: " + ioe);
  }
  return(SKIP_BODY);
 }
}
```


Attributes: The Tag Library Descriptor File

- The tag element must contain a nested attribute element
- The attribute element has three further nested elements
  - name, a required element that defines the case-sensitive attribute name. <name>colour</name>
  - required, a required element that stipulates whether the attribute must always be supplied (true) or is optional (false). E.g. <required>false</required>
- – rtexprvalue, an optional attribute that indicates whether the attribute value can be a JSP expression like <%= expression %> (true) or whether it must be a fixed string (false). The default value is false.


```
<tag>
  <name>welcomeTag</name>
  <tagclass>examples.tags.WelcomeTag</tagclass>
  <bodycontent>empty</bodycontent>
  <info>Outputs a welcome message.</info>
  <attribute>
    <name>colour</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>repeat</name>
    <required>false</required>
  </attribute>
 </tag>
```

Using welcomeTag Tag

```
<HTML>

<BODY>
<H1>Simple Tag Example</H1>

<%@ taglib uri="334-taglib.tld" prefix="334" %>

With no attributes: <334:welcomeTag />
With "colour": <334:welcomeTag colour="red" />
With both attributes: <334:welcomeTag colour="blue" repeat="3" />
</BODY>
</HTML>
```


Including the Tag Body

- Simplest tags
  - <prefix:tagName />
- Tags with attributes
  - <prefix:tagName att1="val1" ... />
- Now

```
<prefix:tagName>
JSP Content
</prefix:tagName>

<prefix:tagName att1="val1" ... >
JSP Content
</prefix:tagName>
```


Using Tag Body: The Tag Handler Class

- doStartTag
  - Usually returns EVAL_BODY_INCLUDE instead of SKIP_BODY
- doEndTag
  - Define this method if you want to take action after handling the body
  - Return EVAL_PAGE


```
package examples.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
public class WelcomeTagB extends TagSupport {
  private int repeat = 1;
  private String colour = "black";
  public void setColour(String colour) {
        this.colour = colour;
  }
  public void setRepeat(String repeat) {
        try {
          this.repeat = Integer.parseInt(repeat);
        } catch(NumberFormatException nfe) {
          this.repeat = 2;
        }
  }
```



```
public int doStartTag() {
  try {
    JspWriter out = pageContext.getOut();
        out.print("<font color=\""+colour+"\">");
        for (int i=0; i<repeat; i++)
          out.print("Welcome ");
  } catch(IOException ioe) {
    System.out.println("Error generating prime: " + ioe);
  }
  return(EVAL_BODY_INCLUDE);
}
```

```
public int doEndTag() {
        try {
          JspWriter out = pageContext.getOut();
          out.print("</font>");
        } catch(IOException ioe) {
          System.out.println("Error in HeadingTag: " + ioe);
        }
        return(EVAL_PAGE); // Continue with rest of JSP page
  }
}
```

```
<font color="red">Welcome
xxx
</font>
```


Using Tag Body: The Tag Library Descriptor File

- Only difference is bodycontent element
  - Should be JSP instead of empty:

```
<tag>
  <name>…</name>
  <tagclass>…</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>…</info>
</tag>
```



```
<tag>
  <name>welcomeTagB</name>
  <tagclass>examples.tags.WelcomeTagB</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>Outputs a welcome message.</info>
  <attribute>
    <name>colour</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>repeat</name>
    <required>false</required>
  </attribute>
</tag>
```

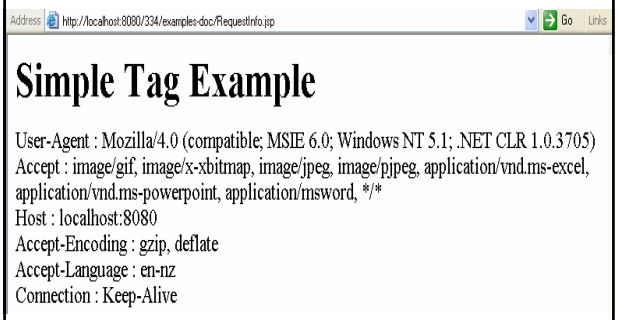

Using welcomeTagB Tag

```
<HTML>

<BODY>
<H1>Simple Tag Example</H1>

<%@ taglib uri="334-taglib.tld" prefix="334" %>

<334:welcomeTagB colour="red">
everybody
</334:welcomeTagB>
</BODY>
</HTML>
```


Obtaining request time information

- The request time information are stored in HttpServletRequest object.
- To obtain the HttpServletRequest object

```
HttpServletRequest request =
  (HttpServletRequest)pageContext.getRequest();
```




```
package examples.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import javax.servlet.http.*;
import javax.servlet.*;
import java.util.*;
public class RequestInfo extends TagSupport {
  public int doStartTag() {
    HttpServletRequest request =
                        (HttpServletRequest)pageContext.getRequest();
    Enumeration headers = request.getHeaderNames();
```



```
try {
        JspWriter out = pageContext.getOut();
        while (headers.hasMoreElements()) {
            String header = (String)headers.nextElement();
            out.print(header+" : " + request.getHeader(header)+"<BR>");
        }
    } catch(IOException ioe) {
       System.out.println("Error generating prime: " + ioe);
    }
    return(SKIP_BODY);
  }
}
```



```
<tag>
  <name>RequestInfo</name>
  <tagclass>examples.tags.RequestInfo</tagclass>
  <bodycontent>Empty</bodycontent>
  <info>Display request information.</info>
 </tag>
```



```
<HTML>
<BODY>
<H1>Simple Tag Example</H1>

<%@ taglib uri="334-taglib.tld" prefix="334" %>

<334:RequestInfo />
</BODY>
</HTML>
```


Manipulating the Tag Body

- No tag body
  - <prefix:tagName />
  - <prefix:tagName att1="val1" ... />
- Previous uses of tag body
  - <prefix:tagName>JSP Content</prefix:tagName>
  - <prefix:tagName att1="val1" ... />
    JSP Content
    </prefix:tagName>
  - Content inserted verbatim
- Now
  - Tag handler class can read/modify/replace tag body

Manipulating Tag Body: The Tag Handler Class

- Extend BodyTagSupport
- New method to override: doAfterBody
- getBodyContent() returns object of type BodyContent that has three key methods
  - getEnclosingWriter -- returns the JspWriter being used by doStartTag and doEndTag
  - getReader -- returns a Reader that can read tag's body
  - getString -- returns a String containing entire tag body

Address http://localhost:8080/334/examples-doc/TableTag.jsp

Table Tag Example

ID	Asg1	Asg2
123	40	50
456	20	60
789	10	90

```
<334:TableTag>
ID Asg1 Asg2
123 40 50
456 20 60
789 10 90
</334:TableTag>
```

```
<TABLE BORDER=1>
<TR>
  <TD>ID</TD>
  <TD>Asg1</TD>
  <TD>Asg2</TD>
 </TR>
 <TR>
  <TD>123</TD>
  <TD>40</TD>
  <TD>50</TD>
 </TR>
 <TR>
  <TD>456</TD>
  <TD>20</TD>
  <TD>60</TD>
 </TR>
 <TR>
  <TD>789</TD>
  <TD>10</TD>
  <TD>90</TD>
 </TR>
</TABLE>
```



```
package examples.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.util.*;
public class TableTag extends BodyTagSupport {
  public int doStartTag() {
    try {
      JspWriter out = pageContext.getOut();
            out.print("<TABLE BORDER=1>\n");
    } catch(IOException ioe) {
      System.out.println("Error generating prime: " + ioe);
    }
    return(EVAL_BODY_TAG);
  }
```



```
public int doEndTag() {
        try {
          JspWriter out = pageContext.getOut();
          out.print("</TABLE>");
        } catch(IOException ioe) {
          System.out.println("Error in HeadingTag: " + ioe);
        }
        return(EVAL_PAGE); // Continue with rest of JSP page
  }
```



```
public int doAfterBody() {
   BodyContent body = getBodyContent();
   StringTokenizer bodyText =
                new StringTokenizer(body.getString(), "\n");
   try {
    JspWriter out = body.getEnclosingWriter();
    while (bodyText.hasMoreTokens()) {
       StringTokenizer rowText =
                new StringTokenizer(bodyText.nextToken(), " ");
       out.print("  <TR>\n");
       while (rowText.hasMoreTokens()) {
          out.print("    <TD>");
          out.print(rowText.nextToken());
          out.print("</TD>\n");
       }
```



```
          out.print("  </TR>\n");
       }
     } catch(IOException ioe) {
       System.out.println("Error in FilterTag: " + ioe);
     }
     return(SKIP_BODY);
   }
  }
```

Manipulating Tag Body: The Tag Library Descriptor File

```
<tag>
  <name>TableTag</name>
  <tagclass>examples.tags.TableTag</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>Display information as a table.</info>
 </tag>
```


Using the TableTag Tag

```
<HTML>
<BODY>
<H1>Table Tag Example</H1>
<%@ taglib uri="334-taglib.tld" prefix="334" %>
<334:TableTag>
ID Asg1 Asg2
123 40 50
456 20 60
789 10 90
</334:TableTag>
</BODY>
</HTML>
```


Manipulating the body multiple times

- If you return EVAL_BODY_TAG from doAfterBody, the server will invoke doAfterBody again. That is, the tag body will be processed again.
- The processing of the tag body stops when you return SKIP_BODY from doAfterBody.
- JSP expressions are permitted as attribute values. They require no special treatment in the tag class handler.

Manipulating the body multiple times: The Tag Handler Class

```
package examples.tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
public class RepeatTag extends BodyTagSupport {
 private int reps;
 public void setReps(String repeats) {
  try {
   reps = Integer.parseInt(repeats);
  } catch(NumberFormatException nfe) {
   reps = 1;
  }
 }
```



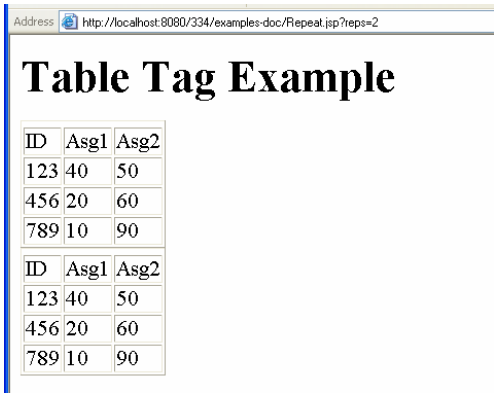
```
public int doAfterBody() {
  if (reps-- >= 1) {
    BodyContent body = getBodyContent();
    try {
      JspWriter out = body.getEnclosingWriter();
      out.println(body.getString());
      body.clearBody(); // Clear for next evaluation
    } catch(IOException ioe) {
      System.out.println("Error in RepeatTag: " + ioe);
    }
    return(EVAL_BODY_TAG);
  } else {
    return(SKIP_BODY);
  }
 }
}
```


Manipulating the body multiple times: The Tag Library Descriptor File

```
<tag>
  <name>repeat</name>
  <tagclass>examples.tags.RepeatTag</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>Repeats body the specified number of times.</info>
  <attribute>
    <name>reps</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
 </tag>
```

Using the repeat Tag

```
<HTML>
<BODY>
<H1>Table Tag Example</H1>
<%@ taglib uri="334-taglib.tld" prefix="334" %>
<334:repeat reps='<%= request.getParameter("reps") %>' >
<334:TableTag>
ID Asg1 Asg2
123 40 50
456 20 60
789 10 90
</334:TableTag>
</334:repeat>
</BODY>
</HTML>
```

Integrating Servlets and JSP

- Why Combine Servlets & JSP?
  - The assumption behind JSP is that a *single* page gives a *single* basic look
  - For complex processing, starting with JSP is awkward

Joint servlet/JSP process

- Original request is answered by a servlet
- Servlet processes request data, does database lookup, business logic, etc.
- Results are placed in beans
- Request is forwarded to a JSP page to format result
- Different JSP pages can be used to handle different types of presentation

Dispatching Requests from Servlets to JSP Pages

- First obtain the servlet context.
  - There is one context per "web application".
  - A "web application" is a collection of servlets and content installed under the server's URL namespace.
    - E.g. all the servlets, JSP file, etc. under the 334 directory.
  - getServletContext()

- Call the getRequestDispatcher method of ServletContext

```
String url = "/examples-doc/Integrate11.jsp";
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher(url);
```

- Call forward to completely transfer control to destination page (no communication with client)

```
dispatcher.forward(request, response);
```

Storing Data for Later Use: The Current Request

- Purpose
  - Storing data that servlet looked up and that JSP page will use only in this request.
- Servlet syntax to store data

```
SomeClass value = new SomeClass(…);
request.setAttribute("key", value);
```

- JSP syntax to retrieve data

```
<jsp:useBean
        id="key"
        class="somepackage.SomeClass"
        scope="request" />
```


Storing Data for Later Use: The Current Session

- Purpose
  - Storing data that servlet looked up and that JSP page will use in this request and in later requests from same client.
- Servlet syntax to store data

```
SomeClass value = new SomeClass(…);
HttpSession session =
request.getSession(true);
session.setAttribute("key", value);
```

- JSP syntax to retrieve data

```
<jsp:useBean
        id="key"
        class="somepackage.SomeClass"
        scope="session" />
```

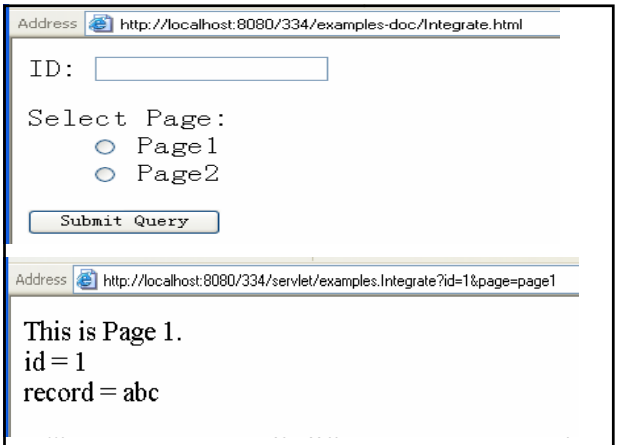

Storing Data for Later Use: The Application

- Purpose
  - Storing data that servlet looked up and that JSP page will use in this request and in later requests from *any* client.
- Servlet syntax to store data

```
SomeClass value = new SomeClass(…);
getServletContext().setAttribute("key",
value);
```

- JSP syntax to retrieve data

```
<jsp:useBean
        id="key"
        class="somepackage.SomeClass"
        scope="application" />
```


```
package examples;
public class Student {
  private String record;
  private int id;

  public Student(String record, int id) {
    this.record = record;
    this.id = id;
  }
  public String getRecord() {
    return record;
  }
  public int getId() {
    return id;
  }
}
```

```
package examples;
public class StudentBean {
   private String record;
   private int id;

   public String getRecord() {
     return record;
   }
   public int getId() {
      return id;
    }
    public void setRecord(String record) {
       this.record = record;
    }
    public void setId(int id) {
      this.id = id;
    }
}
```



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

public class Integrate extends HttpServlet {
  private Student[] student= new Student[2];
  public void init() throws ServletException {
    student[0] = new Student("abc", 1);
    student[1] = new Student("def", 2);
  }
```

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
        int id = Integer.parseInt(request.getParameter("id"));
        StudentBean sb = new StudentBean();
        sb.setRecord(student[id-1].getRecord());
        sb.setId(student[id-1].getId());
        HttpSession session = request.getSession(true);
        session.setAttribute("student", sb);
        String url;
        if (request.getParameter("page").equals("page1"))
                url = "/examples-doc/Integrate1.jsp";
        else   url = "/examples-doc/Integrate2.jsp";
        RequestDispatcher dispatcher =
                getServletContext().getRequestDispatcher(url);
        dispatcher.forward(request, response);
  }
}
```



```
<HTML>
This is Page 1. <BR>
<BODY>
<jsp:useBean  id="student"
              class="examples.StudentBean"
              scope="session" />
id = <jsp:getProperty  name="student"
                       property="id" />
<BR>
record = <jsp:getProperty      name="student"
                               property="record" />

</BODY>
</HTML>
```