JavaServer Pages

JSP

Agenda

• Introducing JavaServer Pages™ (JSP™)
• JSP scripting elements
• The JSP page Directive: Structuring Generated Servlets™
• Including Files in JSP Documents
• Using JavaBeans™ components with JSP
• Creating custom JSP tag libraries
• Integrating servlets and JSP with the MVC architecture
The Need for JSP

• With servlets, it is easy to
  – Read form data
  – Read HTTP request headers
  – Set HTTP status codes and response headers
  – Use cookies and session tracking
  – Share data among servlets
  – Remember data between requests
  – Get fun, high-paying jobs

• But, it sure is a pain to
  – Use those println statements to generate HTML
  – Maintain that HTML

The JSP Framework

• Idea:
  – Use regular HTML for most of page
  – Mark servlet code with special tags
  – Entire JSP page gets translated into a servlet (once), and
    servlet is what actually gets invoked (for each request)

• Example:
  – JSP
    • Thanks for ordering
      <I><%= request.getParameter("title") %></I>
  – URL
    • http://host/OrderConfirmation.jsp?title=Core+Web+Programming
  – Result
    • Thanks for ordering Core Web Programming
Benefits of JSP

- Although JSP technically can't do anything servlets can't do, JSP makes it easier to:
  - Write HTML
  - Read and maintain the HTML
- JSP makes it possible to:
  - Use standard HTML tools such as HomeSite or UltraDev
  - Have different members of your team do the HTML layout and the programming
- JSP encourages you to
  - Separate the (Java) code that creates the content from the (HTML) code that presents it

Advantages of JSP Over Competing Technologies

- Versus ASP or ColdFusion
  - Better language for dynamic part
  - Portable to multiple servers and operating systems
- Versus PHP
  - Better language for dynamic part
  - Better tool support
- Versus WebMacro or Velocity
  - Standard
- Versus pure servlets
  - More convenient to create HTML
  - Can use standard tools (e.g., HomeSite)
  - Divide and conquer
  - JSP developers still need to know servlet programming
Setting Up Your Environment

- Set your CLASSPATH. **Not.**
- Compile your code. **Not.**
- Use packages to avoid name conflicts. **Not.**
- Put JSP page in special directory. **Not.**
  - tomcat_install_dir/webapps/ROOT
  - jrun_install_dir/servers/default/default-app
- Use special URL to invoke JSP page. **Not.**
- Caveats
  - Previous rules about CLASSPATH, install dirs, etc., still apply to regular classes used by JSP

Example

```html
<!DOCTYPE HTML PUBLIC
  "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
  <head>
    <title>JSP Expressions</title>
    <meta name="author" content="Marty Hall">
    <meta name="keywords" content="JSP,expressions,JavaServer,Pages,servlets">
    <meta name="description" content="A quick example of JSP expressions.">
    <link rel="stylesheet" href="JSP-Styles.css" type="text/css">
  </head>
</html>
```
Example (Continued)

```html
<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>
```

Example Result

- **With default setup, if location was**
  - C:\jakarta-tomcat\webapps\ROOT\Expressions.jsp
  - C:\Program Files\Allaire\JRun\servers\default\default-app\Expressions.jsp
- **URL would be**
  - http://localhost/Expressions.jsp
**Most Common Misunderstanding: Forgetting JSP is Server-Side Technology**

- **Very common question**
  - I can’t do such and such with HTML. Will JSP let me do it?
- **Why doesn’t this question make sense?**
  - JSP runs entirely on server
  - It doesn’t change content the browser can handle
- **Similar questions**
  - How do I put an applet in a JSP page?
    Answer: send an `<APPLET...>` tag to the client
  - How do I put an image in a JSP page?
    Answer: send an `<IMG ...>` tag to the client
  - How do I use JavaScript/Acrobat/Shockwave/Etc?
    Answer: send the appropriate HTML tags

**2nd Most Common Misunderstanding: Translation/Request Time Confusion**

- **What happens at page translation time?**
  - JSP constructs get translated into servlet code
- **What happens at request time?**
  - Servlet code gets executed. No interpretation of JSP occurs at request time. The original JSP page is ignored at request time; only the servlet that resulted from it is used
- **When does page translation occur?**
  - Typically, the first time JSP page is accessed after it is modified. This should never happen to real user (developers should test all JSP pages they install).
  - Page translation does not occur for each request
JSP/Servlets in the Real World

• ofoto.com: print and manage digital and conventional photos.

JSP/Servlets in the Real World

• First USA Bank: largest credit card issuer in the world; most on-line banking customers
JSP/Servlets in the Real World

• Delta Airlines: entire Web site, including real-time schedule info

• wine.com: the Internet's leading wine retailer
JSP/Servlets in the Real World

- American Century Investments: more than 70 mutual funds, $90 billion under management, two million investors

- Excite: one of the top five Internet portals; one of the ten busiest sites on the Web
  - AltaVista also uses servlets/JSP on part of their site
JSP Scripting Elements

Uses of JSP Constructs

- Scripting elements calling servlet code directly
- Scripting elements calling servlet code indirectly (by means of utility classes)
- Beans
- Custom tags
- Servlet/JSP combo (MVC architecture)
Types of Scripting Elements

• **Expressions**
  – Format: `<%= expression %>`
  – Evaluated and inserted into the servlet’s output. I.e., results in something like `out.println(expression)`

• **Scriptlets**
  – Format: `<% code %>`
  – Inserted verbatim into the servlet’s `_jspService` method (called by service)

• **Declarations**
  – Format: `<%! code %>`
  – Inserted verbatim into the body of the servlet class, outside of any existing methods

JSP Expressions

• **Format**
  – `<%= Java Expression %>`

• **Result**
  – Expression evaluated, converted to String, and placed into HTML page at the place it occurred in JSP page
  – That is, expression placed in `_jspService` inside `out.print`

• **Examples**
  – Current time: `<%= new java.util.Date() %>`
  – Your hostname: `<%= request.getRemoteHost() %>`

• **XML-compatible syntax**
  – `<jsp:expression>Java Expression</jsp:expression>`
  – XML version not supported by Tomcat 3. Until JSP 1.2, servers are not required to support it.
JSP/Servlet Correspondence

• Original JSP

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

• Possible resulting servlet code

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

Example Using JSP Expressions

```html
<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>
```
Predefined Variables

• **request**
  – The HttpServletRequest (1st arg to doGet)

• **response**
  – The HttpServletResponse (2nd arg to doGet)

• **session**
  – The HttpSession associated with the request (unless disabled with the session attribute of the page directive)

• **out**
  – The stream (of type JspWriter) used to send output to the client

• **application**
  – The ServletContext (for sharing data) as obtained via getServletConfig().getContext().

JSP Scriptlets

• **Format**
  – `<% Java Code %>`

• **Result**
  – Code is inserted verbatim into servlet's _jspService

• **Example**
  – `<%
        String queryData = request.getQueryString();
        out.println("Attached GET data: "+queryData);
    %>`

  – `<% response.setContentType("text/plain"); %>`

• **XML-compatible syntax**
  – `<jsp:scriptlet>Java Code</jsp:scriptlet>`
JSP/Servlet Correspondence

- **Original JSP**
  ```jsp
  <%= foo() %>
  <% bar(); %>
  ```

- **Possible resulting servlet code**
  ```java
  public void _jspService(HttpServletRequest request,
                           HttpServletResponse response)
      throws ServletException, IOException {
    request.setContentType("text/html");
    HttpSession session = request.getSession(true);
    JspWriter out = response.getWriter();
    out.println(foo());
    bar();
    ...}
  ```

Example Using JSP Scriptlets

```html
<!DOCTYPE HTML PUBLIC
  "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
  <head>
    <title>Color Testing</title>
  </head>
  <body bgcolor="<%= bgColor %>">
    String bgColor = request.getParameter("bgColor");
    boolean hasExplicitColor;
    if (bgColor != null) {
      hasExplicitColor = true;
    } else {
      hasExplicitColor = false;
      bgColor = "WHITE";
    }
    ...}
  </body>
</html>
```
JSP Scriptlets: Results

- Format
  - `<%! Java Code %>`

- Result
  - Code is inserted verbatim into servlet's class definition, outside of any existing methods

- Examples
  - `<%! private int someField = 5; %>`
  - `<%! private void someMethod(...) {...} %>`

- XML-compatible syntax
  - `<jsp:declaration>Java Code</jsp:declaration>`
JSP/Servlet Correspondence

• Original JSP

```jsp
<H1>Some Heading</H1>
<%!
  private String randomHeading() {
    return("<H2>" + Math.random() + "</H2>" acompanhado);
  }
%
<%= randomHeading() %> 
```

JSP/Servlet Correspondence

• Possible resulting servlet code

```java
public class xxxx implements HttpJspPage {
  private String randomHeading() {
    return("<H2>" + Math.random() + "</H2>" acompanhado);
  }

  public void _jspService(HttpServletRequest request,
      HttpServletResponse response) 
      throws ServletException, IOException {
    request.setContentType("text/html");
    HttpSession session = request.getSession(true);
    JspWriter out = response.getWriter();
    out.println("<H1>Some Heading</H1>" acompanhado);
    out.println(randomHeading());
    ...
  }
```

```
Example Using JSP Declarations

```html
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.0 Transitional//EN">
<html><head><title>JSP Declarations</title>
<link rel=stylesheet
href="JSP-styles.css"
type="text/css">
</head>
<body>
<h1>JSP Declarations</h1>
<p>%! private int accessCount = 0; %>
<h2>Accesses to page since server reboot:
%s = ++accessCount %></h2>
</body>
</html>
```

JSP Declarations: Result

- After 15 total visits by an arbitrary number of different clients
The JSP page Directive: Structuring Generated Servlets

Purpose of the page Directive

- Give high-level information about the servlet that will result from the JSP page
- Can control
  - Which classes are imported
  - What class the servlet extends
  - What MIME type is generated
  - How multithreading is handled
  - If the servlet participates in sessions
  - The size and behavior of the output buffer
  - What page handles unexpected errors
The import Attribute

• Format
  – `<%@ page import="package.class" %>`
  – `<%@ page import="package.class1,...,package.classN" %>`

• Purpose
  – Generate import statements at top of servlet

• Notes
  – Although JSP pages can be almost anywhere on server, classes used by JSP pages must be in normal servlet dirs
  – For Tomcat, this is install_dir/webapps/ROOT/WEB-INF/classes or ...
  – For JRun, this is install_dir/servers/default/default-app/WEB-INF/classes or ...

Example of import Attribute

```jsp
...<BODY>
<H2>The import Attribute</H2>
<%-- JSP page directive --%>
<%@ page import="java.util.*,cwp.*" %>

<%-- JSP Declaration --%>
<%!
private String randomID() {
  int num = (int)(Math.random()*1000000.0);
  return("id" + num);
}

private final String NO_VALUE = 
"<I>No Value</I>";
%</BODY>
```
Example of import Attribute (Continued)

```jsp
<% 
Cookie[] cookies = request.getCookies();
String oldID = 
    ServletUtilities.getCookieValue(cookies, "userID",
        NO_VALUE);
String newID;
if (oldID.equals(NO_VALUE)) {
    newID = randomID();
} else {
    newID = oldID;
}
LongLivedCookie cookie =
    new LongLivedCookie("userID", newID);
response.addCookie(cookie);
%>

<%-- JSP Expressions --%>
This page was accessed at <%= new Date() %> with a userID
cookie of <%= oldID %>.
</BODY></HTML>
```

Example of import Attribute: Result

- First access

![First access](image1.png)

- Subsequent accesses

![Subsequent access](image2.png)
The contentType Attribute

• Format
  – `<%@ page contentType="MIME-Type" %>`
  – `<%@ page contentType="MIME-Type; charset=Character-Set" %>`

• Purpose
  – Specify the MIME type of the page generated by the servlet that results from the JSP page

Generating Excel Spreadsheets

<table>
<thead>
<tr>
<th>First</th>
<th>Last</th>
<th>Email Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marty</td>
<td>Hall</td>
<td><a href="mailto:hall@corewebprogramming.com">hall@corewebprogramming.com</a></td>
</tr>
<tr>
<td>Larry</td>
<td>Brown</td>
<td><a href="mailto:brown@corewebprogramming.com">brown@corewebprogramming.com</a></td>
</tr>
<tr>
<td>Bill</td>
<td>Gates</td>
<td><a href="mailto:gates@sun.com">gates@sun.com</a></td>
</tr>
<tr>
<td>Larry</td>
<td>Ellison</td>
<td><a href="mailto:ellison@microsoft.com">ellison@microsoft.com</a></td>
</tr>
</tbody>
</table>

`<%@ page contentType="application/vnd.ms-excel" %>`
`<%-- There are tabs, not spaces, between columns. --%>"
The isThreadSafe Attribute

• Format
  – `<%@ page isThreadSafe="true" %>` `<%-- Default --%>`
  – `<%@ page isThreadSafe="false" %>`

• Purpose
  – To tell the system when your code is not threadsafe, so that the system can prevent concurrent access
    • Instructs servlet to implement SingleThreadModel

• Notes
  – Default is true -- system assumes you have synchronized updates to fields & other shared data
  – Supplying a value of false can degrade performance

Example of Non-Threadsafe Code (IDs Must Be Unique)

• What's wrong with this code?

```java
<%! private int idNum = 0; %>
<% String userID = "userID" + idNum;
out.println("Your ID is " + userID + ".");
idNum = idNum + 1;
%>
```
Is isThreadSafe Needed Here?

• No
  `<%! private int idNum = 0; %>`
  `<% synchronized(this) {
      String userID = "userID" + idNum;
      out.println("Your ID is " + userID + ".");
      idNum = idNum + 1;
  } %>`

• Totally safe, better performance in high-traffic environments

Other Attributes of the page Directive

• session
  – Lets you choose not to participate in sessions

• buffer
  – Changes min size of buffer used by JspWriter

• autoflush
  – Requires developer to explicitly flush buffer

• extends
  – Changes parent class of generated servlet

• errorPage
  – Designates a page to handle unplanned errors

• isErrorPage
  – Stipulates that page can be used as error page
Including Files in JSP Documents

Including Pages at Request Time

**Format**
- `<jsp:include page="Relative URL" flush="true" />`

**Purpose**
- To reuse JSP, HTML, or plain text content
- JSP content cannot affect main page: only output of included JSP page is used
- To permit updates to the included content without changing the main JSP page(s)
Including Pages: Example Code

```html
...<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
<TR><TH CLASS="TITLE">
  What's New at JspNews.com</TH></TABLE>
<P>
Here is a summary of our three most recent news stories:
<OL>
  <LI><jsp:include page="news/Item1.html" flush="true" /></LI>
  <LI><jsp:include page="news/Item2.html" flush="true" /></LI>
  <LI><jsp:include page="news/Item3.html" flush="true" /></LI>
</OL>
</BODY>
</HTML>
```

Including Pages: Example Result

![Example Result Image](image-url)
Including Files at Page Translation Time

- **Format**
  - `<%@ include file="Relative URL" %>`

- **Purpose**
  - To reuse JSP content in multiple pages, where JSP content affects main page

- **Notes**
  - Servers are not required to detect changes to the included file, and in practice many don't
  - Thus, you need to change the JSP files whenever the included file changes
  - You can use OS-specific mechanisms such as the Unix "touch" command, or
    - `<%-- Navbar.jsp modified 3/1/02 -->%>
      `<%@ include file="Navbar.jsp" %>`

Reusable JSP Content: ContactSection.jsp

```jsp
<%@ page import="java.util.Date" %>
<%-- The following become fields in each servlet that results from a JSP page that includes this file. --%>
<%!
private int accessCount = 0;
private Date accessDate = new Date();
private String accessHost = "<I>No previous access</I>";
%>
<P>
<HR>
This page © 2000
<A HREF="http://www.my-company.com/">my-company.com</A>. This page has been accessed <%= ++accessCount %> times since server reboot. It was last accessed from <%= accessHost %> at <%= accessDate %>.

<% accessHost = request.getRemoteHost(); %>
<% accessDate = new Date(); %>
```
Using the JSP Content

...<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
   <TR><TH CLASS="TITLE">
         Some Random Page</TH>
   </TR></TABLE>
<P>
Information about our products and services.
<P>
Blah, blah, blah.
<P>
Yadda, yadda, yadda.

<%@ include file="ContactSection.jsp" %>

</BODY>
</HTML>

Using the JSP Content: Result
Uses of JSP Constructs

- Scripting elements calling servlet code directly
- Scripting elements calling servlet code indirectly (by means of utility classes)
- Beans
- Custom tags
- Servlet/JSP combo (MVC architecture)
Background: What Are Beans?

• Classes that follow certain conventions
  – Must have a zero-argument (empty) constructor
  – Should have no public instance variables (fields)
  – Persistent values should be accessed through methods called `getXxx` and `setXxx`
    • If class has method `getTitle` that returns a String, class is said to have a String property named `title`
    • Boolean properties use `isXxx` instead of `getXxx`

• For more on beans, see http://java.sun.com/beans/docs/

Basic Bean Use in JSP

• Format
  – `<jsp:useBean id="name" class="package.Class" />`

• Purpose
  – Allow instantiation of classes without explicit Java syntax

• Notes
  – Simple interpretation: JSP action
    `<jsp:useBean id="book1" class="cwp.Book" />`
    can be thought of as equivalent to the scriptlet
    `<% cwp.Book book1 = new cwp.Book(); %>`
  – But `useBean` has two additional features
    • Simplifies setting fields based on incoming request params
    • Makes it easier to share beans
**Accessing Bean Properties**

- **Format**
  - `<jsp:getProperty name="name" property="property" />`

- **Purpose**
  - Allow access to bean properties (i.e., calls to getXxx methods) without explicit Java programming language code

- **Notes**
  - `<jsp:getProperty name="book1" property="title" />`
  is equivalent to the following JSP expression
  `<%= book1.getTitle() %>`

**Setting Bean Properties: Simple Case**

- **Format**
  - `<jsp:setProperty name="name" property="property" value="value" />`

- **Purpose**
  - Allow setting of bean properties (i.e., calls to setXxx methods) without explicit Java code

- **Notes**
  - `<jsp:setProperty name="book1" property="title" value="Core Servlets and JSP" />`
  is equivalent to the following scriptlet
  `<% book1.setTitle("Core Servlets and JSP"); %>`
Example: StringBean

```java
package cwp;

public class StringBean {
    private String message = "No message specified";

    public String getMessage() {
        return(message);
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

- Installed in normal servlet directory

JSP Page That Uses StringBean

```html
<jsp:useBean id="stringBean" class="cwp.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="Best string bean: Fortex" />
        Value after setting property with setProperty:
        <I><jsp:getProperty name="stringBean" property="message" /></I>
    <LI><% stringBean.setMessage("My favorite: Kentucky Wonder"); %>
        Value after setting property with scriptlet:
        <I><%= stringBean.getMessage() %></I>
</OL>
```
JSP Page That Uses StringBean

**Using JavaBeans with JSP**

1. Initial value (getProperty): No message specified
2. Initial value (JSP expression): No message specified
3. Value after setting property with setProperty: Best string bean: Portax
4. Value after setting property with scriptlet: My favorite: Kentucky Wonder

Associating Bean Properties with Request (Form) Parameters

- If property is a String, you can do
  - `<jsp:setProperty ... value='<%= request.getParameter("...") %>'` /

- Scripting expressions let you convert types, but you have to use Java syntax

- The `param` attribute indicates that:
  - Value should come from specified request param
  - Simple automatic type conversion performed

- Using "*" for the property attribute indicates that:
  - Value should come from request parameter whose name matches property name
  - Simple type conversion should be performed
Setting Bean Properties Case 1: Explicit Conversion & Assignment

```html
<!DOCTYPE ...>
...
<jsp:useBean id="entry"
class="cwp.SaleEntry" />

<%-- getItemID expects a String --%
<jsp:setProperty
   name="entry"
   property="itemID"
   value='<%= request.getParameter("itemID") %>' />}

<%-- getNumItems expects an int --%
<jsp:setProperty
   name="entry"
   property="numItems"
   value='<%= numItemsOrdered %>' />}
```
Setting Bean Properties Case 1: Explicit Conversion & Assignment

```jsp
<% 
double discountCode = 1.0;
try {
   String discountString =
      request.getParameter("discountCode");
discountCode =
   Double.valueOf(discountString).doubleValue();
} catch(NumberFormatException nfe) {} 
%>
<span style="color:red">getDiscountCode expects a double --</span>

<%-- getDiscountCode expects a double --%>
<jsp:setProperty
   name="entry"
   property="discountCode"
   value="<%= discountCode %>">
</jsp:setProperty>
```

Case 2: Associating Individual Properties with Input Parameters

```jsp
<jsp:useBean id="entry"
            class="cwp.SaleEntry" />
<jsp:setProperty
   name="entry"
   property="itemID"
   param="itemID" />
<jsp:setProperty
   name="entry"
   property="numItems"
   param="numItems" />
<jsp:setProperty
   name="entry"
   property="discountCode"
   param="discountCode" />
```
Case 3: Associating All Properties with Input Parameters

```jsp
<jsp:useBean id="entry" class="cwp.SaleEntry" />
<jsp:setProperty name="entry" property="*" />
```

Sharing Beans

- You can use scope attribute to specify where bean is stored
  - `<jsp:useBean id="..." class="..." scope="..." />`
  - Bean still also bound to local variable in _jspService
- Lets multiple servlets or JSP pages share data
- Also permits conditional bean creation
  - Create new object only if you can't find existing one
Values of the scope Attribute

• **page**
  - Default value. Bean object should be placed in the PageContext object for the duration of the current request. Lets methods in same servlet access bean

• **application**
  - Bean will be stored in ServletContext (available through the application variable or by call to getServletContext()). ServletContext is shared by all servlets in the same Web application (or all servlets on server if no explicit Web applications are defined).

• **session**
  - Bean will be stored in the HttpSession object associated with the current request, where it can be accessed from regular servlet code with getAttribute and setAttribute, as with normal session objects.

• **request**
  - Bean object should be placed in the ServletRequest object for the duration of the current request, where it is available by means of getAttribute
Conditional Bean Operations

• **Bean conditionally created**
  - `jsp:useBean` results in new bean object only if no bean with same id and scope can be found
  - If a bean with same id and scope is found, the preexisting bean is simply bound to variable referenced by id

• **Bean properties conditionally set**
  - `<jsp:useBean ... />` replaced by `<jsp:useBean ...>statements</jsp:useBean>`
  - The statements (`jsp:setProperty` elements) are executed only if a new bean is created, not if an existing bean is found

---

Conditional Bean Creation: AccessCountBean

```java
public class AccessCountBean {
    private String firstPage;
    private int accessCount = 1;

    public String getFirstPage() {
        return(firstPage);
    }

    public void setFirstPage(String firstPage) {
        this.firstPage = firstPage;
    }

    public int getAccessCount() {
        return(accessCount);
    }

    public void setAccessCountIncrement(int increment) {
        accessCount = accessCount + increment;
    }
}
```
Conditional Bean Creation: SharedCounts1.jsp

```
<jsp:useBean id="counter"
    class="coreservlets.AccessCountBean"
    scope="application">
    <jsp:setProperty name="counter"
        property="firstPage"
        value="SharedCounts1.jsp" />
</jsp:useBean>

Of SharedCounts1.jsp (this page),
<A HREF="SharedCounts2.jsp">SharedCounts2.jsp</A>, and
<A HREF="SharedCounts3.jsp">SharedCounts3.jsp</A>,
<jsp:getProperty name="counter" property="firstPage" />
was the first page accessed.

Collectively, the three pages have been accessed
<jsp:getProperty name="counter" property="accessCount" />
times.
<jsp:setProperty name="counter"
    property="accessCountIncrement"
    value="1" />
```

Accessing SharedCounts1, SharedCounts2, SharedCounts3

- SharedCounts2.jsp was accessed first.
- Pages have been accessed twelve previous times by an arbitrary number of clients
Creating Custom JSP Tag Libraries

Uses of JSP Constructs

- Scripting elements calling servlet code directly
- Scripting elements calling servlet code indirectly (by means of utility classes)
- Beans
- Custom tags
- Servlet/JSP combo (MVC architecture)
Components That Make Up a Tag Library

- **The Tag Handler Class**
  - Must implement javax.servlet.jsp.tagext.Tag
  - Usually extends TagSupport or BodyTagSupport
  - Goes in same directories as servlet class files and beans

- **The Tag Library Descriptor File**
  - XML file describing tag name, attributes, and implementing tag handler class
  - Goes with JSP file or at arbitrary URL

- **The JSP File**
  - Imports a tag library (referencing descriptor file)
  - Defines tag prefix
  - Uses tags

Defining a Simple Tag Handler Class

- **Extend the TagSupport class**
- **Import needed packages**
  - import javax.servlet.jsp.*;
  - import javax.servlet.jsp.tagext.*;
  - import java.io.*;
- **Override doStartTag**
  - Obtain the JspWriter by means of pageContext.getOut()
  - Use the JspWriter to generate JSP content
  - Return SKIP_BODY
  - Translated into servlet code at page-translation time
  - Code gets called at request time
Defining a Simple Tag Handler Class: Example

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

public class SimplePrimeTag extends TagSupport {
    protected int len = 50;

    public int doStartTag() {
        try {
            JspWriter out = pageContext.getOut();
            BigInteger prime = Primes.nextPrime(Primes.random(len));
            out.print(prime);
        } 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 containing:
  - `name`, whose body defines the base tag name. In this case, I use `<name>simplePrime</name>`
  - `tagclass`, which gives the fully qualified class name of the tag handler. In this case, I use `<tagclass>cwp.tags.SimplePrimeTag</tagclass>`
  - `bodycontent`, which gives hints to development environments. Optional.
  - `info`, which gives a short description. Here, I use `<info>Outputs a random 50-digit prime.</info>`
TLD File for SimplePrimeTag

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib ...>
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>cwp</shortname>
  <info>
  </info>
  <tag>
    <name>simplePrime</name>
    <tagclass>cwp.tags.SimplePrimeTag</tagclass>
    <info>Outputs a random 50-digit prime.</info>
  </tag>
</taglib>
```

Accessing Custom Tags From JSP Files

- **Import the tag library**
  - Specify location of TLD file
    `<%@ taglib uri= "cwp-taglib.tld" prefix= "cwp" %>`
  - Define a tag prefix (namespace)
    `<%@ taglib uri="cwp-taglib.tld" prefix= "cwp" %>`

- **Use the tags**
  - `<prefix:tagName />`
    - Tag name comes from TLD file
    - Prefix comes from taglib directive
  - E.g., `<cwp:simplePrime />`
Using simplePrime Tag

...<H1>Some 50-Digit Primes</H1>
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
<UL>
  <LI><cwp:simplePrime />  
  <LI><cwp:simplePrime />  
  <LI><cwp:simplePrime />  
  <LI><cwp:simplePrime />  
</UL>

Using simplePrime Tag: Result

Some 50-Digit Primes

- 85525203807603181442992536487526237946236109329293
- 548665431365536389373398618035946632307063332731
- 3441941875825795902314053586112120565127145728483
- 145364362188326985683115495683116771875392186140487
Intermediate and Advanced Custom Tags

- Tags with attributes
- Tags that include their body content
- Tags that optionally include their body
- Tags that manipulate their body
- Tags that manipulating their body multiple times (looping tags)
- Nested tags
- See book for details (related chapter online in PDF at Java Developer’s Connection)

Tags with Attributes: prime Tag

...  
<H1>Some N-Digit Primes</H1>
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
<UL>
  <LI>20-digit: <cwp:prime length="20" /></LI>
  <LI>40-digit: <cwp:prime length="40" /></LI>
  <LI>80-digit: <cwp:prime length="80" /></LI>
  <LI>Default (50-digit): <cwp:prime /></LI>
</UL>
Using prime Tag: Result

Some N-Digit Primes

- 20-digit: 3220289383223491679
- 40-digit: 6657206530491561727357099946005524853317
- 80-digit: 4155706927053072733390879462039023652235424155755874460486332222481331561431357
- Default (30-digit): 67648745567655715012021346271431310348120658584901

Including Body Content: heading Tag

```jsp
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
<cwp:heading bgColor="#C0C0C0">
Default Heading
</cwp:heading>
<P>
<cwp:heading bgColor="BLACK" color="WHITE">
White on Black Heading
</cwp:heading>
<P>
<cwp:heading bgColor="#EF8429" fontSize="60" border="5">
Large Bordered Heading
</cwp:heading>
```

Using heading Tag: Result

Default Heading
White on Black Heading
Large Bordered Heading
Heading with Full-Width Background
Heading with Non-Standard Font

Optionally Including Tag Body: debug Tag

```jsp
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
Top of regular page. Blah, blah, blah.
Yadda, yadda, yadda.
</p>
<cwp:debug>
<B>Debug:</B>
<ul>
<li>Current time: <%= new java.util.Date() %>
<li>Requesting hostname: <%= request.getRemoteHost() %>
<li>Session ID: <%= session.getId() %>
</ul>
</cwp:debug>
</p>
Bottom of regular page. Blah, blah, blah.
Yadda, yadda, yadda.
```
Using debug Tag: Results

Using the Debug Tag

Manipulating Tag Body: the filter Tag

```jsp
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
<TABLE BORDER=1 ALIGN="CENTER">
<TR CLASS="COLORED"><TH>Example</TH><TH>Result</TH></TR>
<TD>
<pre>
<cwp:filter>
<EM>Some emphasized text.</EM><BR>
<STRONG>Some strongly emphasized text.</STRONG><BR>
<CODE>Some code.</CODE><BR>
...
</cwp:filter></pre>
</TD>
</TABLE>
```
Using the filter Tag: Results

HTML Logical Character Styles

Physical character styles (i, b, etc.) are rendered consistently in different browsers. Logical character styles, however, may be rendered differently by different browsers. Here's how your browser (Mozilla4.7 [en] (Win98, U)) renders the HTML 4.0 logical character styles:

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;EM&gt;Some emphasized text.&lt;/EM&gt;</td>
<td>Some emphasized text.</td>
</tr>
<tr>
<td>&lt;STRONG&gt;Some strongly emphasized text.&lt;/STRONG&gt;&lt;EM&gt;</td>
<td>Some strongly emphasized text.</td>
</tr>
<tr>
<td>&lt;CODE&gt;Some code.&lt;/CODE&gt;&lt;EM&gt;</td>
<td>Some code.</td>
</tr>
<tr>
<td>&lt;VAR&gt;Some sample text.&lt;/VAR&gt;&lt;EM&gt;</td>
<td>Some sample text.</td>
</tr>
<tr>
<td>&lt;KBD&gt;Some keyboard text.&lt;/KBD&gt;&lt;EM&gt;</td>
<td>Some keyboard text.</td>
</tr>
<tr>
<td>&lt;DFN&gt;A term being defined.&lt;/DFN&gt;&lt;EM&gt;</td>
<td>A term being defined.</td>
</tr>
<tr>
<td>&lt;VAR&gt;A variable.&lt;/VAR&gt;&lt;EM&gt;</td>
<td>A variable.</td>
</tr>
<tr>
<td>&lt;CTT&gt;A citation or reference.&lt;/CTT&gt;</td>
<td>A citation or reference.</td>
</tr>
</tbody>
</table>

Manipulating the Body Multiple Times: the repeat Tag

```jsp
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
<OL>
  <!-- Repeats N times. A null reps value means repeat once. -->
  <cwp:repeat reps='<%= request.getParameter("repeats") %>'>
    <LI><cwp:prime length="40" /></LI>
  </cwp:repeat>
</OL>
```
Using the repeat Tag: Results

Some 40-Digit Primes

Each entry in the following list is the first prime number higher than a randomly selected 40-digit number.

1. 77804572848469290676469059065923259
2. 56549112209695336845971755223274136573
3. 4440230773563930128088671825331783776667
4. 45575110422372539686257258572888541037
5. 50447036124193539525271178623224648577
6. 32798227568416134105141665298217259631
7. 96659852631382074742458325591359401

Nested Tags: the if Tag

```jsp
<%@ taglib uri="cwp-taglib.tld" prefix="cwp" %>
<cwp:if>
  <cwp:condition>true</cwp:condition>
  <cwp:then>Condition is true</cwp:then>
  <cwp:else>Condition is false</cwp:else>
</cwp:if>

Some coin tosses:
<cwp:repeat reps="10">
  <cwp:if>
    <cwp:condition>
      <%= Math.random() < 0.5 %>
    </cwp:condition>
    <cwp:then><B>Heads</B></cwp:then>
    <cwp:else><B>Tails</B></cwp:else>
  </cwp:if>
</cwp:repeat>
```
Using the if Tag: Results

If Tag Example
Condition is true
Request is not using SSL
Some coin tosses:
  Heads
  Tails
  Heads
  Tails
  Heads
  Tails
  Heads
  Heads

Open Source Tag Libraries
http://jakarta.apache.org/taglibs/

- Internationalization (I18N)
- Database access
- Sending email
- JNDITM
- Date/time
- Populating/validating form fields
- Perl regular expressions
- Extracting data from other Web pages
- XSL transformations
- Etc
Integrating Servlets and JSP
The MVC Architecture

Uses of JSP Constructs

- Scripting elements calling servlet code directly
- Scripting elements calling servlet code indirectly (by means of utility classes)
- Beans
- Custom tags
- Servlet/JSP combo (MVC architecture)
Why Combine Servlets & JSP?

- **Typical picture:** use JSP to make it easier to develop and maintain the HTML content
  - For simple dynamic code, call servlet code from scripting expressions
  - For moderately complex cases, use custom classes called from scripting expressions
  - For more complicated cases, use beans and custom tags

- **But, that's not enough**
  - For complex processing, JSP is awkward
  - Despite the convenience of separate classes, beans, and custom tags, the assumption behind JSP is that a single page gives a single basic look

Architecture

- **Approach**
  - Original request is answered by a servlet
  - Servlet processes request data, does database lookup, accesses 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

- **Terminology**
  - Often called the “Model View Controller” architecture or “Model 2” approach to JSP
  - Formalized further with Apache “Struts” framework
    - See http://jakarta.apache.org/struts/
Dispatching Requests

• First, call the getRequestDispatcher method of ServletContext
  – Supply a URL relative to the Web application root
  – Example
    • String url = "/presentations/presentation1.jsp";
      RequestDispatcher dispatcher =
      getServletContext().getRequestDispatcher(url);

• Second
  – Call forward to completely transfer control
to destination page. See following example
  – Call include to insert output of destination page
  and then continue on. See book.

Forwarding Requests: Example Code

```java
public void doGet(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
    String operation = request.getParameter("operation");
    if (operation == null) {
        operation = "unknown";
    }
    if (operation.equals("operation1")) {
        gotoPage("/operations/presentation1.jsp",
                 request, response);
    } else if (operation.equals("operation2")) {
        gotoPage("/operations/presentation2.jsp",
                 request, response);
    } else {
        gotoPage("/operations/unknownRequestHandler.jsp",
                 request, response);
    }
}
```
Forwarding Requests: Example Code (Continued)

```java
private void gotoPage(String address,
                       HttpServletRequest request,
                       HttpServletResponse response)
    throws ServletException, IOException {
    RequestDispatcher dispatcher =
        getServletContext().getRequestDispatcher(address);
    dispatcher.forward(request, response);
}
```

Reminder: JSP useBean Scope Alternatives

- **request**
  - `<jsp:useBean id="..." class="..." scope="request" />

- **session**
  - `<jsp:useBean id="..." class="..." scope="session" />

- **application**
  - `<jsp:useBean id="..." class="..." scope="application" />

- **page**
  - `<jsp:useBean id="..." class="..." scope="page" />
    or just
  - `<jsp:useBean id="..." class="..." />

  - This scope is not used in MVC architecture
Storing Data for Later Use: The Servlet 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);
  // Use RequestDispatcher to forward to JSP page

- **JSP syntax to retrieve data**
  <jsp:useBean
  id="key"
  class="SomeClass"
  scope="request" />

Storing Data for Later Use: The Session Object

- **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);
  // Use RequestDispatcher to forward to JSP page

- **JSP syntax to retrieve data**
  <jsp:useBean
  id="key"
  class="SomeClass"
  scope="session" />
Storing Data for Later Use: The Servlet Context

• **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);
  // Use RequestDispatcher to forward to JSP page

• **JSP syntax to retrieve data**
  
  ```jsp
  <jsp:useBean
      id="key"
      class="SomeClass"
      scope="application"/>
  ```
Review: JSP Introduction

- JSP makes it easier to create/maintain HTML, while still providing full access to servlet code
- JSP pages get translated into servlets
  - It is the servlets that run at request time
  - Client does not see anything JSP-related
- You still need to understand servlets
  - Understanding how JSP really works
  - Servlet code called from JSP
  - Knowing when servlets are better than JSP
  - Mixing servlets and JSP
- Other technologies use similar approach, but aren’t as portable and don’t let you use Java for the “real code”
Uses of JSP Constructs

Simple Application
- Scripting elements calling servlet code directly
- Scripting elements calling servlet code indirectly (by means of utility classes)
- Beans
- Custom tags
- Servlet/JSP combo (MVC architecture)

Complex Application

Review: Calling Java Code Directly: JSP Scripting Elements

- **JSP Expressions**
  - Format: `<%= expression %>`
  - Evaluated and inserted into the servlet’s output.

- **JSP Scriptlets**
  - Format: `<% code %>`
  - Inserted verbatim into the `_jspService` method

- **JSP Declarations**
  - Format: `<%! code %>`
  - Inserted verbatim into the body of the servlet class

**Predefined variables**
- request, response, out, session, application

**Limit the Java code in page**
- Use helper classes, beans, custom tags, servlet/JSP combo
Review: The JSP page Directive: Structuring Generated Servlets

- **The import attribute**
  - Changes the packages imported by the servlet that results from the JSP page

- **The contentType attribute**
  - Specifies MIME type of result
  - Cannot be used conditionally

- **The isThreadSafe attribute**
  - Turns off concurrent access
  - Consider explicit synchronization instead

Review: Including Files in JSP Documents

- `<jsp:include page="Relative URL" flush="true" />`
  - Output inserted into JSP page at request time
  - Cannot contain JSP content that affects entire page
  - Changes to included file do not necessitate changes to pages that use it

- `<%@ include file="Relative URL" %>`
  - File gets inserted into JSP page prior to page translation
  - Thus, file can contain JSP content that affects entire page (e.g., import statements, declarations)
  - Changes to included file might require you to manually update pages that use it
**Review: Using JavaBeans Components with JSP**

- **Benefits of `jsp:useBean`**
  - Hides the Java programming language syntax
  - Makes it easier to associate request parameters with objects (bean properties)
  - Simplifies sharing objects among multiple requests or servlets/JSPs

- **`jsp:useBean`**
  - Creates or accesses a bean

- **`jsp:getProperty`**
  - Puts bean property (i.e. `getXXX` call) into output

- **`jsp:setProperty`**
  - Sets bean property (i.e. passes value to `setXXX`)

**Review: Creating Custom JSP Tag Libraries**

- **For each custom tag, you need**
  - A tag handler class (usually extending `TagSupport` or `BodyTagSupport`)
  - An entry in a Tag Library Descriptor file
  - A JSP file that imports library, specifies prefix, and uses tags

- **Simple tags**
  - Generate output in `doStartTag`, return `SKIP_BODY`

- **Attributes**
  - Define `setAttributeName` method. Update TLD file

- **Body content**
  - `doStartTag` returns `EVAL_BODY_INCLUDE`
  - Add `doEndTag` method
Review: Integrating Servlets and JSP

• **Use MVC (Model 2) approach when:**
  – One submission will result in multiple basic looks
  – Several pages have substantial common processing

• **Architecture**
  – A servlet answers the original request
  – Servlet does the real processing & stores results in beans
    • Beans stored in HttpServletRequest, HttpSession, or ServletContext
  – Servlet forwards to JSP page via forward method of RequestDispatcher
  – JSP page reads data from beans by means of jsp:useBean with appropriate scope (request, session, or application)

More Information

• **Core Servlets and JavaServer Pages**
  – http://www.coreservlets.com
  – More detail on all topics presented here

• **More Servlets and JavaServer Pages**
  – http://www.moreservlets.com
  – New features in servlets 2.3 and JSP 1.2 (filters and listeners), Web applications, security, standard JSP tag library

• **Servlet home page**

• **JavaServer Pages home page**
Questions?