Learning Objectives

- To understand JSP Model 1 and Model 2 architectures
- To create dynamic content using servlets and JavaServer Pages (JSPs)
- To use JavaBeans within the view and controller layers of a web application
- To manage the routing of a webflow
- To maintain user sessions
- To use tags from the JSP Standard Tag Library (JSTL)
- To create XML JSP (JSPX) pages
Java on the Server – The Enterprise Edition

- Java began as an object-oriented programming language with some supporting tools and APIs.
- With the Java 2 platform in 1999, Java was divided into three editions:
  - Java 2 Standard Edition (J2SE)
  - Java 2 Micro Edition (J2ME)
  - Java 2 Enterprise Edition (J2EE)
- In 2004 they were renamed Java SE, Java ME and Java EE.
The Java Enterprise Edition (Java EE)

- The core parts of the standard edition plus many additional APIs that enable us to write enterprise level software
  - distribution, security, transactions, persistence
- Supports web application development.
- Core components of Java EE systems
  - Servlets
  - JavaServer Pages (JSPs)
  - Enterprise JavaBeans (EJB)
  - + many others
Java EE Web Components

Diagram showing the integration of various components:
- Servlets
- JavaBeans
- JSPs
- JNDI
- JDBC

Flow between components:
- Servlets to JavaBeans
- JavaBeans to JSPs
- JNDI
- JDBC
- Database
Model-View-Controller (MVC) Architecture

View
- Heading
- Altitude
- Speed
- Location
...

Controller

Model

Change to model

Updated model

Notification

© Cengage Learning 2008 - 2010
MVC Roles

- **Model**
  - The underlying data
- **View**
  - What is presented to the user
- **Controller**
  - The handling component that manages user interaction and triggers appropriate updates to the model
- **Web application context**
  - Multiple views will be different clients with separate browsers looking at the same application in different ways
MVC, Model 1 and Model 2

- In the original JSP specification (version 0.92) Model 1 and Model 2 architectures were introduced.
- Model 1 architecture
  - a single server side component processes both the HTTP request and the HTTP response.
- Model 2 architecture
  - one server side component processes the request and another processes the response.
- This separation of concerns makes the Model 2 approach more scalable and flexible, but Model 1 is suitable for simple applications.
Front and Presentation Components

- Version 1.1 of the JSP specification describes the two components of the Model 2 architecture as the ‘front component’ and the ‘presentation component’
- the ‘front component’ in a Model 2 architecture can be either a JSP or a servlet
Model 1 and Model 2 Compared

- Model 1:
  - Browser
  - Servlet or JSP
  - Request
  - Response

- Model 2:
  - Browser
  - Servlet or JSP
  - Request
  - JSP
  - Presentation component

Front component

© Cengage Learning 2008 - 2010
JSP Model 1 With a Servlet

- Servlets were the first Java Web application components (1996)
- A servlet is managed by a servlet container/engine
  - Manages the lifecycle of the servlets and handles all the client threads that access them
- A servlet receives (HTTP) requests and returns (HTTP) responses
- The Java Enterprise Edition provides two servlet related packages, ‘javax.servlet’ and ‘javax.servlet.http’
Creating a Servlet

- Java EE includes an HttpServlet class that we subclass.
- Includes a method called ‘service’ that forwards the request to other methods that match the type of HTTP request.
- These methods have to be implemented by the servlet developer for different request types:
  - ‘doGet’ for get requests
  - ‘doPost’ for post requests
Servlet ‘doGet’ Method

- This is the signature of a servlet ‘doGet’ method
  
  ```java
  public void doGet
  (javax.servlet.http.HttpServletRequest request,
   javax.servlet.http.HttpServletResponse response)
  throws javax.servlet.ServletException, java.io.IOException
  ```

- We can gather information about the client’s HTTP request from the HttpServletRequest parameter, and send a response back to the same client using the HttpServletResponse parameter
  - This enables us to create dynamic content
This servlet uses information that is available in the ‘User-Agent’ field of the HTTP request header.

```java
public class UserAgentServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String userAgent = request.getHeader("User-Agent");
        out.println("Your browser sent the following header: <strong>" + userAgent + "</strong>");
        //…etc.
        out.close();
    }
}
```
Deploying a Servlet

- We must provide a URL that clients can use to invoke the servlet
  - In the ‘web.xml’ deployment descriptor

```xml
<servlet>
  <servlet-name>UserAgentServlet</servlet-name>
  <servlet-class>
    com.webhomecover.servlet.UserAgentServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>UserAgentServlet</servlet-name>
  <url-pattern>/useragent</url-pattern>
</servlet-mapping>
```
Invoking the Servlet

- The ‘servlet-name’ element serves as a cross-reference between the ‘servlet’ and ‘servlet-mapping’ elements
  - This is necessary since a single Web application can contain many servlets, and we need to provide each one with a unique name.
- Clients can invoke the servlet by using the ‘url-pattern’ as the resource name, e.g.
  
  http://localhost/webhomecover/useragent
JavaServer Pages (JSPs)

- Servlets are more suited to being ‘front components’ (i.e. processing the request) than they are at being ‘presentation components’ (i.e. generating the response.)
  - Manually writing out HTML tags is very tedious, error prone and hard to maintain
- JavaServer Pages (JSPs) were first introduced in 1999 as a way of overcoming the difficulty of generating mark-up using servlets
  - a JSP is a page of mark-up that can contain some Java programming elements
- JSPs are converted into servlets by the application server’s JSP engine
Writing a JavaServer Page

• A JavaServer Page is usually an (X)HTML page with some dynamic Java content embedded inside it, using special tags that are understood by a JSP compiler

• The JSP tags that enable dynamic content now come in two forms
  - The original syntax (similar to that used in Microsoft Active Server Pages)
  - XML compliant syntax
Converting XHTML to JSP

- Change the file extension
  - e.g. rename ‘welcome.htm’ to ‘welcome.jsp’
- The file remains in the document root of the Web application
- Load into the browser by name
  - http://localhost/webhomecover/welcome.jsp
- The server generates a servlet from the JSP
Writing JSP Expressions

- A JSP expression contains some Java code to provide a value that can be written to the HTTP response
  - This value can come from anything that can be converted to a String in Java, which includes all primitives and objects

- JSP expressions have the following syntax:
  ```
  <%=value %>
  ```
  - e.g. we can create a new ‘java.util.Date’ object, which the expression will convert to a String for display on the page:
  ```
  <%= new java.util.Date() %>
  ```
JSP Page Directives

- JSPs have a package import syntax, which takes the form of a JSP directive
- There are three different types of directive
  - to import a Java class or package we use what is known as a page directive
- This is the page import directive for the ‘java.util.Date’ class
  ```jsp
  <%@page import="java.util.Date" %>
  ```
- With the import in place, our JSP expression need only refer to the class name:
  ```jsp
  <%= new Date() %>
  ```
One of the most useful aspects of the HTTP request is that it can pass parameter data from the client to the server.

```html
<form action="claimservletpage.jsp" method="get">
  <input type="text" id="policynumber" name="policyNumber" />
  ...
  <input type="radio" name="type" id="buildings" value="buildings" />
  ...
  <input type="radio" name="type" id="contents" value="contents" />
  ...
  <textarea id="description" name="description" rows="5" cols="30"></textarea>
  ...
  <input type="submit" value="Submit" />
</form>
```
Retrieving the Parameter Values

- The built in JSP ‘request’ object has a ‘getParameter’ method
- This method takes a String parameter, representing the name of one of the components on the form, and returns the value of that component as a String
  
  ```java
  String policyNumber = request.getParameter("policyNumber");
  ```

- The name of the parameter must exactly match the name used in the HTML form
  - In the example we have used ‘camel case’ to be compatible with JavaBean properties
JSP Scriptlets

- JSP pages can include arbitrary sections of Java code as ‘scriptlets’
- The scriptlet tags looks like this:

  `<% Java code %>`

- Scriplets can use a number of ‘built-in’ objects that are provided by the JSP engine
  - out
    - a JSPWriter object for writing content to the response
  - request
    - an HttpServletRequest object that gives us access to the HTTP request
  - response
    - an HttpServletResponse object that gives us access to the HTTP response
  - session
    - an HttpSession object that represents an individual user’s session
Scriptlet Example

- This scriptlet retrieves the values of the four components of the XHTML form from the request and stores them in String variables
  - It uses the built-in JSP ‘request’ object and its ‘getParameter’ method.

```jsp
<% String policyNumber = request.getParameter("policyNumber");
String claimValue = request.getParameter("claim");
String claimType = request.getParameter("type");
String description = request.getParameter("description"); %>
```
There are one or two processes that may take place in the controller layer, such as data conversion, and validation. Since all the parameters are returned from the ‘request’ object as Strings, we sometimes need to perform conversion processes to change these Strings into more suitable data types.

```java
double valueOfClaim = 0.0;
try {
    valueOfClaim = Double.parseDouble(claimValue);
} catch(NumberFormatException e) {
    e.printStackTrace();
}
```
Presenting Client Data in the View Layer

- The NumberFormat class in the ‘java.text’ package, enables number and currency formatting
  - This scriptlet shows how we can format the ‘valueOfClaim’ variable

```jsp
<%  
    NumberFormat currencyFormat = NumberFormat.getCurrencyInstance();  
    String valueOfClaimFormatted = currencyFormat.format(valueOfClaim);  
%>
```
Using JavaBeans in JSPs

- JSP Model 1 includes JavaBean components that act as intermediaries between the view and controller layers and the underlying model
  - The JavaBean represents the interaction between the view and controller layers of a Web application and the underlying data source
JavaBeans

- A JavaBean is a Java programming component that represents either a Data Transfer Object (DTO) or a process or concept from the business logic layer.

- One of the main features of a JavaBean is that it supports ‘properties’:
  - JavaBeans have properties that are defined by methods.
  - A property is readable if it has a matching ‘getPropertyName’ method.
  - A property is writeable if it has a matching ‘setPropertyName’ method.
  - A property is not necessarily an attribute.
Accessor Methods

• The standard way of writing names for JavaBean accessor methods is to precede the name of a property by ‘get’ or ‘set’ in the method name, where the first letter of the property is in upper case.

```java
public String getDescription()
{
    return claimDescription;
}
public void setDescription (String description)
{
    claimDescription = description;
}
```
ClaimBean

policyNumber: String
claimValue: double
claimType: String
claimDescription: String

getPolicyNumber(): String
getClaim(): String
getType(): String
getDescription(): String
setPolicyNumber(String)
setClaim(String)
setType(String)
setDescription(String)

Encapsulates data formatting
Encapsulates data conversion
JSP Tags for JavaBeans

- JavaBeans can be used in JSPs using special tags that make it very easy to get and set their properties, and even automatically populate them from the HTTP request.

- The tag that either creates or accesses a Java bean is ‘useBean’, which has a number of possible attributes:
  - ‘useBean’ is an empty element, with a ‘jsp’ prefix that specifies its namespace.

```jsp
<jsp:useBean id="claimBean" class="com.webhomecover.beans.ClaimBean" />
```
Setting Properties

- The ‘setProperty’ tag comes in a number of forms
  - This example automatically sets matching properties of the bean from the request parameters
    ```jsp
    <jsp:setProperty name="claimBean" property="*" />
    ```
  - This explicitly sets a named property from a named parameter
    ```jsp
    <jsp:setProperty name="claimBean" property="claimValue" param="claim" />
    ```
  - If the value of a property does not come from a request parameter, then we use the ‘value’ attribute
    ```jsp
    <jsp:setProperty name="claimBean" property="description" value="No Description" />
    ```
Writing Bean Properties to the Response

- We can write bean properties out to the response using the ‘getProperty’ tag

```xml
<p>Here are the details you entered</p>
Policy number: 
<jsp:getProperty name="claimBean" property="policyNumber"/>
<br /> Value of claim:
<jsp:getProperty name="claimBean" property="claim"/>
<br /> Type of insurance policy:
<jsp:getProperty name="claimBean" property="type"/>
<br /> Description of claim:
<jsp:getProperty name="claimBean" property="description"/>
```
JSP Model 2 Architecture

- The interaction model of Model 1 architecture is perfectly workable but is not ideal
  - The server page has multiple responsibilities, handling both the request and the response

- To address this problem we can apply the JSP Model 2 architecture

1. request

Browser

2. Create / populate

JSP (controller)

Application server

3. Data source

4. JavaBean (model)

5. response

JSP (view)
Page Structure Following JSP Model 2 Architecture

- Web-based Model View Controller pattern

[Diagram showing the model-view-controller pattern with 'Form', 'Server page', 'JavaBean', 'Client page', and arrows indicating 'submit', 'forward', and 'builds']
Page Forwarding

- For Model 2 we need to be able to forward from one page to the other.
- Use the ‘jsp:forward’ tag, one of the standard action tags:
  
  `<jsp:forward page="claimresponseserverpage.jsp" />`

- We should always ensure that after forwarding we do not have any more processing in the JSP, since a forward should be the last thing that happens in a page (after all, we don’t expect to be coming back).
Request Scope

- The default scope of a JavaBean is ‘page’
- For multiple JSPs to access the same bean we put the bean in ‘request’ scope

```
<jsp:useBean id="claimBean" class="com.webhomecover.beans.ClaimBean" scope="request"/>
```
The Webflow

- We will look at the webflow design for the ‘get insurance quote’ use case
- This involves a series of pages and conditional page routing
- Also involves ‘including’ server side processes
Including Pages

- Where the same process needs to be called from multiple pages it can be ‘included’ using another JSP.
- After a JSP has been included, control passes back to the first JSP, which then continues with the webflow.
- The tag for an ‘include’ action is similar to the one for the ‘forward’ action:
  ```xml
  <jsp:include page="processquote.jsp" />
  ```
- The role of this JSP is basically that of a command object:
  - Controller pages manage the routing of the application, but command objects perform interactions with the model.
choosequote.jsp

buildingsdetails.jsp

buildingsdetails.jsp

contentsdetails.jsp

processbuildings.jsp

processcontents.jsp

displayquote.jsp

processquote.jsp

processchoice.jsp

Storyboard

[Buildings insurance selected]

[Buildings insurance not selected, Contents insurance selected]

[Contents insurance selected]

[Contents insurance not selected]
Session Management

- HTTP is a stateless protocol
  - Does not maintain the connection after a request/response cycle
- We need to store client session state on the server
- Use the ‘session’ object in JSPs
- An ‘HttpSession’ is basically like a HashTable or HashMap where the keys are Strings and the values are Objects

```java
session.setAttribute("firstName", fname);
```
Accessing Session Attributes

- We can get a reference to an object in a session using the ‘getAttribute’ method
  - The return type of this method is ‘Object’, so we have to cast the returned value to the appropriate type

  String fname = (String)session.getAttribute("firstName");

- To remove an object from the session, use the ‘removeAttribute’ method, passing in the necessary key value

  session.removeAttribute("itemKey");
Session State

- One important aspect of session management is freeing up resources that are no longer required
  ```java
  session.invalidate();
  ```
- At the beginning of a webflow we may want to check if the session that we have acquired from the request is new
- We can send the client back to start the webflow from the beginning by forwarding or redirecting
  ```jsp
  <%
  if(session.isNew())
  {
  %>
  <jsp:redirect page="choosequote.jsp"/>
  <%
  }
  %>
  ```
JavaBeans and Sessions

- The default scope of a JavaBean is the ‘page’
- we can change that using the ‘scope’ attribute of the ‘jsp:useBean’ tag.

```jsp
<jsp:useBean id="claimBean"
class="com.webhomecover.beans.ClaimBean" scope="session"/>
```

- The JSP will look for the bean in the session rather than the page or request, and add it to the session when it creates it.
Routing the Webflow

- The controller JSPs need to manage the routing of the webflow
- This is ‘processchoice.jsp’

```jsp
<jsp:useBean id="choice"
    class="com.webhomecover.beans.QuoteChoice" scope="session" />
<jsp:setProperty name="choice" property="*" />
<% if(choice.getBuildings()) { %>
    <jsp:forward page="buildings.jsp"/>
<% } else {%
    <jsp:forward page="contents.jsp"/>
<% }
}%>`
Process Beans

- Not all JavaBeans are just Data Transfer Objects (DTOs) for sending form data back and forth
- Some beans encapsulate business processes

```java
import com.webhomecover.model;

public class InsuranceQuoter {
    public double getBuildingsQuote(BuildingsDetails details) {
        // Implementation
    }
    public double getContentsQuote(ContentsDetails details) {
        // Implementation
    }
}
```
JSP Standard Tag Library (JSTL)

- Scriptlets can get ugly and confusing, introducing too much Java code into JSPs
- Using a tag library is a good alternative
- The JSTL provides us with a standard set of tags that can be used with any Java application server. These tags are distributed among four libraries:
  - Core
  - Internationalization and formatting
  - Database access
  - XML processing
Using the JSTL

- The JSTL implementation is an open source project
- It has to be downloaded from the Jakarta web site
  - http://jakarta.apache.org/taglibs/
- The two jar files ‘jstl.jar’ and ‘standard.jar’ must be put into the ‘lib’ folder under the WEB-INF folder of the web application
JSTL Library Files

- You have to add some jar files to the ‘lib’ folder of the web application
- There is a ‘taglib’ directive for using the libraries

```jsp
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```
The ‘c’ prefix is the one conventionally used for the core library

<table>
<thead>
<tr>
<th>JSTL core tag name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>c:forEach</td>
<td>Iteration</td>
</tr>
<tr>
<td>c:if</td>
<td>A conditional code block with no alternative action</td>
</tr>
<tr>
<td>c:choose</td>
<td>A conditional code block containing an alternative action</td>
</tr>
<tr>
<td>c:when</td>
<td>Part of the ‘choose’ structure, executed if the ‘choose’ condition is true</td>
</tr>
<tr>
<td>c:otherwise</td>
<td>Part of the ‘choose’ structure, executed if the ‘choose’ condition is false</td>
</tr>
<tr>
<td>c:out</td>
<td>Write to the page</td>
</tr>
<tr>
<td>c:set</td>
<td>Set the value of a variable</td>
</tr>
</tbody>
</table>
The Expression Language

- the JSTL includes the Expression Language (EL),
- An expression is preceded by a dollar sign and surrounded by braces
  \${expression}
- e.g.

  `<c:set var="localname" scope="request" value="no value"/>
  <c:out value="\${localname}"/>

- Within an expression we can use the names of JavaBeans and access
  their properties using the dot operator:
  \${beanname.propertyname}

- For example, if we wanted to access the ‘bedroomCount’ property of
  the ‘building’ bean, the expression would look like this:
  \${building.bedroomCount}`
Using JavaBeans in Expressions

```jsp
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<jsp:useBean id="choice"
    class="com.webhomecover.beans.QuoteChoice" scope="session" />
<jsp:setProperty name="choice" property="buildings" value="false" />
<jsp:setProperty name="choice" property="contents" value="false" />
<jsp:setProperty name="choice" property="*" />

<!-- select which page to go to depending on the check box selected -->
<c:choose>
    <c:when test="${choice.buildings}">
        <jsp:forward page="buildings.jsp"/>
    </c:when>
    <c:otherwise>
        <c:if test="${choice.contents}">
            <jsp:forward page="contents.jsp"/>
        </c:if>
    </c:otherwise>
</c:choose>
```
Indexed Properties

- Bean properties can be indexed, using an array
- Can be useful if we want to store multiple values, e.g. from a select

```html
<select name="interests" id="list" size="4" multiple="multiple">
  <option id="quote">Getting home insurance quotes</option>
  <option id="claim">Making a claim</option>
  <option id="buy">Buying insurance</option>
  <option id="change">Making a change to your policy</option>
</select>
```
Bean with an Indexed Property

- Here is a simple JavaBean (‘InterestBean’) that contains a single indexed property (‘interests’)

```java
public class InterestBean {
    private String[] interests;
    public String[] getInterests() {
        return interests;
    }
    public void setInterests(String[] arr) {
        interests = arr;
    }
}
```
Using the Bean

```html
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
...
<h1>Your interests</h1>
<jsp:useBean id="interestBean" class="com.webhomecover.beans.InterestBean" />
<jsp:setProperty name="interestBean" property="interests" param="interests" />
<div>
<c:forEach var="selected" items="${interestBean.interests}">
<c:out value="${selected}" />
<br />
</c:forEach>
</div>
</body>
</html>
```
XML JSPs

- JSP pages that use the XML syntax are often given a ‘.jspx’ file extension.
- To make it possible to write JSP pages as well-formed XML, the JSP tags that use the non-XML style also have an XML equivalent.

<table>
<thead>
<tr>
<th>JSP tag</th>
<th>Type</th>
<th>JSPX tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;%     %&gt;</td>
<td>scriptlet</td>
<td><a href="">jsp:scriptlet</a>       &lt;/jsp:scriptlet&gt;</td>
</tr>
<tr>
<td>&lt;%=     %&gt;</td>
<td>expression</td>
<td><a href="">jsp:expression</a>   &lt;/jsp:expression&gt;</td>
</tr>
<tr>
<td>&lt;%^@page ... %&gt;</td>
<td>page directive</td>
<td>&lt;jsp:directive.page …&gt;  &lt;/jsp:directive&gt;</td>
</tr>
</tbody>
</table>
Taglibs

- Unfortunately, however, not all conversions to XML style elements are simple.
- If we are using the JSTL, then our pages will include a taglib directive like this:

```jsp
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

- There is no direct XML equivalent, so the namespace of tag libraries has to be added to a root element

```xml
<jsp:root
xmlns:jsp="http://java.sun.com/JSP/Page"
xmlns:c="http://java.sun.com/jsp/jstl/core"
version="2.1">
```
Writing the DOCTYPE

- To convert JSP view pages to JSPX, we would also need to generate the XHTML doctype in an XML compliant way.
- We can use the ‘jsp:output’ tag to do this:

```
<jsp:directive.page contentType="text/html"/>
<jsp:output omit-xml-declaration="false"
doctype-root-element="html"
doctype-public="-//W3C//DTD XHTML 1.1//EN"
doctype-system="http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"/>
```
Three Region Layout

- One feature of JSPs is that they provide an ‘include’ directive that can be used to integrate static XHTML fragments into a generated page:

  `<jsp:directive.include file="filename" />`

- We can use this technique to bring in fragments of the three region layout
Welcome to WebHomeCover.com

Your home and possessions are important.
We give you the very best insurance cover at the lowest prices.

Get an insurance quote
Chapter Summary

- Model 1 and Model 2 architectures
- Servlets
- JSPs
- JavaBeans
- Webflow with user sessions
- JTSLS and EL
- JSPX pages