# J2EE

## Java Server Pages (JSP)

*Presented by Bartosz Sakowicz*

# JSP basics

•JavaServer Pages (JSP) technology enables you to mix regular, static HTML with dynamically generated content from servlets.

•Although what you write often looks more like a regular HTML file than a servlet, behind the scenes, the JSP page is automatically converted to a normal servlet, with the static HTML simply being printed to the output stream associated with the servlet's service method. This translation is normally done the first time the page is requested.

Presented by Bartosz Sakowicz

DMCS TUL

# Suggestion for IE5 or IE6

•If you make an error in the dynamic portion of your JSP page, the system may not be able to properly translate it into a servlet. If your page has such a fatal translation-time error, the server will present an HTML error page describing the problem to the client.

•Internet Explorer 5 typically replaces server-generated error messages with a canned page that it considers friendlier.

•You will need to turn off this “feature” when debugging JSP pages. To do so with Internet Explorer 5 or 6, go to the Tools menu, select Internet Options, choose the Advanced tab, and make sure “Show friendly HTTP error messages” box is not checked.

# JSP constructs

Aside from the regular HTML, there are three main types of JSP constructs that you embed in a page:

- **scripting elements** - scripting elements let you specify Java code that will become part of the resultant servlet
- **directives** - directives let you control the overall structure of the servlet
- **actions** - actions let you specify existing components that should be used and otherwise control the behavior of the JSP engine.

# Scripting elements

JSP scripting elements let you insert code into the servlet that will be generated from the JSP page. There are three forms:

1. **Expressions of the form <%= expression %>**, which are evaluated and inserted into the servlet's output

2. **Scriptlets of the form <% code %>**, which are inserted into the servlet's _jspService method (called by service)

3. **Declarations of the form <%! code %>**, which are inserted into the body of the servlet class, outside of any existing methods

# JSP and HTML comments

If you want a comment to appear in the JSP page but not in the resultant document, use:

**<%-- *JSP Comment* --%>**

HTML comments of the form:

**<!-- *HTML Comment* -->**

are passed through to the resultant HTML normally.

# JSP expressions

**A JSP expression is used to insert values directly into the output.** It has the following form:

**<%= *Java Expression* %>**

The expression is evaluated, converted to a string, and inserted in the page. This evaluation is performed at run time (when the page is requested) and thus has full access to information about the request. For example, the following shows the date/time that the page was requested:

Current time: *<%= new java.util.Date() %>*

# XML syntax

XML authors can use the following alternative syntax for JSP **expressions:**

*<jsp:expression>*
*Java Expression*
*</jsp:expression>*

**scriplets** (on following transparencies)**:**

<jsp:scriptlet>
*Code*
</jsp:scriptlet>

and **declarations** (also on following transparencies):

*<jsp:declaration>*
*Code*
*</jsp:declaration>*

**Note that XML elements, unlike HTML ones, are case sensitive, so be sure to use jsp:expression in lower case.**

# JSP scriplets

•If you want to do something more complex than insert a simple expression, JSP scriptlets let you insert arbitrary code into the servlet's _jspService method (which is called by service).

**Scriptlets have the following form:**

*<% Java Code %>*

•**Scriptlets have access to the same automatically defined variables as expressions.**

•For example, if you want output to appear in the resultant page, you would use the out variable, as in the following example.

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

•In this particular instance, you could have accomplished the same effect more easily by using the following JSP expression:

*Attached GET data: <%= request.getQueryString() %>*

# JSP scriplets(2)

**Scriptlets can perform a number of tasks that cannot be accomplished with expressions alone:**

•include setting response headers and status codes
•invoking side effects such as writing to the server log or updating a database
•executing code that contains loops, conditionals, or other complex constructs.

# JSP scriplets(3)

**Scriptlets can perform a number of tasks that cannot be accomplished with expressions alone:**

•include setting response headers and status codes
•invoking side effects such as writing to the server log or updating a database
•executing code that contains loops, conditionals, or other complex constructs.

**Other features:**

You can set response headers or status codes at various places within a JSP page. Setting headers and status codes is permitted because servlets that result from JSP pages use a special type of PrintWriter (of the more specific class **JspWriter**) that buffers the document before sending it. This buffering behavior can be changed (*autoflush* attribute of the *page* directive).

# Using scriplets to make part of JSP page conditional

Scriptlets need not contain complete Java statements, and blocks left open can affect the static HTML or JSP outside of the scriptlets:

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

When converted to a servlet by the JSP engine, this fragment will result in something similar to the following.

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

# JSP declarations

A JSP declaration lets you define methods or fields that get inserted into the main body of the servlet class (*outside* of the _jspService method that is called by service to process the request).

**A declaration has the following form:**

*<%! Java Code %>*

Since declarations do not generate any output, they are normally used in conjunction with JSP expressions or scriptlets.

# Predefined variables

To simplify code in JSP expressions and scriptlets, you are supplied with eight automatically defined variables (called *implicit objects)*. Since JSP declarations result in code that appears outside of the _jspService method, **these variables are not accessible in declarations.**

**The variables are:**

**request**
This variable is the HttpServletRequest associated with the request

**response**
This variable is the HttpServletResponse associated with the response to the client.

# Predefined variables(2)

**out**

This is the PrintWriter used to send output to the client (precisely this is a buffered version of Print Writer called JspWriter).Note that out is used almost exclusively in scriptlets, since JSP expressions are automatically placed in the output stream and thus rarely need to refer to out explicitly.

**session**

This variable is the HttpSession object associated with the request. Sessions are created automatically, so this variable is bound even if there is no incoming session reference. The one exception is if you use the session attribute of the page directive to turn sessions off. In that case, attempts to reference the session
variable cause errors at the time the JSP page is translated into a servlet.

# Predefined variables(3)

**application**
This variable is the ServletContext as obtained via *getServletConfig().getContext()*. Servlets and JSP pages can store persistent data in the ServletContext object rather than in instance variables. ServletContext has setAttribute and getAttribute methods that let you store arbitrary data associated with specified keys. The difference between storing data in instance variables and storing it in the ServletContext is that the ServletContext is shared by all servlets in the servlet engine.

**config**
This variable is the ServletConfig object for this page.

# Predefined variables(4)

**pageContext**
JSP introduced a new class called PageContext to give a single point of access to many of the page attributes and to provide a convenient place to store shared data. The pageContext variable stores the value of the PageContext object associated with the current page.

**page**
This variable is simply a synonym for this and is not very useful in the Java programming language.

# Directives

**JSP directive affects the overall structure of the servlet that results from the JSP page.** There are two possible forms for directives:

```
<%@ directive attribute="value" %>
<%@ directive attribute1="value1"
              attribute2="value2"
              ...
              attributeN="valueN" %>
```

Single quotes can be substituted for the double quotes around the attribute values, but the quotation marks cannot be omitted altogether. To obtain quote marks within an attribute value, precede them with a back slash, using \’ for ’ and \" for ".

# Directives(2)

**There are three types of directives:**

**page** - the page directive lets you control the structure of the servlet by importing classes, customizing the servlet superclass, setting the content type etc. A page directive can be placed anywhere within the document.

**include** – the include directive lets you insert a file into the servlet class at the time the JSP file is translated into a servlet. An include directive should be placed in the document at the point at which you want the file to be inserted.

**taglib** – the taglib directive can be used to define custom markup tags.

# The Page directive

**The page directive lets you define one or more of the following case-sen-sitive attributes:**

- import
- contentType
- isThreadSafe
- session
- buffer
- autoflush
- extends
- info
- errorPage
- isErrorPage
- language

# The import attribute

The *import* attribute of the page directive lets you specify the packages that should be imported by the servlet into which the JSP page gets translated. If you don't explicitly specify any classes to import, the servlet imports:

- java.lang.*
- javax.servlet.*
- javax.servlet.jsp.*
- javax.servlet.http.*
- possibly some number of server-specific entries.

Never write JSP code that relies on any server-specific classes being imported automatically.

# The import attribute(2)

Use of the import attribute takes one of the following two forms:

*<%@ page import="package.class" %>*
*<%@ page import="package.class1,...,package.classN" %>*

**Example:**

*<%@ page import="java.util.*" %>*

•The import attribute is the only page attribute that is allowed to appear multiple times within the same document.

•It is traditional to place import statements either near the top of the document or just before the first place that the referenced package is used.

# The contentType attribute

The **contentType** attribute sets the Content-Type response header, indicating the MIME type of the document being sent to the client. Use of the contentType attribute takes one of the following two forms:

<%@ page contentType="MIME-Type" %>
<%@ page contentType="MIME-Type; charset=Character-Set" %>

For example, the directive
<%@ page contentType="text/plain" %>
has the same effect as the scriptlet
<% response.setContentType("text/plain"); %>
Unlike regular servlets, where the default MIME type is text/plain, the default for JSP pages is text/html (with a default character set of ISO-8859-1).

# The usage of contentType attribute - example

Generating Excel Spreadshits:

```
<%
String format = request.getParameter("format");
if ((format != null) && (format.equals("excel"))) {
response.setContentType("application/vnd.ms-excel");
}
%>
<TABLE BORDER=1>
<TR><TH></TH><TH>Apples                    <TH>Oranges
<TR><TH>First Quarter          <TD>2307    <TD>4706
<TR><TH>Second Quarter         <TD>2982    <TD>5104
<TR><TH>Third Quarter          <TD>3011    <TD>5220
<TR><TH>Fourth Quarter         <TD>3055    <TD>5287
</TABLE> ...
```

# The isThreadSafe attribute

The **isThreadSafe** attribute controls whether or not the servlet that results from the JSP page will implement the **SingleThreadModel interface**. Use of the isThreadSafe attribute takes one of the following two forms:

<%@ page isThreadSafe="true" %> <%!-- Default --%>

<%@ page isThreadSafe="false" %>

With normal servlets, simultaneous user requests result in multiple threads concurrently accessing the service method of the same servlet instance.

# The isThreadSafe ...(2)

**An example of thread unsafe scriplet**:

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

**And thread safe scriplet:**

```
<%! private int idNum = 0; %>
<%
synchronized(this) {
String userID = "userID" + idNum;
out.println("Your ID is " + userID + ".");
idNum = idNum + 1;
}
%>
```

# The session attribute

The **session** attribute controls whether or not the page participates in HTTP sessions. Use of this attribute takes one of the following two forms:

*<%@ page session="true" %> <%-- Default --%>*
*<%@ page session="false" %>*

A value of **true** (the default) indicates that the predefined variable session (of type HttpSession) should be bound to the existing session if one exists; otherwise, a new session should be created and bound to session. A value of **false** means that no sessions will be used automatically and attempts to access the variable session will result in errors at the time the JSP page is translated into a servlet.

# The buffer attribute

The **buffer** attribute specifies the size of the buffer used by the out variable, which is of type JspWriter (a subclass of PrintWriter). Use of this attribute takes one of two forms:

*<%@ page buffer="sizekb" %>*
*<%@ page buffer="none" %>*

Servers can use a larger buffer than you specify, but not a smaller one. For example, *<%@ page buffer="32kb" %>* means the document content should be buffered and not sent to the client until at least 32 kilobytes have been accumulated or the page is completed. The default buffer size is server specific, but must be at least 8 kilobytes. Be cautious about turning off buffering; doing so requires JSP entries that set headers or status codes to appear at the top of the file, before any HTML content.

# The autoflush attribute

The **autoflush** attribute controls whether the output buffer should be automatically flushed when it is full or whether an exception should be raised when the buffer overflows. Use of this attribute takes one of the following two forms:

*<%@ page autoflush="true" %> <%-- Default --%>*
*<%@ page autoflush="false" %>*

A value of false is illegal when also using buffer="none".

# The extends attribute

The **extends** attribute indicates the superclass of the servlet that will be generated for the JSP page and takes the following form:

*<%@ page extends="package.class" %>*

Use this attribute with extreme caution since the server may be using a custom superclass already.

# The info and language attributes

The **info** attribute defines a string that can be retrieved from the servlet by means of the getServletInfo method. Use of info takes the following form:

*<%@ page info="Some Message" %>*

The **language** attribute is intended to specify the underlying programming language being used, as below:

*<%@ page language=„java" %>*

For now java is both the default and the only legal choice.

# The errorPage and isErrorPage attributes

The **errorPage** attribute specifies a JSP page that should process any exceptions (i.e., something of type Throwable) thrown but not caught in the current page. It is used as follows:

*<%@ page errorPage="Relative URL" %>*

The exception thrown will be automatically available to the designatederror page by means of the **exception** variable.

The **isErrorPage** attribute indicates whether or not the current page can act as the error page for another JSP page. Use of isErrorPage takes one of the following two forms:

*<%@ page isErrorPage="true" %>*
*<%@ page isErrorPage="false" %> <%!-- Default --%>*

# The errorPage ... - example

**ComputeSpeed.jsp:**

```
...
<%@ page errorPage="SpeedErrors.jsp" %>
...
<%!
// Note lack of try/catch for NumberFormatException if
// value is null or malformed.
private double toDouble(String value) {
return(Double.valueOf(value).doubleValue());
}
...
double param = toDouble(request.getParameter(„param"));
... %>
```

# The errorPage...– example(2)

**SpeedErrors.jsp:**

*...*
***<%@ page isErrorPage="true" %>***
*...*
*ComputeSpeed.jsp reported the following error:*
*<%= **exception** %>. This problem occurred in the following place:*
*<% **exception**.printStackTrace(new PrintWriter(out)); %>*
*...*

# XML syntax for directives

JSP permits you to use an alternative XML-compatible syntax for directives.These constructs take the following form:

*<jsp:directive.directiveType attribute="value" />*

For example, the XML equivalent of:

*<%@ page import="java.util.*" %>*

is:

*<jsp:directive.page import="java.util.*" />*

# The include directive

You use the include directive to include a file in the main JSP document at the time the document is translated into a servlet (which is typically the first time it is accessed). The syntax is as follows:

*<%@ include file="Relative URL" %>*

- **If the included file changes, all the JSP files that use it need to be updated.**
- **The file is included at page translation time.**
- **Included file can contain JSP constructs (such as field or method declarations) that affect the main page as a whole.**

# Including files at request time

• The **jsp:include** action includes files at the time of the client request and thus does not require you to update the main file when an included file changes.

• At request time the page has already been translated into a servlet, so the included files cannot contain JSP.

The jsp:include element has two required attributes: page (a relative URL referencing the file to be included) and flush (which *must* have the value true).

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

# Java plug-in

To include ordinary applet just use the normal HTML APPLET tag. However, these applets must use JDK 1.1 or JDK 1.02 since neither Netscape 4.x nor Internet Explorer 5.x support the Java 2 platform (i.e., JDK 1.2). This lack of support imposes several restrictions on applets:

- In order to use Swing, you must send the Swing files over the network. This process is time consuming and fails in Internet Explorer 3 and Netscape 3.x and 4.01-4.05 (which only support JDK 1.02), since Swing depends on JDK 1.1.
- You cannot use Java 2D.
- You cannot use the Java 2 collections package.
- Your code runs more slowly, since most compilers for the Java 2 platform are significantly improved over their 1.1 predecessors.

# Java plug-in(2)

Sun developed a browser plug-in for Netscape and Internet Explorer that lets you use the Java 2 platform for applets in a variety of browsers. This plug-in is available at http://java.sun.com/products/plugin/, and also comes bundled with JDK 1.2.2 and later.

- It is not reasonable to expect users on the WWW at large to download and install it just to run your applets.
- It is a reasonable alternative for fast corporate intranets.

# Java plug-in(3)

• The normal **APPLET** tag will not work with the plug-in, since browsers are specifically designed to use only their builtin virtual •machine when they see APPLET.

• You have to use a long **OBJECT** tag for Internet Explorer and an equally long **EMBED** tag for Netscape.

• Since you typically don't know which browser type will be accessing your page, you have to either include both OBJECT and EMBED (placing the EMBED within the COMMENT section of OBJECT) or identify the browser type at the time of the request and conditionally build the right tag. This process is time consuming.

• **The jsp:plugin element instructs the server to build a tag appropriate for applets that use the plug-in.**

# The jsp:plugin element

The simplest way to use **jsp:plugin** is to supply four attributes: type, code, width, and height. You supply a value of applet for the type attribute and use the other three attributes in exactly the same way as with the APPLET element, with two exceptions: the attribute names are case sensitive, and single or double quotes are always required around the attribute values. For example, you could replace:

*<APPLET CODE="MyApplet.class„ WIDTH=475 HEIGHT=350>*
*</APPLET>*

with:

```
<jsp:plugin    type="applet"
               code="MyApplet.class"
               width="475"
               height="350">
</jsp:plugin>
```

# The jsp:plugin attributes

**type**

For applets, this attribute should have a value of applet. However, the Java Plug-In also permits you to embed JavaBeans elements in Web pages. Use a value of bean in such a case.

**code**

This attribute is used identically to the CODE attribute of APPLET, specifying the top-level applet class file that extends Applet or JApplet.

**width**

This attribute is used identically to the WIDTH attribute of APPLET, specifying the width in pixels to be reserved for the applet.

**height**

This attribute is used identically to the HEIGHT attribute of APPLET, specifying the height in pixels to be reserved for the applet.

# The jsp:plugin attributes(2)

**codebase**

This attribute is used identically to the CODEBASE attribute of APPLET, specifying the base directory for the applets. The code attribute is interpreted relative to this directory. As with the APPLET element, if you omit this attribute, the directory of the current page is used as the default. In the case of JSP, this default location is the directory where the original JSP file resided, not the system-specific location of the servlet that results from the JSP file.

**align**

This attribute is used identically to the ALIGN attribute of APPLET and IMG, specifying the alignment of the applet within the Web page. Legal values are *left, right, top, bottom*, and *middle.*

# The jsp:plugin attributes(3)

**hspace**

This attribute is used identically to the HSPACE attribute of APPLET, specifying empty space in pixels reserved on the left and right of the applet.

**vspace**

This attribute is used identically to the VSPACE attribute of APPLET, specifying empty space in pixels reserved on the top and bottom of the applet.

**archive**

This attribute is used identically to the ARCHIVE attribute of APPLET, specifying a JAR file from which classes and images should be loaded.

# The jsp:plugin attributes(4)

**name**

This attribute is used identically to the NAME attribute of APPLET, specifying a name to use for inter-applet communication or for identifying the applet to scripting languages like JavaScript.

**title**

This attribute is used identically to the very rarely used TITLE attribute of APPLET, specifying a title that could be used for a tool-tip or for indexing.

**jreversion**

This attribute identifies the version of the Java Runtime Environment (JRE) that is required. The default is 1.1.

# The jsp:plugin attributes(5)

**iepluginurl**
This attribute designates a URL from which the plug-in for Internet Explorer can be downloaded. Users who don't already have the plug-in installed will be prompted to download it from this location. The default value will direct the user to the Sun site, but for intranet use you might want to direct the user to a local copy.
**nspluginurl**
This attribute designates a URL from which the plug-in for Netscape can be downloaded.

# The jsp:param and jsp:params elements

• The **jsp:param** element is used with jsp:plugin in a manner similar to the way that PARAM is used with APPLET, specifying a name and value that are accessed from within the applet by getParameter.

• jsp:param follows XML syntax, attribute'names must be lower case, attribute values must be enclosed in single or double quotes, and the element must end with />, not just >.

• All **jsp:param** entries must be enclosed within a **jsp:params** element.

# The jsp:param and jsp:params elements(2)

**Example:**

You can replace:

```
<APPLET CODE="MyApplet.class„ WIDTH=475 HEIGHT=350>
<PARAM NAME="PARAM1" VALUE="VALUE1">
<PARAM NAME="PARAM2" VALUE="VALUE2">
</APPLET>
```

with:

```
<jsp:plugin type="applet„ code="MyApplet.class„ width="475"
height="350">
<jsp:params>
        <jsp:param name="PARAM1" value="VALUE1" />
        <jsp:param name="PARAM2" value="VALUE2" />
</jsp:params>
</jsp:plugin>
```

# The jsp:fallback element

The **jsp:fallback** element provides alternative text to browsers that do not support OBJECT or EMBED. You use this element in almost the same way as you would use alternative text placed within an APPLET element. For example, you would replace:

```
<APPLET CODE="MyApplet.class" WIDTH=475 HEIGHT=350>
<B>Error: this example requires Java.</B>
</APPLET>
```

with:

```
<jsp:plugin type="applet„ code="MyApplet.class"
        width="475" height="350">
        <jsp:fallback>
                <B>Error: this example requires Java.</B>
        </jsp:fallback>
</jsp:plugin>
```

# The jsp:plugin example

**Lets consider following example:**

```
<jsp:plugin type="applet” code=„MyApplet.class"
                    width="475" height="350">
        <jsp:params>
                <jsp:param name="MESSAGE" value="Your Msg" />
        </jsp:params>
</jsp:plugin>
```

# The jsp:plugin example(2)

**The HTML output would be like this:**

*<**OBJECT** classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93„ width="475" height="350" codebase="http://java.sun.com/products/plugin/1.2.2/jin-stall-1_2_2-win.cab#Version=1,2,2,0">*
*<PARAM name="java_code" value=„MyApplet.class">*
*<PARAM name="type" value="application/x-java-applet;">*
*<PARAM name="MESSAGE" value="Your Msg">*
*<COMMENT>*
*<**EMBED** type="application/x-java-applet;" width="475" height="350" pluginspage="http://java.sun.com/products/plugin/" java_code=„MyApplet.class" MESSAGE="Your Msg" >*
*<NOEMBED> </COMMENT> </NOEMBED></EMBED>*
*</OBJECT>*

# JavaBeans

**The JavaBeans API** provides a standard format for Java classes.

Visual manipulation tools and other programs can automatically discover information about classes that follow this format and can then create and manipulate the classes without the user having to explicitly write any code.

Presented by Bartosz Sakowicz

DMCS TUL

# JavaBeans(2)

**To be a JavaBean class must follow particular rules:**

**1. A bean class must have a zero-argument constructor.**
You can satisfy this requirement either by explicitly defining such a constructor or by omitting all constructors, which results in an empty constructor being created automatically. The empty constructor will be called when JSP elements create beans.

**2. A bean class should have no public instance variables.** You should use private instance variables and accessor methods**.** Use of accessor methods lets you impose constraints on variable values (e.g., have the setSpeed method of your Car class disallow negative speeds) or allows you to change your inter-nal data structures without changing the class interface.

# JavaBeans(3)

**3. Persistent values should be accessed through methods called ge*tXxx* and se*tXxx*.** For example, if your Car class stores the current number of passengers, you might have methods named getNumPassengers (which takes no arguments and returns an int) and setNumPassengers (which takes an int and has a void return type). In such a case, the Car class is said to have a *property* named numPassengers (notice the lowercase n in the property name, but the uppercase N in the method names).

If the class has a ge*tXxx* method but no corresponding se*tXxx*, the class is said to have a read-only property named *xxx*. The one exception to this naming convention is with boolean properties: they use a method called is*Xxx* to look up their values.

# JavaBean example

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

# The jsp:usebean element

The **jsp:useBean** action lets you load a bean to be used in the JSP page. The simplest syntax for specifying that a bean should be used is:

*<jsp:useBean id="name" class="package.Class" />*

This usually means “instantiate an object of the class specified by Class, and bind it to a variable with the name specified by id.” For example, the JSP action:

*<jsp:useBean id="book1" class="package.Book" />*
can normally be thought of as equivalent to the scriptlet:

*<% package.Book book1 = new package.Book(); %>*

# Accessing bean properties

Once you have a bean, you can access its properties with **jsp:getProperty**, which takes a *name* attribute that should match the *id* given in *jsp:useBean* and a *property* attribute that names the property of interest. Alternatively, you could use a JSP expression and explicitly call a method on the object that has the variable name specified with the id attribute:

*<jsp:getProperty name="book1" property="title" /> OR:*
*<%= book1.getTitle() %>*

The first approach is preferable in this case, since the syntax is more accessible to Web page designers who are not familiar with the Java programming language. Direct access to the variable is useful when you are using loops, conditional statements, and methods not represented as properties.

# Setting bean properties

To modify bean properties, you normally use **jsp:setProperty**. This action has several different forms, but with the simplest form you just supply three attributes: name (which should match the id given by jsp:useBean), property (the name of the property to change), and value (the new value). An alternative to using the jsp:setProperty action is to use a scriptlet that explicitly calls methods on the bean object. For example:

```
<jsp:setProperty      name="book1"
                      property="title"
                      value="JSP" />
```

OR:

```
<% book1.setTitle("JSP"); %>
```

# Setting bean properties(2)

Most JSP attribute values have to be fixed strings, but the value and name attributes of jsp:setProperty are permitted to be request-time expressions. Example:

*<jsp:setProperty*
*name="entry"*
*property="itemID"*
***value='<%= request.getParameter("itemID") %>'*** */>*

Note that \’ and \" can be used to represent single or double quotes within an attribute value.

# Associating property with input parmeter

Setting the itemID property was easy since its value is a String.
Setting the numbers is a bit more problematic:

```
<%
        int numItemsOrdered = 1;
        try {
                numItemsOrdered =
                Integer.parseInt(request.getParameter("numItems"));
        } catch(NumberFormatException nfe) { }
%>
```

It is possible to use alternate JSP code:

```
<jsp:setProperty
        name="entry"
        property="numItems"
        value="<%= numItemsOrdered %>" />
```

# Associating property with input parmeter(2)

• JSP automatically performs type conversion from strings to numbers, characters, and boolean values.

• Instead of using the value attribute, you use param to name an input parameter.

• The value of this parameter is automatically used as the value of the property, and simple type conversions are performed automatically.

• If the specified input parameter is missing from the request, no action is taken (the system does not pass null to the associated property).

# Automatic type conversions

Type conversions when properties are associated with input parameters:

| **Property type** | **Conversion routine** |
|---|---|
| boolean | Boolean.valueOf(paramString).booleanValue() |
| Boolean | Boolean.valueOf(paramString) |
| byte | Byte.valueOf(paramString).byteValue() |
| Byte | Byte.valueOf(paramString) |
| char | Character.valueOf(paramString).charValue() |
| Character | Character.valueOf(paramString) |
| double | Double.valueOf(paramString).doubleValue() |
| Double | Double.valueOf(paramString) |

*( int, Integer, float, Float, long, Long by analogy)*

# Associating all properties with input parmeters

**JSP associate *all* properties with identically named input parameters. Example:**

*<jsp:setProperty name="entry" property="*" />*

• No action is taken when an input parameter is missing.

• Automatic type conversion does not guard against illegal values as effectively as does manual type conversion. Consider error pages when using automatic type conversion.

• Since both property names and input parameters are case sensitive, **the property name and input parameter must match exactly.**

# Sharing beans

Until now we  have treated the objects that were created with jsp:use-Bean as local variables in the _jspService method.

**Beans can be  are also stored in one of  four different locations, depending on the value of the optional scope attribute of jsp:useBean.**

# The scope attribute

The scope attribute has the following possible values:

**page**

This is the default value. It indicates that, in addition to being bound to a local variable, the bean object should be placed in the **PageContext** object for the duration of the current request. In principle, storing the object there means that servlet code can access it by calling getAttribute on the predefined pageContext variable.

In practice, beans created with page scope are almost always accessed by jsp:getProperty, jsp:setProperty, scriptlets, or expressions later in the same page.

# The scope attribute(2)

**application**

This very useful value means that, in addition to being bound to a local variable, the bean will be stored in the shared **ServletContext** available through the predefined application variable or by a call to getServletContext().

The ServletContext is shared by all servlets in the same Web application (or all servlets in the same server or servlet engine if no explicit Web applications are defined).

Values in the ServletContext can be retrieved by the getAttribute method.

# The scope attribute(3)

**session**

This value means that, in addition to being bound to a local variable, the bean will be stored in the **HttpSession** object associated with the current request, where it can be retrieved with getValue.

Attempting to use scope="session" causes an error at page translation time when the page directive stipulates that the current page is not participating in sessions.

# The scope attribute(4)

**request**

This value signifies that, in addition to being bound to a local variable, the bean object should be placed in the **ServletRequest** object for the duration of the current request.

# Conditional bean creation

• jsp:useBean element results in a new bean being instantiated
•only if no bean with the same id and scope can be found.

• If a bean with the same id and scope is found, the preexisting bean is simply bound to the variable referenced by id.

• A typecast is performed if the preexisting bean is of a more specific type than the bean being declared, and a *ClassCastException* results if this typecast is illegal.

# Conditional bean creation(2)

Instead of
*<jsp:useBean ... />*
you can use
*<jsp:useBean ...> statements </jsp:useBean>*

• The point of using the second form is that the statements between the jsp:useBean start and end tags are executed **only** if a new bean is created, ***not*** if an existing bean is used.

• This conditional execution is convenient for setting initial bean properties for beans that are shared by multiple pages.
•Since you don't know which page will be accessed first, you don't know which page should contain the initialization code - they can all contain the code, but only the page first accessed actually executes it.

# Conditional bean creation(3)

**Example:**

```
<jsp:useBean          id="counter"
                      class="package.AccessCountBean"
                      scope="application">
                      <jsp:setProperty
                            name="counter"
                            property="firstPage"
                            value="Current Page Name" />
</jsp:useBean>
```

# Custom tags

• In JSP you are able to define your own **tags**.

• You define how the tag, its attributes, and its body are interpreted, then group your tags into collections called **tag libraries** that can be used in any number of JSP files.

• The ability to define tag libraries in this way permits Java developers to boil down complex server-side behaviors into simple and easy-to-use elements that content  developers can easily incorporate into their JSP pages.

# Custom tags(2)

**In order to use custom JSP tags, you need to define three separate components:**

- The **tag handler** class that defines the tag's behavior

- The **tag library descriptor file** that maps the XML element names to the tag implementations

- The JSP file that uses the tag library.

# Tag handler class

• When defining a new tag, your first task is to define a Java class that tells the system what to do when it sees the tag.

• This class must implement the **javax.servlet.jsp.tagext.Tag** interface. This is usually accomplished by extending the TagSupport or BodyTagSupport class (these classes implements Tag interface).

# The basic tag

• Tags that either have no body should extend the **TagSupport** class.
• This is a built-in class in the javax.servlet.jsp.tagext package that implements the **Tag** interface and contains much of the standard functionality basic tags need.
• Because of other classes you will use, your tag should normally import classes in the *javax.servlet.jsp* and *java.io* packages as well:

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
```

# The basic tag(2)

• For a tag without attributes or body you need to override the **doStartTag method**, which defines code that gets called at request time where the element's start tag is found.
• To generate output, the method should obtain the JspWriter (the specialized PrintWriter available in JSP pages through use of the predefined out variable) from the pageContext field by means of **getOut**.
• In addition to the getOut method, the pageContext field (of type PageContext) has methods for obtaining other data structures associated with the request. The most important ones are **getRequest,getResponse, getServletContext,** and **getSession.**
• Since the print method of JspWriter throws IOException, the print statements should be inside a try/catch block.
• If your tag does not have a body, your doStartTag should return the **SKIP_BODY constant**.

# Tag handler class example

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
public class ExampleTag extends TagSupport {
        public int doStartTag() {
                try {
                        JspWriter out = pageContext.getOut();
                        out.print("Custom tag example ");
                } catch(IOException ioe) {
                System.out.println("Error in ExampleTag: " + ioe);
                }
                return(SKIP_BODY);
        }
}
```

# Tag library descriptor file

The general format of a descriptor file should contain an XML version identifier followed by a *DOCTYPE declaration* followed by a *taglib* container element. The important part in the taglib element is the tag element. For tags without attributes, the tag element should contain four elements between ***<tag>* and *</tag>*:**
**name**, whose body defines the base tag name to which the prefix of the taglib directive will be attached. **Example:**

*<name>example</name>*

**tagclass**, which gives the fully qualified class name of the tag handler. **Example:**

*<tagclass>dmcs.tags.ExampleTag </tagclass>*

**info**, which gives a short description. **Example:**

*<info>Outputs a short String</info>*

**bodycontent**, which should have the value EMPTY for tags without bodies. **Example:**

*<bodycontent>EMPTY</bodycontent>*

# Tag library descriptor file(2)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP
Tag Library 1.1//EN" "http://java.sun.com/j2ee/dtds/web-
jsptaglibrary_1_1.dtd">
<!-- a tag library descriptor -->
<taglib>
<tlibversion>1.0</tlibversion>
<jspversion>1.1</jspversion>
<shortname>mytags</shortname>
<info> My tag library </info>
<tag>
<name>example</name>
<tagclass>dmcs.tags.ExampleTag</tagclass>
<info>Outputs a short String</info>
<bodycontent>EMPTY</bodycontent>
</tag></taglib>
```

# The JSP file

Before the first use of your tag, you need to use the taglib directive. This directive has the following form:

*<%@ taglib uri="..." prefix="..." %>*

**Example:**

***...***
*<%@ taglib uri="mytags-taglib.tld" prefix="**mytags**" %>*
***...***
*<BODY>*
*<H1>**<mytags:example />**</H1>*
***...***

# Tags with attributes

Tag with attributes example:

***<prefix:name attribute1="value1" attribute2="value2" ... />***

• Use of an attribute called attribute1 results in a call to a method called setAttribute1 in class that extends TagSupport (or implements the Tag interface).

• The attribute value is supplied to the method as a String. To add support for tag attribute named attribute1 is necessary to implement the following method:

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

**An attribute of attributeName (lowercase a) corresponds to a method called setAttributeName (uppercase A).**

# Tags with attributes(2)

**The Tag Library Descriptor File**

Tag attributes must be declared inside the tag element by means of an attribute element. The attribute element has three nested elements that can appear between <*attribute*> and <*/attribute*>:

**name**, a required element that defines the case-sensitive attribute name.

**required**, a required element that stipulates whether the attribute must always be supplied (true) or is optional (false).

**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.

# Tags with attributes(3)

**The Tag Library Descriptor File example:**

**...**

```
<tag>
        <name>exampleAttr</name>
        <tagclass>dmcs.tags.MyTagAttr</tagclass>
        <info>
                Outputs a short String provided as an argument
        </info>
        <bodycontent>EMPTY</bodycontent>
        <attribute>
                <name>shortString</name>
                <required>false</required>
        </attribute>
</tag>
```

**...**

# Tags with body

**Tag with body example:**

*<prefix:tagname>*
*body*
*</prefix:tagname>*

• To instruct the system to make use of the body that occurs between the new element's start and end tags, your doStartTag method should return **EVAL_BODY_INCLUDE** (instead of SKIP_BODY).

• The body content can contain JSP scripting elements, directives, and actions, just like the rest of the page. The JSP constructs are translated into servlet code at page translation time, and that code is invoked at request time.

# Tag with body example

**Example of fully configured tag replacing HTML <H1> to <H6> elements.**

**Tag handler class:**

```
// all needed imports
public class HeadingTag extends TagSupport {
        private String bgColor; // The one required attribute
        private String color = null;
        private String align="CENTER";
        private String fontSize="36";
        private String fontList="Arial, Helvetica, sans-serif";
        private String border="0";
        private String width=null;
        public void setBgColor(String bgColor) {
                this.bgColor = bgColor;
        } // and so on the rest of set methods...
```

***verte-->***

# Tag with body example(2)

```
public int doStartTag() {
        try {
                JspWriter out = pageContext.getOut();
                out.print("<TABLE BORDER=" + border +
                        " BGCOLOR=\"" + bgColor + "\"" +
                        " ALIGN=\"" + align + "\"");
                if (width != null)
                        out.print(" WIDTH=\"" + width + "\"");
                                out.print("><TR><TH>");
                out.print("<SPAN STYLE=\"" +
                        "font-size: " + fontSize + "px; " +
                        "font-family: " + fontList + "; ");
                if (color != null) out.println("color: " + color + ";");
                out.print("\"> "); // End of <SPAN ...>
        } catch(IOException ioe) {//do Sth }
        return(EVAL_BODY_INCLUDE);   // Include tag body
}
```

**verte -->**

# Tag with body example(3)

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

# Tag with body example(4)

**Tag library descriptor file:**

**...**

*<tag>*

*<name>heading</name>*

*<tagclass>dmcs.tags.HeadingTag</tagclass>*

*<info>Outputs a 1-cell table used as a heading.</info>*

***<bodycontent>***

***JSP <!-- instead of EMPTY -->***

***</bodycontent>***

*<attribute>*

*<name>bgColor</name>*

*<required>true</required> <!--bgColor is required-->*

*</attribute>*

***...** <!-- the rest of attributes -->*

*</tag>*

# Tag with body example(5)

**The JSP file:**

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

# Optionally including tag body

- **It is possible to use request time information to decide whether or not to include the tag body.**

- Although the body can contain JSP that is interpreted at page translation time, the result of that translation is servlet code that can be invoked or ignored at request time.

- **To include body optionally return EVAL_BODY_INCLUDE or SKIP_BODY depending on the value of some request time expression.**

- To read a request you should use getRequest method to obtain the ServletRequest from the automatically defined pageContext field of TagSupport. Than you can cast it to HttpServletRequest in order to use more specialized methods.

# Optionally ... - example

**DebugTag.java :**

```
public class DebugTag extends TagSupport {

public int doStartTag() {
        ServletRequest request =
        pageContext.getRequest();
        String debugFlag = request.getParameter("debug");
        if ((debugFlag != null) &&
                (!debugFlag.equalsIgnoreCase("false"))) {
                return(EVAL_BODY_INCLUDE);
        } else {
                return(SKIP_BODY);
        }
}}
```

# Optionally ... - example(2)

**Tag library descriptor file:**

```
...
<tag>
        <name>debug</name>
        <tagclass>dmcs.tags.DebugTag</tagclass>
        <info>Includes body only if debug param is set.</info>
        <bodycontent>JSP</bodycontent>
</tag>
...
```

# Optionally ... - example(3)

**JSP file:**

```
...
<%@ taglib uri="mytags-taglib.tld" prefix="mytags" %>
...
<mytags:debug>
<B>Debug:</B>
<UL>
<LI>Current time: <%= new java.util.Date() %>
<LI>Requesting hostname:<%= request.getRemoteHost() %>
<LI>Session ID: <%= session.getId() %>
</UL>
</mytags:debug>
...
```

# Processing the tag body

To process tag body you should subclass **BodyTagSupport.** The two important new methods defined by BodyTagSupport are:

• **doAfterBody**, a method that you should override to handle the manipulation of the tag body. This method should normally return SKIP_BODY when it is done, indicating that no further body processing should be performed.

• **getBodyContent**, a method that returns an object of type BodyContent that encapsulates information about the tag body.

# Processing the tag body(2)

The **BodyContent** class has three important methods:

•**getEnclosingWriter**, a method that returns the JspWriter being used by doStartTag and doEndTag.

•**getReader**, a method that returns a Reader that can read the tag's body.

•**getString**, a method that returns a String containing the entire tag body.

# Processing ... - example

```
public class FilterTag extends BodyTagSupport {
        public int doAfterBody() {
                BodyContent body = getBodyContent();
                String filteredBody =
                ServletUtilities.filter(body.getString());
                try {
                        JspWriter out = body.getEnclosingWriter();
                        out.print(filteredBody);
                } catch(IOException ioe) {
                        System.out.println("Error in FilterTag: " + ioe);
                }
// SKIP_BODY means that all is done. If you wanted to evaluate
// and handle the body again, you'd return EVAL_BODY_TAG.
                return(SKIP_BODY);
        }
}
```

# Processing ... - example(2)

**Tag library descriptor file:**

```
<tag>
        <name>filter</name>
        <tagclass>coreservlets.tags.FilterTag</tagclass>
        <info>Replaces HTML-specific characters in body.</info>
        <bodycontent>JSP</bodycontent>
</tag>
```

**JSP file:**

```
<%@ taglib uri="mytags-taglib.tld" prefix="mytags" %>
...
<mytags:filter>
< > % &
</mytags:filter>
```

# Processing tag body multiple times

Tags that process the body content multiple times should start by extending BodyTagSupport and implementing doStartTag, doEndTag, and **doAfterBody**.

**If doAfterBody returns  EVAL_BODY_TAG, the tag body is evaluated again, resulting in a new call to doAfterBody. This process continues until doAfterBody returns SKIP_BODY.**

# Processing ... - example

**A tag that repeats the body the specified number of times:**

```
public class RepeatTag extends BodyTagSupport {
        private int reps;
        public void setReps(String repeats) {
                try {
                        reps = Integer.parseInt(repeats);
                } catch(NumberFormatException nfe) {
                reps = 1;
                }
        }
```

***verte-->***

# Processing ... - example(2)

```
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);
        }
}}
```

# Processing ... - example(3)

**Tag library descriptor file:**

```
<tag>
        <name>repeat</name>
        <tagclass>dmcs.tags.RepeatTag</tagclass>
        <info>Repeats body the specified number of times.</info>
        <bodycontent>JSP</bodycontent>
        <attribute>
                <name>reps</name>
                <required>true</required>
                <!-- rtexprvalue indicates whether attribute
                can be a JSP expression. -->
                <rtexprvalue>true</rtexprvalue>
        </attribute>
</tag>
```

# Processing ... - example(4)

**JSP file:**

```
...
<%@ taglib uri="mytags-taglib.tld" prefix="mytags" %>
...
<OL>
<!-- Repeats N times. A null reps value means repeat once. -->
<mytags:repeat reps='<%= request.getParameter("repeats") %>'>
<LI>List item
</mytags:repeat>
</OL>
...
```

# Nested tags

• Class definitions for nested tags can extend either **TagSupport** or **BodyTagSupport** (as necessary).

• Nested tags can use **findAncestorWithClass** to find the tag in which they are nested. This method takes a reference to the current class (e.g., ***this***) and the Class object of the enclosing class (e.g., EnclosingTag.class) as arguments. If no enclosing class is found, the method in the nested class can throw a **JspTagException** that reports the problem.

• If one tag wants to store data that a later tag will use, it can place that data in the instance of the enclosing tag. The definition of the enclosing tag should provide methods for storing and accessing this data.

# Template for nested tags

```
public class OuterTag extends TagSupport {
        public void setSomeValue(SomeClass arg) { ... }
        public SomeClass getSomeValue() { ... }
}
public class FirstInnerTag extends BodyTagSupport {
        public int doStartTag() throws JspTagException {
        OuterTag parent =
        (OuterTag)findAncestorWithClass(this, OuterTag.class);
        if (parent == null) {
                throw new JspTagException("nesting error");
        } else {
                parent.setSomeValue(...);
        }
        return(EVAL_BODY_TAG);
        } }
```

# Nested tags - example

**Lets make following structure:**

*<mytags:if>*
*<mytags:condition><%= someExpression %></mytags:condition>*
*<mytags:then>JSP to include if condition is true</mytags:then>*
*<mytags:else>JSP to include if condition is false</mytags:else>*
*</mytags:if>*

# Nested tags - example(2)

```
public class IfTag extends TagSupport {
        private boolean condition;
        private boolean hasCondition = false;
        public void setCondition(boolean condition) {
                this.condition = condition;
                hasCondition = true;          }
        public boolean getCondition() {
                return(condition);     }
        public void setHasCondition(boolean flag) {
                this.hasCondition = flag;     }
        /** Has the condition field been explicitly set? */
        public boolean hasCondition() {
                return(hasCondition);         }
        public int doStartTag() {
        return(EVAL_BODY_INCLUDE);  }
}
```

# Nested tags - example(3)

```
public class IfConditionTag extends BodyTagSupport {
        public int doStartTag() throws JspTagException {
                IfTag parent =
                (IfTag)findAncestorWithClass(this, IfTag.class);
                if (parent == null)
                throw new JspTagException("cond. not inside if");
                return(EVAL_BODY_TAG);          }
        public int doAfterBody() {
                IfTag parent =
                (IfTag)findAncestorWithClass(this, IfTag.class);
                String bodyString = getBodyContent().getString();
                if (bodyString.trim().equals("true")) {
                        parent.setCondition(true);
                } else { parent.setCondition(false); }
                return(SKIP_BODY);          }
}
```

# Nested tags - example(4)

```
public class IfThenTag extends BodyTagSupport {
        public int doStartTag() throws JspTagException {
        IfTag parent =
        (IfTag)findAncestorWithClass(this, IfTag.class);
        if (parent == null) {
                throw new JspTagException("then not inside if");
        } else if (!parent.hasCondition()) {
                String warning =
                "condition tag must come before then tag";
                throw new JspTagException(warning);
        }
        return(EVAL_BODY_TAG);
        }
```

***verte-->***

# Nested tags - example(5)

```
public int doAfterBody() {
        IfTag parent =
        (IfTag)findAncestorWithClass(this, IfTag.class);
        if (parent.getCondition()) {
                try {
                        BodyContent body = getBodyContent();
                        JspWriter out = body.getEnclosingWriter();
                        out.print(body.getString());
                } catch(IOException ioe) {
                System.out.println("Error in IfThenTag: " + ioe);
                }
        }
        return(SKIP_BODY);
} }
```

# Nested tags - example(6)

```
public class IfElseTag extends BodyTagSupport {
        public int doStartTag() throws JspTagException {
        IfTag parent =
        (IfTag)findAncestorWithClass(this, IfTag.class);
        if (parent == null) {
                throw new JspTagException("else not inside if");
        } else if (!parent.hasCondition()) {
                String warning =
                "condition tag must come before else tag";
                throw new JspTagException(warning);
        }
        return(EVAL_BODY_TAG);
        }
```

***verte-->***

# Nested tags - example(7)

```
public int doAfterBody() {
        IfTag parent =
        (IfTag)findAncestorWithClass(this, IfTag.class);
        if (!parent.getCondition()) {
                try {
                        BodyContent body = getBodyContent();
                        JspWriter out = body.getEnclosingWriter();
                        out.print(body.getString());
                } catch(IOException ioe) {
                System.out.println("Error in IfElseTag: " + ioe);
                }
        }
        return(SKIP_BODY);
}}
```

# Nested tags - example(8)

**Tag library descriptor file:**

```
<tag>
        <name>if</name>
        <tagclass>dmcs.tags.IfTag</tagclass>
        <info>if/condition/then/else tag.</info>
        <bodycontent>JSP</bodycontent>
</tag>
<tag>
        <name>condition</name>
        <tagclass>dmcs.tags.IfConditionTag</tagclass>
        <info>condition part of if/condition/then/else tag.</info>
        <bodycontent>JSP</bodycontent>
</tag>
<!-- then and else tags by analogy -->
```

# Nested tags - example(9)

**JSP file:**

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

<mytags:if>
<mytags:condition>
        <%= request.isSecure() %>
</mytags:condition>
<mytags:then>Request is using SSL (https)</mytags:then>
<mytags:else>Request is not using SSL</mytags:else>
</mytags:if>
```