Introduction to JSP technology

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Tutorial tips

Should I take this tutorial?

This tutorial introduces the fundamentals of JavaServer Pages (JSP) technology. The goals are to give you a solid grasp of the basics and enable you to start writing your own JSP solutions. In particular, this tutorial will:

- Discuss the fundamental elements that define JSP technology: concepts, syntax, and semantics
- Identify and exemplify each element
- Use short, specific, topical examples to illustrate each element and clearly illuminate important issues related to that element

This is an introductory tutorial on JavaServer Pages technology and is intended for new or novice JSP programmers. The emphasis is on simple explanations rather than exhaustive coverage of every single option. If you're an expert on JSP technology, this tutorial is probably not for you.

One aspect of JSP technology that this tutorial does not address is programming style. This tutorial is strictly an overview of the JSP page syntax and semantics.

Although this tutorial introduces JSP technology from the beginning, you'll get the most out of it if you have some understanding of HTML and Java. Finally, in order to author and test your own JSP solutions, you will need a suitable environment, such as *IBM WebSphere* or *Jakarta*.

Getting help

For technical questions about the content of this tutorial, contact the author, Noel J. Bergman, at *noel@devtech.com*.

Noel's background in object-oriented programming spans more than 20 years, including participation on the original CORBA and Common Object Services Task Forces. He has consistently received high marks as a favored speaker at the *Colorado Software Summit* as well as other industry conferences, and is in demand as a mentor, providing customized consulting and mentoring services based upon each client's specific problem domain.

At present, Noel is involved in developing interactive database-backed Web sites using open source freeware. To that end, Noel is also a co-author of GNUJSP, an open source implementation of JSP, and originator of the *JSP Developer's Guide Web site*.

Section 2. JSP technology overview

Background

JSP technology is one of the most powerful, easy to use, and fundamental tools in a Web site developer's toolbox. JSP technology combines HTML and XML with Java servlet (server application extension) and JavaBeans technologies to create a highly productive environment for developing and deploying reliable, interactive, high-performance, platform-independent Web sites.

JSP technology facilitates creation of dynamic content on the server. It is part of the Java platform's integrated solution for server-side programming, which provides a portable alternative to other server-side technologies, such as CGI. JSP technology integrates numerous Java application technologies, such as Java servlet, JavaBeans, JDBC, and Enterprise JavaBeans. It also separates information presentation from application logic and fosters a reusable-component model of programming.

A common question is when to use JSP technology versus other Java server-side technologies. Unlike on the client side, which has numerous technologies competing for dominance (including HTML and applets, Document Object Model [DOM]-based strategies such as Weblets and Doclets, and more), the server side has relatively little overlap among the various Java server-side technologies and very clean models of interaction.

Who uses JSP technology? Are there any real-world, mission-critical deployments of JSP technology on the Internet? *Gist*, *Delta Air Lines*, *The Sharper Image*, *Tire Rack*, and many others rely upon JSP technology, with major new sites appearing on a frequent basis. Within the *IBM WebSphere product line*, the preferred technology and programming model uses JSP pages (although WebSphere supports other server-side programming technologies as well). The *IBM WebSphere Studio* product includes wizards to help build JSP pages. As allowed by the JSP specification, WebSphere supports multiple scripting languages for JSP pages, and IBM's implementation is being migrated into Jakarta.

What is JSP technology?

What exactly is JSP technology? Let's consider the answer to that from two different perspectives: that of an HTML designer and that of a Java programmer.

If you are an HTML designer, you can look at JSP technology as extending HTML to provide you with the ability to seamlessly embed snippets of Java code within your HTML pages. These bits of Java code generate dynamic content, which is embedded within the other HTML/XML content you author. Even better, JSP technology provides the means by which programmers can create new HTML/XML tags and JavaBeans components, which provide new features for HTML designers without those designers needing to learn how to program.

Note: A common misconception is that Java code embedded in a JSP page is transmitted with the HTML and executed by the user agent (such as a browser). This is not the case. A JSP page is translated into a Java servlet and executed on the server. JSP statements embedded in the JSP page become part of the servlet generated from the JSP page. The resulting servlet is executed on the server. It is never visible to the user agent.

If you are a Java programmer, you can look at JSP technology as a new, higher-level means to writing servlets. Instead of directly writing servlet classes and then emitting HTML from your servlets, you write HTML pages with Java code embedded in them. The JSP environment takes your page and dynamically compiles it. Whenever a user agent requests that page from the Web server, the servlet that was generated from your JSP code is executed, and the results are returned to the user.

A simple JSP page

Let's move this discussion from the abstract to the concrete by looking at a couple of very simple JSP pages. The first example is a JSP version of Hello World.

HelloWorld.jsp (*live demo*)

```
<HTML>
<HEAD><TITLE>Hello World JSP Example</TITLE></HEAD>
<BODY>
Hello World.
</BODY>
</HTML>
```

This is just normal HTML. JSP syntax extends HTML; it does not replace it. Whatever static content you have in your page is passed, unchanged, from the server to the client. In this case, we do not have any dynamic content, so our static content is passed, unchanged, to the browser. The difference between treating this as a JSP page and treating it as a normal HTML page is that a normal HTML page is just transmitted from the Web server to the client, whereas a JSP page is translated into a servlet, and when that servlet is executed, the response from the servlet contains the HTML. The content seen by the user is identical; only the mechanism used on the server is different.

A dynamic JSP page

Next, let's add some dynamic content to our simple example. In addition to displaying "Hello World," our example page will also show the current time. The revised JSP page looks like this, with new additions highlighted:

HelloWorld2.jsp (*live demo*)

```
<HTML>
<HEAD><TITLE>Hello World JSP Example w/Current Time</TITLE></HEAD>
<BODY>
Hello World. The local server time is <%= new java.util.Date() %>.
</BODY>
</HTML>
```

The odd-looking bit of text, `<%= new java.util.Date() %>`, is a JSP expression. We'll explain expressions shortly. For now, just understand that when output is prepared for the client, the server's current time is acquired, automatically converted to a `String` object, and embedded in-place.

Note: It is the server's time, not the client's time, that is displayed. Unlike a JavaScript

element, which is executed on the client computer, JSP code executes on the server, exists in the context of the server, and is completely transparent to the client.

An alternative JSP syntax

Another syntax is defined for JSP elements, using XML tags instead of <% tags. The same example, written using XML tags, looks like this:

HelloWorld3.jsp (*live demo*)

```
<HTML>
<HEAD><TITLE>Hello World JSP Example w/ Current Time</TITLE></HEAD>
<BODY>
Hello World. The local server time is
<jsp:expression> new java.util.Date() </jsp:expression>.
</BODY>
</HTML>
```

The XML syntax is less compact, but this time it is clear that even if you don't know what a particular JSP expression is, you are looking at one. These two different ways of writing expressions are synonymous. The meaning and resulting servlet code are identical.

JSP pages are servlets

Technically, a JSP page is compiled into a *Java servlet*. If you are not familiar with the Java servlet technology, you may wish to review the *servlets tutorial*. Although it would be possible for a JSP implementation to generate bytecode files directly from the JSP source code, typically each JSP page is first translated into the Java source code for a servlet class, and then that servlet is compiled. The resulting servlet is invoked to handle all requests for that page. Typically, the page translation step is performed the first time a given JSP page is requested, and then only when that page's source code changes thereafter. Otherwise, the resulting servlet is simply executed, providing very quick delivery of content to the user.

The JSP specification defines the JSP language and defines the JSP run-time environment, but it does not define the translation environment. In other words, it defines what your JSP source file looks like and it defines the run-time environment, including classes and interfaces, for the generated servlet, but it does not define *how* the JSP source is turned into the servlet, nor does it enforce how that servlet must be deployed.

Page translation

The intimate details of page translation are largely up to the authors of any particular implementation of JSP technology. Normally, this is of limited consequence; all implementations of JSP technology must conform to the same standard so that your pages are portable. Occasionally, however, there may be an operational difference that is outside the specification.

GNUJSP, an open source implementation of JSP, takes a fairly straightforward approach. When you request a JSP page, GNUJSP receives a request from the Web server. GNUJSP looks in the file system to find the JSP page. If it finds that page, it checks to see if it needs to

be recompiled: if the page has been changed since the last time it was compiled or if the compiled servlet is not found in the cache maintained by GNUJSP, GNUJSP translates the JSP source code into Java source code for a servlet class and compiles the resulting source code into a binary servlet. GNUJSP then executes the generated servlet. This means that you can rapidly evolve your JSP pages; GNUJSP will automatically recompile them when, and only when, necessary.

The *Jakarta* project contains the reference implementation of both Java servlet and JSP technologies. Taking a different approach from GNUJSP, Jakarta encourages the deployment of Web applications in a kind of JAR file known as a Web App Repository (WAR) file, which is part of version 2.2 and later revisions of the Java servlet specification.

Within the WAR file are all of the resources that make up the Web application. Deployment descriptors within the WAR file are used to map from Uniform Resource Identifiers (URIs) to resources. This is a more complex environment to configure and maintain, but it has numerous benefits for large-scale commercial deployment, because you can deploy a fully configured, precompiled (sourceless) Web application as a single, binary archive.

The `JspPage` interface

You may find it useful to know that each generated page is a servlet class that supports the `JspPage` interface (technically, the class supports a protocol-dependent descendent, such as `HttpJspPage`). `JspPage` extends `Servlet` and is the essential contract between the JSP page and the JSP container.

A full discussion of the `JspPage` interface is not necessary here. The essential information is that the primary method, which handles requests, is `_jspService`. The `_jspService` method is generated during page translation. Any expressions, scriptlets, and actions you write in your JSP page affect the generated implementation of the page's `_jspService` method. Any declarations affect the definition of the generated servlet class.

What's next?

Having seen that authoring a JSP page is little different from authoring standard HTML, let's look more closely at the syntax for JSP pages. The remainder of this tutorial provides a brief overview of the entirety of the JSP syntax and semantics.

Note: Technically, the JSP specification permits languages other than Java to be used as a JSP page's scripting language. At present, the scripting language must be Java; however, you should understand that wherever we talk about embedding Java code within a JSP page, in the future you may be able to use other languages.

Section 3. JSP syntax

JSP syntactic elements

The following table shows the four broad categories of core syntax elements defined by the JSP specification.

Type of element	Element content
Template content on page 9	Everything in your JSP page's source file that is not a JSP element. Includes all static content.
Directives on page 13	Instructions you place in your JSP page to tell the JSP implementation how to go about building your page, such as to include another file.
Scripting elements	Declarations on page 17, Expressions on page 22 , and Scriptlets on page 23 , which are used to embed Java code into your JSP pages.
Actions on page 26	Actions provide high-level functionality, in the form of custom XML-style tags, to a JSP page without exposing the scripting language. Standard actions include those to create, modify, and otherwise use JavaBeans within your JSP page.

JSP element syntax

Each of the different JSP elements is written in a different way. The following table roughly indicates the syntax for each kind of JSP element. This is just an outline so that you get the overall flavor of JSP syntax. We will cover the formal specification of JSP syntax in more detail later in this tutorial. Remember that there are two different ways to write most elements: using the <% syntax and using the XML syntax.

Element	<% Syntax	XML Syntax
Output comments on page 9	<!-- *visible comment* -->	<!-- *visible comment* -->
Hidden comments on page 10	<%-- *hidden comment* --%>	<%-- *hidden comment* --%>
Declarations on page 17	<%! *Java declarations* %>	<jsp:declaration> *Java Language declarations* </jsp:declaration>
Expressions on page 22	<%= *A Java expression* %>	<jsp:expression> *A Java expression* </jsp:expression>
Scriptlets on page 23	<% *Java statements* %>	<jsp:scriptlet> *Java statements*

23		</jsp:scriptlet>
Directives on page 13	<%@ ... %>	<jsp:directive.*type* ... />
Actions on page 26		<jsp:*action* ... /> or <jsp:*action*> ... </jsp:*action*>

Section 4. Template content

What is template content?

Template content is not, strictly speaking, a specific element. Rather, template content is everything in your JSP page that is not an actual JSP element. Recall our first HelloWorld.jsp example (A simple JSP page on page 4); *everything* on that page was template content. The JSP specification says that everything that is not an actual JSP element is template content and is to be passed, unchanged, into the output stream. For example, consider the following variation of our Hello World example:

HelloWorld4.jsp (*live demo*)

```
<jsp:directive.page import="java.util.Date"/>
<HTML>
<HEAD><TITLE>Hello World JSP Example w/Current Time</TITLE></HEAD>
<BODY>
Hello World. The local server time is <%= new Date() %>.
</BODY>
</HTML>
```

This is the same as an earlier example, except that we have included a JSP directive to import the `java.util.Date` class, so that we don't have to fully qualify the package name when we create a `Date` object. You might assume, therefore, that since we haven't changed the template content and have only specified an import statement, the output from this JSP page is the same as we received previously. However, that assumption would be incorrect.

The fact is that we did change the template content: the carriage return following the `page` directive is new, and becomes part of the template content. You can see this for yourself by viewing the HTML source for the live demo.

Remember: Everything that is not part of a JSP element is template content.

Comments

There are three ways to place comments in a JSP page:

- Write an HTML comment.
- Write a JSP comment.
- Embed a comment within a scripting element.

Output comments

The first way is to write a standard HTML comment. They follow this format:

```
<!- I am an HTML comment -->
```

This comment is just part of the template content. Nothing fancy; it is just plain old HTML. It is sometimes referred to as a *visible comment*, because the JSP container treats it just like

any other piece of template content and passes it unchanged into the output stream. The fact that it is a comment is really an issue for the user agent (for example, your browser). In point of fact, there are many HTML comments that are not really comments at all. For example, the HTML standard recommends that JavaScript statements be embedded within a comment, to maintain compatibility with pre-JavaScript browsers. Browsers that support JavaScript process those comments as normal statements, whereas browsers that do not support JavaScript ignore the comments.

Of course, because an HTML comment is just another piece of template content to the JSP container, you can embed JSP expressions within it. For example, you could write the following:

HelloWorld5.jsp (*live demo*)

```
<!-- The time is <%= new java.util.Date() %> -->
<HTML>
<HEAD><TITLE>Hello World JSP Example</TITLE></HEAD>
<BODY>
Hello World.
</BODY>
</HTML>
```

This results in the date being embedded within the HTML comment. Nothing would be apparent in a browser view, but you can see the comment by viewing the HTML source for the live demo.

Hidden comments

The second way to write a comment is to use the JSP notation:

```
<%-- I am a comment --%>
```

This comment is referred to as a *hidden comment*, because it is an actual JSP comment, and does not appear in the generated servlet. The user will never see the comment, nor have any way to know of its existence. Repeating our Hello World example, but this time with a JSP comment would yield the following:

HelloWorld6.jsp (*live demo*)

```
<%-- This is our oft-repeated Hello World example: --%>
<HTML>
<HEAD><TITLE>Hello World JSP Example</TITLE></HEAD>
<BODY>
Hello World.
</BODY>
</HTML>
```

Again, remember that everything that is not a JSP element is template content. The carriage return following the comment is not part of the JSP comment element; it is part of the template content, and is therefore visible in the browser.

Scripting language comments

The third way to write a comment is to embed a language-specific comment within a scripting element. For example, you could write the following:

HelloWorld7.jsp (*live demo*)

```
<jsp:directive.page import="java.util.*"/>
<jsp:declaration>
/* date is a new member variable, initialized when we are instantiated.
It can be used however we want, such as:
    out.println("Instantiated at " + date); // display our "birth" date
*/
Date date = new Date();
</jsp:declaration>
<HTML>
<HEAD><TITLE>Hello World JSP Example</TITLE></HEAD>
<BODY>
Hello World.<BR>
This page was loaded into memory at <%= date %>.<BR>
The current time is <%= new Date() %>
</BODY>
</HTML>
```

This version has a comment embedded within a scripting element, the declaration. In fact, there is also a nested comment: the `//` comment within the `/* ... */` block comment. The only type of nested comment allowed by the JSP specification is a scripting language comment, and even then only when the scripting language (for example, Java) permits nested comments.

Quoting and escaping rules

If <% is a magic token that the JSP container understands as the beginning of a JSP element, how do we put a <% into the output stream? For example, if we want to use JSP technology to write a tutorial on the subject of JSP technology, how do we write a page so that its output contains the source for a JSP page? Won't those JSP elements be processed as well?

The solution to this problem is called *escaping*, and entails encoding the content in a modified form. The following table illustrates the set of escaping rules that are part of the JSP specification. It indicates that when we want a <% to be passed along literally instead of being processed, we write it as `<\%` in the template content.

Within this element type ...	If we want to write ...	We escape the text as ...
Template content	<%	<\%
Scripting element	%>	%\>
Element attribute	'	\'
Element attribute	"	\"
Element attribute	\	\\
Element attribute	<%	<\%

Element attribute	%>	%\>

Using escape rules

The following example illustrates some uses of these escaping sequences and illustrates an important point regarding HTML.

HelloWorld8.jsp (*live demo*)

```
<HTML>
<HEAD><TITLE>Hello World JSP Escape Sequences</TITLE></HEAD>
<!--
This page was run at <%= new java.util.Date() %>, according
to <\%=new Date() %\>
-->
<BODY>
<P>Hello World.<BR>
The local server time is <%= new java.util.Date() %>.<BR>
The time is computed using the JSP expression
<I>&lt;%=new Date()%&gt;.</I></P>
<P>The escaping sequence in the comment uses the JSP defined
escape sequence, but that won't work within the normal HTML
content. Why not? Because HTML also has escaping rules.</P>
<P>The most important HTML escaping rules are that '&lt;' is
encoded as <I>"&amp;lt;"</I>, '&gt;' is encoded as
<I>"&amp;gt;"</I>, and '&amp;' is encoded as <I>"&amp;amp;"</I>.
</P>
</BODY>
</HTML>
```

HTML "entities"

As illustrated in the previous example, you need to be aware not just of the rules for JSP pages, but also of the rules for the template content. In this case, we are writing HTML content, and HTML also has escaping rules. These rules say that you cannot write <\% within HTML, except within a comment. Why not? Because HTML treats the < as a special character; it marks the beginning of an HTML tag. Accordingly, HTML requires that you encode certain special characters, such as those listed in the following table.

Desired Character	HTML Escape Sequence
<	<
>	>
&	&

There are many other useful escape sequences for HTML. Part of the HTML 4.01 specification includes a complete list of the official HTML escape sequences, known in the HTML standard as *character entity references*.

Section 5. Directives

JSP directives

Directives provide additional information to the JSP container and describe attributes for your page. For example, directives are used to import Java packages for use within your page, to include files, and to access libraries of custom tags.

Directive	Purpose
page	Controls properties of the JSP page
include	Includes the contents of a file into the JSP page at translation time
taglib	Makes a custom tag library available within the including page

The general syntax for a directive is:

```
<%@ directive [...]%>
```

or

```
<jsp:directive.directive [...] />
```

For clarity, I always use the XML syntax for directives.

The page directive

The page directive provides instructions to the JSP container regarding how you want it to build your JSP page. The directive provides a means for setting page attributes and is written as:

```
<%@ page [attribute="value"*] %>
```

or

```
<jsp:directive.page [attribute="value"*] />
```

You will most likely use the import attribute, as we saw in an earlier example:

```
<jsp:directive.page import="java.util.Date"/>
<HTML>
<HEAD><TITLE>Hello World JSP Example w/Current Time</TITLE></HEAD>
<BODY>
Hello World.
The local server time is <%= new Date() %>.
</BODY>
</HTML>
```

This causes java.util.Date to be imported (as in the Java statement `import java.util.Date`) into your JSP page.

page directive attributes

Other page attributes control page buffering, exception handling, the scripting language, and so forth.

Attribute	Value	Default
language	java	java
extends	superclass	Implementation-dependent. This tells the JSP container to use the specified class as the superclass for the generated servlet. **Use with caution**. Each implementation provides its own default class; overriding the default can impact the quality of the implementation.
import	java-import-list	`java.lang.*, javax.servlet.http.*, javax.servlet.*, javax.servlet.jsp.*` Add any additional packages or types that you need imported for your page.
session	true | false	true
buffer	Kilobytes or none	8 KB or more
autoFlush	true | false	true
isThreadSafe	true | false	true
info	text	None
errorPage	URL for errorPage	None
isErrorPage	true | false	false
contentType	MIME type;encoding	text/html

Basically, the `page` directive is a catch-all for any information about a JSP page that you might need to describe to the JSP container. The good news is that the default values for all of the `page` directive attributes are quite reasonable, and you will rarely need to change them.

The `include` directive

The `include` directive includes the content of the named file directly into your JSP page's source, as it is compiled. The two ways to write a JSP `include` directive are:

```
<%@ include file="filename" %>
```

or

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

The `include` directive differs from the `jsp:include` action. The directive includes the content of a file at translation time, analogous to a C/C++ `#include`, whereas the action includes the output of a page into the output stream at request time.

The `include` directive is very useful for including reusable content into your JSP page, such as common footer and header elements. For example, a common footer element on our JSP pages is a last-changed indicator, so that people know when the page was last updated. We implement that within an included file. Another page, which we often include, forces a user to log in before they have access to the content of a page.

`include` directive example

Let's consider a very simple use of the `include` directive. So far, none of our examples has included a copyright notice. Assume that `/copyright.html` contains a common copyright notice for inclusion in our sample pages. We can include `/copyright.html` as a header within our JSP pages. This looks like the following:

HelloWorld9.jsp (*live demo*)

```
<jsp:directive.include file="/copyright.html" />
<HTML>
<HEAD><TITLE>Hello World JSP Example</TITLE></HEAD>
<BODY>
Hello World.
</BODY>
</HTML>
```

The included file simply becomes part of our JSP page when the page is translated, which you can see by viewing the HTML source from the live demo.

The `taglib` directive

The ability to create and use custom tag libraries is one of the most powerful and useful features provided by JSP technology. This feature enables developers to define new XML-style tags for use by page designers; the JSP specification refers to these new tags as new actions. These custom tags allow nonprogrammers to use entirely new programmed capabilities without requiring such page designers to understand the Java programming language or any other scripting language; all of the details can be encapsulated within the custom tags. In addition, JSP pages that use custom tags can have a much cleaner separation of interface design and business logic implementation than can JSP pages that have a lot of embedded Java code (scriptlets and/or expressions).

For example, thus far we have made frequent use of the Java `Date` class to display the current time. Instead, we could define a custom DATE tag, which might look like this:

```
<jsp:directive.taglib uri="jdg.tld" prefix="jdg" />
<HTML>
<HEAD><TITLE>Hello World JSP Example w/Current Time</TITLE></HEAD>
<BODY>
```

```
Hello World.
The local server time is <jdg:DATE />.
</BODY>
</HTML>
```

The first line is a `taglib` directive. The `uri` attribute tells the JSP container where to find the `taglib` definition. The `prefix` attribute tells the JSP container that we will use `jdg:` as a prefix for the tag; this prefix is mandatory and prevents namespace collisions. The `jsp` prefix is reserved by Sun.

The tag `<jdg:DATE />` results in a custom tag handler class being used, where we can put the current date and time into the output stream. The page designer doesn't have to know anything about Java code; instead, the designer simply uses the custom DATE tag we have documented.

The process of creating a custom tag library is outside the scope of this tutorial.

Section 6. Declarations

JSP declarations

Declarations declare new data and function members for use within the JSP page. These declarations become part of the resulting servlet class generated during page translation. You can write a JSP declaration in two ways, as follows:

```
<%! java declarations %>
```

or

```
<jsp:declaration> java declarations </jsp:declaration>
```

For clarity, I never use the <% syntax for declarations; the XML syntax is self-descriptive and far more clear to the reader than remembering which JSP element uses an exclamation point.

JSP declaration example

Here is a version of Hello World that uses a JSP declaration to declare a class variable and some class functions. They are declared as static because they are not related to any specific instance of the JSP page.

HelloWorld10.jsp (*live demo*)

```
<jsp:directive.page import="java.util.Date"/>
<jsp:declaration>
private static String loadTime = new Date().toString();
private static String getLoadTime() { return loadTime; }
private static String getCurrentTime() { return new Date().toString(); }
</jsp:declaration>
<HTML>
<HEAD><TITLE>Hello World JSP Example</TITLE></HEAD>
<BODY>
Hello World.<BR>
This page was loaded into memory at <%= getLoadTime() %>.<BR>
The current time is <%= getCurrentTime() %>
</BODY>
</HTML>
```

The first highlighted section is a JSP declaration. Recall that each JSP page is compiled into a Java servlet class. JSP declarations allow us to declare new members for that class. In the example, we declare a new class variable, `loadTime`, and two class functions, `getLoadTime()` and `getCurrentTime()`. Then, in the body of our JSP page, we refer to our new functions.

JSP page initialization and termination

JSP containers provide a means for pages to initialize their state. A page may declare optional initialization and cleanup routines. The `jspInit` method will be called before the

page is asked to service any requests. The JSP container will call the `jspDestroy` method if it needs the page to release resources. The following are the signatures for these methods:

```
public void jspInit();
public void jspDestroy();
```

We place these optional methods within a JSP declaration just as we would any other methods.

`jspInit()` and `jspDestroy()` example

This example is identical to the JSP declaration example on page 17 , except that instead of presenting time in local server time, we translate time into GMT using a `SimpleDateFormat` object. Because there is no need to construct this object more than once, we initialize it in `jspInit()` and dereference it in `jspDestroy()`.

HelloWorld11.jsp (*live demo*)

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

<jsp:declaration>
private static DateFormat formatter;
private static String loadTime;
private static String getLoadTime() { return loadTime; }
private static String getCurrentTime() { return toGMTString(new Date()); }

private static String toGMTString(Date date)
{
    return formatter.format(date);
}

public void jspInit()
{
    formatter = new SimpleDateFormat("d MMM yyyy HH:mm:ss 'GMT'", Locale.US);
    formatter.setTimeZone(TimeZone.getTimeZone("GMT"));
    loadTime = toGMTString(new Date());
}

public void jspDestroy()
{
    formatter = null;
}

</jsp:declaration>
<HTML>
<HEAD><TITLE>Hello World JSP Example</TITLE></HEAD>
<BODY>
Hello World.<BR>
This page was initialized at <%= getLoadTime() %>.<BR>
The current time is <%= getCurrentTime() %>
</BODY>
</HTML>
```

Section 7. Implicit objects

Implicit objects

There are a number of objects predefined by JSP architecture. They provide access to the run-time environment. The implicit objects are local to the generated `_jspService` method (see The JspPage interface on page 6). Scriptlets and expressions, which effect the `_jspService` method, have access to the implicit objects, but declarations, which effect the generated class, do not.

You can write a lot of JSP pages and never have a need to refer to the implicit objects directly. Most of the implicit objects are standard Servlet API components. This section lists and briefly discusses each, so that you have an overview of the standard objects available to expressions and scriptlets.

Implicit objects

Object	Class	Purpose
`out`	`javax.servlet.jsp.JspWriter`	The output stream
`request`	`javax.servlet.ServletRequest`	Provides access to details regarding the request and requester
`response`	`javax.servlet.ServletResponse`	Provides access to the servlet output stream, and other response data
`session`	`javax.servlet.http.HttpSessior`	Supports the illusion of a client session within the HTTP protocol
`pageContext`	`javax.servlet.jsp.PageContext`	Used extensively by the generated JSP code to access the JSP environment and user beans
`config`	`javax.servlet.ServletConfig`	Servlet configuration information
`page`	`java.lang.Object`	The page itself
`application`	`javax.servlet.ServletContext`	Represents the Web application; provides access to logging methods
`exception`	`java.lang.Throwable`	*For error pages only*; the exception that caused the page to be invoked

The `out` object

The `out` object is an instance of `javax.servlet.jsp.JspWriter`. The `JspWriter` emulates `java.io.PrintWriter`, but supports buffering like `java.io.BufferedWriter`. The usual `java.io.PrintWriter` methods are available with the modification that the `JspWriter` methods throw `java.io.IOException`.

The `out` object is infrequently referenced directly by a JSP page author. The most common use would be within a scriptlet or passed as a parameter to another method.

The request object

The request object is a standard Servlet object that is a protocol-dependent subclass of javax.servlet.ServletRequest. In other words, for the HTTP protocol the request object will be an instance of javax.servlet.http.HttpServletRequest.

The request object provides access to details regarding the request and requester. A common method is request.getParameter(), although jsp:setProperty mitigates the need, but there are many other useful methods.

The response object

The response object is a standard Servlet object that is a protocol-dependent subclass of javax.servlet.ServletResponse. In other words, for the HTTP protocol the response object will be an instance of javax.servlet.http.HttpServletResponse.

The response object provides access to the servlet output stream. It also allows response headers to be set, including cookies, content type, cache control, refresh, redirection, and so on, and it supports URL encoding as an aid to session tracking when cookies aren't available.

The session object

The session object is a standard Servlet object that is an instance of the javax.servlet.http.HttpSession class. This object helps to support the illusion of a client session within the HTTP protocol.

JSP page authors essentially get sessions for free. Simply use session scope when using jsp:useBean on page 27 to work with any session-specific beans.

The pageContext object

The pageContext object is an instance of javax.servlet.jsp.PageContext. It is used extensively by the generated JSP code to access the JSP environment and user beans.

The pageContext object provides a uniform access method to the various JSP objects and beans, regardless of scope. It also provides the means through which the out object is acquired, so that an implementation can supply a customer JspWriter, and provides the JSP interface to include and forward functionality.

The config object

The config object is a standard Servlet object that is an instance of the javax.servlet.ServletConfig class. It provides access to the ServletContext and to any servlet initialization parameters. You'll rarely use this object.

The page object

The page object is the JSP page object that is currently executing the request.

As discussed in The JspPage interface on page 6 , the page object is a javax.servlet.jsp.JspPage interface descendent. javax.servlet.jsp.HttpJspPage is used for the HTTP protocol.

The methods of the JspPage interface are:

```
void jspInit();// allows user action when initialized
void jspDestroy();// allows user action when destroyed
public void _jspService(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException;
```

As we saw in the jspInit() and jspDestroy() example on page 18 , you may declare the jspInit() and jspDestroy() methods. _jspService() is generated for you.

The application object

The application object is a standard servlet object that is an instance of the javax.servlet.ServletContext class. The ServletContext is also used to get the RequestDispatcher.

The resource methods:

```
URL getResource(String path);
InputStream getResourceAsStream(String path);
```

are especially useful because they isolate you from details of Web storage (for example, file system, JAR or WAR file).

You'll most likely use this object to gain access to logging:

```
application.log(String);
application.log(String, Throwable);
```

The exception object

The exception object is an instance of java.lang.Throwable. The exception object is present only in an error page, defined as such by the page directive.

This object contains the exception that caused the request to be forwarded to the error page.

Section 8. Expressions

JSP expressions

JSP expressions should be subliminally familiar to you by now. We have been using expressions since the second JSP example, when we wrote the following:

```
The local server time is <%= new java.util.Date() %>.
```

The text `<%= new java.util.Date() %>` is a JSP expression.

Valid JSP expressions

Expressions in Java code are statements that result in a value. *Chapter 15* of the Java Language Specification discusses expressions in detail. A JSP expression is a Java expression that is evaluated at request-time, converted to a `String`, and written into the output stream. JSP expressions are written either as:

```
<%= a-java-expression %>
```

or as

```
<jsp:expression> a-java-expression </jsp:expression>
```

This is the one aspect of JSP syntax where I am likely to use the <% syntax instead of the XML syntax.

Important: Note that you do *not* terminate the expression with semicolon.

During the execution of the JSP page, the result of a JSP expression is emitted in-place, which means that the text of the JSP `expression` statement is replaced by its value in the same location on the page.

Section 9. Scriptlets

JSP scriptlets

So far, we've talked about how to declare new data and function members, and how to use Java expressions to create dynamic content for our page. But how do we add logic flow, such as looping and branching, to our page? How do we do more than simply evaluate expressions? That is what scriptlets can do for us.

Scriptlets are what their name implies: they are (more or less) small sets of statements written in the scripting language, which, with the exception of WebSphere as mentioned earlier, means written in Java code.

Scriptlets are written either as:

```
<% java-statements %>
```

or as

```
<jsp:scriptlet>
   java-statements
</jsp:scriptlet>
```

Once again, I always prefer to use the XML syntax when writing scriptlets.

Scriptlet example

The following JSP page uses scriptlets to execute differently depending upon which browser you use:

Scriptlet.jsp (*live demo*)

```
<jsp:scriptlet>
String userAgent = (String) request.getHeader("user-agent");
</jsp:scriptlet>
<HTML>
<HEAD><TITLE>JSP Scriptlet Example</TITLE></HEAD>
<BODY>
<jsp:scriptlet>
if (userAgent.indexOf("MSIE") != -1)
{
</jsp:scriptlet>
<p>You are using Internet Microsoft Explorer.</p>
<jsp:scriptlet>
}
else
{
</jsp:scriptlet>
<p>You are not using Internet Microsoft Explorer. You are using <%= userAgent %></p>
<jsp:scriptlet>
}
</jsp:scriptlet>
</BODY>
```

```
</HTML>
```

Local variables

There were several scriptlets in the Scriptlet.jsp example. The first scriptlet:

```
<jsp:scriptlet>
String userAgent = (String) request.getHeader("user-agent");
</jsp:scriptlet>
```

shows that any Java statement that can appear within a function body is permissible, including a variable declaration.

Program logic

The remaining scriptlets within the Scriptlet.jsp example implement some simple conditional logic. The key sequence, reformatted for brevity, in this JSP page is:

```
<jsp:scriptlet> if (condition) { </jsp:scriptlet>
  JSP statements and template content
<jsp:scriptlet> } else { </jsp:scriptlet>
  JSP statements and template content
<jsp:scriptlet> } </jsp:scriptlet>
```

The first block of JSP statements and template content is effective if the condition is true, and the second block is effective if the condition is false.

This is a common construct, providing for conditional execution of the content of a JSP page. The important thing to notice is that the scriptlets need not be complete statements; they can be fragments, so long as the fragments form complete statements when stitched together in context. In this sequence, the first scriptlet contains a conditional test, and leaves the compound statement in an open state. Therefore, all of the JSP statements that follow are included in the compound statement until the second scriptlet closes the block, and opens the `else` block. The third scriptlet finally closes the compound statement for the `else` clause.

In similar fashion, you can deploy looping constructs within scriptlets.

Scriptlet caveats

Exercise moderation with scriptlets. It is very easy to get carried away writing scriptlets. Scriptlets allow us to write almost arbitrary Java programs within a JSP page. This is not generally in keeping with good JSP page design. Also, as the use of JSP custom tags becomes more mature, JSP authors should expect to see many uses of scriptlets replaced with custom tags.

One reason for this approach is to keep programming logic separate from presentation. Also, many Web page designers are not Java programmers, so scriptlets embedded in Web pages can be confusing.

Java programmers, however, can take advantage of JSP scriptlets as a rapid prototyping tool. You can develop logic as a scriptlet, let the JSP container dynamically re-build your page as you prototype your solution, and create custom tag handlers after you have debugged your logic.

Furthermore, tools such as the one which generated this tutorial, can also make use of JSP scriptlets. This tutorial was authored as a pure XML document. The Tutorial DTD defines the allowable tutorial entities such as Sections, Panels, Paragraphs, and Code Listings. The HTML and PDF forms of this document were automatically generated from the XML source. All of the HTML tags, including the JavaScript portions of the HTML, were inserted by the tool and transparent to the author of the tutorial. Likewise, JSP scriptlets could have been embedded while still preserving a separation between programmed logic and authored content.

Section 10. Actions

Actions

Actions provide a higher level of functionality than the declarations, expressions, and scriptlets we've seen thus far. Unlike the scripting elements, actions are independent of any scripting language. In many respects, JSP actions are like built-in custom tags. In fact, only XML syntax is defined for actions; there is no equivalent <% syntax.

There are three categories of standard actions:

- Those for using JavaBeans components
- Those that control run-time forwarding/including
- Those that prepare HTML for the Java plug-in

In addition, Java developers can create libraries of custom actions, which are made available as tags within a JSP document through the use of the `taglib` directive.

JSP pages and JavaBeans

JSP technology provides a remarkably well-designed integration between JavaBeans and HTML forms. The `jsp:useBean`, `jsp:setProperty`, and `jsp:getProperty` actions work together to achieve this integration.

Action	Purpose
jsp:useBean on page 27	Prepare a bean for use within the JSP page
jsp:setProperty on page 28	Set one or more bean properties on a bean
jsp:getProperty on page 29	Output the value of the bean property as a `String`

What is a Bean?

Although a full treatment of the JavaBeans component architecture is outside the scope of this tutorial, it is easy to explain the essential convention required to interact with `jsp:useBean`, `jsp:setProperty`, and `jsp:getProperty`.

For any given value (called a *property*) of type *T* named *N* that you want to make settable, the class must have a method whose signature is:

```
public void setN(T);
```

For any given property of type *T* named *N* that you want to make gettable, the class must have a method whose signature is:

```
public T getN();
```

jsp:useBean

jsp:useBean tells the JSP page that you want a bean of a given name (which may be a request-time expression) and scope. You also provide creation information. The JSP page checks to see if a bean of that name and scope already exists. If not, the bean is instantiated and registered.

The jsp:useBean tag is written either as:

```
<jsp:useBean id="name" scope="Bean-Scope" Bean-Specification/>
```

or

```
<jsp:useBean id="name" scope="Bean-Scope" Bean-Specification>
   creation-body
</jsp:useBean>
```

If the bean needs to be created and you use the second form of jsp:useBean, the statements that make up the *creation-body* are also executed.

The object made available by useBean is also known as a *scripting variable* and is available to other scripting elements within the JSP page invoking jsp:useBean.

Bean-Scope

The jsp:useBean action makes the bean available as a scripting variable available within the page, but what is the overall lifespan of the bean? Is it re-created each time? Is there a unique copy of the bean for each session?

That is the purpose of the scope attribute. The bean remains available throughout the lifetime of the specified scope, which must be one of the following:

Scope	Duration
page	The bean will be good **only within the defining JSP page** and will be re-created for each new request.
request	The bean will be good throughout that request and is available to included or forwarded pages.
session	The bean will be associated with the particular session responsible for its creation and is good for the lifetime of the session.
application	The bean is common to all sessions and is good until the Web application terminates.

Bean-Specification

The Bean-Specification attributes are extremely flexible, and cover a wide range of options, as illustrated by the following table:

Specification	Meaning
`class="className"`	`class` is the implementation class for the object.
`type="typename" class="className"`	`type` is the type to be used for the bean within the page, and must be compatible with the class. `class` is the implementation class for the object.
`type="typeName" beanName="beanName"`	`type` is the type to be used for the bean within the page. `beanName` is the name of an existing bean, and will be passed to `java.beans.Beans.instantiate()`. The `beanName` may be a JSP expression, whose value is computed at request time. Such an expression must use the <%-syntax.
`type="typeName"`	`type` is the type to be used for the bean within the page.

jsp:setProperty

The `jsp:setProperty` action is a high-level, scripting-language-independent method for setting the values of a scripting variable.

The syntax for the `jsp:setProperty` action is:

```
<jsp:setProperty name="beanName" propertyExpression />
```

The value used for `beanName` is the name that was used for the `id` attribute in the `jsp:useBean` action, or a name similarly assigned by a custom tag. So, following a `jsp:useBean` statement like:

```
<jsp:useBean id = "myName" ... />
```

A subsequent `jsp:setProperty` (or `jsp:getProperty`) action would use:

```
<jsp:setProperty name = "myName" ... />
```

The propertyExpression attribute

The `propertyExpression` for `jsp:setProperty` can take one of several forms, as shown in the following table.

Property Expression	Meaning
property="*"	All bean properties for which there is an HTTP request parameter with the same name will be automatically set to the value of the request parameter. This is probably the single most frequently used form of `jsp:setProperty`, typically used in conjunction with an HTTP form.
property="propertyName"	Sets just that property to the corresponding request parameter.
property="propertyName" param="parameterName"	Sets the specified property to the specified request parameter.
property="propertyName"	Sets the specified property to the specified string value, which will be

value="propertyValue"	coerced to the property's type. The value may be a JSP expression, whose value is computed at request time. Such an expression must use the <%-syntax.

jsp:getProperty

The `jsp:getProperty` action is the counterpart to the `jsp:setProperty` action. Just as `jsp:setProperty` is used to set values into a scripting variable, `jsp:getProperty` is used to get the values from a scripting variable.

The syntax for the `jsp:getProperty` action is:

```
<jsp:getProperty name="name" property="propertyName" />
```

The result of the `jsp:getProperty` action is that the value of the specified bean property is converted to a `String`, which is then written into the `out` object. This is essentially the same as a JSP expression of:

```
<%= beanName.getProperty() %>
```

Using the bean-related actions

Here is a simple, self-contained example of using the bean-related actions:

HelloWorld13.jsp (*live demo*)

```
<HTML>
<jsp:declaration>
// this is a local "helper" bean for processing the HTML form
static public class localBean
{
   private String value;
   public String getValue()         { return value;}
   public void setValue(String s)   { value = s; }
}
</jsp:declaration>

<jsp:useBean id="localBean" scope="page" class="localBean" >
<%-- Every time we create the bean, initialize the string  --%>
<jsp:setProperty name="localBean" property="value" value="World" />
</jsp:useBean>

<%-- Whatever HTTP parameters we have,
     try to set an analogous bean property  --%>
<jsp:setProperty name="localBean" property="*" />

<HEAD><TITLE>HelloWorld w/ JavaBean</TITLE></HEAD>
<BODY>
<CENTER>
<P><H1>Hello
<jsp:getProperty name='localBean' property='value'/></H1></P>
<FORM method=post>
Enter a name to be greeted:
```

```
<INPUT TYPE="text" SIZE="32" NAME="value"
VALUE="<jsp:getProperty name='localBean' property='value'/>">
<BR>
<INPUT TYPE="submit" VALUE="Submit">
</FORM>
</CENTER>
</BODY>
</HTML>
```

Explanation: The local bean

This is our Hello World example, enhanced so that instead of greeting the world, we can tell it whom to greet.

The first change is that we declared a JavaBean within a declaration:

```
static public class localBean
{
   private String value;
   public String getValue()         { return value; }
   public void setValue(String s)   { value = s; }
}
```

Yes, you can do that, and it is convenient for creating helper beans. There are drawbacks to declaring beans within a JSP page, but locally declared beans can also be very convenient under specific circumstances. Our bean has a single `String` property named `value`.

Important: You must always use `page` scope with any locally declared beans.

Explanation: Using the `jsp:useBean` tag

The next thing that we do is use a `jsp:useBean` action, so that we can use a bean within our JSP page:

```
<jsp:useBean id="localBean" scope="page" class="localBean" >
<%-- Every time we create the bean, initialize the string --%>
   <jsp:setProperty name="localBean" property="value" value="World" />
</jsp:useBean>
```

The action tells the JSP container that we want to use a bean named localBean, that the bean will be used only within this page, and that the class of bean is localBean. If the bean does not already exist, it is created for us. Notice the lack of a closing `/` at the end of the `jsp:useBean` tag. Instead, there is a </jsp:useBean> tag later on. Everything that appears between the opening and closing tags is considered the action body. The body is executed if, and only if, the bean is instantiated by the `jsp:useBean` tag. In this case, since the bean exists only for the lifetime of the page, each request creates it anew, and therefore we will always execute the body.

Explanation: Using the `jsp:setProperty` tag

The body of our action consists of a single `jsp:setProperty` tag:

```
<jsp:setProperty name="localBean" property="value" value="World" />
```

The `jsp:setProperty` tag in the body names our bean and indicates that it wants to set the property named `value` to World. This means that the default will be to greet the world. But how do we greet someone else? That is where the HTML form and the other `jsp:setProperty` tag come into play:

```
<%-- Whatever HTTP parameters we have,
     try to set an analogous bean property  --%>
<jsp:setProperty name="localBean" property="*" />
```

As the comment implies, the second `jsp:setProperty` tag uses a bit of JSP magic on our behalf. It takes whatever fields you've named in an HTML form and, if your bean has a property of the same name, its value is set to the value submitted via the form. Right now, we just have one field, but when you have complex forms, you will really appreciate the simplicity of this single tag.

Explanation: Using the `jsp:getProperty` tag

Next, our JSP page displays a greeting, but with a twist. Instead of simply saying "Hello World," the page asks the bean whom it should greet, and displays that name:

```
Hello <jsp:getProperty name='localBean' property='value'/>
```

The `jsp:getProperty` tag is similar to the `jsp:setProperty` tag; it takes the name of the bean, and the name of the property. The result of the `jsp:getProperty` tag is just like that of a JSP expression: the value of the property is converted to a string and written into the output stream. So, whatever value is placed into our bean will be used as the name to be greeted.

Explanation: The HTML Form

That brings us to the final part of our example, the HTML form itself:

```
<FORM method=post>
Enter a name to be greeted:
<INPUT TYPE="text" SIZE="32" NAME="value"
VALUE="<jsp:getProperty name='localBean' property='value'/>">
<BR>
<INPUT TYPE="submit" VALUE="Submit">
</FORM>
```

There is nothing really special going on here, now that you understand how these tags work. The form simply defines a single input field. Whatever value we enter into that field will be put into our bean by the `jsp:setProperty` tag. The form also uses a `jsp:getProperty` tag to initialize the field, as a convenience.

JSP pages can be smart forms

Did you notice that there is not a form action associated with our sample form?

The form submits back to our JSP page, which handles the form directly. JSP pages do not force you to submit a form's content to a CGI script, or other third party, for processing. Often, a FORM tag has an action parameter, which tells the form to which URL the form data should be submitted, encoded either as GET parameters or POST data. However, when you leave off the action parameter, the FORM defaults to submitting it back to the current URL. This is why the form embedded in our JSP page is posted back to our JSP page for processing.

The `jsp:include`, `jsp:forward`, and `jsp:param` actions

The `jsp:include` and `jsp:forward` actions allow us to use the output from other pages within (or instead of, respectively) a JSP page's content.

Action	Purpose
jsp:include on page 32	Include the referenced resource's content within the including page's content
jsp:forward on page 33	Substitute the referenced resource's content for the forwarding page's content
jsp:param on page 33	Pass a parameter to the resource referenced by the enclosing `jsp:include` or `jsp:forward`

`jsp:include`

The `page` parameter tells the JSP container to include another resource's content into the stream. The resource is specified by a relative URL, and can be either dynamic (for example, a JSP page, servlet, or CGI script) or static (for example, an HTML page).

The difference between the `jsp:include` action and the `include` directive is that the action dynamically inserts the content of the specified resource at request time, whereas the directive physically includes the content of the specified file into the translation unit at translation time.

The `jsp:include` action is written as:

```
<jsp:include page="relativeURL" flush="true" />
```

The `flush` parameter tells JSP that whatever output we've written into the stream so far should be committed. The reason for this is that the output could be buffered, and we need to flush our buffer before we let someone else write to the output stream.

Prior to JSP version 1.2, `flush` is required, and the mandatory value is `true`. JSP v1.2 and later default `flush` to `false`, and so `flush` can be optional.

jsp:forward

The `jsp:forward` action tells the JSP container that we want to forward the request to another resource, whose content will substitute for our own. The resource is specified by a relative URL, and can be either dynamic (for example, a JSP page, servlet, or CGI script) or static (for example, an HTML page).

The `jsp:forward` action is written as:

```
<jsp:forward page="relativeURL" />
```

The `page` parameter specifies to which resource the request should be (re)directed. The resource is specified by a relative URL and can be either dynamic (for example, a JSP page, servlet, or CGI script) or static (for example, an HTML page).

The `jsp:forward` action is only permitted when we have not yet committed any content to the output stream.

---

jsp:param

The `jsp:param` action provides parameters, which can be passed to the target page of an include or forward action, and is written as:

```
<jsp:param name="name" value="value" />
```

The parameters are [name, value] pairs, which are passed through the `request` object to the receiving resource.

---

jsp:include example

The following example illustrates the `jsp:include` action:

UseHeader.jsp (*live demo*)

```
<HTML>
<jsp:include page="head.jsp" flush="true">
<jsp:param name="html-title" value="JSP Include Action" />
</jsp:include>
<BODY>
</BODY>
</HTML>
```

head.jsp

```
<HEAD>
<TITLE>
<%=(request.getParameter("html-title") != null) ?
    request.getParameter("html-title") : "UNTITLED"%>
</TITLE>
```

```
<META HTTP-EQUIV="Copyright" NAME="copyright" CONTENT="Copyright (C) 2001 Noel J. Bergma
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
</HEAD>
```

The example is made up of two separately compiled JSP pages, UseHeader.jsp and head.jsp. Since the JSP page generates no body content, the only thing you'll notice when you look at the browser view is the title, but the source view shows the HEAD section.

When we request UseHeader.jsp, that JSP page in turn invokes head.jsp to generate the HEAD section of the resulting HTML output. The generated HEAD section includes only a few tags in this example, but we could add as many as we want.

The `jsp:param` action is used to pass our desired title to head.jsp, where we use the request object to get it. If the parameter is not provided, the page will still generate a default TITLE tag for us. You can see how you could provide support for keywords, descriptions, and so forth with appropriate default values for your Web sites.

The `jsp:plugin` action

Political and legal battles within the software industry have resulted in a situation where the JVM built into a browser is often outdated. The solution to the problem of out-of-date or missing browser support for Java is to be able to plug newer versions of the JVM into the browser. That is what the Java Plug-in provides.

The `jsp:plugin` action is a specialized tag, whose sole purpose is to generate the appropriate <OBJECT> or <EMBED> tag to load the *Java Plug-in* software as necessary, and then load a specified applet or JavaBean.

The `jsp:plugin` action is written as:

```
<jsp:plugin type="bean|applet" code="objectCode" codebase="URL" [align="alignment"]
[archive="archiveList"] [height="height"] [hspace="hspace"]
[jreversion="JRE-version"] [name="componentName"] [vspace="vspace"]
[width="width"] [nspluginurl="URL"] [iepluginurl="URL"]>

[<jsp:params> <jsp:param name="name" value="value" />* </jsp:params>]

[<jsp:fallback> arbitrary content </jsp:fallback> ]

</jsp:plugin>
```

All attributes except for `type`, `jreversion`, `nspluginurl`, and `iepluginurl` are defined by the HTML specification for the <OBJECT> tag.

Attribute	Meaning
`type`	The type of component (an Applet or a JavaBean)
`jreversion`	The JRE version that the component requires
`nspluginurl`	A URL from which to download a Navigator-specific plugin
`iepluginurl`	A URL from which to download an IE-specific plugin

jsp:plugin example

The following jsp:plugin and <APPLET> tag are equivalent.

jsp:plugin:

```
<jsp:plugin type="applet"
code="com.devtech.applet.PhotoViewer.class"
codebase="/applets"
archive="gallets.jar"
WIDTH="266" HEIGHT="392" ALIGN="TOP" >
<jsp:params>
<jsp:param name="image" value="1" />
<jsp:param name="borderWidth" value="10" />
<jsp:param name="borderColor" value="#999999" />
<jsp:param name="bgColor" value="#000000" />
</jsp:params>
<jsp:fallback>
<P>Unable to start Java plugin</P>
</jsp:fallback>
</jsp:plugin>
```

<APPLET> tag:

```
<APPLET WIDTH="266" HEIGHT="392"
BORDER="0" ALIGN="TOP"
CODE="com.devtech.applet.PhotoViewer.class"
CODEBASE="/applets/" ARCHIVE="gallets.jar">
    <PARAM NAME="ARCHIVE" VALUE="gallets.jar">
    <PARAM NAME="image" VALUE="1">
    <PARAM NAME="borderWidth" VALUE="10">
    <PARAM NAME="borderColor" VALUE="#999999">
    <PARAM NAME="bgColor" VALUE="#000000">
</APPLET>
```

As seen, jsp:action is very similar to the <APPLET> tag, slightly more verbose, but permits the server to generate the appropriate tags based upon the requesting browser.

Section 11. Summary

Wrap up

We have completed an overview of the JSP specification. It is important to understand that this introduction presented what we *can* do with JSP technology. This is not necessarily the same as what we *should* do. The former deals with syntax and grammar, the latter deals with philosophy and methodology.

For example, one of the benefits of using JSP technology is separating presentation from business logic. To that end, many JSP practitioners maintain that it is bad to put any Java code into a JSP page, on the grounds that it is not maintainable by non-programming Web designers. In the extreme, the only JSP features they permit are the use of the action tags and custom tag libraries (and the attendant directives). Some will permit "simple" expressions; others will not.

This introductory material follows the approach that one does not teach a subject by avoiding its features in an effort to advance a particular methodology. Ignoring debated aspects of a technology places the reader at a disadvantage. Furthermore, the target audience for this tutorial is not the nonprogramming Web designer; it is the Java programmer who wants to employ JSP technology in content delivery solutions. It is just as likely that a reader may be involved in developing, say, new WYSIWYG editor tools that generate JSP pages that use JSP constructs internally while at the same time hiding them from the user. And it is certainly the case that rapid prototyping may employ different tools from those used in production.

We abide by the philosophy that it is better to teach all of the options; illustrating why and when certain approaches are more or less desirable than others in practice. As this is only an introductory tutorial, we are not able to delve into such topics as programming models that you can use to implement JSP solutions, or how we can use JSP technology to manage the look and feel associated with our content. We recommend that you look at other documents and more advanced tutorials, which will provide such information.

Resources

Here are some JSP resources you may find helpful:

- JSP is designed to clearly separate the roles of Web designer and application developer. Get an overview of the architectural decisions that make this possible in "*JSP architecture*" (*developerWorks*, February 2001).

- This IBM tutorial, "*Using JSPs and custom tags within VisualAge for Java and WebSphere Studio*," shows you how to build JSP custom tag libraries using VisualAge for Java and WebSphere Studio.

- IBM's alphaWorks offers the *JSP Format Bean Library*, a collection of beans that support JSP. Refer to this overview of the *JSP Format Bean Library Project*.

- IBM's Marshall Lamb shows how to use embedded Java code to dynamically build XML templates in "*Generate dynamic XML using JSP technology*" (*developerWorks*, December 2000).

- In "*"Build better Web sites using the Translator pattern"* (*developerWorks*, February 2001), IBM's Donald S. Bell explains how to keep your JSP pages, servlets, and business objects happily separate, but still talking.

- Learn how JSP technology can be used to *build dynamic Web sites using WebSphere Studio*.

- For a complete explanation of the design and construction of Toot-O-Matic, the Java tool used to build our tutorials, read *"XML training wheels"* (developerWorks, June 2001), by IBM's Doug Tidwell.

- For the latest JSP information and downloads, go to *Sun's JSP page.*

- For more on Jakarta, go to the *Jakarta page* on the Apache Software Foundation Web site.

Your feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

For questions about the content of this tutorial, contact the author, Noel J. Bergman, at *noel@jspdevguide.com*.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials.

developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.