Introduction to JavaServer Pages

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. About this tutorial

What is this tutorial about?

This tutorial introduces the fundamentals of JavaServer Pages (JSP) technology 2.0 and provides an update to the original tutorial written by Noel J. Bergman that discussed JSP 1.2. This tutorial will give you a solid grasp of JSP basics and enable you to start writing your own JSP solutions. In particular, this tutorial will:

- Discuss the fundamental elements that define JSP, such as 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.

Should I take this tutorial?

This is an introductory tutorial intended for new or novice JSP programmers who have a solid grasp of Java programming and HTML. The emphasis is on simple explanations, rather than exhaustive coverage of every option. If you're a JSP expert, this tutorial is probably not for you. However, it will serve as a reference for knowledgeable developers wishing to shift to JSP 2.0 from earlier versions.

Note that this tutorial is strictly an overview of the JSP syntax and semantics; it does not address programming style.

Web container requirements

The JSP 2.0 specification requires J2SE 1.3 or later for stand-alone containers and version 1.4 for containers that are part of a J2EE 1.4 environment. To get started with JSP 2.0, the Web container must also support the Java Servlet 2.4 specification. All JSP containers must be able to run in a J2SE 1.4 environment.

Also, the JSP 2.0 specification uses the Servlet 2.4 specification for its Web semantics.

About the authors

Abhinav Chopra is a senior software engineer with IBM Global Services India Ltd. He has extensive experience in Java-, J2EE- and XML-related technologies. His areas of expertise include designing and developing n-tier enterprise applications. He has presented Java- and XML-related topics in department-level talks, and holds professional certifications from IBM and Sun Microsystems. Contact him at *abchopra@in.ibm.com* (mailto:abchopra@in.ibm.com) .

Noel J. Bergman's background in object-oriented programming spans more than 20 years. He participated in the original CORBA and Common Object Services task forces, and consistently receives high marks as a favored speaker at the *Colorado Software Summit* (http://www.softwaresummit.com/) , as well as other industry conferences. He provides customized IT consulting and mentoring services based on each client's specific problem domain. He is currently involved in using open source freeware to develop interactive database-backed Web sites. He is also a co-author of GNUJSP, an open source implementation of JSP, and is the originator of the *JSP Developer's Guide* (http://www.jspdevguide.com/) Web site. Contact him at *noel@jspdevguide.com* (mailto:noel@jspdevguide.com) .

Section 2. JSP overview

Background

JSP is one of the most powerful, easy-to-use, and fundamental tools in a Web-site developer's toolbox. JSP 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 facilitates the 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 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 involves deciding whether to use JSP vs. other Java server-side technologies. Unlike 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 clean models of interaction.

Who uses JSP? Are there any real-world, mission-critical deployments of JSP on the Internet? Delta Air Lines and CNN are two of the many businesses that rely upon JSP, and major new sites are frequently appearing.

What is JSP?

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

If you are an HTML designer, you can look at JSP as an extension of HTML that gives you 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. Even better, JSP provides the means by which programmers can create new HTML/XML tags and JavaBeans components that offer new features for HTML designers without requiring you to learn how to program.

Note: A common misconception is that Java code embedded in a JSP is transmitted with the HTML and executed by the user agent, such as a browser. This is not the case. A JSP is translated into a Java servlet and executed on the server. JSP statements embedded in the JSP become part of the servlet generated from the JSP. 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 as a new higher-level way to write servlets. Instead of directly writing servlet classes and 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 generated from your JSP code is executed, and the results are returned to the user.

A simple JSP

Let's move this discussion from the abstract to the concrete by examining a couple of simple JSPs. The first example is a JSP version of Hello World:

HelloWorld.jsp

```
<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. The server passes whatever static content you have on your page, unchanged, to the client. In this case, no dynamic content exists, so the static content is passed, unchanged, to the browser. The difference between treating this as a JSP 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 is translated into a servlet, and when that servlet is executed, the response from the servlet contains the HTML. The user sees identical content; only the mechanism used on the server is different.

A dynamic JSP

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

HelloWorld2.jsp

```
<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 later. For now, just understand that when output is prepared for the client, the server's current time is acquired and is 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 executes 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

```
<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 ways of writing expressions are synonymous. The meaning and resulting servlet code are identical.

JSPs are servlets

Technically, a JSP is compiled into a Java servlet (see Resources on page 62 for more on Java servlets). Although it's possible for a JSP implementation to generate bytecode files directly from the JSP source code, typically, each JSP is first translated into the Java source code for a servlet class, and that servlet is compiled. The resulting servlet is invoked to handle all requests for that page. The page translation step is usually performed the first time a given JSP is requested and only when that page's source code changes thereafter. Otherwise, the resulting servlet is simply executed, providing quick delivery of content to the user.

The JSP specification defines the JSP language and 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 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. Normally, this is of limited consequence; all implementations of JSP must conform to the same standard so that your pages are portable. Occasionally, however, there might be an operational difference outside the specification.

GNUJSP, an open source implementation of JSP, takes a fairly straightforward approach. When you request a JSP, GNUJSP receives a request from the Web server. GNUJSP looks in the file system to find the JSP. 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 JSPs; GNUJSP automatically recompiles them when -- and only when -- necessary.

The Apache 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 type 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.

The WAR file contains all of the resources that make up the Web application. You use deployment descriptors within the WAR file 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 might 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 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 affect the generated implementation of the page's `_jspService()` method. Any declarations affect the definition of the generated servlet class.

What's next?

Now that you've seen how authoring a JSP is different from authoring standard HTML, let's look more closely at the syntax for JSPs. The remainder of this tutorial provides a brief overview of all JSP syntax and semantics.

Note: Technically, the JSP specification permits you to use other languages as a JSP's scripting language. At the present time, the scripting language must be the Java language. At the present time, the scripting language must be Java. However, understand that in regard to embedding Java code within a JSP, you might be able to use other languages in the future.

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 11	Everything in your JSP's source file that is not a JSP element; includes all static content
Directives on page 16	Instructions you place in your JSP to tell the JSP implementation how to build your page, such as whether to include another file
Scripting elements	Declarations on page 21, Expressions on page 24, and Scriptlets on page 25 used to embed Java code into your JSPs
Actions on page 45	Actions provide high-level functionality, in the form of custom XML-style tags, to a JSP without exposing the scripting language. Standard actions include those to create, modify, and otherwise use JavaBeans within your JSP.

JSP element syntax

Each of the 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 you get the overall flavor of JSP syntax. We cover the formal specification of JSP syntax in more detail later. Remember you can use the <% syntax and the XML syntax to write most elements.

Element	<% Syntax	XML Syntax
Output comments on page 12	<!-- *visible comment* -->	<!-- *visible comment* -->
Hidden comments on page 12	<%-- *hidden comment* --%>	<%-- *hidden comment* --%>
Declarations on page 21	<%! *Java declarations* %>	<jsp:declaration> *Java language declarations* </jsp:declaration>
Expressions on page 24	<%= *A Java expression* %>	<jsp:expression> *A Java expression* </jsp:expression>
Scriptlets on page 25	<% *Java statements* %>	<jsp:scriptlet>

		Java statements </jsp:scriptlet>
Directives on page 16	<%@ ... %>	<jsp:directive. *type* ... />
Actions on page 45		<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 that is not an actual JSP element. Recall our first HelloWorld.jsp example (A simple JSP on page 5); *everything* on that page was template content. The JSP specification states that everything that is not an actual JSP element is template content and should be passed, unchanged, into the output stream. For example, consider the following variation of our Hello World example:

HelloWorld4.jsp

```
<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 we included a JSP directive to import the `java.util.Date` class so you don't have to fully qualify the package name when you 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 is the same as we received previously. However, that assumption is 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 confirm this by viewing the HTML source for the live demo.

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

Comments

You can place comments in a JSP in three ways:

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

Output comments

With the first way, you write a standard HTML comment, following this format:

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

This comment is part of the template content. Nothing fancy here; 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 fact, many HTML comments are not really comments at all. For example, the HTML standard recommends that you embed JavaScript statements 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 can write the following:

HelloWorld5.jsp

```
<!-- 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

You can write a comment a second way by using the JSP notation:

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

This comment is known as a *hidden comment* because it is an actual JSP comment and does not appear in the generated servlet. The user never sees the comment and won't have any way to know of its existence. Repeating the Hello World example, but this time with a JSP comment, yields the following:

HelloWorld6.jsp

```
<%-- 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 can write the following:

HelloWorld7.jsp

```
<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 permits nested comments.

Quoting and escaping rules

<% is a magic token the JSP container understands as the beginning of a JSP element. How do you put a <% into the output stream? For example, if you want to use JSP to write a tutorial on the subject of JSP, how do you write a page so

that its output contains the source for a JSP? 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 is part of the JSP specification. It indicates that when you want a <% to be passed along literally instead of being processed, you write it as <\% in the template content.

Within this element type ...	If you want to write ...	You 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

```
<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 must be aware of not only the rules for JSPs but also the rules for the template content. In this case, you write HTML content, and HTML also has escaping rules. These rules say that you cannot write <\% within HTML, except within a comment. Why not? 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**
<	&lt;
>	&gt;
&	&amp;

Many other useful escape sequences for HTML exist. Part of the HTML V4.01 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. We describe three directives normally used in JSPs below. In addition to those, JSP 2.0 introduced three more. These directives are available only to tag files and are, therefore, introduced in Tag files on page 56.

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

Typically, you use the directives described above to import Java packages for use within your page, include files, or access libraries of custom tags.

Note that directives do not produce any output to the current output stream.

The general syntax for a directive is:

```
<%@ directive [attr="value"]%>
```

or

```
<jsp:directive. directive [attr="value"] />
```

The **page** directive

The `page` directive defines various page-dependent properties and communicates these to the JSP container. The `page` directive does this by setting page attributes as follows:

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

or

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

Most often, you will use the import attribute, as you 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 directive causes `java.util.Date` to be imported (as in the Java statement `import java.util.Date`) into your JSP.

---

page directive attributes

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

Attribute	Use
`language`	This attribute defines the scripting language to be used in the scriptlets, expressions, and declarations within the body of the translation unit (the JSP and any files included using the include directive described later). The only language supported by JSP specification 2.0 is Java, so this is the default value for this `page` attribute.
`extends`	Specifies the superclass of the class to which this JSP is transformed. It should be defined as a fully qualified class name. It is implementation-dependent, and you should use it with caution. Each implementation provides its own default class; overriding the default can affect the quality of the implementation.
`import`	Describes all Java types available as imports. It follows the same rules as standard Java language `import` statements. By default, `java.lang.*`, `javax.servlet.http.*`, `javax.servlet.*`, and `javax.servlet.jsp.*` are imported. You can add any additional packages or types you need imported for your page.
`session`	Indicates whether the page requires participation in an HttpSession. By default, the value is true. You have the option of specifying the value as false (in that case, any reference to HttpSession in the page will not be resolved).
`buffer`	This page attribute indicates whether the content output from the page will be buffered. By default, the page output is buffered with an implementation buffer size no smaller than 8 KB.
`autoFlush`	Specifies whether the buffered output should be flushed automatically (true value) when the buffer is filled, or whether an exception should be raised (false value) to indicate buffer overflow. The default value is true.
`isThreadSafe`	Indicates whether the resulting servlet is threadsafe. The JSP 2.0 specification advises against using `isThreadSafe` because the generated servlet might implement the `SingleThreadModel` interface, which has been deprecated in the Servlet 2.4 specification.
`info`	Defines a `string` that can be incorporated into the translated page.
`isErrorPage`	Indicates if the current JSP is the intended target of another JSP's

	. The default value is false.
`errorPage`	Defines a URL to a resource that will catch the current page's exception.
`contentType`	Defines the MIME type and the character encoding for the response of the JSP. Values are either of the form "TYPE" or "TYPE;charset= CHARSET".
`pageEncoding`	Defines a URL to a resource that will catch the current page's exception.
`isELIgnored`	Defines whether expression language (EL) expressions are ignored or evaluated for this page and translation unit. The default value varies depending on the web.xml version. For details, see the section on JSP configuration on page30 .

The `page` directive is a catch-all for any information about a JSP 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's source, as it is compiled. The two ways to write a JSP `include` directive:

```
<%@ 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 useful for including reusable content into your JSP, such as common footer and header elements. For example, a common footer element on your JSPs is a last-changed indicator, so people know when the page was last updated. You implement that within an included file. Another page, often included, forces users to log in before they have access to a page's content.

`include` directive example

Let's consider a simple demonstration 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 JSPs. This looks like the following:

HelloWorld9.jsp

```
<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 your JSP when the page is translated.

The **`taglib`** directive

The ability to create and use custom tag libraries is one of the most powerful and useful features JSP offers. 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 the details can be encapsulated within the custom tags. In addition, JSPs that use custom tags have a much cleaner separation of interface design and business logic implementation than JSPs with a lot of embedded Java code (scriptlets 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 you 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 the use of a custom tag handler class. In this handler class, the current date and time are put 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 this tutorial's scope (see Resources on page 62 for more information about taglibs).

Section 6. Declarations

JSP declarations

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

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

or

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

For clarity, 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:

HelloWorld10.jsp

```
<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 is compiled into a Java servlet class. JSP declarations allow you 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, we refer to our new functions.

JSP initialization and termination

JSP containers provide a means for pages to initialize their state. A page might declare optional initialization and cleanup routines. The `jspInit()` method is called before the page is asked to service any requests. The JSP container calls 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();
```

You place these optional methods within a JSP declaration just as you would any other method.

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

This example is identical to the JSP declaration example on page 21, 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, you initialize it in `jspInit()` and de-reference it in `jspDestroy()`:

HelloWorld11.jsp

```
<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. 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 (see Resources on page 62). A JSP expression is a Java expression evaluated at request time, converted to a `String`, and written into the output stream. JSP expressions are written either as:

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

or

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

This is the one aspect of JSP syntax where you will likely use the <% syntax instead of the XML syntax.

Important: You do *not* terminate the expression with a semicolon.

During the execution of the JSP, the JSP expression's result 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 8. Scriptlets

JSP scriptlets

So far, we've discussed how to declare new data and function members, and how to use Java expressions to create dynamic content for our page. But how do you add logic flow, such as looping and branching, to your page? How do you do more than simply evaluate expressions? That is when scriptlets come into play.

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 executed at request processing time. Whether they produce any output into the output stream depends on the code in the scriptlet.

Scriptlets are written as:

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

or

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

Once again, using the XML syntax when writing scriptlets is preferable.

Scriptlet example

The following JSP uses scriptlets to execute differently depending on the browser you use:

Scriptlet.jsp

```
<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 are several scriptlets in the Scriptlet.jsp example. The first scriptlet shows that any Java statement that can appear within a function body is permissible, including a variable declaration:

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

Program logic

The remaining scriptlets within the Scriptlet.jsp example implement some simple conditional logic. The key sequence, reformatted for brevity, in this JSP 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 JSP content. The important thing to notice is that the scriptlets don't have to 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 easy to get carried away writing scriptlets. Scriptlets allow you to write almost arbitrary Java programs within a JSP. This is not generally in keeping with good JSP design. Also, as the use of JSP custom tags becomes more mature, JSP authors should expect to see many scriptlet uses replaced with custom tags.

One reason for using custom tags 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.

Section 9. Error handling

Translation-time processing errors

Errors might be encountered while translating JSP source into the corresponding servlet class by a JSP container. This translation can happen:

- After the JSP has been deployed into the JSP container and before the client requests the JSP.
- When the client requests a JSP.

In the first case, error processing is implementation-dependent, and the JSP specification does not cover this. However, in both cases, if translation errors are encountered, the JSP container should return the appropriate error code for the client requests.

For HTTP protocols, the error status code 500 is returned.

Request-time processing errors

Request-time processing errors occur during the request processing by a JSP. These errors can occur in the body of the generated servlet or in any other code called from the body of the JSP implementation class.

These exceptions might be caught and handled (as appropriate) in the body of the JSP implementation class.

Any uncaught exceptions thrown in the body of the JSP implementation class result in the forwarding of the client request and uncaught exception to the errorPage URL specified by the JSP. The next section discusses how you can use JSPs as error pages.

Using JSPs as error pages

A JSP is considered an error page if it sets the page directive's `isErrorPage` attribute to true. Such JSP URLs are generally defined as errorPage URLs in other JSPs. If an uncaught exception occurs in any of the JSPs defining errorPage JSPs, the control is transferred to these error pages.

Here is an example of an error page:

MyErrorPage.jsp

```
<%@ page isErrorPage="true" contentType="text/html" %>
A fatal error has occurred. Please call the help desk.
```

The following JSP, MyErrorThrower.jsp, throws the error captured by MyErrorPage.jsp:

MyErrorThrower.jsp

```
<%@ page errorPage="MyErrorPage.jsp" contentType="text/html" %>
<% int totalCost = 200; int numberOfVehicles =0;
  int costOfVehicle = totalCost/numberOfVehciles ; %>
```

An `ArithmeticException` in the above code causes an unhandled exception to be thrown. This results in a request being forwarded to MyErrorPage.jsp.

Error-page element

You can also define error pages for JSPs in the web.xml file. The web.xml snippet below shows how to do this:

```
<error-page>
<exception-code>500</exception-code>
<location>/MyErrorPage.jsp</location>
</error-page>
```

Section 10. JSP configuration

Changes in web.xml

JSP configuration information is specified in the deployment descriptor (WEB-INF/web.xml). As of the Java Servlet 2.4 and JSP 2.0 specifications, the web.xml is defined using XML schema instead of DTD. The obvious benefit of this is that now the order of the elements in web.xml does not matter.

Also, the deployment descriptor includes many new configuration options. We describe these configuration options below.

`jsp-config` element

You use the `jsp-config` element to provide the global configuration information for the JSP files in a Web application. Schema defines two sub-elements of this element:

- **`taglib`** -- Defines the taglib mapping
- **`jsp-property-group`** -- Defines the groups of JSP files

`taglib element`

The `taglib` element provides a mapping between the taglib URIs and TLD resource paths. The `taglib` element has two sub-elements:

- **`taglib-uri`** -- This element describes a URI identifying a tag library used in the Web application. It can be an absolute URI or a relative URI. Note that same `taglib-uri` value cannot be repeated in a web.xml.

- **`taglib-location`** -- This sub-element defines the actual TLD resource path for the `taglib-uri`.

As an example, consider that a JSP uses a TLD resource defined as `test-taglib.tld`. Now, this JSP can declare the taglib URI as:

```
<%@ taglib prefix="test"
uri="http://ibm.com/test-example-taglib"%>
```

In this case, the entry in web.xml should look like this:

```
<taglib>
<taglib-uri>
http://ibm.com/test-example-taglib
</taglib-uri>
<taglib-location>
/WEB-INF/jsp/test-taglib.tld
</taglib-location>
</taglib>
```

`jsp-property-group` element

A `jsp-property-group` element represents a collection of properties that apply to a set of files that represent JSPs. These properties are defined in one or more `jsp-property-group` elements in the web.xml.

The properties that can be described in a `jsp-property-group` include:

- **`el-ignored:`** Controls disabling of EL evaluation
- **`scripting-invalid:`** Controls disabling of scripting elements
- **`page-encoding:`** Indicates page-encoding information
- **`include-prelude:`** Prelude automatic includes
- **`include-coda:`** Coda automatic includes
- **`is-xml:`** Indicates that a resource is a JSP document

Let's examine each of these properties.

---

el-ignored

Specifications prior to JSP 2.0 did not identify the EL pattern *`${expr}`*, so in JSPs written before JSP 2.0, there might be situations where the intention is not to activate EL expression evaluation but to pass such patterns verbatim.

For Web applications delivered using a web.xml that uses the Servlet 2.4 format, the default mode evaluates EL expressions. The default mode for JSPs in a Web application delivered using a web.xml based on Servlet 2.3 or earlier formats ignores EL expressions.

The default mode can be explicitly changed by setting the value of `el-ignored` to true or false. The snippet shown below deactivates EL evaluation for the configtest.JSP inside folder jsp2:

```
<jsp-property-group>
<url-pattern>jsp2/configtest.jsp</url-pattern>
<el-ignored>true</el-ignored>
```

```
</jsp-property-group>
```

---

scripting-invalid

With the introduction of EL in JSP specification 2.0, JSP authors might want to disallow scripting elements within JSPs. By default, scripting is enabled. You can disable it by setting the value of the `scripting-invalid` property to false.

For example, the snippet shown below deactivates scripting for the configtest.JSP inside folder jsp2:

```
<jsp-property-group>
<url-pattern>jsp2/configtest.jsp</url-pattern>
<scripting-invalid>true</scripting-invalid>
</jsp-property-group>
```

Page encoding

You can use page encoding to set the `pageEncoding` property of a group of JSPs defined using the `jsp-property-group` element. If different values are given for the `pageEncoding` attribute of the JSP directive and in a JSP configuration element matching the page, a translation time error will occur.

For example, the snippet shown below sets the page encoding for a JSP group inside the jsp2 folder:

```
<jsp-property-group>
<url-pattern>jsp2/*.jsp</url-pattern>
<page-encoding>ISO-8859-1</page-encoding>
</jsp-property-group>
```

---

include-prelude

The value of `include-prelude` is a context-relative path that must correspond to an element in the Web application. When the element is present, the given path is automatically included (as in an include directive) at the beginning of the JSP in the jsp-property-group.

For example, the JSP fragment topFragment.jspf is automatically included at the beginning of all JSPs inside the jsp2 folder:

```
<jsp-property-group>
<url-pattern>jsp2/*.jsp</url-pattern>
<include-prelude>/jsp2/topFragment.jspf</include-prelude>
</jsp-property-group>
```

---

include-coda

The value of `include-coda` is a context-relative path that must correspond to an element in the Web application. When the element is present, the given path is automatically included (as in an include directive) at the end of the JSP in the `jsp-property-group`.

For example, the JSP fragment endFragment.jspf is automatically included at the end of all JSPs inside the jsp2 folder:

```
<jsp-property-group>
<url-pattern>jsp2/*.jsp</url-pattern>
<include-prelude>/jsp2/endFragment.jspf</include-prelude>
</jsp-property-group>
```

---

is-xml

This JSP configuration element denotes that a group of files are JSP documents and must, therefore, be interpreted as XML documents.

For example, the web.xml snippet shown below indicates that files with extension .jspx are XML documents:

```
<jsp-property-group>
<url-pattern>*.jspx</url-pattern>
<is-xml>true</is-xml>
</jsp-property-group>
```

Section 11. Implicit objects

Implicit objects

Many objects are 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 7). Scriptlets and expressions, which affect the `_jspService()` method, have access to the implicit objects, but declarations, which affect the generated class, do not.

You can write numerous JSPs and never have a need to refer to the implicit objects directly. Most of the implicit objects are standard Servlet API components. This next section lists and briefly discusses each, so 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.HttpSession`	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`.

A JSP author infrequently references the `out` object directly. The most common use is within a scriptlet or as a parameter passed to another method.

For example, the code below uses out object to print "Example of out implicit object" on the JSP:

```
<% out.println("Example of out implicit object"); %>
```

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 is an instance of `javax.servlet.http.HttpServletRequest`.

The `request` object provides access to details regarding the request and requester. For example, the code below fetches the value of the request parameter named username:

```
<% String userName = request.getParameter("username"); %>
```

The **`response`** object

The `response` object is a standard servlet object that is a protocol-dependent subclass of `javax.servlet.ServletResponse`. For the HTTP protocol, the `response` object is an instance of `javax.servlet.http.HttpServletResponse`.

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

For example, the code below sets the content type for a response:

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

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 support the illusion of a client session within the HTTP protocol.

JSP authors essentially get sessions for free. Simply use `session` scope when using **`jsp:useBean`** on page 46 to work with any session-specific beans.

The code below shows how to use session object to fetch session attribute username:

```
<% String userName = (String)session.getAttribute("username"); %>
```

The **`pageContext`** object

The `pageContext` object is an instance of `javax.servlet.jsp.PageContext`. The generated JSP code uses it extensively 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 an implementation can supply a custom `JspWriter`, and provides the JSP interface to `include` and `forward` functionality.

As shown below, you can also use `pageContext` to set attributes or to fetch exceptions and process as required:

```
<% java.lang.Exception e = pageContext.getException()

 if (e.getMessage().equals("testException") { %>
```

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 object executing the request.

As discussed in The JspPage interface on page 7, 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:

```
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 you saw in the **`jspInit()`** and **`jspDestroy()`** example on page 22, you might 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. You also use `ServletContext` 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 causes the request to forward to the error page.

Section 12. Objects and scopes

Application-specific objects

It is recommended that application behavior be encapsulated in objects whenever possible. This helps page designers focus on the presentation issues. Apart from the predefined implicit objects, a JSP can access, create, and modify server-side objects. You can create and use objects within a JSP in many ways:

- Instance and class variables of the JSP's servlet class are created in declarations and accessed in scriptlets and expressions.
- Local variables of the JSP's servlet class are created and used in scriptlets and expressions.
- Attributes of scope objects (see the discussion below on objects and scopes) are created and used in scriptlets and expressions.
- You can also create a JavaBeans component in a declaration or scriptlet and invoke the methods of a JavaBeans component in a scriptlet or expression.

Object scopes

Scope describes what entities can access the object. Objects can be defined in any of the following scopes:

page

Objects with page scope are accessible only within the page where they are created. All references to such an object are released after the JSP sends the response back to the client or the request is forwarded somewhere else. References to objects with page scope are stored in the `pageContext` object.

request

Objects with request scope are accessible from pages processing the same request where they were created. References to the object are released after the request processes. In particular, if the request is forwarded to a resource in the same runtime, the object is still reachable. References to objects with request scope are stored in the `request` object.

session

Objects with session scope are accessible from pages processing requests in the same session as the one in which they were created. It is illegal to define an object with session scope from within a page that is not session-aware. All references to the object are released after the associated session ends.

application

Objects with application scope are accessible from pages processing requests in the same application as the one in which they were created. The `application` object is the servlet context obtained from the servlet configuration object.

Section 13. Expression language

Expression language

Expression language is used for run-time assignment of values to action element attributes. It was first introduced as part of Java Standard Tag Library (JSTL) 1.0 specification, but is now part of the JSP 2.0 specification.

As part of JSTL, you could only use expression language with JSTL actions. Now, as an integral part of the JSP 2.0 specification, you can use EL with template text, as well as with standard and custom actions.

Expression language is inspired by both ECMAScript and XPath expression languages, and uses the features of both languages, as well as introduces some new ones. For example, expression language performs data-type conversions automatically.

The main advantage of using EL in JSPs is to enforce writing scriptless JSPs. You can do this through the configuration element `scripting-invalid`. As discussed in JSP configuration on page 30, making the value of this element true allows the use of expression language, but prohibits the user from using Java scriptlets, Java expressions, or Java declaration elements within JSPs.

Let's cover some EL basics.

Implicit objects

EL expressions support several implicit objects. The following table describes these objects.

Implicit object	Description
`pageScope`	A map of all page-scoped variables and their values
`requestScope`	A map of all request-scoped variables and their values
`sessionScope`	A map of all session-scoped variables and their values
`applicationScope`	A map of all application-scoped variables and their values
`pageContext`	An object of the `pageContext` class
`param`	A map of all request parameter values wherein each parameter is mapped to a single String value
`paramValues`	A map of all request parameter values wherein each parameter is mapped to a String array
`header`	A map of all request header values wherein each header is mapped to a single String value

`headerValues`	A map of all request header values wherein each header is mapped to a String array
`cookie`	A map of all request cookie values wherein each cookie is mapped to a single `javax.servlet.http.Cookie` value
`initParam`	A map of all application initialization parameter values wherein each parameter is mapped to a single String value

---

Examples of implicit objects

Let's consider an example detailing the use of implicit objects. The table below describes how you can access and resolve information like request attributes, session attributes, and request parameters (first column of the table) using EL (second column of the table). The third column displays the result of the resolution.

Source JSP parameter/attribute	**Target JSP expression**	**Result**
`request.setAttribute ("reqAttr", "iAmReqAttr");`	`${requestScope["reqAttr"]}`	iAmReqAttr
`session.setAttribute ("sessAttr", "iAmSessAttr");`	`${sessionScope["sessAttr"]}`	iAmSessAttr
`request parameter "customerName" having value "Martin"`	`${param.customerName}`	Martin
`Header browser information`	`${header.user-agent}`	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.1.4322)
`Header host information`	`${header["host"]}`	localhost:8888

Syntax of expression language

The syntax of expression language is quite simple:

- You can use a `[]` operator to access properties of JavaBeans objects, lists, or arrays of objects.
- You can also use a `.` operator to access the properties of a JavaBean object.
- You can use arithmetic operators for computations.

- You can use standard Java relational operators for relational comparisons.
- Logical operators are also available for your use.
- Literals of boolean, integer, floating point, string, and null are available.
- You can also use conditional operators for conditional processing.

Let's discuss the above syntactic rules in detail.

Arithmetic operators

Expression language can use the following five arithmetic operators to act on integer and floating point values:

- `+` operator for addition
- `-` operator for subtraction
- `*` operator for multiplication
- `/` operator for division
- `%` operator for remainder

The table below describes the arithmetic operators through some examples.

Expression using arithmetic operator	**Result of the expression**
`${2.7 + 5.6}`	8.3
`${-2 - 7}`	-9
`${9/2}`	4.5
`${10%4}`	2

Relational operators

Relational operators operate on two operands and always return a boolean result. You can use the following relational operators with EL:

- == operator for evaluating "equal condition"
- != operator for evaluating "are not equal condition"
- < operator for evaluating "less than" condition
- > operator for evaluating "greater than" condition
- <= operator for evaluating "less than equal to" condition
- >= operator for evaluating "greater than equal to" condition

The table below describes the relational operators through some examples.

Expression using relational operator	Result of the expression
`${10 == 2*5}`	true
`${10 > 4}`	true
`${10 < 10/2}`	false
`${10 <= 20/2}`	true
`${10 != 2*5}`	false

Logical operators

Logical operators operate on two expressions and always return a boolean value. Expression language can use the following logical operators:

- && operator returns true if both expressions evaluate to true.
- || operator returns true if one of the expressions evaluate to true.
- ! operator returns the inverse of the evaluation result.

The table below describes the logical operators through some examples.

Expression using logical operator	Result of the expression
`${10 > 4 && 4 < 16}`	true
`${10 > 4 && 16 < 5}`	false
`${10 > 4 || 16 < 5}`	true
`${!(10 > 4)}`	false

Conditional operator

This operator is used for conditional processing. Depending on the boolean result of the condition, one of the two possible results is returned.

The following examples show conditional operator usage.

Expression using conditional operator	Result of the expression
${(5<6) ? 5 : 6}	5
${(5>6) ? 5 : 6}	6

[] and . operators

As discussed above, you use `[]` and `.` operators to look for a named property in a bean or in a collection. Let's consider an example where `customer` is the name of a bean with property `SSN`.

To access the property `SSN`, you can use the following expression:

```
${customer['SSN']}
```

The value within the brackets must be a string literal for the property's name or a variable that holds the property's name.

You can even use a complete EL expression that resolves to a property.

Here are a few general rules that exist while evaluating `expr-a[expr-b]`:

- Evaluate `expr-a` into `value-a`.
- If `value-a`is `null`, return `null`.
- Evaluate `expr-b` into `value-b`.
- If `value-b` is `null`, return `null`.
- If `value-a` is a map, list, or array, then evaluate whether `value-b` resolves to a property for it.

Using the `.` operator, the alternative syntax could be:

```
${customer.SSN}
```

Section 14. Actions

Actions

Actions provide a higher level of functionality than the declarations, expressions, and scriptlets you'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 that use JavaBeans components
- Those that control run-time forwarding/including
- Those that prepare HTML for the Java plug-in

Also, JSP 2.0 introduced two standard actions you can use only in tag files. These actions are <jsp:invoke> and <jsp:doBody>. For a complete discussion of tag files, refer to Tag files on page 56.

JSPs and JavaBeans

JSP 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 46	Prepare a bean for use within the JSP
`jsp:setProperty` on page 47	Set one or more bean properties on a bean
`jsp:getProperty` on page 48	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 this tutorial's scope, 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* you want to make

settable, the class must have a method whose signature is:

```
public void set N (T);
```

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

```
public T get N ();
```

---

jsp:useBean

`jsp:useBean` tells the JSP that you want a bean of a given name (which might be a request-time expression) and scope. You also provide creation information. The JSP 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 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 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 recreated 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* and will be recreated 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` might 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 the following table shows.

Property expression	Meaning
`property="*"`	All bean properties for which there is an HTTP request parameter with the same name is 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" value="propertyValue"`	Sets the specified property to the specified string value, which will be coerced to the property's type. The value might 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 you use `jsp:setProperty` to set values into a scripting variable, you use `jsp:getProperty` 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 converts 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 that uses the bean-related actions:

HelloWorld13.jsp

```
<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, but locally declared beans can also be quite 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

Next, you use a `jsp:useBean` action, so you can use a bean within your JSP:

```
<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 you 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 you. 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, because the bean exists only for the lifetime of the page, each request creates it anew, and, therefore, you 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 your bean and indicates that it wants to set the property named `value` to World. This means that the default is to greet the world. But how do you greet someone else? That is when the HTML form and the other `jsp:setProperty` tag enter the scenario:

```
<%-- Whatever HTTP parameters you 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 your 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, you only 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, your JSP 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 similar to 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 your bean is 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 particularly special going on here, now that you understand how these tags work. The form simply defines a single input field. Whatever value you enter into that field is put into your bean by the `jsp:setProperty` tag. The form also uses a `jsp:getProperty` tag to initialize the field, as a convenience.

JSPs can be smart forms

Did you notice that no form action is associated with the sample form?

The form submits back to your JSP, which handles the form directly. JSPs 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 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 your JSP is posted back to your JSP for processing.

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

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

Action	Purpose
`jsp:include` on page 52	Include the referenced resource's content within the including page's content
`jsp:forward` on page 52	Substitute the referenced resource's content for the forwarding page's content
`jsp:param` on page 53	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 dynamic (for example, a JSP, 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 the JSP that whatever output you've written into the stream so far should be committed. The reason for this is that the output could be buffered, and you need to flush your buffer before you let someone else write to the output stream.

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

`jsp:forward`

The `jsp:forward` action tells the JSP container that you want to forward the request to another resource whose content will substitute for your content. The resource is specified by a relative URL and can be dynamic or static.

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 dynamic or static.

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

`jsp:param`

The `jsp:param` action provides parameters that 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 that are passed through the `request` object to the receiving resource.

`jsp:include` example

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

UseHeader.jsp

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

The included page, head.jsp, is as follows:

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. Bergman/DEVTECH All Rights Reserved." >
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
</HEAD>
```

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

When you request UseHeader.jsp, that JSP 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 you could add as many as you want.

You use the `jsp:param` action to pass your desired title to head.jsp, where you use the request object to get it. If the parameter is not provided, the page still generates a default TITLE tag for you. 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 the ability to plug newer JVM versions 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, 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 the component requires
`nspluginurl`	A URL from which to download a Navigator-specific plug-in
`iepluginurl`	A URL from which to download an IE-specific plug-in

`jsp:plugin` example

The following `jsp:plugin` and <APPLET> tags 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 above, `jsp:action` is similar to the <APPLET> tag. It's slightly more verbose, but permits the server to generate the appropriate tags based on the requesting browser.

Section 15. Tag files

Overview of tag files

JSP 2.0 introduced tag files. These are essentially tag extensions you can invoke via your JSPs.

Unlike with JSP 1.x, where page authors had to write tag handlers to generate tag extensions, with JSP 2.0, page authors do not need Java knowledge to write tag extensions. Now with tag files, you can write tag extensions using only JSP syntax. It echoes the purist belief that page authors should concentrate on presentation and not on server-side languages to write Web pages.

Let's discuss tag file basics.

Basics of tag files

The required file extension for tag files is .tag. Similar to how JSPs can contain JSP fragments, tag files can also contain tag fragments. The extension for such tag fragments is .tagx.

For a JSP container to recognize these tag files, you must place them in one of the two possible locations:

- If your JSP files and tag files are part of a JAR file, you can place tag files in the /META-INF/tags/ directory (or a subdirectory of /META-INF/tags/) in a JAR file installed in the /WEB-INF/lib/ directory of the Web application.

- If your JSP files and tag files are not part of a JAR file, you can place your tag files in the /WEB-INF/tags/ directory (or a subdirectory of /WEB-INF/tags/) of the Web application.

Tag files placed in locations other than the ones above are not considered tag extensions, and, according to JSP 2.0 specification, the JSP container will ignore them.

Tag tiles bundled in a JAR require a tag library descriptor (TLD). Per the JSP 2.0 specification, tag files not described in TLD but packaged in the JAR must be ignored by the JSP container.

To describe tags within a tag library, JSP 2.0 introduced a new TLD element: `<tag-file>`. This element requires `<name>` and `<path>` sub-elements that define the tag name and the full path of the tag file from the root of the JAR.

Directives applicable to tag files

JSP 2.0 introduced three new directives: the `<tag>` directive, the `<attribute>` directive, and the `<variable>` directive. These directives are available only to tag files. Let's examine these directives in detail.

Directive	Usage
`tag`	The tag directive is similar to the page directive, but applies to tag files instead of JSPs. Apart from attributes like language, import, and pageEncoding that are also present in page directive, the tag directive has some attributes applicable to tag files only. These are the important ones: ° **`display-name`:** It is a short name intended to be displayed by tools. Defaults to the name of the tag file without the .tag extension. ° **`body-content`:** Provides information on the content of the body of this tag. Defaults to scriptless. ° **`dynamic-attributes`:** The presence of this attribute indicates this tag supports additional attributes with dynamic names. A discussion on dynamic attributes is outside this tutorial's scope.
`attribute`	The attribute directive allows the declaration of custom action attributes. Usage of the attribute directive is explained in more detail in the next section. You can use the following attributes with the attribute directive: ° **`name`:** This is the unique name of the attribute being declared. This is a required attribute. ° **`required`:** Whether this attribute is required (true) or optional (false). The default is false. ° **`fragment`:** Whether this attribute is a fragment that should be evaluated by the tag handler (true) or a normal attribute that should be evaluated by the container prior to being passed to the tag handler. ° **`rtexprvalue`:** Whether the attribute's value might be dynamically calculated at run time by a scriptlet expression. ° **`type`:** The run-time type of the attribute's value. Defaults to `java.lang.String` if not specified.

	◦ **`description`:** Description of the attribute.
`variable`	The variable directive defines the details of a variable exposed by the tag handler to the calling page.

Tag file implementation

Let's discuss a simple example that displays the total premium for an automobile insurance policy. First assume that this automobile policy can have `liability` coverage or `thirdparty` coverage, and the total premium is the sum of these coverages.

So, you have a JSP that is the parent page and a tag file that takes individual premiums for `liability` coverage and `thirdparty` coverage and displays the total premium:

automobile-policy.jsp

```
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
<p>
Automobile-Policy Page
</p>
<tags:coverages liability="2000" thirdparty="4500" >
<p>
The total premium will be:
</p>
</tags:coverages>
```

The above JSP defines tags as the prefix for the tag files contained inside the /WEB-INF/tags folder. `tags:coverages` invokes the tag file coverages.tag. Notice that `tags:coverages` passes the values of `liability` premium and `thirdparty` premium as attributes. The tag file coverages.tag calculates the total premium value:

coverages.tag

```
<%@ tag body-content="scriptless" %>
<%@ attribute name="liability" required="false" %>
<%@ attribute name="thirdparty" required="false" %>
<jsp:doBody/>
<p> ${liability + thirdparty} </p>
```

The code above first defines the `tag` directive that provides information that the tag's content is scriptless. It does not contain any scripting elements. The lines below use the `attribute` directive and define the two attributes `liability` and `thirdparty`. `required` has a value of false that suggests that these attributes are not mandatory.

The `<jsp:doBody/>` is a new standard action introduced in JSP 2.0. This action suggests that the JSP contains content between `<tags:coverages>` and `</tags:coverages>`. If the JSP does not contain any content, you must define the value of the `body-content` attribute as empty instead of scriptless.

The last line in the tag file uses JSP 2.0 expression language to calculate the total premium.

To run this code, place coverages.tag inside the /WEB-INF/tags/ folder of the Web application.

Section 16. Alternative technologies

Competing technologies

For many years, JSP has enjoyed a near total monopoly as the preferred technology for generating client-side presentation logic. However, many other technologies are now competing for a foothold in this arena. Some of these are:

- **WebMacro --** A 100-percent Java open source template language
- **Tapestry --** A powerful, open source, all-Java language framework for creating leading-edge Web applications in the Java language
- **FreeMarker --** A template engine written purely in the Java language
- **Apache Cocoon --** A Web development framework built around the concepts of separation of concerns and component-based Web development
- **Velocity --** An open source templating solution

Velocity

Of the above technologies, Velocity is the one creating ripples in Java/open source circles. So, let's look at Velocity more closely.

Velocity is an open source templating solution you can use in place of JSP for creating dynamic Web pages. Working with Velocity involves a template that contains the static portion of the output and placeholders for the dynamic content.

For example, a Velocity template might contain lines like these:

```
<html>
<head>
</head>
<body>
#set( $this = "Velocity") This page is generated using $this.
</body>
</html>
```

As you can see, the template is a complete HTML file. The last line above contains a placeholder `$this`. This placeholder is resolved at run time by a templating engine, and the output is something like this:

```
This page is generated using Velocity.
```

This seems easy and straightforward. Yes, it is.

Velocity has several advantages over JSP:

- Velocity's biggest advantage is that it is interpreted at run time. This means you can read templates from anywhere, even from a database, and the changes are instantly reflected.
- Velocity helps enforce a clean separation between the view layer and the model/control layers.
- Velocity, which caches templates for speed, reportedly has performance comparable or better than JSP.
- Velocity templates are not limited to HTML, and you can use it to produce any type of text output, including XML, SQL, ASCII, and PostScript.

But JSP scores over Velocity in other areas:

- JSP 2.0 supports JSTL. So, JSP has the advantage of existing taglibs that make usage easy.
- In JSP 2.0, expression language makes coding JSPs simple.
- In JSP 2.0, tag files make writing custom tags easy.
- JSP inherently works with server-side code.
- JSP has long been a standard with a large following of experienced programmers.

Ultimately, the decision to use Velocity or JSP depends on the programming model you're developing. In fact, new technologies drive the industry. For more information about Velocity, see Resources on page62 .

Section 17. Wrap up and resources

Summary

We have completed an overview of the JSP specification, including updates in JSP 2.0. It is important to understand that this introduction presented what you *can* do with JSP. This is not necessarily the same as what you *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 is separating presentation from business logic. To that end, many JSP practitioners maintain that it's bad to put any Java code into a JSP, on the grounds that it's not maintainable by nonprogramming 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 permit "simple" expressions; others do 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, this tutorial's target audience is not the nonprogramming Web designer; it is the Java programmer who wants to employ JSP in content delivery solutions. It is just as likely that a reader might be involved in developing, say, new WYSIWYG editor tools that generate JSPs that use JSP constructs internally while at the same time hiding them from the user. And it is certainly the case that rapid prototyping might 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 you can use to implement JSP solutions or how you can use JSP to manage the look and feel associated with your content. We recommend that you review other documents and more advanced tutorials that offer such information.

Resources

- Download the Code_IntroToJSP.zip source code used in this tutorial.
- Download the *JSP 2.0 specification* from the Sun JSP site.
- 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).

- Refer to the article *Developing Web applications with JavaServer Pages 2.0* for further discussion of JSP 2.0.

- See *Creating JSP 2.0 tag files* (Oracle Technology Network) for an in-depth discussion about JSP 2.0 tag files.

- Refer to *JSP 2.0: The new deal, Part 1* (ONJava.com, November 2003) for new features of JSP 2.0.

- For more on Velocity, see *Client and server-side templating with Velocity* (developerWorks, February 2004).

- Read about JSP taglibs in Noel J. Bergman's *JSP taglibs: Better usability by design* (developerWorks, December 2001).

- Brett McLaughlin shows you how make the best use of JSP in his article *JSP best practices: Combine JavaBeans components and JSP* (developerWorks, May 2003).

- See the entire *JSP best practices series* by Brett McLaughlin.

- Learn more about expression language in *A JSTL primer, Part 1: The expression language* (developerWorks, February 2003). The remainder of the series offers some excellent reading, as well: *Part 2* (March 2003), examines flow control and URL management through custom tags; *Part 3* (April 2003) covers formatting and internationalization through custom tags; and *Part 4* (May 2003) introduces custom tag libraries for exchanging XML and database content in JSP pages.

- *Chapter 15* of the Java Language Specification examines expressions in detail.

- Rick Hightower's comprehensive tutorial will have you *Mastering JSP custom tags* (developerWorks, June 2004) in no time.

- Interested in Java servlets? Roy Miller's *Introduction to Java Servlet technology* (developerWorks, February 2005) will get you started out right.

Your feedback

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.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .