JavaServer Pages (JSP) 1.0

---

1. Overview

JavaServer Pages (JSP) lets you separate the dynamic part of your pages from the static HTML. You simply write the regular HTML in the normal manner, using whatever Web-page-building tools you normally use. You then enclose the code for the dynamic parts in special tags, most of which start with "<%" and end with "%>". For example, here is a section of a JSP page that results in something like "Thanks for ordering *Core Web Programming*" for a URL of `http://host/OrderConfirmation.jsp?title=Core+Web+Programming`:

```
Thanks for ordering
<I><%= request.getParameter("title") %></I>
```

You normally give your file a .jsp extension, and typically install it in any place you could place a normal Web page. Although what you write often looks more like a regular HTML file than a servlet, behind the scenes, the JSP page just gets converted to a normal servlet, with the static HTML simply being printed to the output stream associated with the servlet's `service` method. This is normally done the first time the page is requested, and developers can simply request the page themselves when first installing it if they want to be sure that the first real user doesn't get a momentary delay when the JSP page is translated to a servlet and the servlet is compiled and loaded. Note also that many Web servers let you define aliases that so that a URL that appears to reference an HTML file really points to a servlet or JSP page.

Aside from the regular HTML, there are three main types of JSP constructs that you embed in a page: scripting elements, directives, and actions. *Scripting elements* let you specify Java code that will become part of the resultant servlet, *directives* let you control the overall structure of the servlet, and *actions* let you specify existing components that should be used, and otherwise control the behavior of the JSP engine. To simplify the scripting elements, you have access to a number of predefined variables such as `request` in the snippet above.

Note that this tutorial covers version 1.0 of the JSP specification. JSP has changed dramatically since version 0.92, and although these changes were almost entirely for the better, you should note that version 1.0 JSP pages are almost totally incompatible with the earlier JSP engines. Note that this JSP tutorial is part of a larger tutorial on servlets and JSP at http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/.

2. Syntax Summary

JSP Element	Syntax	Interpretation	Notes
JSP Expression	`<%= expression %>`	Expression is evaluated and placed in output.	XML equivalent is `<jsp:expression>` `expression` `</jsp:expression>`. Predefined variables are `request`, `response`, `out`, `session`, `application`, `config`, and `pageContext` (available in scriptlets also).
JSP Scriptlet	`<% code %>`	Code is inserted in `service` method.	XML equivalent is `<jsp:scriptlet>` `code` `</jsp:scriptlet>`.
JSP Declaration	`<%! code %>`	Code is inserted in body of servlet class, outside of `service` method.	XML equivalent is `<jsp:declaration>` `code` `</jsp:declaration>`.
JSP `page` Directive	`<%@ page att="val" %>`	Directions to the servlet engine about general setup.	XML equivalent is `<jsp:directive.page att="val"\>`. Legal attributes, with default values in bold, are: • import="*package.class*" • contentType="*MIME-Type*" • isThreadSafe="**true**|false" • session="**true**|false" • buffer="*size*kb|none" • autoflush="**true**|false" • extends="*package.class*" • info="*message*" • errorPage="*url*" • isErrorPage="true|**false**" • language="java"
JSP `include` Directive	`<%@ include file="url" %>`	A file on the local system to be included when the JSP page is translated into a servlet.	XML equivalent is `<jsp:directive.include file="url"\>`. The URL must be a relative one. Use the `jsp:include` action to include a file at request time instead of translation time.
JSP Comment	`<%-- comment --%>`	Comment; ignored when JSP page is translated into	If you want a comment in the resultant HTML, use regular HTML comment syntax of `<-- comment -->`.

		servlet.	
The `jsp:include` Action	`<jsp:include page="`*`relative URL`*`" flush="true"/>`	Includes a file at the time the page is requested.	If you want to include the file at the time the page is translated, use the `page` directive with the `include` attribute instead. **Warning:** on some servers, the included file must be an HTML file or JSP file, as determined by the server (usually based on the file extension).

The `jsp:useBean` Action	`<jsp:useBean att=val*/>` or
	`<jsp:useBean att=val*>`
	`...`
	`</jsp:useBean>`
		Find or build a Java Bean.	Possible attributes are:
			• id="*name*"
			• scope="**page**|request|session|application"
			• class="*package.class*"
			• type="*package.class*"
			• beanName="*package.class*"

The `jsp:setProperty` Action	`<jsp:setProperty att=val*/>`	Set bean properties, either explicitly or by designating that value comes from a request parameter.	Legal attributes are • name="*beanName*" • property="*propertyName*|*" • param="*parameterName*" • value="*val*"
The `jsp:getProperty` Action	`<jsp:getProperty name="`*`propertyName`*`" value="`*`val`*`"/>`	Retrieve and output bean properties.	
The `jsp:forward` Action	`<jsp:forward page="`*`relative URL`*`"/>`	Forwards request to another page.	

The `jsp:plugin` Action	`<jsp:plugin attribute="`*`value`*`"*>`
	`...`
	`</jsp:plugin>`
		Generates `OBJECT` or `EMBED` tags, as appropriate to the browser type, asking that an applet be run using the Java Plugin.	

3. Template Text: Static HTML

In many cases, a large percent of your JSP page just consists of static HTML, known as *template text*. In all respects except one, this HTML looks just like normal HTML, follows all the same syntax rules, and is simply "passed through" to the client by the servlet created to handle the page. Not only does the HTML *look* normal, it can be *created* by whatever tools you already are using for building Web pages. For example, I used Allaire's HomeSite for most of the JSP pages in this tutorial.

The one minor exception to the "template text is passed straight through" rule is that, if you want to have "`<%`" in the output, you need to put "`<\%`" in the template text.

4. JSP Scripting Elements

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

1. Expressions of the form `<%= expression %>` that are evaluated and inserted into the output,
2. Scriptlets of the form `<% code %>` that are inserted into the servlet's `service` method, and
3. Declarations of the form `<%! code %>` that are inserted into the body of the servlet class, outside of any existing methods.

Each of these is described in more detail below.

4.1 JSP Expressions

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

```
<%= Java Expression %>
```

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

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

To simplify these expressions, there are a number of predefined variables that you can use. These implicit objects are discussed in more detail later, but for the purpose of expressions, the most important ones are:

- `request`, the `HttpServletRequest`;
- `response`, the `HttpServletResponse`;
- `session`, the `HttpSession` associated with the request (if any); and
- `out`, the `PrintWriter` (a buffered version of type `JspWriter`) used to send output to the client.

Here's an example:

```
Your hostname: <%= request.getRemoteHost() %>
```

Finally, note that XML authors can use an alternative syntax for JSP expressions:

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

Remember that XML elements, unlike HTML ones, are case sensitive. So be sure to use lowercase.

4.2 JSP Scriptlets

If you want to do something more complex than insert a simple expression, JSP *scriptlets* let you insert arbitrary code into the servlet method that will be built to generate the page. Scriptlets have the following form:
<% Java Code **%>**
Scriptlets have access to the same automatically defined variables as expressions. So, for example, if you want output to appear in the resultant page, you would use the `out` variable.
```
<%
String queryData = request.getQueryString();
out.println("Attached GET data: " + queryData);
%>
```
Note that code inside a scriptlet gets inserted *exactly* as written, and any static HTML (template text) before or after a scriptlet gets converted to `print` statements. This means that scriptlets need not contain complete Java statements, and blocks left open can affect the static HTML outside of the scriptlets. For example, the following JSP fragment, containing mixed template text and scriptlets
```
<% if (Math.random() < 0.5) { %>
Have a <B>nice</B> day!
<% } else { %>
Have a <B>lousy</B> day!
<% } %>
```
will get converted to something like:
```
if (Math.random() < 0.5) {
  out.println("Have a <B>nice</B> day!");
} else {
  out.println("Have a <B>lousy</B> day!");
}
```
If you want to use the characters "%>" inside a scriptlet, enter "%\>" instead. Finally, note that the XML equivalent of `<% Code %>` is
```
<jsp:scriptlet>
Code
</jsp:scriptlet>
```

4.3 JSP Declarations

A JSP *declaration* lets you define methods or fields that get inserted into the main body of the servlet class (outside of the `service` method processing the request). It has the following form:
<%! Java Code **%>**
Since declarations do not generate any output, they are normally used in conjunction with JSP expressions or scriptlets. For example, here is a JSP fragment that prints out the number of times the current page has been requested since the server booted (or the servlet class was changed and reloaded):
```
<%! private int accessCount = 0; %>
Accesses to page since server reboot:
<%= ++accessCount %>
```
As with scriptlets, if you want to use the characters "%>", enter "%\>" instead. Finally, note that the XML equivalent of `<%! Code %>` is
```
<jsp:declaration>
Code
</jsp:declaration>
```

5. JSP Directives

A JSP *directive* affects the overall structure of the servlet class. It usually has the following form:
<%@ directive attribute="value" **%>**
However, you can also combine multiple attribute settings for a single directive, as follows:
```
<%@ directive attribute1="value1"
              attribute2="value2"
              ...
              attributeN="valueN" %>
```
There are two main types of directive: `page`, which lets you do things like import classes, customize the servlet superclass, and the like; and `include`, which lets you insert a file into the servlet class at the time the JSP file is translated into a servlet. The specification also mentions the `taglib` directive, which is not supported in JSP version 1.0, but is intended to let JSP authors define their own tags. It is expected that this will be the main new contribution of JSP 1.1.

5.1 The JSP page Directive

The `page` directive lets you define one or more of the following case-sensitive attributes:

- `import="`*`package.class`*`"` or `import="`*`package.class1,...,package.classN`*`"`. This lets you specify what packages should be imported. For example:
  `<%@ page import="java.util.*" %>`
  The `import` attribute is the only one that is allowed to appear multiple times.
- `contentType="`*`MIME-Type`*`"` or
  `contentType="`*`MIME-Type`*`; charset=`*`Character-Set`*`"`
  This specifies the MIME type of the output. The default is `text/html`. For example, the directive
  `<%@ page contentType="text/plain" %>`
  has the same effect as the scriptlet
  `<% response.setContentType("text/plain"); %>`
- `isThreadSafe="true|false"`. A value of `true` (the default) indicates normal servlet processing, where multiple requests can be processed simultaneously with a single servlet instance, under the assumption that the author synchronized access to instance variables. A value of `false` indicates that the servlet should implement `SingleThreadModel`, with requests either delivered serially or with simultaneous requests being given separate servlet instances.
- `session="true|false"`. A value of `true` (the default) indicates that the predefined variable `session` (of type `HttpSession`) should be bound to the existing session if one exists, otherwise a new session should be created and bound to

it. A value of `false` indicates that no sessions will be used, and attempts to access the variable `session` will result in errors at the time the JSP page is translated into a servlet.
- `buffer="`*`size`*`kb|none"`. This specifies the buffer size for the `JspWriter` `out`. The default is server-specific, but must be at least `8kb`.
- `autoflush="true|false"`. A value of `true`, the default, indicates that the buffer should be flushed when it is full. A value of `false`, rarely used, indicates that an exception should be thrown when the buffer overflows. A value of `false` is illegal when also using `buffer="none"`.
- `extends="`*`package.class`*`"`. This indicates the superclass of servlet that will be generated. Use this with extreme caution, since the server may be using a custom superclass already.
- `info="`*`message`*`"`. This defines a string that can be retrieved via the `getServletInfo` method.
- `errorPage="`*`url`*`"`. This specifies a JSP page that should process any `Throwable`s thrown but not caught in the current page.
- `isErrorPage="true|false"`. This indicates whether or not the current page can act as the error page for another JSP page. The default is `false`.
- `language="java"`. At some point, this is intended to specify the underlying language being used. For now, don't bother with this since `java` is both the default and the only legal choice.

The XML syntax for defining directives is
`<jsp:directive.`*`directiveType`* `attribute=value />`
For example, the XML equivalent of
`<%@ page import="java.util.*" %>`
is
`<jsp:directive.page import="java.util.*" />`

5.2 The JSP `include` Directive

This directive lets you include files at the time the JSP page is translated into a servlet. The directive looks like this:
`<%@ include file="relative url" %>`
The URL specified is normally interpreted relative to the JSP page that refers to it, but, as with relative URLs in general, you can tell the system to interpret the URL relative to the home directory of the Web server by starting the URL with a forward slash. The contents of the included file are parsed as regular JSP text, and thus can include static HTML, scripting elements, directives, and actions.

For example, many sites include a small navigation bar on each page. Due to problems with HTML frames, this is usually implemented by way of a small table across the top of the page or down the left-hand side, with the HTML repeated for each page in the site. The `include` directive is a natural way of doing this, saving the developers from the maintenance nightmare of actually copying the HTML into each separate file. Here's some representative code:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Servlet Tutorial: JavaServer Pages (JSP) 1.0</TITLE>
<META NAME="author" CONTENT="webmaster@somesite.com">
<META NAME="keywords" CONTENT="...">
<META NAME="description" CONTENT="...">
<LINK REL=STYLESHEET
      HREF="Site-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<%@ include file="/navbar.html" %>

<!-- Part specific to this page ... -->

</BODY>
</HTML>
```

Note that since the `include` directive inserts the files at the time the page is translated, if the navigation bar changes, you need to re-translate all the JSP pages that refer to it. This is a good compromise in a situation like this, since the navigation bar probably changes infrequently, and you want the inclusion process to be as efficient as possible. If, however, the included files changed more often, you could use the `jsp:include` action instead. This includes the file at the time the JSP page is requested, and is discussed in the tutorial section on JSP actions.

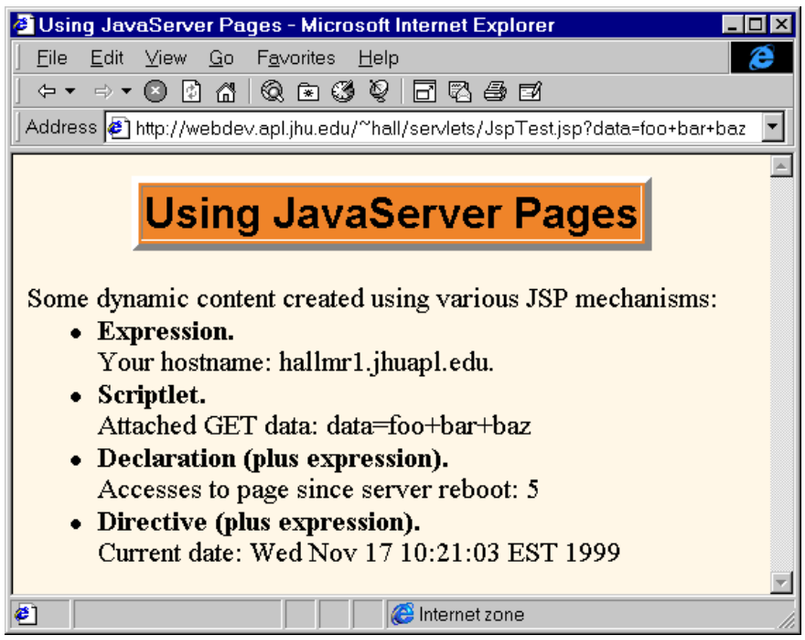
6. Example Using Scripting Elements and Directives

Here is a simple example showing the use of JSP expressions, scriptlets, declarations, and directives. You can also download the source or try it on-line.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Using JavaServer Pages</TITLE>

<META NAME="author" CONTENT="Marty Hall -- hall@apl.jhu.edu">
<META NAME="keywords"
      CONTENT="JSP,JavaServer Pages,servlets">
<META NAME="description"
      CONTENT="A quick example of the four main JSP tags.">
<LINK REL=STYLESHEET
      HREF="My-Style-Sheet.css"
      TYPE="text/css">
</HEAD>

<BODY BGCOLOR="#FDF5E6" TEXT="#000000" LINK="#0000EE"
      VLINK="#551A8B" ALINK="#FF0000">
```

```
<CENTER>
<TABLE BORDER=5 BGCOLOR="#EF8429">
  <TR><TH CLASS="TITLE">
      Using JavaServer Pages</TABLE>
</CENTER>
<P>

Some dynamic content created using various JSP mechanisms:
<UL>
  <LI><B>Expression.</B><BR>
      Your hostname: <%= request.getRemoteHost() %>.
  <LI><B>Scriptlet.</B><BR>
      <% out.println("Attached GET data: " +
                     request.getQueryString()); %>
  <LI><B>Declaration (plus expression).</B><BR>
      <%! private int accessCount = 0; %>
      Accesses to page since server reboot: <%= ++accessCount %>
  <LI><B>Directive (plus expression).</B><BR>
      <%@ page import = "java.util.*" %>
      Current date: <%= new Date() %>
</UL>


</BODY>
</HTML>
```

Here's a typical result:



7. Predefined Variables

To simplify code in JSP expressions and scriptlets, you are supplied with eight automatically defined variables, sometimes called *implicit objects*. The available variables are `request`, `response`, `out`, `session`, `application`, `config`, `pageContext`, and `page`. Details for each are given below.

7.1 request

This is the `HttpServletRequest` associated with the request, and lets you look at the request parameters (via `getParameter`), the request type (`GET`, `POST`, `HEAD`, etc.), and the incoming HTTP headers (cookies, `Referer`, etc.). Strictly speaking, request is allowed to be a subclass of `ServletRequest` other than `HttpServletRequest`, if the protocol in the request is something other than HTTP. This is almost never done in practice.

7.2 response

This is the `HttpServletResponse` associated with the response to the client. Note that, since the output stream (see `out` below) is buffered, it *is* legal to set HTTP status codes and response headers, even though this is not permitted in regular servlets once any output has been sent to the client.

7.3 out

This is the `PrintWriter` used to send output to the client. However, in order to make the `response` object (see the previous section) useful, this is a buffered version of `PrintWriter` called `JspWriter`. Note that you can adjust the buffer size, or even turn buffering off, through use of the `buffer` attribute of the `page` directive. This was discussed in Section 5. Also note that `out` is used almost exclusively in scriptlets, since JSP expressions automatically get placed in the output stream, and thus rarely need to refer to `out` explicitly.

7.4 session

This is the `HttpSession` object associated with the request. Recall that sessions are created automatically, so this variable is bound

even if there was no incoming session reference. The one exception is if you use the `session` attribute of the `page` directive (see Section 5) to turn sessions off, in which case attempts to reference the session variable cause errors at the time the JSP page is translated into a servlet.

7.5 application

This is the `ServletContext` as obtained via `getServletConfig().getContext()`.

7.6 config

This is the `ServletConfig` object for this page.

7.7 pageContext

JSP introduced a new class called `PageContext` to encapsulate use of server-specific features like higher performance `JspWriter`s. The idea is that, if you access them through this class rather than directly, your code will still run on "regular" servlet/JSP engines.

7.8 page

This is simply a synonym for `this`, and is not very useful in Java. It was created as a placeholder for the time when the scripting language could be something other than Java.

8. Actions

JSP *actions* use constructs in XML syntax to control the behavior of the servlet engine. You can dynamically insert a file, reuse JavaBeans components, forward the user to another page, or generate HTML for the Java plugin. Available actions include:

- `jsp:include` - Include a file at the time the page is requested. See Section 8.1.
- `jsp:useBean` - Find or instantiate a JavaBean. See Section 8.2 for an overview, and Section 8.3 for details.
- `jsp:setProperty` - Set the property of a JavaBean. See Section 8.4.
- `jsp:getProperty` - Insert the property of a JavaBean into the output. See Section 8.5.
- `jsp:forward` - Forward the requester to a new page. See Section 8.6.
- `jsp:plugin` - Generate browser-specific code that makes an `OBJECT` or `EMBED` tag for the Java plugin. See Section 8.7.

These actions are described in more detail below. Remember that, as with XML in general, the element and attribute names are case sensitive.

8.1 The `jsp:include` Action

This action lets you insert files into the page being generated. The syntax looks like this:

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

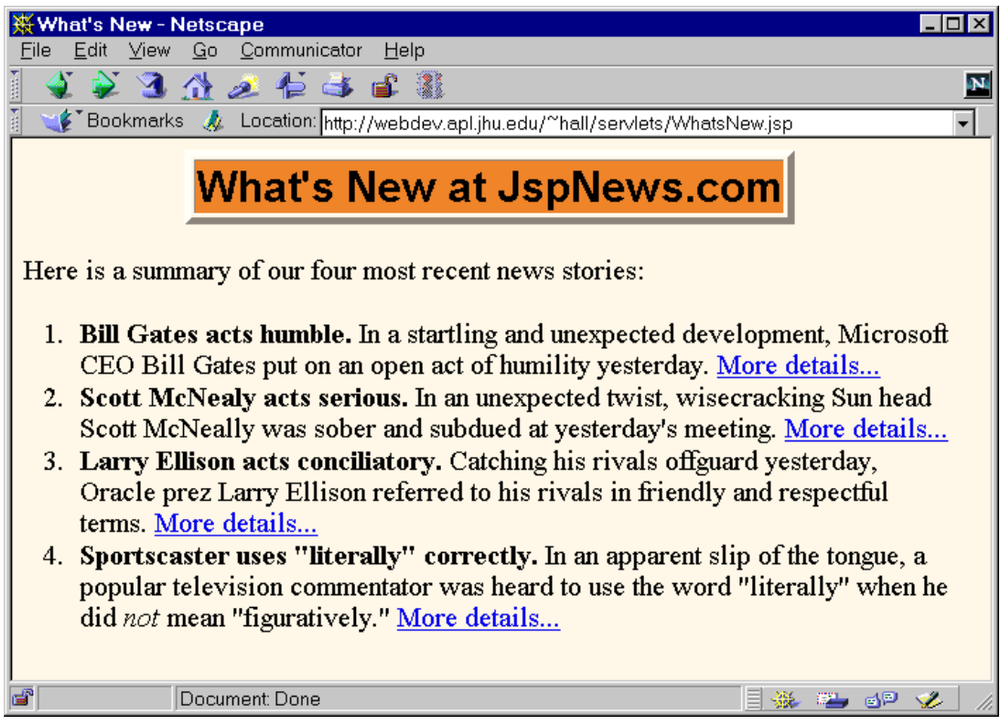
Unlike the include directive, which inserts the file at the time the JSP page is translated into a servlet, this action inserts the file at the time the page is requested. This pays a small penalty in efficiency, and precludes the included page from containing general JSP code (it cannot set HTTP headers, for example), but it gains significantly in flexibility. For example, here is a JSP page that inserts four different snippets into a "What's New?" Web page. Each time the headlines change, authors only need to update the four files, but can leave the main JSP page unchanged.

WhatsNew.jsp

You can also download the source or try it on-line.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>What's New</TITLE>
<LINK REL=STYLESHEET
      HREF="My-Style-Sheet.css"
      TYPE="text/css">
</HEAD>

<BODY BGCOLOR="#FDF5E6" TEXT="#000000" LINK="#0000EE"
      VLINK="#551A8B" ALINK="#FF0000">

<CENTER>
<TABLE BORDER=5 BGCOLOR="#EF8429">
  <TR><TH CLASS="TITLE">
      What's New at JspNews.com</TABLE>
</CENTER>
<P>

Here is a summary of our four most recent news stories:
<OL>
  <LI><jsp:include page="news/Item1.html" flush="true"/>
  <LI><jsp:include page="news/Item2.html" flush="true"/>
  <LI><jsp:include page="news/Item3.html" flush="true"/>
  <LI><jsp:include page="news/Item4.html" flush="true"/>
</OL>
</BODY>
</HTML>
```

Here's a typical result:



8.2 The `jsp:useBean` Action

This action lets you load in a JavaBean to be used in the JSP page. This is a a very useful capability because it lets you exploit the reusability of Java classes without sacrificing the convenience that JSP adds over servlets alone. The simplest syntax for specifying that a bean should be used is:

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

This usually means "instantiate an object of the class specified by `class`, and bind it to a variable with the name specified by `id`." However, as we'll see shortly, you can specify a `scope` attribute that makes the bean associated with more than just the current page. In that case, it is useful to obtain references to existing beans, and the `jsp:useBean` action specifies that a new object is instantiated only if there is no existing one with the same `id` and `scope`. Now, once you have a bean, you can modify its properties via `jsp:setProperty`, or by using a scriptlet and calling a method explicitly on the object with the variable name specified earlier via the `id` attribute. Recall that with beans, when you say "this bean has a property of type*X* called `foo`", you really mean "this class has a method called `getFoo` that returns something of type *X*, and another method called `setFoo` that takes an *X* as an argument." The `jsp:setProperty` action is discussed in more detail in the next section, but for now note that you can either supply an explicit `value`, give a `param` attribute to say that the value is derived from the named request parameter, or just list the property to indicate that the value should be derived from the request parameter with the same name as the property. You read existing properties in a JSP expression or scriptlet by calling the appropriate *`getXxx`* method, or more commonly, by using the `jsp:getProperty` action.

Note that the class specified for the bean must be in the server's *regular* class path, not the part reserved for classes that get automatically reloaded when they change. For example, in the Java Web Server, it and all the classes it uses should go in the `classes` directory or be in a jar file in the `lib` directory, not be in the `servlets` directory.

Here is a very simple example that loads a bean and sets/gets a simple `String` parameter.

BeanTest.jsp

You can also download the source or try it on-line.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Reusing JavaBeans in JSP</TITLE>
<LINK REL=STYLESHEET
      HREF="My-Style-Sheet.css"
      TYPE="text/css">
</HEAD>

<BODY>

<CENTER>
<TABLE BORDER=5>
  <TR><TH CLASS="TITLE">
      Reusing JavaBeans in JSP</TABLE>
</CENTER>
<P>

<jsp:useBean id="test" class="hall.SimpleBean" />
<jsp:setProperty name="test"
                 property="message"
                 value="Hello WWW" />

<H1>Message: <I>
<jsp:getProperty name="test" property="message" />
</I></H1>
```

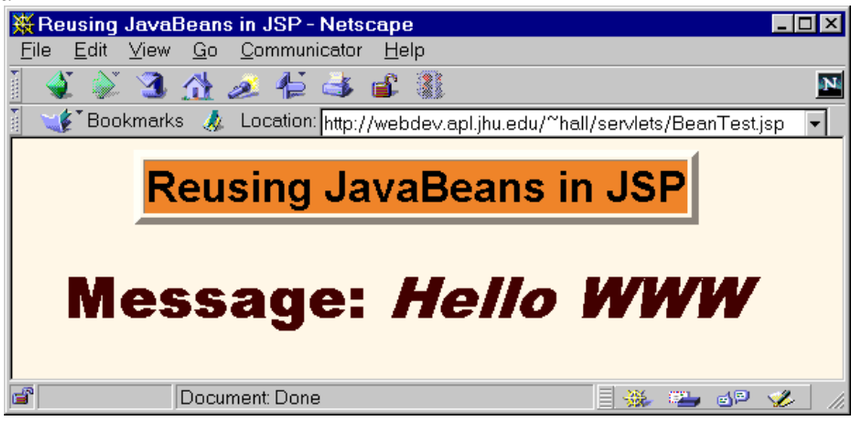
```
</BODY>
</HTML>
```

SimpleBean.java

Here's the source code for the bean used in the BeanTest JSP page. You can also download the source.

```
package hall;

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

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

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

Here's a typical result:



8.3 More `jsp:useBean` Details

The simplest way to use a bean is to use

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

to load the bean, then use `jsp:setProperty` and `jsp:getProperty` to modify and retrieve bean properties. However, there are two other options. First, you can use the container format, namely

```
<jsp:useBean ...>
  Body
</jsp:useBean>
```

to indicate that the *Body* portion should be executed only when the bean is first instantiated, not when an existing bean is found and used. As discussed below, beans can be shared, so not all `jsp:useBean` statements result in a new bean being instantiated. Second, in addition to `id` and `class`, there are three other attributes that you can use: `scope`, `type`, and `beanName`. These attributes are summarized in the following table.

Atribute	Usage
`id`	Gives a name to the variable that will reference the bean. A previous bean object is used instead of instantiating a new one if one can be found with the same `id` and `scope`.
`class`	Designates the full package name of the bean.
`scope`	Indicates the context in which the bean should be made available. There are four possible values: `page`, `request`, `session`, and `application`. The default, `page`, indicates that the bean is only available on the current page (stored in the `PageContext` of the current page). A value of `request` indicates that the bean is only available for the current client request (stored in the `ServletRequest` object). A value of `session` indicates that the object is available to all pages during the life of the current `HttpSession`. Finally, a value of `application` indicates that it is available to all pages that share the same `ServletContext`. The reason that the scope matters is that a `jsp:useBean` entry will only result in a new object being instantiated if there is no previous object with the same `id` and `scope`. Otherwise the previously existing object is used, and any `jsp:setParameter` elements or other entries between the `jsp:useBean` start and end tags will be ignored.
`type`	Specifies the type of the variable that will refer to the object. This must match the classname or be a superclass or an interface that the class implements. Remember that the *name* of the variable is designated via the `id` attribute.
`beanName`	Gives the name of the bean, as you would supply it to the `instantiate` method of `Beans`. It is permissible to supply a `type` and a `beanName`, and omit the `class` attribute.

8.4 The `jsp:setProperty` Action

You use `jsp:setProperty` to give values to properties of beans that have been referenced earlier. You can do this in two contexts. First, you can use `jsp:setProperty` after, but outside of, a `jsp:useBean` element, as below:

```
<jsp:useBean id="myName" ... />
...
<jsp:setProperty name="myName"
                 property="someProperty" ... />
```

In this case, the `jsp:setProperty` is executed regardless of whether a new bean was instantiated or an existing bean was found. A second context in which `jsp:setProperty` can appear is inside the body of a `jsp:useBean` element, as below:

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

```
  <jsp:setProperty name="myName"
                   property="someProperty" ... />
</jsp:useBean>
```
Here, the `jsp:setProperty` is executed only if a new object was instantiated, not if an existing one was found.

There are four possible attributes of `jsp:setProperty`:

Attribute	Usage
`name`	This required attribute designates the bean whose property will be set. The `jsp:useBean` element must appear before the `jsp:setProperty` element.
`property`	This required attribute indicates the property you want to set. However, there is one special case: a value of `"*"` means that all request parameters whose names match bean property names will be passed to the appropriate setter methods.
`value`	This optional attribute specifies the value for the property. String values are automatically converted to numbers, `boolean`, `Boolean`, `byte`, `Byte`, `char`, and `Character` via the standard `valueOf` method in the target or wrapper class. For example, a value of `"true"` for a `boolean` or `Boolean` property will be converted via `Boolean.valueOf`, and a value of `"42"` for an `int` or `Integer` property will be converted via `Integer.valueOf`. You can't use both `value` and `param`, but it is permissible to use neither. See the discussion of `param` below.

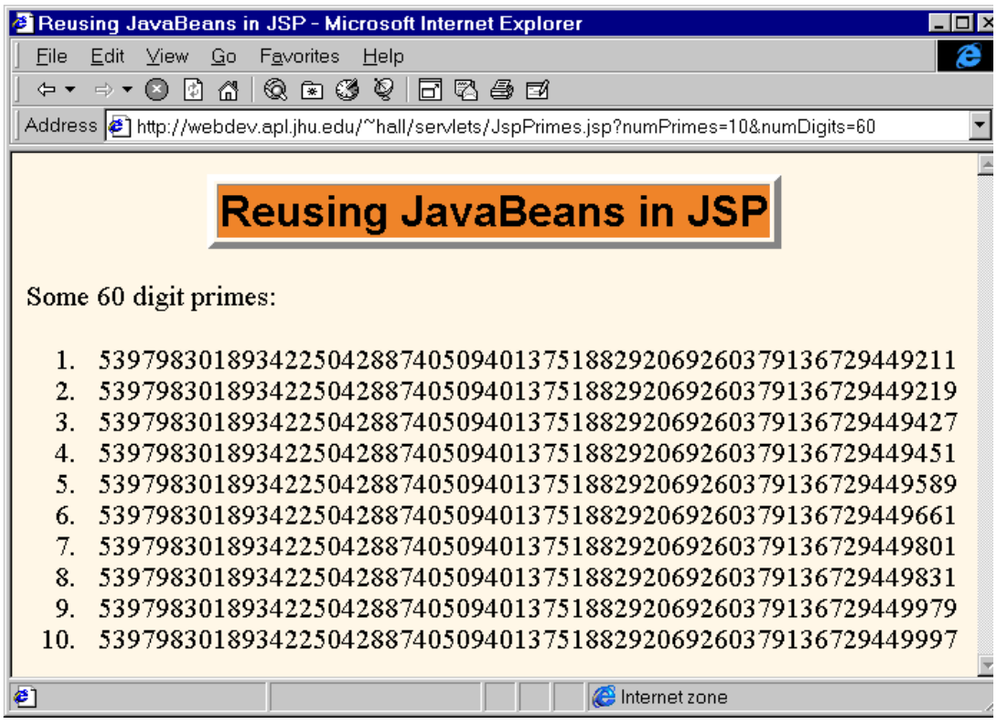
`param`	This optional attribute designates the request parameter from which the property should be derived. If the current request has no such parameter, nothing is done: the system does *not* pass `null` to the setter method of the property. Thus, you can let the bean itself supply default values, overriding them only when the request parameters say to do so. For example, the following snippet says "set the `numberOfItems` property to whatever the value of the `numItems` request parameter is, if there is such a request parameter. Otherwise don't do anything."
	`<jsp:setProperty name="orderBean"`
	`property="numberOfItems"`
	`param="numItems" />`
	If you omit both `value` and `param`, it is the same as if you supplied a `param` name that matches the `property` name. You can take this idea of automatically using the request property whose name matches the property one step further by supplying a property name of `"*"` and omitting both `value` and `param`. In this case, the server iterates through available properties and request parameters, matching up ones with identical names.

Here's an example that uses a bean to create a table of prime numbers. If there is a parameter named `numDigits` in the request data, it is passed into the bean's `numDigits` property. Likewise for `numPrimes`.

JspPrimes.jsp

To download the JSP source, right click on the source code link. You can also download the source code for the `NumberedPrimes` bean referenced by the `jsp:useBean` element. Browse the source code directory for other Java classes used by `NumberedPrimes`. The best way to try it out on-line is to start with the HTML page that acts as a front end to it.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Reusing JavaBeans in JSP</TITLE>
<LINK REL=STYLESHEET
      HREF="My-Style-Sheet.css"
      TYPE="text/css">
</HEAD>

<BODY>

<CENTER>
<TABLE BORDER=5>
  <TR><TH CLASS="TITLE">
      Reusing JavaBeans in JSP</TABLE>
</CENTER>
<P>

<jsp:useBean id="primeTable" class="hall.NumberedPrimes" />
<jsp:setProperty name="primeTable" property="numDigits" />
<jsp:setProperty name="primeTable" property="numPrimes" />

Some <jsp:getProperty name="primeTable" property="numDigits" />
digit primes:
<jsp:getProperty name="primeTable" property="numberedList" />

</BODY>
</HTML>
```
Here's a typical result:



8.5 The `jsp:getProperty` Action

This element retrieves the value of a bean property, converts it to a string, and inserts it into the output. The two required attributes are name, the name of a bean previously referenced via jsp:useBean, and property, the property whose value should be inserted. Here's an example; for more examples, see Sections 8.2 and 8.4.

```
<jsp:useBean id="itemBean" ... />
...
<UL>
  <LI>Number of items:
      <jsp:getProperty name="itemBean" property="numItems" />
  <LI>Cost of each:
      <jsp:getProperty name="itemBean" property="unitCost" />
</UL>
```

8.6 The `jsp:forward` Action

This action lets you forward the request to another page. It has a single attribute, `page`, which should consist of a relative URL. This could be a static value, or could be computed at request time, as in the two examples below.

```
<jsp:forward page="/utils/errorReporter.jsp" />
<jsp:forward page="<%= someJavaExpression %>" />
```

8.7 The `jsp:plugin` Action

This action lets you insert the browser-specific `OBJECT` or `EMBED` element needed to specify that the browser run an applet using the Java plugin.

9. Comments and Character Quoting Conventions

There are a small number of special constructs you can use in various cases to insert comments or characters that would otherwise be treated specially. Here's a summary:

Syntax	Purpose
`<%-- comment --%>`	A JSP comment. Ignored by JSP-to-scriptlet translator. Any embedded JSP scripting elements, directives, or actions are ignored.
`<!-- comment -->`	An HTML comment. Passed through to resultant HTML. Any embedded JSP scripting elements, directives, or actions are executed normally.
`<\%`	Used in template text (static HTML) where you really want "<%".
`%\>`	Used in scripting elements where you really want "%>".
`\'`	A single quote in an attribute that uses single quotes. Remember, however, that you can use either single or double quotes, and the other type of quote will then be a regular character.
`\"`	A double quote in an attribute that uses double quotes. Remember, however, that you can use either single or double quotes, and the other type of quote will then be a regular character.
`%\>`	%> in an attribute.
`<\%`	<% in an attribute.