


---

# CONTROLLING THE STRUCTURE OF GENERATED SERVLETS: THE JSP PAGE DIRECTIVE

## Topics in This Chapter

- 
- Understanding the purpose of the `page` directive
  - Designating which classes are imported
  - Specifying the MIME type of the page
  - Generating Excel spreadsheets
  - Participating in sessions
  - Setting the size and behavior of the output buffer
  - Designating pages to handle JSP errors
  - Controlling threading behavior
  - Using XML-compatible syntax for directives

### Training courses from the book's author:

<http://courses.coreservlets.com/>

- *Personally* developed and taught by Marty Hall
- Available onsite at *your* organization (any country)
- Topics and pace can be customized for your developers
- Also available periodically at public venues
- Topics include Java programming, beginning/intermediate servlets and JSP, advanced servlets and JSP, Struts, JSF/MyFaces, Ajax, GWT, Ruby/Rails and more. Ask for custom courses!

---

# Chapter

# 12

## Training courses from the book's author:

<http://courses.coreservlets.com/>

- *Personally* developed and taught by Marty Hall
- Available onsite at *your* organization (any country)
- Topics and pace can be customized for your developers
- Also available periodically at public venues
- Topics include Java programming, beginning/intermediate servlets and JSP, advanced servlets and JSP, Struts, JSF/MyFaces, Ajax, GWT, Ruby/Rails and more. Ask for custom courses!

A JSP *directive* affects the overall structure of the servlet that results from the JSP page. The following templates show the two possible forms for directives. Single quotes can be substituted for the double quotes around the attribute values, but the quotation marks cannot be omitted altogether. To obtain quotation marks within an attribute value, precede them with a backslash, using `\'` for `'` and `\"` for `"`.

```
<%@ directive attribute="value" %>
```

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

In JSP, there are three main types of directives: `page`, `include`, and `taglib`. The `page` directive lets you control the structure of the servlet by importing classes, customizing the servlet superclass, setting the content type, and the like. A `page` directive can be placed anywhere within the document; its use is the topic of this chapter. The second directive, `include`, lets you insert a file into the JSP page at the time the JSP file is translated into a servlet. An `include` directive should be placed in the document at the point at which you want the file to be inserted; it is discussed in Chapter 13. The third directive, `taglib`, defines custom markup tags; it is discussed at great length in Volume 2 of this book, where there are several chapters on custom tag libraries.

The `page` directive lets you define one or more of the following case-sensitive attributes (listed in approximate order of frequency of use): `import`, `contentType`, `pageEncoding`, `session`, `isELIgnored` (JSP 2.0 only), `buffer`, `autoFlush`,

info, errorPage, isErrorPage, isThreadSafe, language, and extends. These attributes are explained in the following sections.

## 12.1 The import Attribute

The import attribute of the page directive lets you specify the packages that should be imported by the servlet into which the JSP page gets translated. As discussed in Section 11.3 (Limiting the Amount of Java Code in JSP Pages) and illustrated in Figure 12-1, using separate utility (helper) classes makes your dynamic code easier to write, maintain, debug, test, and reuse.

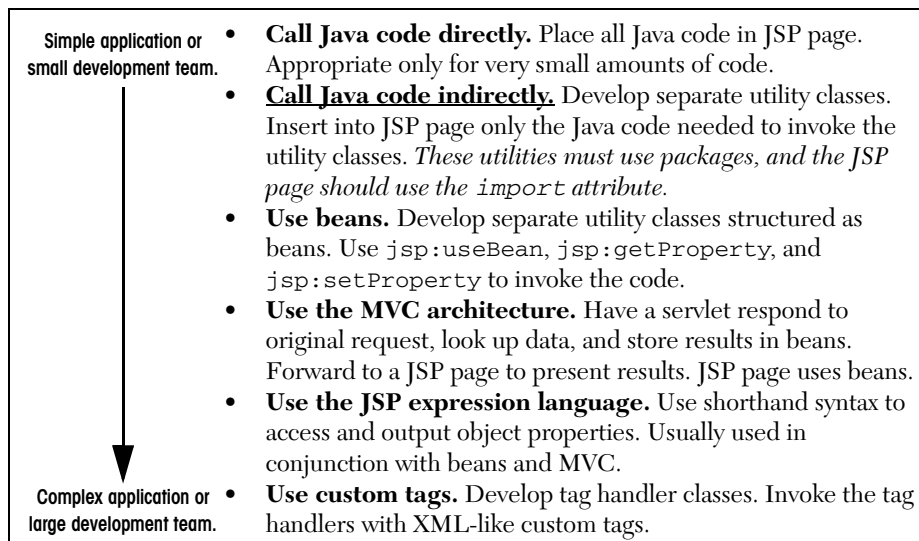


Figure 12-1 Strategies for invoking dynamic code from JSP.

When you use utility classes, remember that they should *always* be in packages. For one thing, packages are a good strategy on any large project because they help protect against name conflicts. With JSP, however, packages are absolutely required. The reason is that, in the absence of packages, classes you reference are assumed to be in the same package as the current class. For example, suppose that a JSP page contains the following scriptlet.

```
<% Test t = new Test(); %>
```

Now, if `Test` is in an imported package, there is no ambiguity. But, if `Test` is not in a package or the package to which `Test` belongs is not explicitly imported, then

the system will assume that `Test` is in the same package as the autogenerated servlet. The problem is that the autogenerated servlet's package is not known! It is quite common for servers to create servlets whose package is determined by the directory in which the JSP page is placed. Other servers use different approaches. So, you simply cannot rely on packageless classes working properly. The same argument applies to beans (Chapter 14), since beans are just classes that follow some simple naming and structure conventions.

### Core Approach

---

*Always put your utility classes and beans in packages.*

---



By default, the servlet imports `java.lang.*`, `javax.servlet.*`, `javax.servlet.jsp.*`, `javax.servlet.http.*`, and possibly some number of server-specific entries. Never write JSP code that relies on any server-specific classes being imported automatically; doing so makes your code nonportable.

Use of the `import` attribute takes one of the following two forms.

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

For example, the following directive signifies that all classes in the `java.util` package should be available to use without explicit package identifiers.

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

The `import` attribute is the only page attribute that is allowed to appear multiple times within the same document. Although `page` directives can appear anywhere within the document, it is traditional to place `import` statements either near the top of the document or just before the first place that the referenced package is used.

Note that, although the JSP pages go in the normal HTML directories of the server, the classes you write that are used by JSP pages must be placed in the special Java-code directories (e.g., `.../WEB-INF/classes/directoryMatchingPackageName`). See Sections 2.10 (Deployment Directories for Default Web Application: Summary) and 2.11 (Web Applications: A Preview) for information on these directories.

For example, Listing 12.1 presents a page that illustrates each of the three scripting elements from the previous chapter. The page uses three classes not in the standard JSP import list: `java.util.Date`, `coreservlets.CookieUtilities` (see Listing 8.3), and `coreservlets.LongLivedCookie` (see Listing 8.4). So, to simplify references to these classes, the JSP page uses

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

Figures 12–2 and 12–3 show some typical results.

**Listing 12.1** ImportAttribute.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>The import Attribute</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H2>The import Attribute</H2>
<!-- JSP page Directive --%>
<%@ page import="java.util.*,coreservlets.*" %>
<!-- JSP Declaration --%>
<%!
private String randomID() {
    int num = (int)(Math.random()*10000000.0);
    return("id" + num);
}
private final String NO_VALUE = "<I>No Value</I>";
%>
<!-- JSP Scriptlet --%>
<%
String oldID =
    CookieUtilities.getCookieValue(request, "userID", NO_VALUE);
if (oldID.equals(NO_VALUE)) {
    String newID = randomID();
    Cookie cookie = new LongLivedCookie("userID", newID);
    response.addCookie(cookie);
}
%>
<!-- JSP Expressions --%>
This page was accessed on <%= new Date() %> with a userID
cookie of <%= oldID %>.
</BODY></HTML>
```

---

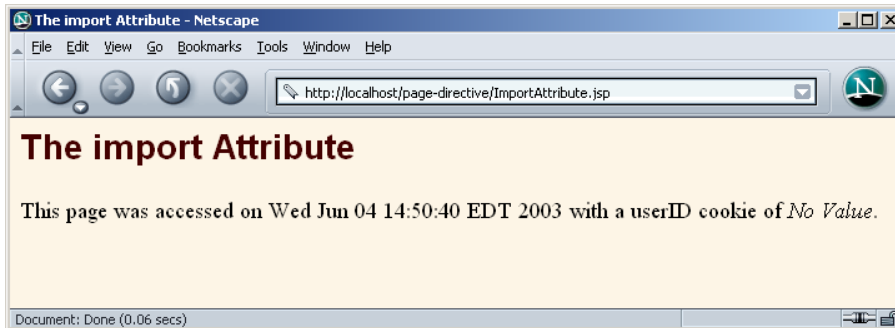


Figure 12-2 ImportAttribute.jsp when first accessed.

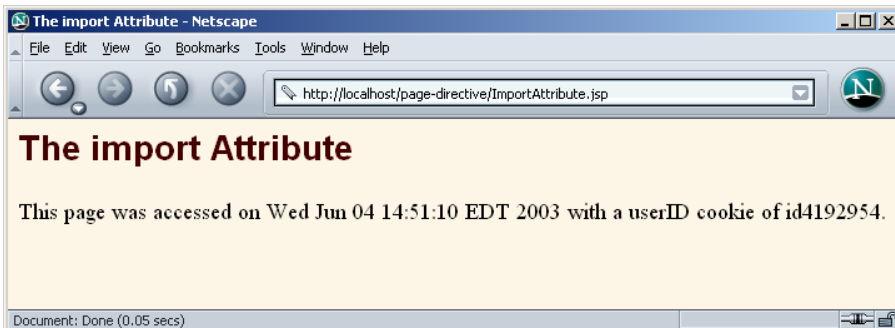


Figure 12-3 ImportAttribute.jsp when accessed in a subsequent request.

## 12.2 The contentType and pageEncoding Attributes

The `contentType` attribute sets the `Content-Type` response header, indicating the MIME type of the document being sent to the client. For more information on MIME types, see Table 7.1 (Common MIME Types) in Section 7.2 (Understanding HTTP 1.1 Response Headers).

Use of the `contentType` attribute takes one of the following two forms.

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

For example, the directive

```
<%@ page contentType="application/vnd.ms-excel" %>
```

has the same basic effect as the scriptlet

```
<% response.setContentType("application/vnd.ms-excel"); %>
```

The first difference between the two forms is that `response.setContentType` uses explicit Java code (an approach some developers try to avoid), whereas the `page` directive uses only JSP syntax. The second difference is that directives are parsed specially; they don't directly become `_jspService` code at the location at which they appear. This means that `response.setContentType` can be invoked conditionally whereas the `page` directive cannot be. Setting the content type conditionally is useful when the same content can be displayed in different forms—for an example, see the next section (Conditionally Generating Excel Spreadsheets).

Unlike regular servlets, for which the default MIME type is `text/plain`, the default for JSP pages is `text/html` (with a default character set of `ISO-8859-1`). Thus, JSP pages that output HTML in a Latin character set need not use `contentType` at all. If you want to change both the content type and the character set, you can do the following.

```
<%@ page contentType="someMimeType; charset=someCharacterSet" %>
```

However, if you only want to change the character set, it is simpler to use the `pageEncoding` attribute. For example, Japanese JSP pages might use the following.

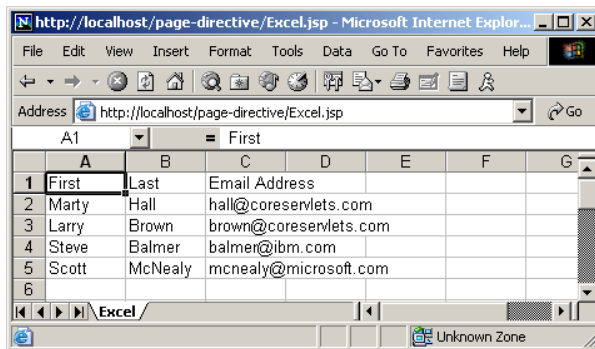
```
<%@ page pageEncoding="Shift_JIS" %>
```

## Generating Excel Spreadsheets

Listing 12.2 shows a JSP page that uses the `contentType` attribute and tab-separated data to generate Excel output. Note that the `page` directive and comment are at the bottom so that the carriage returns at the ends of the lines don't show up in the Excel document. (Note: JSP does not ignore white space—JSP usually generates HTML in which most white space is ignored by the browser, but JSP itself maintains the white space and sends it to the client.) Figure 12-4 shows the result in Internet Explorer on a system that has Microsoft Office installed.

**Listing 12.2** Excel.jsp

```
First   Last   Email Address
Marty  Hall   hall@coreservlets.com
Larry  Brown  brown@coreservlets.com
Steve  Balmer balmer@ibm.com
Scott  McNealy mcnealy@microsoft.com
<%@ page contentType="application/vnd.ms-excel" %>
<%-- There are tabs, not spaces, between columns. --%>
```

**Figure 12-4** Excel document (Excel.jsp) in Internet Explorer.

## 12.3 Conditionally Generating Excel Spreadsheets

In most cases in which you generate non-HTML content with JSP, you know the content type in advance. In those cases, the `contentType` attribute of the `page` directive is appropriate: it requires no explicit Java syntax and can appear anywhere in the page.

Occasionally, however, you may want to build the same content, but change the listed content type depending on the situation. For example, many word processing and desktop publishing systems can import HTML pages. So, you could arrange to have the page come up either in the publishing system or in the browser, depending on the content type you send. Similarly, Microsoft Excel can import tables that are represented in HTML with the `TABLE` tag. This capability suggests a simple method of returning either HTML or Excel content, depending on which the user prefers: just use an HTML table and set the content type to `application/vnd.ms-excel` only if the user requests the results in Excel.



Unfortunately, this approach brings to light a small deficiency in the page directive: attribute values cannot be computed at runtime, nor can page directives be conditionally inserted as can template text. So, the following attempt results in Excel content regardless of the result of the `checkUserRequest` method.

```
<% boolean usingExcel = checkUserRequest(request); %>
<% if (usingExcel) { %>
<%@ page contentType="application/vnd.ms-excel" %>
<% } %>
```

Fortunately, there is a simple solution to the problem of conditionally setting the content type: just use scriptlets and the normal servlet approach of `response.setContentType`, as in the following snippet:

```
<%
String format = request.getParameter("format");
if ((format != null) && (format.equals("excel"))) {
    response.setContentType("application/vnd.ms-excel");
}
%>
```

For example, we once worked on a project that displayed financial (budget) information to authorized users. The data could be displayed in a table in a regular Web page if the user merely wanted to review it, or it could be placed into an Excel spreadsheet if the user wanted to put it into a report. When we first joined the project, there were two entirely separate pieces of code for each task. We changed it to build the same HTML table either way and to merely change the content type. Voila!

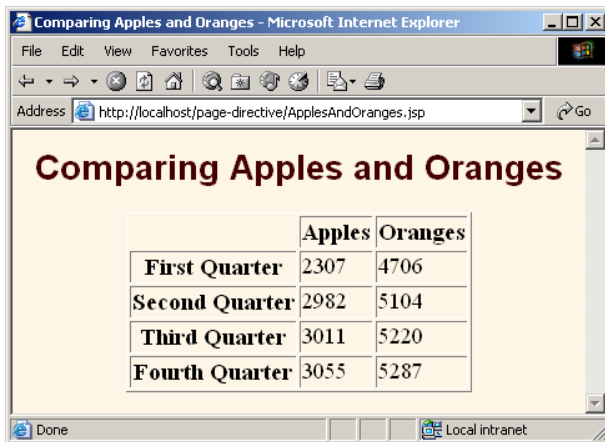
Listing 12.3 shows a page that uses this approach; Figures 12–5 and 12–6 show the results. In a real application, of course, the data would almost certainly come from a database. We use static values here for simplicity, but see Chapter 17 (Accessing Databases with JDBC) for information on talking to relational databases from servlets and JSP pages.

### Listing 12.3 ApplesAndOranges.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Comparing Apples and Oranges</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
```

**Listing 12.3** ApplesAndOranges.jsp (continued)

```
<BODY>
<CENTER>
<H2>Comparing Apples and Oranges</H2>
<%
String format = request.getParameter("format");
if ((format != null) && (format.equals("excel"))) {
    response.setContentType("application/vnd.ms-excel");
}
%>
<TABLE BORDER=1>
  <TR><TH></TH>          <TH>Apples<TH>Oranges
  <TR><TH>First Quarter <TD>2307 <TD>4706
  <TR><TH>Second Quarter<TD>2982 <TD>5104
  <TR><TH>Third Quarter <TD>3011 <TD>5220
  <TR><TH>Fourth Quarter<TD>3055 <TD>5287
</TABLE>
</CENTER></BODY></HTML>
```



**Figure 12-5** The default result of ApplesAndOranges.jsp is HTML content.

The screenshot shows a web browser window with the address bar containing `http://localhost/page-directive/ApplesAndOranges.jsp?format=excel...`. The browser displays an Excel spreadsheet with the following data:

	Apples	Oranges
First Quarter	2307	4706
Second Quarter	2982	5104
Third Quarter	3011	5220
Fourth Quarter	3055	5287

**Figure 12-6** Specifying `format=excel` for `ApplesAndOranges.jsp` results in Excel content.

## 12.4 The session Attribute

The `session` attribute controls whether the page participates in HTTP sessions. Use of this attribute takes one of the following two forms.

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

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

Using `session="false"` may save significant amounts of server memory on high-traffic sites. However, note that using `session="false"` does not *disable* session tracking—it merely prevents the JSP page from creating *new* sessions for users who don't have them already. So, since sessions are *user specific*, not *page specific*, it doesn't do any good to turn off session tracking for one page unless you also turn it off for related pages that are likely to be visited in the same client session.

## 12.5 The isELIgnored Attribute

The `isELIgnored` attribute controls whether the JSP 2.0 Expression Language (EL) is ignored (`true`) or evaluated normally (`false`). This attribute is new in JSP 2.0; it is illegal to use it in a server that supports only JSP 1.2 or earlier. The default value of the attribute depends on the version of `web.xml` you use for your Web application. If your `web.xml` specifies servlets 2.3 (corresponding to JSP 1.2) or earlier, the default is `true` (but it is still legal to change the default—you are permitted to use this attribute in a JSP-2.0-compliant server regardless of the `web.xml` version). If your `web.xml` specifies servlets 2.4 (corresponding to JSP 2.0) or later, the default is `false`. Use of this attribute takes one of the following two forms.

```
<%@ page isELIgnored="false" %>
<%@ page isELIgnored="true" %>
```

JSP 2.0 introduced a concise expression language for accessing request parameters, cookies, HTTP headers, bean properties and `Collection` elements from within a JSP page. For details, see Chapter 16 (Simplifying Access to Java Code: The JSP 2.0 Expression Language). Expressions in the JSP EL take the form `${expression}`. Normally, these expressions are convenient. However, what would happen if you had a JSP 1.2 page that, just by happenstance, contained a string of the form `${...}`? In JSP 2.0, this could cause problems. Using `isELIgnored="true"` prevents these problems.

## 12.6 The buffer and autoFlush Attributes

The `buffer` attribute specifies the size of the buffer used by the `out` variable, which is of type `jspWriter`. Use of this attribute takes one of two forms:

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

Servers can use a larger buffer than you specify, but not a smaller one. For example, `<%@ page buffer="32kb" %>` means the document content should be buffered and not sent to the client until at least 32 kilobytes are accumulated, the page is completed, or the output is explicitly flushed (e.g., with `response.flushBuffer`). The default buffer size is server specific, but must be at least 8 kilobytes. Be cautious about turning off buffering; doing so requires JSP elements that set headers or status codes to appear at the top of the file, before any HTML content. On the other hand,

disabling buffering or using a small buffer is occasionally useful when it takes a very long time to generate each line of the output; in this scenario, users would see each line as soon as it is ready, rather than waiting even longer to see groups of lines.

The `autoFlush` attribute controls whether the output buffer should be automatically flushed when it is full (the default) or whether an exception should be raised when the buffer overflows (`autoFlush="false"`). Use of this attribute takes one of the following two forms.

```
<%@ page autoFlush="true" %> <!-- Default --%>  
<%@ page autoFlush="false" %>
```

A value of `false` is illegal when `buffer="none"` is also used. Use of `autoFlush="false"` is exceedingly rare when the client is a normal Web browser. However, if the client is a custom application, you might want to guarantee that the application either receives a complete message or no message at all. A value of `false` could also be used to catch database queries that generate too much data, but it is generally better to place that logic in the data access code, not the presentation code.

## 12.7 The `info` Attribute

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

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

## 12.8 The `errorPage` and `isErrorPage` Attributes

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

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

The exception thrown will automatically be available to the designated error page by means of the `exception` variable.

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

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

For example, Listing 12.4 shows a JSP page that computes speed based on distance and time parameters. The page neglects to check whether the input parameters are missing or malformed, so an error could easily occur at runtime. However, the page designates `SpeedErrors.jsp` (Listing 12.5) as the page to handle errors that occur in `ComputeSpeed.jsp`, so the user does not receive the typical terse JSP error messages. Note that `SpeedErrors.jsp` is placed in the `WEB-INF` directory. Because servers prohibit direct client access to `WEB-INF`, this arrangement prevents clients from accidentally accessing `SpeedErrors.jsp` directly. When an error occurs, `SpeedErrors.jsp` is accessed by the *server*, not by the *client*: error pages of this sort do not result in `response.sendRedirect` calls, and the client sees only the URL of the originally requested page, not the URL of the error page.

Figures 12-7 and 12-8 show results when good and bad input parameters are received, respectively.

Note that the `errorPage` attribute designates *page-specific* error pages. To designate error pages that apply to an entire Web application or to various categories of errors within an application, use the `error-page` element in `web.xml`. For details, see Volume 2 of this book.

#### Listing 12.4 `ComputeSpeed.jsp`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Computing Speed</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<%@ page errorPage="/WEB-INF/SpeedErrors.jsp" %>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
    Computing Speed</TABLE>
<%!
// Note lack of try/catch for NumberFormatException if
// value is null or malformed.
```

**Listing 12.4** ComputeSpeed.jsp (*continued*)

```
private double toDouble(String value) {
    return(Double.parseDouble(value));
}
%>
<%
double furlongs = toDouble(request.getParameter("furlongs"));
double fortnights = toDouble(request.getParameter("fortnights"));
double speed = furlongs/fortnights;
%>
<UL>
    <LI>Distance: <%= furlongs %> furlongs.
    <LI>Time: <%= fortnights %> fortnights.
    <LI>Speed: <%= speed %> furlongs per fortnight.
</UL>
</BODY></HTML>
```

**Listing 12.5** SpeedErrors.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Error Computing Speed</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<%@ page isErrorPage="true" %>
<TABLE BORDER=5 ALIGN="CENTER">
    <TR><TH CLASS="TITLE">
        Error Computing Speed</TH>
</TR>
</TABLE>
<P>
ComputeSpeed.jsp reported the following error:
<I><%= exception %></I>. This problem occurred in the
following place:
<PRE>
<%@ page import="java.io.*" %>
<% exception.printStackTrace(new PrintWriter(out)); %>
</PRE>
</BODY></HTML>
```



Figure 12–7 ComputeSpeed.jsp when it receives legal values.

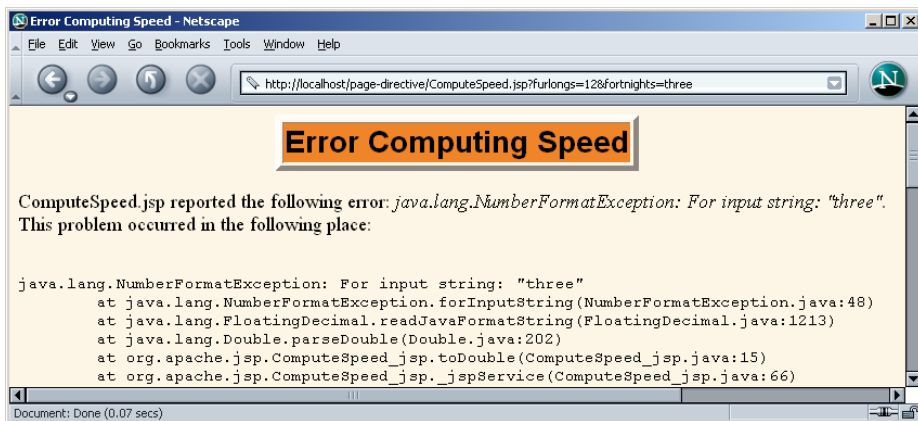


Figure 12–8 ComputeSpeed.jsp when it receives illegal values. Note that the address line shows the URL of ComputeSpeed.jsp, not SpeedErrors.jsp.

## 12.9 The `isThreadSafe` Attribute

The `isThreadSafe` attribute controls whether the servlet that results from the JSP page will allow concurrent access (the default) or will guarantee that no servlet instance processes more than one request at a time (`isThreadSafe="false"`). Use of the `isThreadSafe` attribute takes one of the following two forms.

```

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



Unfortunately, the standard mechanism for preventing concurrent access is to implement the `SingleThreadModel` interface (Section 3.7). Although `SingleThreadModel` and `isThreadSafe="false"` were recommended in the early days, recent experience has shown that `SingleThreadModel` was so poorly designed that it is basically useless. So, you should avoid `isThreadSafe` and use explicit synchronization instead.



### Core Warning

*Do not use `isThreadSafe`. Use explicit synchronization instead.*

To understand why `isThreadSafe="false"` is a bad idea, consider the following non-thread-safe snippet to compute user IDs. It is not thread safe since a thread could be preempted after reading `idNum` but before updating it, yielding two users with the same user ID.

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

The code should have used a `synchronized` block. This construct is written

```
synchronized(someObject) { ... }
```

and means that once a thread enters the block of code, no other thread can enter the same block (or any other block marked with the same object reference) until the first thread exits. So, the previous snippet should have been written in the following manner.

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

There are two reasons why this explicitly synchronized version is superior to the original version with the addition of `<%@ page isThreadSafe="false" %>`.

First, the explicitly synchronized version will probably have much better performance if the page is accessed frequently. The reason is that most JSP pages are not CPU limited but are I/O limited. So, while the system waits for I/O (e.g., a response from a database, the result of an EJB call, output sent over the network to the user), it should be doing something else. Since most servers implement `SingleThreadModel` by queueing up requests and handling them one at a time, high-traffic JSP pages can be *much* slower with this approach.

Even worse, the version that uses `SingleThreadModel` might not even get the right answer! Rather than queueing up requests, servers are permitted to implement `SingleThreadModel` by making a pool of servlet instances, as long as no instance is invoked concurrently. This, of course, totally defeats the purpose of using fields for persistence, since each instance would have a different field (instance variable), and multiple users could still get the same user ID. Defining the `idNum` field as `static` does not solve the problem either; the `this` reference would be different for each servlet instance, so the protection would be ineffective.

These problems are basically intractable. Give up. Forget `SingleThreadModel` and `isThreadSafe="false"`. Synchronize your code explicitly instead.

## 12.10 The extends Attribute

The `extends` attribute designates the superclass of the servlet that will be generated for the JSP page. It takes the following form.

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

This attribute is normally reserved for developers or vendors that implement fundamental changes to the way in which pages operate (e.g., to add in personalization features). Ordinary mortals should steer clear of this attribute except when referring to classes provided by the server vendor for this purpose.

## 12.11 The language Attribute

At some point, the `language` attribute is intended to specify the scripting language being used, as below.

```
<%@ page language="cobol" %>
```

For now, don't bother with this attribute since `java` is both the default and the only legal choice.

## 12.12 XML Syntax for Directives

If you are writing XML-compatible JSP pages, you can use an alternative XML-compatible syntax for directives as long as you don't mix the XML syntax and the classic syntax in the same page. These constructs take the following form:

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

For example, the XML equivalent of

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

is

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