Oracle9*i*

Support for JavaServer Pages Reference

Release 2 (9.2)

March 2002

Part No. A96657-01

Oracle9*i* Support for JavaServer Pages Reference, Release 2 (9.2)

Part No. A96657-01

Primary Author: Brian Wright

Contributing Author: Michael Freedman

Contributors: Julie Basu, Alex Yiu, Sunil Kunisetty, Gael Stevens, Ping Guo, YaQing Wang, Song Lin, Hal Hildebrand, Jasen Minton, Matthieu Devin, Jose Alberto Fernandez, Olga Peschansky, Jerry Schwarz, Clement Lai, Shinji Yoshida, Kenneth Tang, Robert Pang, Kannan Muthukkaruppan, Ralph Gordon, Shiva Prasad, Sharon Malek, Jeremy Lizt, Kuassi Mensah, Susan Kraft, Sheryl Maring, Ellen Barnes, Angie Long, Sanjay Singh, Olaf van der Geest

Contents

Send Us Your Comments

Oracle9*i* Support for JavaServer Pages Reference, Release 2 (9.2)

Part No. A96657-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: jpgcomment_us@oracle.com
- FAX: (650) 506-7225 Attn: Java Platform Group, Information Development Manager
- Postal service:
  Oracle Corporation
  Java Platform Group, Information Development Manager
  500 Oracle Parkway, Mailstop 4op9
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This document introduces and explains the Oracle implementation of JavaServer Pages (JSP) technology, specified by Sun Microsystems. It summarizes standard features, as specified by Sun, but focuses primarily on Oracle implementation details and value-added features.

The Oracle JSP container provided with Oracle9*i* release 2 is a complete implementation of the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

This preface contains these topics:

- Intended Audience
- Documentation Accessibility
- Organization
- Related Documentation
- Conventions

Important: Version 1.1.2.4 of the Oracle JSP container is supplied with Oracle9*i* release 2.

Intended Audience

This document is intended for developers interested in creating Web applications based on JavaServer Pages technology. It assumes that working Web and servlet environments already exist, and that readers are already familiar with the following:

- general Web technology
- general servlet technology
- how to configure their Web server and servlet environments
- HTML
- Java
- Oracle JDBC (for JSP applications accessing an Oracle database)
- Oracle SQLJ (for JSP database applications using SQLJ)

While some information about standard JSP 1.1 technology and syntax is provided in Chapter 1 and elsewhere, there is no attempt at completeness in this area. For additional information about standard JSP 1.1 features, consult the Sun Microsystems *JavaServer Pages Specification, Version 1.1* or other appropriate reference materials.

The JSP 1.1 specification relies on a servlet 2.2 environment, and servlet 2.2 features are discussed in this document. However, the JSP container provided with Oracle9*i* has special features for earlier servlet environments, and there is special discussion of these features in Chapter 9 as they relate to servlet 2.0 environments, particularly Apache JServ, which is also included with Oracle9*i*.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle Corporation is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation This documentation may contain links to Web sites of other companies or organizations that Oracle Corporation does not own or control. Oracle Corporation neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Organization

This document contains:

Chapter 1, "General Overview"

This chapter highlights standard JSP 1.1 technology. (It is not intended as a complete reference.)

Chapter 2, "Overview of the Oracle JSP Implementation"

This chapter summarizes Oracle JSP features and extensions, and introduces the JSP and servlet containers and the Web server provided with Oracle9*i* release 2.

Chapter 3, "Basics"

This chapter introduces basic JSP programming considerations and provides a starter sample for database access.

Chapter 4, "Key Considerations"

This chapter discusses a variety of general programming and configuration issues the developer should be aware of.

Chapter 5, "Oracle-Specific Programming Extensions"

This chapter covers Oracle-specific (non-portable) extensions supplied by the Oracle JSP container with Oracle9*i* release 2.

Chapter 6, "JSP Translation and Deployment"

This chapter covers Oracle JSP translation and deployment features and issues, and documents the `ojspc` pre-translation tool.

Chapter 7, "JSP Tag Libraries"

This chapter introduces the basic JSP 1.1 framework for custom tag libraries.

Chapter 8, "Oracle JSP Globalization Support"

This chapter discusses standard and Oracle-specific features for globalization support.

Chapter 9, "Oracle JSP in Apache JServ"

This appendix provides details of how to use the Oracle JSP container in the JServ servlet 2.0 environment, including required files, deployment, configuration, and special programming considerations.

Appendix A, "Getting Started in Alternative Environments"

This appendix covers configuration steps for alternative environments—Tomcat, from the Apache Software Foundation, and the Sun Microsystems JSWDK.

Appendix B, "Third Party Licenses"

This appendix includes the Third Party License for third party products included with Oracle9*i* release 2 and discussed in this document.

Related Documentation

Also available from the Oracle Java Platform group, for Oracle9*i* releases:

- *Oracle9i Java Developer's Guide*

  This book introduces the basic concepts of Java in Oracle9*i* and provides general information about server-side configuration and functionality. Information that pertains to the Oracle database Java environment in general, rather than to a particular product such as JDBC or SQLJ, is in this book.

- *Oracle9i JDBC Developer's Guide and Reference*

  This book covers programming syntax and features of the Oracle implementation of the JDBC standard (for Java Database Connectivity). This includes an overview of the Oracle JDBC drivers, details of the Oracle implementation of JDBC 1.22, 2.0, and 3.0 features, and discussion of Oracle JDBC type extensions and performance extensions.

- *Oracle9i SQLJ Developer's Guide and Reference*

  This book covers the use of SQLJ to embed static SQL operations directly into Java code, covering SQLJ language syntax and SQLJ translator options and features. Both standard SQLJ features and Oracle-specific SQLJ features are described.

- *Oracle9i JPublisher User's Guide*

  This book describes how to use the Oracle JPublisher utility to translate object types and other user-defined types to Java classes. If you are developing SQLJ or JDBC applications that use object types, VARRAY types, nested table types, or object reference types, then JPublisher can generate custom Java classes to map to them.

- *Oracle9i Java Stored Procedures Developer's Guide*

  This book discusses Java stored procedures—programs that run directly in the Oracle9*i* database. With stored procedures (functions, procedures, triggers, and SQL methods), Java developers can implement business logic at the server level, thereby improving application performance, scalability, and security.

The following OC4J documents, for Oracle9*i* Application Server releases, are also available from the Oracle Java Platform group:

- *Oracle9iAS Containers for J2EE User's Guide*

  This book provides some overview and general information for OC4J; primer chapters for servlets, JSP pages, and EJBs; and general configuration and deployment instructions.

- *Oracle9iAS Containers for J2EE Support for JavaServer Pages Reference*

  This book provides information for JSP developers who want to run their pages in OC4J. It includes a general overview of JSP standards and programming considerations, as well as discussion of Oracle value-added features and steps for getting started in the OC4J environment.

- *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*

  This book provides conceptual information and detailed syntax and usage information for tag libraries, JavaBeans, and other Java utilities provided with OC4J.

- *Oracle9iAS Containers for J2EE Servlet Developer's Guide*

  This book provides information for servlet developers regarding use of servlets and the servlet container in OC4J. It also documents relevant OC4J configuration files.

- *Oracle9iAS Containers for J2EE Services Guide*

  This book provides information about basic Java services supplied with OC4J, such as JTA, JNDI, and the Oracle9*i* Application Server Java Object Cache.

- *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide and Reference*

  This book provides information about the EJB implementation and EJB container in OC4J.

The following documents are from the Oracle Server Technologies group:

- *Oracle9i XML Developer's Kits Guide - XDK*
- *Oracle9i Application Developer's Guide - Fundamentals*
- *Oracle9i Supplied Java Packages Reference*
- *Oracle9i Supplied PL/SQL Packages and Types Reference*
- *PL/SQL User's Guide and Reference*
- *Oracle9i SQL Reference*
- *Oracle9i Net Services Administrator's Guide*
- *Oracle Advanced Security Administrator's Guide*
- *Oracle9i Database Reference*
- *Oracle9i Database Error Messages*
- *Oracle9i Sample Schemas*

The following documents from the Oracle9*i* Application Server group may also be of some interest:

- *Oracle9i Application Server Administrator's Guide*
- *Oracle Enterprise Manager Administrator's Guide*
- *Oracle HTTP Server Administration Guide*
- *Oracle9i Application Server Performance Guide*
- *Oracle9i Application Server Globalization Support Guide*

- *Oracle9iAS Web Cache Administration and Deployment Guide*
- *Oracle9i Application Server: Migrating from Oracle9i Application Server 1.x*

The following are available from the Oracle9*i* JDeveloper group:

- JDeveloper online help
- JDeveloper documentation on the Oracle Technology Network:

  `http://otn.oracle.com/products/jdev/content.html`

In North America, printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

`http://www.oraclebookshop.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://otn.oracle.com/admin/account/membership.html`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://otn.oracle.com/docs/index.htm`

To access the database documentation search engine directly, please visit

`http://tahiti.oracle.com`

The following Oracle Technology Network (OTN) resources are available for further information about JavaServer Pages:

- OTN Web site for Java servlets and JavaServer Pages:

  `http://otn.oracle.com/tech/java/servlets/`

- OTN JSP discussion forums, accessible through the following address:

  `http://www.oracle.com/forums/forum.jsp?id=399160`

The following resources are available from Sun Microsystems:

- Web site for JavaServer Pages, including the latest specifications:

  `http://java.sun.com/products/jsp/index.html`

- Web site for Java Servlet technology, including the latest specifications:

  `http://java.sun.com/products/servlet/index.html`

- `jsp-interest` discussion group for JavaServer Pages

  To subscribe, send an e-mail to `listserv@java.sun.com` with the following line in the body of the message:

  ```
  subscribe jsp-interest yourlastname yourfirstname
  ```

  It is recommended, however, that you request only the daily digest of the posted e-mails. To do this add the following line to the message body as well:

  ```
  set jsp-interest digest
  ```

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Italics	Italic typeface indicates book titles or emphasis, or terms that are defined in the text.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.

Convention	Meaning	Example
`UPPERCASE monospace (fixed-width) font`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.
`lowercase monospace (fixed-width) font`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the data files and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true.` Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase italic monospace (fixed-width) font`	Lowercase italic monospace font represents place holders or variables.	You can specify the *`parallel_clause`*. Run *`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = ’MIGRATE’;
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`

|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}`
		`[COMPRESS | NOCOMPRESS]`

`. . .`	Horizontal ellipsis points indicate either:	
	■ That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS `*`subquery`*`;`
	■ That you can repeat a portion of the code	`SELECT `*`col1`*`, `*`col2`*`, ... , `*`coln`*` FROM employees;`

Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);`
		`acct    CONSTANT NUMBER(4) := 3;`

`Italics`	Italicized text indicates place holders or variables for which you must supply particular values.	`CONNECT SYSTEM/`*`system_password`*
		`DB_NAME = `*`database_name`*

`UPPERCASE`	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

`lowercase`	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

1 General Overview

This chapter reviews standard features and functionality of JavaServer Pages technology. For further information, consult the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

(For an overview of Oracle-specific JSP features, see Chapter 2, "Overview of the Oracle JSP Implementation".)

The following topics are covered here:

- Introduction to JavaServer Pages
- JSP Execution
- Overview of JSP Syntax Elements

Introduction to JavaServer Pages

JavaServer Pages(TM) is a technology specified by Sun Microsystems as a convenient way of generating dynamic content in pages that are output by a Web application (an application running on a Web server).

This technology, which is closely coupled with Java servlet technology, allows you to include Java code snippets and calls to external Java components within the HTML code (or other markup code, such as XML) of your Web pages. JavaServer Pages (JSP) technology works nicely as a front-end for business logic and dynamic functionality in JavaBeans and Enterprise JavaBeans (EJBs).

JSP code is distinct from other Web scripting code, such as JavaScript, in a Web page. Anything that you can include in a normal HTML page can be included in a JSP page as well.

In a typical scenario for a database application, a JSP page will call a component such as a JavaBean or Enterprise JavaBean, and the bean will directly or indirectly access the database, generally through JDBC or perhaps SQLJ.

A JSP page is translated into a Java servlet before being executed (typically on demand, but sometimes in advance), and it processes HTTP requests and generates responses similarly to any other servlet. JSP technology offers a more convenient way to code the servlet.

Furthermore, JSP pages are fully interoperable with servlets—JSP pages can include output from a servlet or forward to a servlet, and servlets can include output from a JSP page or forward to a JSP page.

What a JSP Page Looks Like

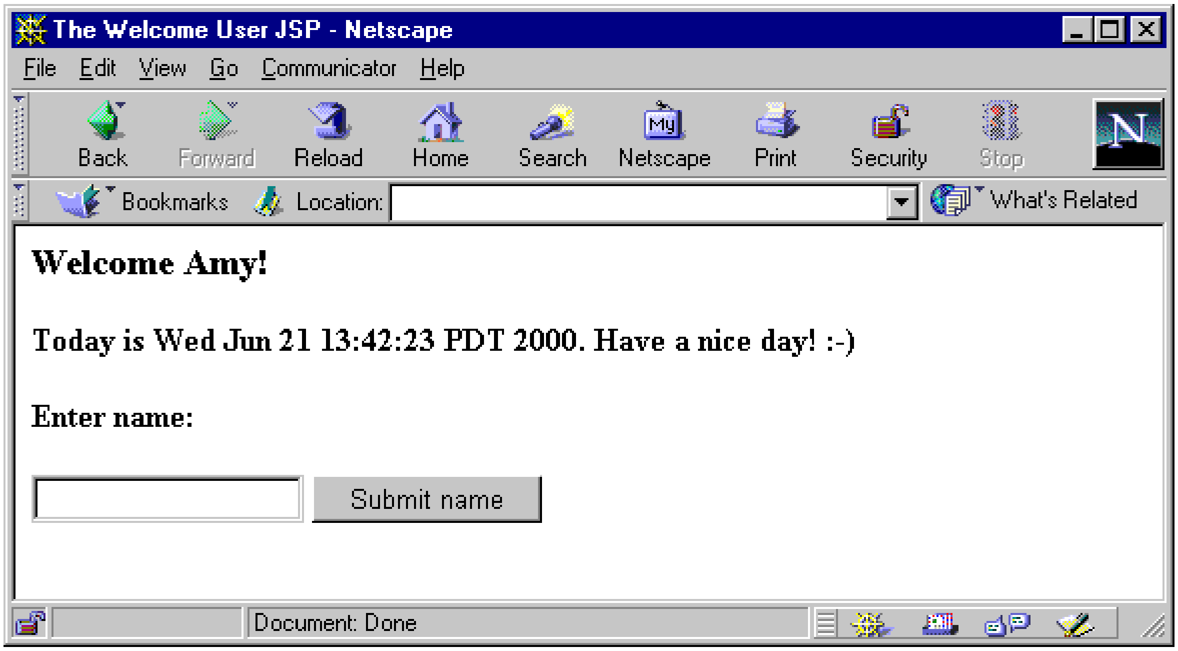
Here is an example of a simple JSP page. (For an explanation of JSP syntax elements used here, see "Overview of JSP Syntax Elements" on page 1-10.)

```
<HTML>
<HEAD><TITLE>The Welcome User JSP</TITLE></HEAD>
<BODY>
<% String user=request.getParameter("user"); %>
<H3>Welcome <%= (user==null) ? "" : user %>!</H3>
<P><B> Today is <%= new java.util.Date() %>. Have a nice day! :-)</B></P>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
```

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

In a JSP page, Java elements are set off by tags such as <% and %>, as in the preceding example. In this example, Java snippets get the user name from an HTTP request object, print the user name, and get the current date.

This JSP page will produce the following output if the user inputs the name "Amy":


The Welcome User JSP - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: What's Related
Enter name:
Submit name
Document: Done

Welcome Amy!

Today is Wed Jun 21 13:42:23 PDT 2000. Have a nice day! :-)

Convenience of JSP Coding Versus Servlet Coding

Combining Java code and Java calls into an HTML page is more convenient than using straight Java code in a servlet. JSP syntax gives you a shortcut for coding dynamic Web pages, typically requiring much less code than Java servlet syntax. Following is an example contrasting servlet code and JSP code.

Servlet Code

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class Hello extends HttpServlet
{
   public void doGet(HttpServletRequest rq, HttpServletResponse rsp)
   {
      rsp.setContentType("text/html");
      try {
         PrintWriter out = rsp.getWriter();
         out.println("<HTML>");
         out.println("<HEAD><TITLE>Welcome</TITLE></HEAD>");
         out.println("<BODY>");
         out.println("<H3>Welcome!</H3>");
         out.println("<P>Today is "+new java.util.Date()+".</P>");
         out.println("</BODY>");
         out.println("</HTML>");
      } catch (IOException ioe)
      {
        // (error processing)
      }
   }
}
```

JSP Code

```
<HTML>
<HEAD><TITLE>Welcome</TITLE></HEAD>
<BODY>
<H3>Welcome!</H3>
<P>Today is <%= new java.util.Date() %>.</P>
</BODY>
</HTML>
```

Note how much simpler JSP syntax is. Among other things, it saves Java overhead such as package imports and `try...catch` blocks.

Additionally, the JSP translator automatically handles a significant amount of servlet coding overhead for you in the `.java` file that it outputs, such as directly or indirectly implementing the standard `javax.servlet.jsp.HttpJspPage` interface and adding code to acquire an HTTP session.

Also note that because the HTML of a JSP page is not embedded within Java print statements as is the case in servlet code, you can use HTML authoring tools to create JSP pages.

Separation of Business Logic from Page Presentation: Calling JavaBeans

JSP technology allows separating the development efforts between the HTML code that determines static page presentation, and the Java code that processes business logic and presents dynamic content. It therefore becomes much easier to split maintenance responsibilities between presentation and layout specialists who may be proficient in HTML but not Java, and code specialists who may be proficient in Java but not HTML.

In a typical JSP page, most Java code and business logic will *not* be within snippets embedded in the JSP page—instead, it will be in JavaBeans or Enterprise JavaBeans that are invoked from the JSP page.

JSP technology offers the following syntax for defining and creating an instance of a JavaBeans class:

```
<jsp:useBean id="pageBean" class="mybeans.NameBean" scope="page" />
```

This example creates an instance, `pageBean`, of the `mybeans.NameBean` class (the `scope` parameter will be explained later in this chapter).

Later in the page, you can use this bean instance, as in the following example:

```
Hello <%= pageBean.getNewName() %> !
```

(This prints "Hello Julie !", for example, if the name "Julie" is in the `newName` attribute of `pageBean`, which might occur through user input.)

The separation of business logic from page presentation allows convenient division of responsibilities between the Java expert who is responsible for the business logic and dynamic content—this developer owns and maintains the code for the `NameBean` class—and the HTML expert who is responsible for the static presentation and layout of the Web page that the application user sees—this developer owns and maintains the code in the `.jsp` file for this JSP page.

Tags used with JavaBeans—`useBean` to declare the JavaBean instance and `getProperty` and `setProperty` to access bean properties—are further discussed in "JSP Actions and the <jsp: > Tag Set" on page 1-18.

JSP Pages and Alternative Markup Languages

JavaServer Pages technology is typically used for dynamic HTML output, but the Sun Microsystems *JavaServer Pages Specification, Version 1.1* also supports additional types of structured, text-based document output. A JSP translator does not process text outside of JSP elements, so any text that is appropriate for Web pages in general is typically appropriate for a JSP page as well.

A JSP page takes information from an HTTP request and accesses information from a data server (such as through a SQL database query). It combines and processes this information and incorporates it as appropriate into an HTTP response with dynamic content. The content can be formatted as HTML, DHTML, XHTML, or XML, for example.

For information about XML support, see "XML-Alternative Syntax" on page 4-17. You can also refer to the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference* for information about the JML `transform` tag.

JSP Execution

This section provides a top-level look at how a JSP is run, including on-demand translation (the first time a JSP page is run), the role of the *JSP container* and the servlet container, and error processing.

Note: The term *JSP container* is used in the Sun Microsystems *JavaServer Pages Specification, Version 1.1*, replacing the term *JSP engine* that was used in earlier specifications. The two terms are synonymous.

JSP Containers in a Nutshell

A JSP container is an entity that translates, executes, and processes JSP pages and delivers requests to them.

The exact make-up of a JSP container varies from implementation to implementation, but it will consist of a servlet or collection of servlets. The JSP container, therefore, is executed by a servlet container.

A JSP container may be incorporated into a Web server if the Web server is written in Java, or the container may be otherwise associated with and used by the Web server.

JSP Pages and On-Demand Translation

Presuming the typical on-demand translation scenario, a JSP page is usually executed as follows:

1. The user requests the JSP page through a URL ending with a `.jsp` file name.
2. Upon noting the `.jsp` file name extension in the URL, the servlet container of the Web server invokes the JSP container.
3. The JSP container locates the JSP page and translates it if this is the first time it has been requested. Translation includes producing servlet code in a `.java` file and then compiling the `.java` file to produce a servlet `.class` file.

   The servlet class generated by the JSP translator subclasses a class (provided by the JSP container) that implements the `javax.servlet.jsp.HttpJspPage` interface. The servlet class is referred to as the *page implementation class*. This document will refer to instances of page implementation classes as *JSP page instances*.

Translating a JSP page into a servlet automatically incorporates standard servlet programming overhead into the generated servlet code, such as implementing the `HttpJspPage` interface and generating code for its service method.

4. The JSP container triggers instantiation and execution of the page implementation class.

The servlet (JSP page instance) will then process the HTTP request, generate an HTTP response, and pass the response back to the client.

Note: The preceding steps are loosely described for purposes of this discussion. As mentioned earlier, each vendor decides how to implement its JSP container, but it will consist of a servlet or collection of servlets. For example, there may be a front-end servlet that locates the JSP page, a translation servlet that handles translation and compilation, and a wrapper servlet class that is subclassed by each page implementation class (because a translated page is not a pure servlet and cannot be run directly by the servlet container). A servlet container is required to run each of these components.

Requesting a JSP Page

A JSP page can be requested either directly—through a URL—or indirectly—through another Web page or servlet.

Directly Request a JSP Page

As with a servlet or HTML page, the end-user can request a JSP page directly by URL. For example, assume you have a `HelloWorld` JSP page that is located under the `myapp` application root directory in the Web server, as follows:

```
myapp/dir1/HelloWorld.jsp
```

If it uses port 8080 of the Web server, you can request it with the following URL:

`http://`*`hostname`*`:8080/myapp/dir1/HelloWorld.jsp`

(The application root directory is specified in the servlet context of the application.)

The first time the end-user requests `HelloWorld.jsp`, the JSP container triggers both translation and execution of the page. With subsequent requests, the JSP container triggers page execution only; the translation step is no longer necessary.

Indirectly Requesting a JSP Page

JSP pages, like servlets, can also be executed indirectly—linked from a regular HTML page or referenced from another JSP page or from a servlet.

When invoking one JSP page from a JSP statement in another JSP page, the path can be either relative to the application root—known as *context-relative* or *application-relative*—or relative to the invoking page—known as *page-relative*. An application-relative path starts with "/"; a page-relative path does not.

Be aware that, typically, neither of these paths is the same path as used in a URL or HTML link. Continuing the example in the preceding section, the path in an HTML link is the same as in the direct URL request, as follows:

```
<a href="/myapp/dir1/HelloWorld.jsp" /a>
```

The application-relative path in a JSP statement is:

```
<jsp:include page="/dir1/HelloWorld.jsp" flush="true" />
```

The page-relative path to invoke `HelloWorld.jsp` from a JSP page in the same directory is:

```
<jsp:forward page="HelloWorld.jsp" />
```

("JSP Actions and the <jsp: > Tag Set" on page 1-18 discusses the `jsp:include` and `jsp:forward` statements.)

Overview of JSP Syntax Elements

You have seen a simple example of JSP syntax in "What a JSP Page Looks Like" on page 1-2; now here is a top-level list of syntax categories and topics:

- *directives*—These convey information regarding the JSP page as a whole.
- *scripting elements*—These are Java coding elements such as declarations, expressions, scriptlets, and comments.
- *objects* and *scopes*—JSP objects can be created either explicitly or implicitly and are accessible within a given scope, such as from anywhere in the JSP page or the session.
- *actions*—These create objects or affect the output stream in the JSP response (or both).

This section introduces each category, including basic syntax and a few examples. For more information, see the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

Notes: There are XML-compatible alternatives to the syntax for JSP directives, declarations, expressions, and scriptlets. See "XML-Alternative Syntax" on page 4-17.

Directives

Directives provide instruction to the JSP container regarding the entire JSP page. This information is used in translating or executing the page. The basic syntax is as follows:

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

The JSP 1.1 specification supports the following directives:

- `page`—Use this directive to specify any of a number of page-dependent attributes, such as the scripting language to use, a class to extend, a package to import, an error page to use, or the JSP page output buffer size. For example:

  ```
  <%@ page language="java" import="packages.mypackage" errorPage="boof.jsp" %>
  ```

  or, to set the JSP page output buffer size to 20kb (the default is 8kb):

  ```
  <%@ page buffer="20kb" %>
  ```

or, to unbuffer the page:

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

Notes:

- A JSP page using an error page must be buffered. Forwarding to an error page clears the buffer (not outputting it to the browser).
- For the Oracle JSP container, `java` is the default language setting. It is good programming practice to set it explicitly, however.

- `include`—Use this directive to specify a resource that contains text or code to be inserted into the JSP page when it is translated. Specify the path of the resource relative to the URL specification of the JSP page.

  Example:

  ```
  <%@ include file="/jsp/userinfopage.jsp" %>
  ```

  The `include` directive can specify either a page-relative or context-relative location. (See "Requesting a JSP Page" on page 1-8 for related discussion.)

Notes:

- The `include` directive, referred to as a "static include", is comparable in nature to the `jsp:include` action discussed later in this chapter, but takes effect at JSP translation time instead of request time. See "Static Includes Versus Dynamic Includes" on page 4-6.
- The `include` directive can be used only between pages in the same servlet context.

- `taglib`—Use this directive to specify a library of custom JSP tags that will be used in the JSP page. Vendors can extend JSP functionality with their own sets of tags. This directive indicates the location of a *tag library description* file and a prefix to distinguish use of tags from that library.

Example:

```
<%@ taglib uri="/oracustomtags" prefix="oracust" %>
```

Later in the page, use the `oracust` prefix whenever you want to use one of the tags in the library (presume this library includes a tag `dbaseAccess`):

```
<oracust:dbaseAccess ... >
...
</oracust:dbaseAccess>
```

As you can see, this example uses XML-style start-tag and end-tag syntax.

JSP tag libraries and tag library description files are introduced later in this chapter, in "Tag Libraries" on page 1-23, and discussed in detail in Chapter 7, "JSP Tag Libraries".

Scripting Elements

JSP scripting elements include the following categories of Java code snippets that can appear in a JSP page:

- *declarations*—These are statements declaring methods or member variables that will be used in the JSP page.

  A JSP declaration uses standard Java syntax within the `<%!...%>` declaration tags to declare a member variable or method. This will result in a corresponding declaration in the generated servlet code. For example:

  ```
  <%! double f1=0.0; %>
  ```

  This example declares a member variable, `f1`. In the servlet class code generated by the JSP translator, `f1` will be declared at the class top level.

  Note: Method variables, as opposed to member variables, are declared within JSP scriptlets as described below. See "Method Variable Declarations Versus Member Variable Declarations" on page 4-11 for more information.

- *expressions*—These are Java expressions that are evaluated, converted into string values as appropriate, and displayed where they are encountered on the page.

  A JSP expression does *not* end in a semi-colon, and is contained within `<%=...%>` tags.

Example:

```
<P><B> Today is <%= new java.util.Date() %>. Have a nice day! </B></P>
```

Note: A JSP expression in a request-time attribute, such as in a `jsp:setProperty` statement, need not be converted to a string value.

- *scriptlets*—These are portions of Java code intermixed within the markup language of the page.

  A scriptlet, or code fragment, may consist of anything from a partial line to multiple lines of Java code. You can use them within the HTML code of a JSP page to set up conditional branches or a loop, for example.

  A JSP scriptlet is contained within `<%...%>` scriptlet tags, using normal Java syntax.

  Example 1:

```
<% if (pageBean.getNewName().equals("")) { %>
   I don't know you.
<% } else { %>
   Hello <%= pageBean.getNewName() %>.
<% } %>
```

  Three one-line JSP scriptlets are intermixed with two lines of HTML (one of which includes a JSP expression, which does *not* require a semi-colon). Note that JSP syntax allows HTML code to be the code that is conditionally executed within the `if` and `else` branches (inside the Java brackets set out in the scriptlets).

  The preceding example assumes the use of a JavaBean instance, `pageBean`.

  Example 2:

```
<% if (pageBean.getNewName().equals("")) { %>
   I don't know you.
   <% empmgr.unknownemployee();
} else { %>
   Hello <%= pageBean.getNewName() %>.
   <% empmgr.knownemployee();
} %>
```

This example adds more Java code to the scriptlets. It assumes the use of a JavaBean instance, `pageBean`, and assumes that some object, `empmgr`, was previously instantiated and has methods to execute appropriate functionality for a known employee or an unknown employee.

Note: Use a JSP scriptlet to declare method variables, as opposed to member variables, as in the following example:

```
<% double f2=0.0; %>
```

This scriptlet declares a method variable, `f2`. In the servlet class code generated by the JSP translator, `f2` will be declared as a variable within the service method of the servlet.

Member variables are declared in JSP declarations as described above.

For a comparative discussion, see "Method Variable Declarations Versus Member Variable Declarations" on page 4-11.

- *comments*—These are developer comments embedded within the JSP code, similar to comments embedded within any Java code.

  Comments are contained within `<%--...--%>` tags.

  Example:

  ```
  <%-- Execute the following branch if no user name is entered. --%>
  ```

JSP Objects and Scopes

In this document, the term *JSP object* refers to a Java class instance declared within or accessible to a JSP page. JSP objects can be either:

- *explicit*—Explicit objects are declared and created within the code of your JSP page, accessible to that page and other pages according to the `scope` setting you choose.

or:

- *implicit*—Implicit objects are created by the underlying JSP mechanism and accessible to Java scriptlets or expressions in JSP pages according to the inherent `scope` setting of the particular object type.

Scopes are discussed below, in "Object Scopes".

Explicit Objects

Explicit objects are typically JavaBean instances declared and created in `jsp:useBean` action statements. The `jsp:useBean` statement and other action statements are described in "JSP Actions and the <jsp: > Tag Set" on page 1-18, but an example is also shown here:

```
<jsp:useBean id="pageBean" class="mybeans.NameBean" scope="page" />
```

This statement defines an instance, `pageBean`, of the `NameBean` class that is in the `mybeans` package. The scope parameter is discussed in "Object Scopes" below.

You can also create objects within Java scriptlets or declarations, just as you would create Java class instances in any Java program.

Object Scopes

Objects in a JSP page, whether explicit or implicit, are accessible within a particular *scope*. In the case of explicit objects, such as a JavaBean instance created in a `jsp:useBean` action statement, you can explicitly set the scope with the following syntax (as in the example in the preceding section, "Explicit Objects"):

scope="*scopevalue*"

There are four possible scopes:

- `scope="page"`—The object is accessible only from within the JSP page where it was created.

  Note that when the user refreshes the page while executing a JSP page, new instances will be created of all page-scope objects.

- `scope="request"`—The object is accessible from any JSP page servicing the same HTTP request that is serviced by the JSP page that created the object.
- `scope="session"`—The object is accessible from any JSP page sharing the same HTTP session as the JSP page that created the object.
- `scope="application"`—The object is accessible from any JSP page used in the same Web application (within any single Java virtual machine) as the JSP page that created the object.

Implicit Objects

JSP technology makes available to any JSP page a set of *implicit objects*. These are Java class instances that are created automatically by the JSP mechanism and that allow interaction with the underlying servlet environment.

The following implicit objects are available. For information about methods available with these objects, refer to the Sun Microsystems Javadoc for the noted classes and interfaces at the following location:

`http://java.sun.com/products/servlet/2.2/javadoc/index.html`

- `page`

  This is an instance of the JSP page implementation class that was created when the page was translated, and that implements the interface `javax.servlet.jsp.HttpJspPage`; `page` is synonymous with `this` within a JSP page.

- `request`

  This represents an HTTP request and is an instance of a class that implements the `javax.servlet.http.HttpServletRequest` interface, which extends the `javax.servlet.ServletRequest` interface.

- `response`

  This represents an HTTP response and is an instance of a class that implements the `javax.servlet.http.HttpServletResponse` interface, which extends the `javax.servlet.ServletResponse` interface.

  The `response` and `request` objects for a particular request are associated with each other.

- `pageContext`

  This represents the *page context* of a JSP page, which is provided for storage and access of all `page` scope objects of a JSP page instance. A `pageContext` object is an instance of the `javax.servlet.jsp.PageContext` class.

  The `pageContext` object has `page` scope, making it accessible only to the JSP page instance with which it is associated.

- `session`

  This represents an HTTP session and is an instance of the `javax.servlet.http.HttpSession` class.

- `application`

  This represents the servlet context for the Web application and is an instance of the `javax.servlet.ServletContext` class.

The `application` object is accessible from any JSP page instance running as part of any instance of the application within a single JVM. (The programmer should be aware of the server architecture regarding use of JVMs.)

- `out`

  This is an object that is used to write content to the output stream of a JSP page instance. It is an instance of the `javax.servlet.jsp.JspWriter` class, which extends the `java.io.Writer` class.

  The `out` object is associated with the `response` object for a particular request.

- `config`

  This represents the servlet configuration for a JSP page and is an instance of a class that implements the `javax.servlet.ServletConfig` interface. Generally speaking, servlet containers use `ServletConfig` instances to provide information to servlets during initialization. Part of this information is the appropriate `ServletContext` instance.

- `exception` (JSP error pages only)

  This implicit object applies only to JSP error pages—these are pages to which processing is forwarded when an exception is thrown from another JSP page; they must have the `page` directive `isErrorPage` attribute set to `true`.

  The implicit `exception` object is a `java.lang.Exception` instance that represents the uncaught exception that was thrown from another JSP page and that resulted in the current error page being invoked.

  The `exception` object is accessible only from the JSP error page instance to which processing was forwarded when the exception was encountered.

  For an example of JSP error processing and use of the `exception` object, see "JSP Runtime Error Processing" on page 3-16.

Using an Implicit Object

Any of the implicit objects discussed in the preceding section might be useful. The following example uses the `request` object to retrieve and display the value of the `username` parameter from the HTTP request:

```
<H3> Welcome <%= request.getParameter("username") %> ! <H3>
```

JSP Actions and the <jsp: > Tag Set

JSP action elements result in some sort of action occurring while the JSP page is being executed, such as instantiating a Java object and making it available to the page. Such actions may include the following:

- creating a JavaBean instance and accessing its properties
- forwarding execution to another HTML page, JSP page, or servlet
- including an external resource in the JSP page

Action elements use a set of standard JSP tags that begin with "`<jsp:`" syntax. Although the tags described earlier in this chapter that begin with "<%" syntax are sufficient to code a JSP page, the "`<jsp:`" tags provide additional functionality and convenience.

Action elements also use syntax similar to that of XML statements, with similar "begin" and "end" tags such as in the following example:

```
<jsp:sampletag attr1="value1" attr2="value2" ... attrN="valueN">
...body...
</jsp:sampletag>
```

or, where there is no body, the action statement is terminated with an empty tag:

```
<jsp:sampletag attr1="value1", ..., attrN="valueN" />
```

The JSP specification includes the following standard action tags, which are introduced and briefly discussed here:

- `jsp:useBean`

  The `jsp:useBean` action creates an instance of a specified JavaBean class, gives the instance a specified name, and defines the scope within which it is accessible (such as from anywhere within the current JSP page instance).

  Example:

  ```
  <jsp:useBean id="pageBean" class="mybeans.NameBean" scope="page" />
  ```

  This example creates a page-scoped instance `pageBean` of the `mybeans.NameBean` class. This instance is accessible only from the JSP page instance that creates it.

- `jsp:setProperty`

  The `jsp:setProperty` action sets one or more bean properties. The bean must have been previously specified in a `useBean` action. You can directly specify a value for a specified property, or take the value for a specified property from an associated HTTP request parameter, or iterate through a series of properties and values from the HTTP request parameters.

The following example sets the `user` property of the `pageBean` instance (defined in the preceding `useBean` example) to a value of "Smith":

```
<jsp:setProperty name="pageBean" property="user" value="Smith" />
```

The following example sets the `user` property of the `pageBean` instance according to the value set for a parameter called `username` in the HTTP request:

```
<jsp:setProperty name="pageBean" property="user" param="username" />
```

If the bean property and request parameter have the same name (`user`), you can simply set the property as follows:

```
<jsp:setProperty name="pageBean" property="user" />
```

The following example results in iteration over the HTTP request parameters, matching bean property names with request parameter names and setting bean property values according to the corresponding request parameter values:

```
<jsp:setProperty name="pageBean" property="*" />
```

Important: For `property="*"`, the JSP 1.1 specification does not stipulate the order in which properties are set. If order matters, and if you want to ensure that your JSP page is portable, you should use a separate `jsp:setProperty` statement for each property.

Also, if you use separate `jsp:setProperty` statements, then the Oracle JSP translator can generate the corresponding `setXXX()` methods directly. In this case, introspection only occurs during translation. There will be no need to introspect the bean during runtime, which would be somewhat more costly.

- `jsp:getProperty`

  The `jsp:getProperty` action reads a bean property value, converts it to a Java string, and places the string value into the implicit `out` object so that it can be displayed as output. The bean must have been previously specified in a `jsp:useBean` action. For the string conversion, primitive types are converted directly and object types are converted using the `toString()` method specified in the `java.lang.Object` class.

The following example puts the value of the `user` property of the `pageBean` bean into the `out` object:

```
<jsp:getProperty name="pageBean" property="user" />
```

- `jsp:param`

  You can use the `jsp:param` action in conjunction with `jsp:include`, `jsp:forward`, or `jsp:plugin` actions (described below).

  For `jsp:forward` and `jsp:include` statements, a `jsp:param` action optionally provides key / value pairs for parameter values in the HTTP request object. New parameters and values specified with this action are added to the request object, with new values taking precedence over old.

  The following example sets the request object parameter `username` to a value of `Smith`:

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

  Note: The `jsp:param` tag is not supported for `jsp:include` or `jsp:forward` in the JSP 1.0 specification.

- `jsp:include`

  The `jsp:include` action inserts additional static or dynamic resources into the page at request time as the page is displayed. Specify the resource with a relative URL (either page-relative or application-relative).

  As of the Sun Microsystems *JavaServer Pages Specification, Version 1.1*, you must set `flush` to `true`, which results in the buffer being flushed to the browser when a `jsp:include` action is executed. (The `flush` attribute is mandatory, but a setting of `false` is currently invalid.)

  You can also have an action body with `jsp:param` settings, as shown in the second example.

  Examples:

  ```
  <jsp:include page="/templates/userinfopage.jsp" flush="true" />
  ```

or:

```
<jsp:include page="/templates/userinfopage.jsp" flush="true" >
   <jsp:param name="username" value="Smith" />
   <jsp:param name="userempno" value="9876" />
</jsp:include>
```

Note that the following syntax would work as an alternative to the preceding:

```
<jsp:include page="/templates/userinfopage.jsp?username=Smith&userempno=9876" flush="true" />
```

Notes:

- The `jsp:include` action, known as a "dynamic include", is similar in nature to the `include` directive discussed earlier in this chapter, but takes effect at request time instead of translation time. See "Static Includes Versus Dynamic Includes" on page 4-6.
- The `jsp:include` action can be used only between pages in the same servlet context.

- `jsp:forward`

  The `jsp:forward` action effectively terminates execution of the current page, discards its output, and dispatches a new page—either an HTML page, a JSP page, or a servlet.

  The JSP page must be buffered to use a `jsp:forward` action; you cannot set `buffer="none"`. The action will clear the buffer, not outputting contents to the browser.

  As with `jsp:include`, you can also have an action body with `jsp:param` settings, as shown in the second example.

  Examples:

```
<jsp:forward page="/templates/userinfopage.jsp" />
```

  or:

```
<jsp:forward page="/templates/userinfopage.jsp" >
   <jsp:param name="username" value="Smith" />
   <jsp:param name="userempno" value="9876" />
</jsp:forward>
```

Notes:

- The difference between the `jsp:forward` examples here and the `jsp:include` examples earlier is that the `jsp:include` examples insert `userinfopage.jsp` within the output of the current page; the `jsp:forward` examples stop executing the current page and display `userinfopage.jsp` instead.
- The `jsp:forward` action can be used only between pages in the same servlet context.
- The `jsp:forward` action results in the original `request` object being forwarded to the target page. As an alternative, if you do not want the `request` object forwarded, you can use the `sendRedirect(String)` method specified in the standard `javax.servlet.http.HttpServletResponse` interface. This sends a temporary redirect response to the client using the specified redirect-location URL. You can specify a relative URL; the servlet container will convert the relative URL to an absolute URL.

- `jsp:plugin`

  The `jsp:plugin` action results in the execution of a specified applet or JavaBean in the client browser, preceded by a download of Java plugin software if necessary.

  Specify configuration information, such as the applet to run and the codebase, using `jsp:plugin` attributes. The JSP container might provide a default URL for the download, but you can also specify attribute `nspluginurl="`*`url`*`"` (for a Netscape browser) or `iepluginurl="`*`url`*`"` (for an Internet Explorer browser).

  Use nested `jsp:param` actions within `<jsp:params>` and `</jsp:params>` start and end tags to specify parameters to the applet or JavaBean. (Note that these `jsp:params` start and end tags are *not* necessary when using `jsp:param` in a `jsp:include` or `jsp:forward` action.)

  Use `<jsp:fallback>` and `</jsp:fallback>` start and end tags to delimit alternative text to execute if the plugin cannot run.

The following example, from the *Sun Microsystems JavaServer Pages Specification, Version 1.1*, shows the use of an applet plugin:

```
<jsp:plugin type=applet code="Molecule.class" codebase="/html" >
   <jsp:params>
      <jsp:param name="molecule" value="molecules/benzene.mol" />
   </jsp:params>
   <jsp:fallback>
      <p> Unable to start the plugin. </p>
   </jsp:fallback>
</jsp:plugin>
```

Many additional parameters—such as `ARCHIVE`, `HEIGHT`, `NAME`, `TITLE`, and `WIDTH`—are allowed in the `jsp:plugin` action statement as well. Use of these parameters is according to the general HTML specification.

Tag Libraries

In addition to the standard JSP tags discussed previously in this section, the JSP 1.1 specification lets vendors define their own *tag libraries* and also lets vendors implement a framework allowing customers to define their own tag libraries.

A tag library defines a collection of custom tags and can be thought of as a JSP sub-language. Developers can use tag libraries directly, in manually coding a JSP page, but they might also be used automatically by Java development tools. A standard tag library must be portable between different JSP container implementations.

Import a tag library into a JSP page using the `taglib` directive, introduced in "Directives" on page 1-10.

Key concepts of standard JavaServer Pages support for JSP tag libraries include the following topics:

- tag handlers

  A *tag handler* describes the semantics of the action that results from use of a custom tag. A tag handler is an instance of a Java class that implements either the `Tag` or `BodyTag` interface (depending on whether the tag uses a body between a start tag and an end tag) in the standard `javax.servlet.jsp.tagext` package.

- scripting variables

  Custom tag actions can create server-side objects available for use by the tag itself or by other scripting elements such as scriptlets. This is accomplished by creating or updating *scripting variables*.

  Details regarding scripting variables that a custom tag defines must be specified in a subclass of the standard `javax.servlet.jsp.tagext.TagExtraInfo` abstract class. This document refers to such a subclass as a *tag-extra-info class*. The JSP container uses instances of these classes during translation.

- tag library description files

  A *tag library description* (TLD) file is an XML document that contains information about a tag library and about individual tags of the library. The file name of a TLD has the `.tld` extension.

  A JSP container uses the TLD file in determining what action to take when it encounters a tag from the library.

- use of `web.xml` for tag libraries

  The Sun Microsystems *Java Servlet Specification, Version 2.2* describes a standard deployment descriptor for servlets—the `web.xml` file. JSP applications can use this file in specifying the location of a JSP tag library description file.

  For JSP tag libraries, the `web.xml` file can include a `taglib` element and two subelements: `taglib-uri` and `taglib-location`.

For information about these topics, see "Standard Tag Library Framework" on page 7-2. For further information, see the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

For information about tag libraries provided by Oracle, see the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

2

Overview of the Oracle JSP Implementation

Oracle JSP 1.1.x.x releases are complete implementations of the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

This chapter provides an overview of the Oracle JSP implementation, including programmatic extensions, and support in both Oracle and non-Oracle environments.

The following topics are covered here:

Important: Version 1.1.2.4 of the Oracle JSP container is supplied with Oracle9*i* release 2.

Overview of JSP and Servlet Containers and Web Server with Oracle9*i*

This section introduces the Oracle JSP container and servlet environment supplied with Oracle9*i* release 2, and also discusses the role of the Oracle HTTP Server and `mod_jserv` component in running database-access Web applications.

Note Regarding Desupport of J2EE in the Oracle9*i* Database:

With the introduction of Oracle9*i* Application Server Containers for J2EE (OC4J)—a new, lighter-weight, easier-to-use, faster, and certified J2EE container—Oracle will desupport the Java 2 Enterprise Edition (J2EE) and CORBA stacks from the database, starting with Oracle9*i* database release 2. However, the database-embedded Java VM (Oracle JVM) will still be present and will continue to be enhanced to offer Java 2 Standard Edition (J2SE) features, Java stored procedures, JDBC, and SQLJ in the database. As of Oracle9*i* database release 2 (9.2.0), Oracle will no longer support the following technologies in the database:

- the J2EE stack, consisting of:
  Enterprise Beans (EJB) container
  JavaServer Pages (JSP) container
  Oracle9*i* Servlet Engine (OSE)
- the embedded Common Object Request Broker Architecture (CORBA) framework, based on Visibroker for Java

Customers will no longer be able to deploy servlets, JSP pages, EJBs, and CORBA objects in Oracle databases. Oracle9*i* database release 1 (9.0.1) is the last database release to support the J2EE and CORBA stack. Oracle is encouraging customers to migrate existing J2EE applications running in the database to OC4J.

JSP Container and Servlet Environment Provided with Oracle9*i*

Version 1.1.2.4 of the Oracle JSP container is supplied with Oracle9*i* release 2. This version is fully compliant with the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

The supplied servlet environment with Oracle9*i* release 2 is Apache JServ, a servlet 2.0 environment. The JSP 1.1 specification calls for a servlet 2.1(b) or later environment, but you can use the Oracle JSP version 1.1.2.4 container with any servlet 2.0 or later environment—extensions have been built in to the JSP container to emulate some servlet 2.2 functionality. This is further discussed in "Portability and Functionality Across Servlet Environments" on page 2-5.

Special considerations in using the Oracle JSP container with a servlet 2.0 environment are discussed in Chapter 9, "Oracle JSP in Apache JServ". This chapter also documents JSP and JServ configuration.

Other Servlet Environments

Instead of using JServ, you can acquire any of several other servlet environments. Options include the following, in particular:

- Oracle9*i*AS Containers for J2EE (OC4J)

  OC4J is supplied with the Oracle9*i* Application Server and is also available in a standalone version from the Oracle Technology Network (`http://otn.oracle.com`).

- Tomcat

  Tomcat, from the Apache Software Foundation, includes a servlet 2.2 environment. It also includes a JSP 1.1 reference implementation, but you can use the Oracle JSP container on top of it.

Because you can use a servlet 2.2 environment with the Oracle JSP container, and because the JSP container itself emulates some servlet 2.2 functionality, many servlet 2.2 features are discussed in this document, even though the environment supplied with Oracle9*i* release 2 is servlet 2.0.

OC4J documentation is available with Oracle9*i* Application Server releases and from the Oracle Technology Network.

For configuration information for Tomcat, as well as for the Sun Microsystems JavaServer Web Developer's Kit (JSWDK), refer to Appendix A, "Getting Started in Alternative Environments".

Role of the Oracle HTTP Server

Oracle HTTP Server, powered by the Apache Web server, is included with the Oracle9*i* database as the HTTP entry point for Web applications accessing the database. Database access is through Apache add-on modules.

The remainder of this section covers the following topics:

- Use of Apache Mods
- More About mod_jserv

Use of Apache Mods

In using the Oracle HTTP Server, dynamic content is delivered through various Apache *mod* components provided either by Apache or by other vendors such as Oracle. (Static content is usually delivered from the file system.) An Apache mod is typically a module of C code, running in the Apache address space, that passes requests to a particular mod-specific processor. The mod software will have been written specifically for use with the particular processor.

To access Oracle9*i* data from JSP pages or servlets running in the JServ servlet environment that is provided with Oracle9*i*, use `mod_jserv`. This mod was developed by Apache and is provided with Oracle9*i*.

Note: Many additional Apache "mod" components are available for use in an Apache environment, provided by Apache for general use or by Oracle for Oracle-specific use, but they are not relevant for JSP applications.

More About mod_jserv

The `mod_jserv` component delegates HTTP requests to JSP pages or servlets running in the JServ servlet container in a middle-tier JVM. Oracle9*i* release 2 supplies the JServ servlet container, which supports the servlet 2.0 specification. The middle-tier environment may or may not be on the same physical host as the back-end Oracle9*i* database.

Communication between `mod_jserv` and middle-tier JVMs uses a proprietary Apache JServ protocol over TCP/IP. The `mod_jserv` component can delegate requests to multiple JVMs in a pool for load balancing.

JSP applications running in middle-tier JVMs use the Oracle JDBC OCI driver or Thin driver to access the database.

Servlet 2.0 environments (as opposed to servlet 2.1 or 2.2 environments) have issues that require special consideration. See "Considerations for JServ Servlet Environments" on page 9-20.

Refer to Apache documentation for `mod_jserv` configuration information. (This documentation is provided with Oracle9*i*.)

Portability and Functionality Across Servlet Environments

The Oracle JavaServer Pages implementation is highly portable across server platforms and servlet environments. It also supplies a framework for Web applications in older servlet environments, where servlet context behavior was not yet sufficiently defined.

Oracle JSP Portability

The Oracle JSP container can run on any servlet environment that complies with version 2.0 or higher of the Sun Microsystems *Java Servlet Specification*. This is in contrast to most JSP implementations, which require a servlet 2.1(b) or higher implementation. The Oracle JSP container provides functionality equivalent to what is lacking in older servlet environments.

Furthermore, the Oracle JSP container is independent of the server environment and its servlet implementation. This is in contrast to vendors who deliver their JSP implementation as part of their servlet implementation instead of as a standalone product.

This portability makes it much easier to run JSP pages in both your development environment and the target environment, as opposed to having to use a different JSP implementation on your development system because of any server or servlet platform limitations. There are usually benefits to developing on a system with the same JSP container as the target server; but realistically speaking, there is usually some variation between environments.

Oracle JSP Extended Functionality for Servlet 2.0 Environments

Because of interdependence between servlet specifications and JSP functionality, Sun Microsystems has tied versions of the *JavaServer Pages Specification* to particular versions of the *Java Servlet Specification*. According to Sun, JSP 1.0 requires a servlet 2.1(b) implementation, and JSP 1.1 requires a servlet 2.2 implementation.

The servlet 2.0 specification was limited in that it provided only a single servlet context per Java virtual machine, instead of a servlet context for each application. The servlet 2.1 specification allowed, but did not mandate, a separate servlet context for each application. The servlet 2.1(b) and servlet 2.2 specifications mandated separate servlet contexts.

The Oracle JSP container, however, offers functionality that emulates the application support provided with the servlet 2.1(b) specification. This allows a full application framework in a servlet 2.0 environment such as JServ. This includes providing applications with distinct `ServletContext` and `HttpSession` objects.

This extended support is provided through a file, `globals.jsa`, that acts as a JSP application marker, application and session event handler, and centralized location for application-global declarations and directives. (For information, see "Oracle JSP Application and Session Support for JServ" on page 9-26.)

Because of this extended functionality, the Oracle JSP container is not limited by the underlying servlet environment.

Oracle9*i* JDeveloper Support for the Oracle JSP Container

Some visual Java programming tools now support JSP coding. In particular, Oracle9*i* JDeveloper supports the Oracle JSP container and includes the following features:

- integration of the Oracle JSP container to support the full application development cycle—editing, debugging, and running JSP pages
- debugging of deployed JSP pages
- an extensive set of data-enabled and Web-enabled JavaBeans, known as JDeveloper Web beans
- the JSP Element Wizard, which offers a convenient way to add predefined Web beans to a page
- support for incorporating custom JavaBeans
- a deployment option for JSP applications that rely on the JDeveloper Business Components for Java (BC4J)

See "Deployment of JSP Pages with Oracle9i JDeveloper" on page 6-29 for more information about JSP deployment support.

For debugging, JDeveloper can set breakpoints within JSP page source and can follow calls from JSP pages into JavaBeans. This is much more convenient than manual debugging techniques, such as adding print statements within the JSP page to output state into the response stream (for viewing in your browser) or to the server log (through the `log()` method of the implicit `application` object).

For information about JDeveloper, refer to their online help, or to the following Web site:

`http://otn.oracle.com/products/jdev/content.html`

Support for the Oracle JSP Container in Non-Oracle Environments

You should be able to install and run the Oracle JSP container on any server environment supporting servlet specification 2.0 or higher. In particular, it has been tested in the following environments as of release 1.1.2.4:

- Apache Software Foundation Apache JServ 1.1 (also provided with Oracle9*i*)

  This is a Web server and servlet 2.0 environment without a JSP environment. To run JSP pages, you must install a JSP environment on top of it.

- Sun Microsystems JSWDK 1.0 (JavaServer Web Developer's Kit)

  This is a Web server with the servlet 2.1 and JavaServer Pages 1.0 reference implementations. You can, however, install the Oracle JSP container on top of the JSWDK servlet environment to replace the original JSP environment.

- Apache Software Foundation Tomcat 3.1

  This cooperative effort between Sun Microsystems and the Apache Software Foundation is a Web server with the servlet 2.2 and JavaServer Pages 1.1 reference implementations. You can, however, install the Oracle JSP container on top of the Tomcat servlet environment to replace the original JSP environment. You can also run Tomcat in conjunction with the Apache Web server instead of using the Tomcat Web server.

Overview of Oracle JSP Programmatic Extensions

This section provides an overview of the following Oracle-specific programming extensions supported by the Oracle JSP container:

- support for SQLJ, a standard syntax for embedding SQL statements directly into Java code
- extended globalization support
- `JspScopeListener` for event handling
- `globals.jsa` file for application support in servlet 2.0 environments

The Oracle JSP container also provides the following extended functionality, documented in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*, through custom tag libraries and custom JavaBeans that are generally portable to other JSP environments:

- extended types implemented as JavaBeans that can have a specified scope
- integration with XML and XSL
- data-access JavaBeans
- the Oracle JSP Markup Language (JML) custom tag library, which reduces the level of Java proficiency required for JSP development
- a custom tag library for SQL functionality

All these features are introduced in the following subsections.

Overview of Oracle-Specific Extensions

The Oracle JSP extensions listed in this section, documented later in this manual, are not portable to other JSP environments.

SQLJ Support in the Oracle JSP Container

Dynamic server pages commonly include data extracted from databases; however, JavaServer Pages technology does not offer built-in support to facilitate database access. JSP developers typically must rely on the standard Java Database Connectivity (JDBC) API or a custom set of database JavaBeans.

SQLJ is a standard syntax for embedding static SQL instructions directly in Java code, greatly simplifying database-access programming. The Oracle JSP container and its translator support SQLJ programming in JSP scriptlets.

SQLJ statements are indicated by the `#sql` token. You can trigger the Oracle JSP translator to invoke the Oracle SQLJ translator by using the file name extension `.sqljsp` for the JSP source code file.

For more information, see "Oracle JSP Support for Oracle SQLJ" on page 5-3.

Extended Globalization Support in the Oracle JSP Container

Oracle9*i* release 2 provides extended globalization support for servlet environments that cannot encode multibyte request parameters and bean property settings.

For such environments, the Oracle JSP container supports the `translate_params` configuration parameter, which can be enabled to direct the JSP container to override the servlet container and do the encoding itself.

For more information, see "Oracle JSP Extended Support for Multibyte Parameter Encoding" on page 8-5.

JspScopeListener for Event Handling

Oracle9*i* release 2 provides the `JspScopeListener` interface for lifecycle management of Java objects of various scopes within a JSP application.

Standard servlet and JSP event-handling is provided through the `javax.servlet.http.HttpSessionBindingListener` interface, but this handles session-based events only. The Oracle `JspScopeListener` can handle page-based, request-based, and application-based events as well.

For more information, see "Oracle JSP Event Handling with JspScopeListener" on page 5-2.

globals.jsa File for Application Support (Servlet 2.0)

For servlet 2.0 environments, where servlet contexts are not fully defined, the Oracle JSP container defines a file, `globals.jsa`, to extend servlet application support.

Within any single Java virtual machine, there can be a `globals.jsa` file for each application (or, equivalently, for each servlet context). This file supports the concept of Web applications through use as an application location marker. Based on `globals.jsa` functionality, the Oracle JSP container can also mimic servlet context and HTTP session behavior for servlet environments, where such behavior is not sufficiently defined.

The `globals.jsa` file also provides a vehicle for global Java declarations and JSP directives across all JSP pages of an application.

For more information, see "Oracle JSP Application and Session Support for JServ" on page 9-26.

Overview of JSP Tag Libraries and JavaBeans Provided with Oracle9*i*

The Oracle extensions discussed in this section are implemented through standard tag libraries or custom JavaBeans and are generally portable to other JSP environments.

These features are documented in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Extended Type JavaBeans

JSP pages generally rely on core Java types in representing scalar values. However, neither of the following type categories is fully suitable for use in JSP pages:

- primitive types such as `int`, `float`, and `double`

  Values of these types cannot have a specified scope—they cannot be stored in a JSP scope object (for `page`, `request`, `session`, or `application` scope), because only objects can be stored in a scope object.

- wrapper classes in the standard `java.lang` package, such as `Integer`, `Float`, and `Double`

  Values of these types are objects, so they can theoretically be stored in a JSP scope object. However, they cannot be declared in a `jsp:useBean` action, because the wrapper classes do not follow the JavaBean model and do not provide a zero-argument constructor.

  Additionally, instances of the wrapper classes are immutable. To change a value, you must create a new instance and assign it appropriately.

To work around these limitations, Oracle9*i* release 2 provides the `JmlBoolean`, `JmlNumber`, `JmlFPNumber`, and `JmlString` JavaBean classes in package `oracle.jsp.jml` to wrap the most common Java types.

Integration with XML and XSL

You can use JSP syntax to generate any text-based MIME type, not just HTML code. In particular, you can dynamically create XML output. When you use JSP pages to generate an XML document, however, you often want a stylesheet applied to the XML data before it is sent to the client. This is difficult in JavaServer Pages technology, because the standard output stream used for a JSP page is written directly back through the server.

Oracle9*i* release 2 provides special tags in its sample JML tag library to specify that all or part of a JSP page should be transformed through an XSL stylesheet before it is output. You can use this JML tag multiple times in a single JSP page if you want to specify different style sheets for different portions of the page.

In addition, the Oracle JSP translator supports XML-alternative syntax as specified in the Sun Microsystems *JavaServer Pages Specification, Version 1.1*. For information, see "XML-Alternative Syntax" on page 4-17.

Custom Data-Access JavaBeans

Oracle9*i* release 2 supplies a set of custom JavaBeans for use in database access. The following beans are provided in the `oracle.jsp.dbutil` package:

- `ConnBean`—Open a simple database connection.
- `ConnCacheBean`—Use the Oracle connection caching implementation for database connections.
- `DBBean`—Execute a database query.
- `CursorBean`—Provide general DML support for queries and for `UPDATE`, `INSERT`, and `DELETE` statements.

SQL Custom Tag Library

Oracle9*i* release 2 provides a custom tag library for SQL functionality. The following tags are provided:

- `dbOpen`—Open a database connection.
- `dbClose`—Close a database connection.
- `dbQuery`—Execute a query.
- `dbCloseQuery`—Close the cursor for a query.
- `dbNextRow`—Move to the next row of the result set.
- `dbExecute`—Execute any SQL DML or DDL statement.

Oracle JSP Markup Language (JML) Custom Tag Library

Although the Sun Microsystems *JavaServer Pages Specification, Version 1.1* supports scripting languages other than Java, Java is the primary language used. Even though JavaServer Pages technology is designed to separate the dynamic/Java development effort from the static/HTML development effort, it is no doubt still a hindrance if the Web developer does not know any Java, especially in small development groups where no Java experts are available.

Oracle9*i* release 2 provides custom tags as an alternative—the JSP Markup Language (JML). The Oracle JML sample tag library provides an additional set of JSP tags so that you can script your JSP pages without using Java statements. JML provides tags for variable declarations, control flow, conditional branches, iterative loops, parameter settings, and calls to objects. The JML tag library also supports XML functionality, as noted previously.

The following example shows use of the `jml:for` tag, repeatedly printing "Hello World" in progressively smaller headings (H1, H2, H3, H4, H5):

```
<jml:for id="i" from="<%= 1 %>" to="<%= 5 %>" >
     <H<%=i%>>
            Hello World!
     </H<<%=i%>>
</jml:for>
```

Note: Oracle JSP versions preceding the JSP 1.1 specification used an Oracle-specific compile-time implementation of the JML tag library. This implementation is still supported as an alternative to the standard runtime implementation.

JSP Execution Models

As mentioned earlier, you can use the Oracle JSP framework in a variety of server environments. The Oracle JSP container offers two distinct execution models:

- The JSP container typically translates pages on demand before triggering their execution, as is also true with the JSP implementations of most other vendors.
- In some scenarios, however, the developer translates the pages in advance and deploys the translated and compiled results. The `ojspc` command-line tool is available to translate the pages. Then, when the end-user requests the JSP page, it is executed directly, with no translation necessary.

On-Demand Translation Model

JSP pages usually run in an on-demand translation model. This includes typical usage with the JServ servlet environment.

When a JSP page is requested from a Web server that incorporates the Oracle JSP container, the servlet `oracle.jsp.JspServlet` is instantiated and invoked (assuming proper Web server configuration). This servlet can be thought of as the front-end of the Oracle JSP container.

`JspServlet` locates the JSP page, translates and compiles it if necessary (if the page implementation class does not exist or has an earlier timestamp than the JSP page source), and triggers its execution.

Note that the Web server must be properly configured to map the `*.jsp` file name extension (in a URL) to `JspServlet`. The steps to accomplish this for JServ are discussed in detail in "Mapping JSP File Name Extensions for JServ" on page 9-6.

Pre-Translation Model

Developers may want to pre-translate their JSP pages before deploying them, for reasons such as the following:

- It can save time for the end-users when they request a JSP page, because translation at execution time is not necessary.
- It is also useful if you want to deploy binary files only, perhaps because the software is proprietary and you do not want to expose the code.

For more information, see "General Use of ojspc for Pre-Translation" on page 6-13 and "Deployment of Binary Files Only" on page 6-27.

Oracle9*i* release 2 supplies the `ojspc` command-line utility for pre-translating JSP pages. This utility has options that allow you to set an appropriate base directory for the output files, depending on how you want to deploy the application. The `ojspc` utility is documented in "Details of the ojspc Pre-Translation Tool" on page 6-14.

3 Basics

This chapter discusses key basic issues for JSP development, followed by a JSP "starter sample" for data access.

The following topics are included:

- Application Root and Doc Root Functionality
- Overview of JSP Applications and Sessions
- JSP-Servlet Interaction
- JSP Resource Management
- JSP Runtime Error Processing
- JSP Starter Sample for Data Access

Notes:

- JSP configuration, including specifics for a JServ environment, are covered in "Getting Started in a JServ Environment" on page 9-2.
- JSP pages will run with any standard browser supporting HTTP 1.0 or higher. The JDK or other Java environment in the end-user's Web browser is irrelevant, because all the Java code in a JSP page is executed in the Web server or data server.

Application Root and Doc Root Functionality

This section provides an overview of application roots and doc roots, distinguishing between servlet 2.2 functionality and servlet 2.0 functionality.

Application Roots in Servlet 2.2 Environments

As mentioned earlier, the servlet 2.2 specification provides for each application to have its own servlet context. Each servlet context is associated with a directory path in the server file system, which is the base path for modules of the application. This is the *application root*. Each application has its own application root.

This is similar to how a Web server uses a *doc root* as the root location for HTML pages and other files belonging to a Web application.

For an application in a servlet 2.2 environment, there is a one-to-one mapping between the application root (for servlets and JSP pages) and the doc root (for static files, such as HTML files)—they are essentially the same thing.

Note that a servlet URL has the following general form:

```
http://host[:port]/contextpath/servletpath
```

When a servlet context is created, a mapping is specified between the application root and the *context path* portion of a URL.

For example, consider an application with the application root `/home/dir/mybankappdir`, which is mapped to the context path `mybank`. Further assume the application includes a servlet whose servlet path is `loginservlet`. This servlet can be invoked as follows:

```
http://host[:port]/mybank/loginservlet
```

(The application root directory name itself is not visible to the end-user.)

To continue this example for an HTML page in this application, the following URL points to the file `/home/dir/mybankappdir/dir1/abc.html`:

```
http://host[:port]/mybank/dir1/abc.html
```

For each servlet environment there is also a *default* servlet context. For this context, the context path is simply "/", which is mapped to the default servlet context application root.

For example, assume the application root for the default context is `/home/mydefaultdir`, and a servlet with the servlet path `myservlet` uses the default context. Its URL would be as follows (again, the application root directory name itself is not visible to the user):

```
http://host[:port]/myservlet
```

(The default context is also used if there is no match for the context path specified in a URL.)

Continuing this example for an HTML file, the following URL points to the file `/home/mydefaultdir/dir2/def.html`:

```
http://host[:port]/dir2/def.html
```

Oracle Implementation of Application Root Functionality in Servlet 2.0 Environments

Apache JServ and other servlet 2.0 environments have no concept of application roots, because there is only a single application environment. The Web server doc root is effectively the application root.

For Apache, the doc root is typically some `.../htdocs` directory. In addition, it is possible to specify "virtual" doc roots through `alias` settings in the `httpd.conf` configuration file.

In a servlet 2.0 environment, the Oracle JSP container offers the following functionality regarding doc roots and application roots:

- By default, the Oracle JSP container uses the doc root as an application root.
- Through the Oracle `globals.jsa` mechanism, you can designate a directory under the doc root to serve as an application root for any given application. This is accomplished by placing a `globals.jsa` file as a marker in the desired directory. (See "Overview of globals.jsa Functionality" on page 9-26.)

Overview of JSP Applications and Sessions

This section provides a brief overview of how JSP applications and sessions are supported by the Oracle JSP container.

General Application and Session Support in the Oracle JSP Container

The Oracle JSP container uses underlying servlet mechanisms for managing applications and sessions. For servlet 2.1 and servlet 2.2 environments, these underlying mechanisms are sufficient, providing a distinct servlet context and session object for each JSP application.

Using the servlet mechanisms becomes problematic, however, in a servlet 2.0 environment such as JServ. The concept of a Web application was not well defined in the servlet 2.0 specification, so in a servlet 2.0 environment there is only one servlet context per servlet container. Additionally, there is one session object only per servlet container. However, for JServ and other servlet 2.0 environments, Oracle provides extensions to optionally allow distinct servlet contexts and session objects for each application. (This is unnecessary for Web servers hosting just a single application.)

Note: For additional information relevant to JServ and other servlet 2.0 environments, see "Considerations for JServ Servlet Environments" on page 9-20 and "Overview of globals.jsa Functionality" on page 9-26.

JSP Default Session Requests

Generally speaking, servlets do *not* request an HTTP session by default. However, JSP page implementation classes *do* request an HTTP session by default. You can override this by setting the `session` parameter to `false` in a JSP `page` directive, as follows:

```
<%@ page ... session="false" %>
```

JSP-Servlet Interaction

Although coding JSP pages is convenient in many ways, some situations call for servlets. One example is when you are outputting binary data, as discussed in "Reasons to Avoid Binary Data in JSP Pages" on page 4-16.

Therefore, it is sometimes necessary to go back and forth between servlets and JSP pages in an application. This section discusses how to accomplish this, covering the following topics:

- Invoking a Servlet from a JSP Page
- Passing Data to a Servlet Invoked from a JSP Page
- Invoking a JSP Page from a Servlet
- Passing Data Between a JSP Page and a Servlet
- JSP-Servlet Interaction Samples

Important: This discussion assumes a servlet 2.2 environment. Appropriate reference is made to other sections of this document for related considerations for JServ and other servlet 2.0 environments.

Invoking a Servlet from a JSP Page

As when invoking one JSP page from another, you can invoke a servlet from a JSP page through the `jsp:include` and `jsp:forward` action tags. (See "JSP Actions and the <jsp: > Tag Set" on page 1-18.) Following is an example:

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

When this statement is encountered during page execution, the page buffer is output to the browser and the servlet is executed. When the servlet has finished executing, control is transferred back to the JSP page and the page continues executing. This is the same functionality as for `jsp:include` actions from one JSP page to another.

And as with `jsp:forward` actions from one JSP page to another, the following statement would clear the page buffer, terminate the execution of the JSP page, and execute the servlet:

```
<jsp:forward page="/servlet/MyServlet" />
```

Important: You cannot include or forward to a servlet in JServ or other servlet 2.0 environments; you would have to write a JSP wrapper page instead. For information, see "Dynamic Includes and Forwards in JServ" on page 9-20.

Passing Data to a Servlet Invoked from a JSP Page

When dynamically including or forwarding to a servlet from a JSP page, you can use a `jsp:param` tag to pass data to the servlet (the same as when including or forwarding to another JSP page).

A `jsp:param` tag is used within a `jsp:include` or `jsp:forward` tag. Consider the following example:

```
<jsp:include page="/servlet/MyServlet" flush="true" >
   <jsp:param name="username" value="Smith" />
   <jsp:param name="userempno" value="9876" />
</jsp:include>
```

For more information about the `jsp:param` tag, see "JSP Actions and the <jsp: > Tag Set" on page 1-18.

Alternatively, you can pass data between a JSP page and a servlet through an appropriately scoped JavaBean or through attributes of the HTTP request object. Using attributes of the request object is discussed later, in "Passing Data Between a JSP Page and a Servlet" on page 3-8.

Note: The `jsp:param` tag was introduced in the JSP 1.1 specification.

Invoking a JSP Page from a Servlet

You can invoke a JSP page from a servlet through functionality of the standard `javax.servlet.RequestDispatcher` interface. Complete the following steps in your code to use this mechanism.

1. Get a servlet context instance from the servlet instance:

   ```
   ServletContext sc = this.getServletContext();
   ```

2. Get a request dispatcher from the servlet context instance, specifying the page-relative or application-relative path of the target JSP page as input to the `getRequestDispatcher()` method:

   ```
   RequestDispatcher rd = sc.getRequestDispatcher("/jsp/mypage.jsp");
   ```

   Prior to or during this step, you can optionally make data available to the JSP page through attributes of the HTTP request object. See "Passing Data Between a JSP Page and a Servlet" below for information.

3. Invoke the `include()` or `forward()` method of the request dispatcher, specifying the HTTP request and response objects as arguments. For example:

   ```
   rd.include(request, response);
   ```

   or:

   ```
   rd.forward(request, response);
   ```

   The functionality of these methods is similar to that of `jsp:include` and `jsp:forward` actions. The `include()` method only temporarily transfers control; execution returns to the invoking servlet afterward.

   Note that the `forward()` method clears the output buffer.

Notes:

- The request and response objects would have been obtained earlier using standard servlet functionality, such as the `doGet()` method specified in the `javax.servlet.http.HttpServlet` class.
- This functionality was introduced in the servlet 2.1 specification.

Passing Data Between a JSP Page and a Servlet

The preceding section, "Invoking a JSP Page from a Servlet", notes that when you invoke a JSP page from a servlet through the request dispatcher, you can optionally pass data through the HTTP request object.

You can accomplish this using either of the following approaches.

- You can append a query string to the URL when you obtain the request dispatcher, using "?" syntax with *`name`*=*`value`* pairs. For example:

  ```
  RequestDispatcher rd =
         sc.getRequestDispatcher("/jsp/mypage.jsp?username=Smith");
  ```

  In the target JSP page (or servlet), you can use the `getParameter()` method of the implicit `request` object to obtain the value of a parameter set in this way.

- You can use the `setAttribute()` method of the HTTP request object. For example:

  ```
  request.setAttribute("username", "Smith");
  RequestDispatcher rd = sc.getRequestDispatcher("/jsp/mypage.jsp");
  ```

  In the target JSP page (or servlet), you can use the `getAttribute()` method of the implicit `request` object to obtain the value of a parameter set in this way.

Notes:

- This functionality was introduced in the servlet 2.1 specification. Be aware that the semantics are different between the servlet 2.1 specification and the servlet 2.2 specification—in a servlet 2.1 environment a given attribute can be set only once.
- Mechanisms discussed in this section can be used instead of the `jsp:param` tag to pass data from a JSP page to a servlet.

JSP-Servlet Interaction Samples

This section provides a JSP page and a servlet that use functionality described in the preceding sections. The JSP page `Jsp2Servlet.jsp` includes the servlet `MyServlet`, which includes another JSP page, `welcome.jsp`.

Code for Jsp2Servlet.jsp

```
<HTML>
<HEAD> <TITLE> JSP Calling Servlet Demo </TITLE> </HEAD>
<BODY>

<!-- Forward processing to a servlet -->
<% request.setAttribute("empid", "1234"); %>
<jsp:include page="/servlet/MyServlet?user=Smith" flush="true"/>

</BODY>
</HTML>
```

Code for MyServlet.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.PrintWriter;
import java.io.IOException;

public class MyServlet extends HttpServlet {
    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)
      throws IOException, ServletException {
      PrintWriter out= response.getWriter();
      out.println("<B><BR>User:" + request.getParameter("user"));
      out.println
          (", Employee number:" + request.getAttribute("empid") + "</B>");
      this.getServletContext().getRequestDispatcher("/jsp/welcome.jsp").
        include(request, response);
    }
}
```

Code for welcome.jsp

```
<HTML>
<HEAD> <TITLE> The Welcome JSP  </TITLE> </HEAD>
<BODY>

<H3> Welcome! </H3>
<P><B> Today is <%= new java.util.Date() %>.  Have a nice day! </B></P>
</BODY>
</HTML>
```

JSP Resource Management

The `javax.servlet.http` package offers a standard mechanism for managing session resources. Additionally, Oracle provides extensions for managing application, session, page, and request resources.

Standard Session Resource Management with HttpSessionBindingListener

A JSP page must appropriately manage resources acquired during its execution, such as JDBC connection, statement, and result set objects. The standard `javax.servlet.http` package provides the `HttpSessionBindingListener` interface and `HttpSessionBindingEvent` class to manage session-scoped resources. Through this mechanism, a session-scoped query bean could, for example, acquire a database cursor when the bean is instantiated and close it when the HTTP session is terminated. (The example in "JSP Starter Sample for Data Access" on page 3-19 opens and closes the connection for each query, which adds overhead.)

This section describes use of the `HttpSessionBindingListener` `valueBound()` and `valueUnbound()` methods.

Note: The bean instance must register itself in the event notification list of the HTTP session object, but the `jsp:useBean` statement takes care of this automatically.

The valueBound() and valueUnbound() Methods

An object that implements the `HttpSessionBindingListener` interface can implement a `valueBound()` method and a `valueUnbound()` method, each of which takes an `HttpSessionBindingEvent` instance as input. These methods are called by the servlet container—the `valueBound()` method when the object is stored in the session; the `valueUnbound()` method when the object is removed from the session or when the session times-out or becomes invalid. Usually, a developer will use `valueUnbound()` to release resources held by the object (in the example below, to release the database connection).

Note: Oracle9*i* release 2 provides extensions for additional resource management, allowing you to program JavaBeans to manage page-scoped, request-scoped, or application-scoped resources as well as session-scoped resources. See "Oracle JSP Event Handling with JspScopeListener" on page 5-2.

"JDBCQueryBean JavaBean Code" below provides a sample JavaBean that implements `HttpSessionBindingListener` and a sample JSP page that calls the bean.

JDBCQueryBean JavaBean Code

Following is the sample code for `JDBCQueryBean`, a JavaBean that implements the `HttpSessionBindingListener` interface. (It uses the JDBC OCI driver for its database connection; use an appropriate JDBC driver and connection string if you want to run this example yourself.)

`JDBCQueryBean` gets a search condition through the HTML request (as described in "The UseJDBCQueryBean JSP Page" on page 3-13), executes a dynamic query based on the search condition, and outputs the result.

This class also implements a `valueUnbound()` method (as specified in the `HttpSessionBindingListener` interface) that results in the database connection being closed at the end of the session.

```
package mybeans;

import java.sql.*;
import javax.servlet.http.*;

public class JDBCQueryBean implements HttpSessionBindingListener
{
  String searchCond = "";
  String result = null;

  public void JDBCQueryBean() {
  }

  public synchronized String getResult() {
    if (result != null) return result;
    else return runQuery();
  }
```

```
public synchronized void setSearchCond(String cond) {
  result = null;
  this.searchCond = cond;
}

private Connection conn = null;

private String runQuery() {
  StringBuffer sb = new StringBuffer();
  Statement stmt = null;
  ResultSet rset = null;
  try {
    if (conn == null) {
      DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
      conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                         "scott", "tiger");

    }

    stmt = conn.createStatement();
    rset = stmt.executeQuery ("SELECT ename, sal FROM scott.emp "+
          (searchCond.equals("") ? "" : "WHERE " + searchCond ));
    result = formatResult(rset);
    return result;

  } catch (SQLException e) {
    return ("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
  }
  finally {
    try {
      if (rset != null) rset.close();
      if (stmt != null) stmt.close();
    }
    catch (SQLException ignored) {}
  }
}

private String formatResult(ResultSet rset) throws SQLException  {
  StringBuffer sb = new StringBuffer();
  if (!rset.next())
    sb.append("<P> No matching rows.<P>\n");
  else {
    sb.append("<UL><B>");
    do {  sb.append("<LI>" + rset.getString(1) +
          " earns $ " + rset.getInt(2) + "</LI>\n");
      } while (rset.next());
      sb.append("</B></UL>");
    }
    return sb.toString();
  }

  public void valueBound(HttpSessionBindingEvent event) {
    // do nothing -- the session-scoped bean is already bound
  }

  public synchronized void valueUnbound(HttpSessionBindingEvent event) {
    try {
      if (conn != null) conn.close();
    }
    catch (SQLException ignored) {}
  }
}
```

Note: The preceding code serves as a sample only. This is not necessarily an advisable way to handle database connection pooling in a large-scale Web application.

The UseJDBCQueryBean JSP Page

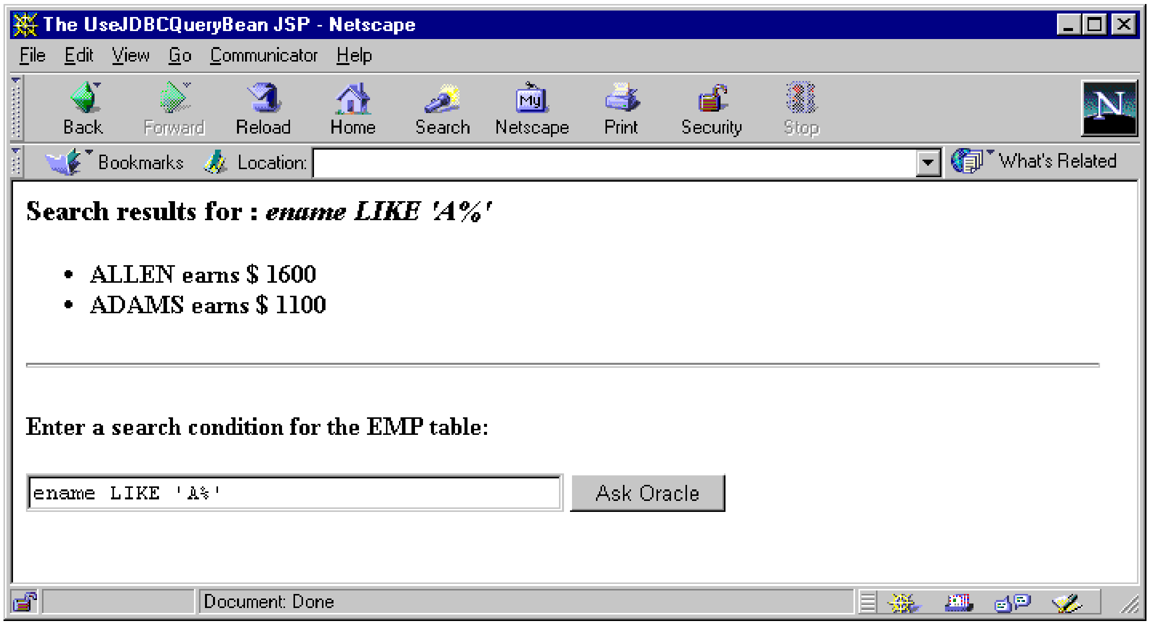
The following JSP page uses the `JDBCQueryBean` JavaBean defined in "JDBCQueryBean JavaBean Code" above, invoking the bean with `session` scope. It uses `JDBCQueryBean` to display employee names that match a search condition entered by the user.

`JDBCQueryBean` gets the search condition through the `jsp:setProperty` command in this JSP page, which sets the `searchCond` property of the bean according to the value of the `searchCond` request parameter input by the user through the HTML form. (The HTML `INPUT` tag is what specifies that the search condition entered in the form be named `searchCond`.)

```
<jsp:useBean id="queryBean" class="mybeans.JDBCQueryBean" scope="session" />
<jsp:setProperty name="queryBean" property="searchCond" />

<HTML>
<HEAD> <TITLE> The UseJDBCQueryBean JSP  </TITLE> </HEAD>
<BODY BGCOLOR="white">

<% String searchCondition = request.getParameter("searchCond");
```

```
   if (searchCondition != null) { %>
      <H3> Search results for : <I> <%= searchCondition %> </I> </H3>
      <%= queryBean.getResult() %>
      <HR><BR>
<% }  %>

<B>Enter a search condition for the EMP table:</B>

<FORM METHOD="get">
<INPUT TYPE="text" NAME="searchCond" VALUE="ename LIKE 'A%' " SIZE="40">
<INPUT TYPE="submit" VALUE="Ask Oracle">
</FORM>

</BODY>
</HTML>
```

Following is sample input and output for this page:


The UseJDBCQueryBean JSP - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location:
What's Related

Search results for : ***ename LIKE 'A%'***

- ALLEN earns $ 1600
- ADAMS earns $ 1100

Enter a search condition for the EMP table:

ename LIKE 'A%'

Ask Oracle

Document: Done

Advantages of HttpSessionBindingListener

In the preceding example, an alternative to the `HttpSessionBindingListener` mechanism would be to close the connection in a `finalize()` method in the JavaBean. The `finalize()` method would be called when the bean is garbage-collected after the session is closed. The `HttpSessionBindingListener` interface, however, has more predictable behavior than a `finalize()` method. Garbage collection frequency depends on the memory consumption pattern of the application. By contrast, the `valueUnbound()` method of the `HttpSessionBindingListener` interface is called reliably at session shutdown.

Overview of Oracle Extensions for Resource Management

Oracle provides the following extensions for managing application and session resources as well as page and request resources:

- `JspScopeListener`—for managing application, session, page, or request resources

  For information, see "Oracle JSP Event Handling with JspScopeListener" on page 5-2.

- `globals.jsa` application and session events—for start and end events for applications and sessions, typically in a servlet 2.0 environment such as JServ

  See "The globals.jsa Event Handlers" on page 9-31 for information.

JSP Runtime Error Processing

While a JSP page is executing and processing client requests, runtime errors can occur either inside the page or outside the page (such as in a called JavaBean). This section describes the JSP error processing mechanism and provides a simple example.

Using JSP Error Pages

Any runtime error encountered during execution of a JSP page is handled using the standard Java exception mechanism in one of two ways:

- You can catch and handle exceptions in a Java scriptlet within the JSP page itself, using standard Java exception-handling code.
- Exceptions you do not catch in the JSP page will result in forwarding of the request and uncaught exception to an error page. This is the preferred way to handle JSP errors.

You can specify the URL of an error page by setting the `errorPage` parameter in a `page` directive in the originating JSP page. (For an overview of JSP directives, including the `page` directive, see "Directives" on page 1-10.)

An error page must have a `page` directive setting the `isErrorPage` parameter to `true`.

The exception object describing the error is a `java.lang.Exception` instance that is accessible in the error page through the implicit `exception` object.

Only an error page can access the implicit `exception` object. (For information about JSP implicit objects, including the `exception` object, see "Implicit Objects" on page 1-15.)

See "JSP Error Page Example" below for an example of error page usage.

Note: There is ambiguity in the JSP 1.1 specification regarding exception types that can be handled through the JSP mechanism.

In the Oracle JSP container, a page implementation class generated by the translator can handle an instance of the `java.lang.Exception` class or a subclass, but cannot handle an instance of the `java.lang.Throwable` class or any subclass other than `Exception`. A `Throwable` instance will be thrown by the JSP container to the servlet container.

The ambiguity is expected to be addressed in the JSP 1.2 specification. The Oracle behavior will be modified appropriately in a future release.

JSP Error Page Example

The following example, `nullpointer.jsp`, generates an error and uses an error page, `myerror.jsp`, to output contents of the implicit `exception` object.

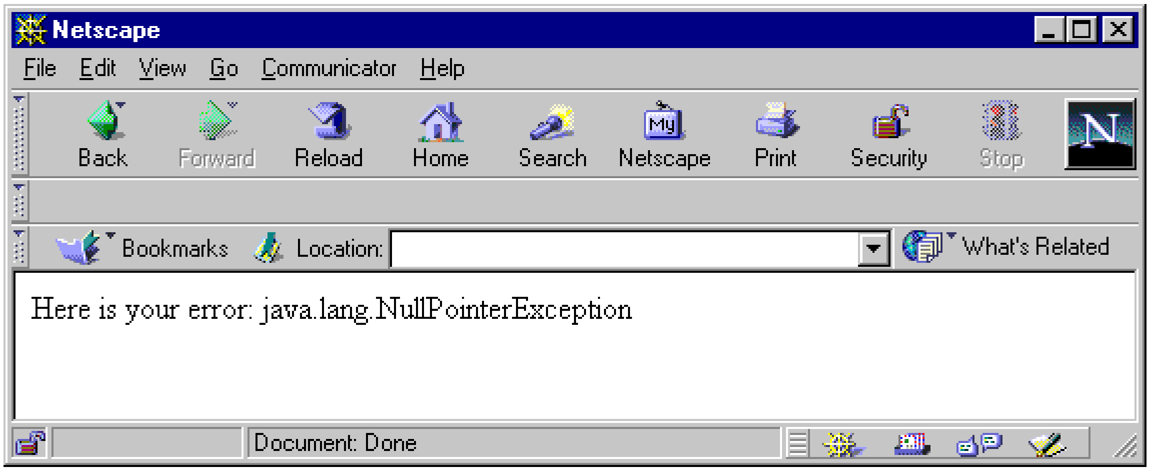
Code for nullpointer.jsp

```
<HTML>
<BODY>
<%@ page errorPage="myerror.jsp" %>
Null pointer is generated below:
<%
   String s=null;
   s.length();
%>
</BODY>
</HTML>
```

Code for myerror.jsp

```
<HTML>
<BODY>
<%@ page isErrorPage="true" %>
Here is your error:
<%= exception %>
</BODY>
</HTML>
```

This example results in the following output:


Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: What's Related
Document: Done

```
Here is your error: java.lang.NullPointerException
```

Note: The line "Null pointer is generated below:" in `nullpointer.jsp` is not output when processing is forwarded to the error page. This shows the difference between `jsp:include` and `jsp:forward` functionality—with `jsp:forward`, the output from the "forward-to" page *replaces* the output from the "forward-from" page.

JSP Starter Sample for Data Access

Chapter 1, "General Overview", provides a couple of simple JSP examples; however, if you are using the Oracle JSP container, you presumably want to access an Oracle database. This section offers a more interesting sample that uses standard JDBC code in a JSP page to perform a query.

Because the JDBC API is simply a set of Java interfaces, JavaServer Pages technology directly supports its use within JSP scriptlets.

Notes:

- Oracle JDBC provides several driver alternatives: 1) the JDBC OCI driver for use with an Oracle client installation; 2) a 100%-Java JDBC Thin driver that can be used in essentially any client situation, including applets; 3) a JDBC server-side Thin driver to access one Oracle database from within another Oracle database; and 4) a JDBC server-side internal driver to access the database within which the Java code is running (such as from a Java stored procedure). For more information about Oracle JDBC, see the *Oracle9i JDBC Developer's Guide and Reference.*
- The Oracle JSP container also supports SQLJ (embedded SQL in Java) for static SQL operations. This is discussed in "Oracle JSP Support for Oracle SQLJ" on page 5-3.

The following example creates a query dynamically from search conditions the user enters through an HTML form (typed into a box and entered with an `Ask Oracle` button). To perform the specified query, it uses JDBC code in a method called `runQuery()` that is defined in a JSP declaration. It also defines a method `formatResult()` within the JSP declaration to produce the output. The `runQuery()` method uses the `scott` schema with password `tiger`.

The HTML `INPUT` tag specifies that the string entered in the form be named `cond`. Therefore, `cond` is also the input parameter to the `getParameter()` method of the implicit `request` object for this HTTP request, and the input parameter to the `runQuery()` method (which puts the `cond` string into the `WHERE` clause of the query).

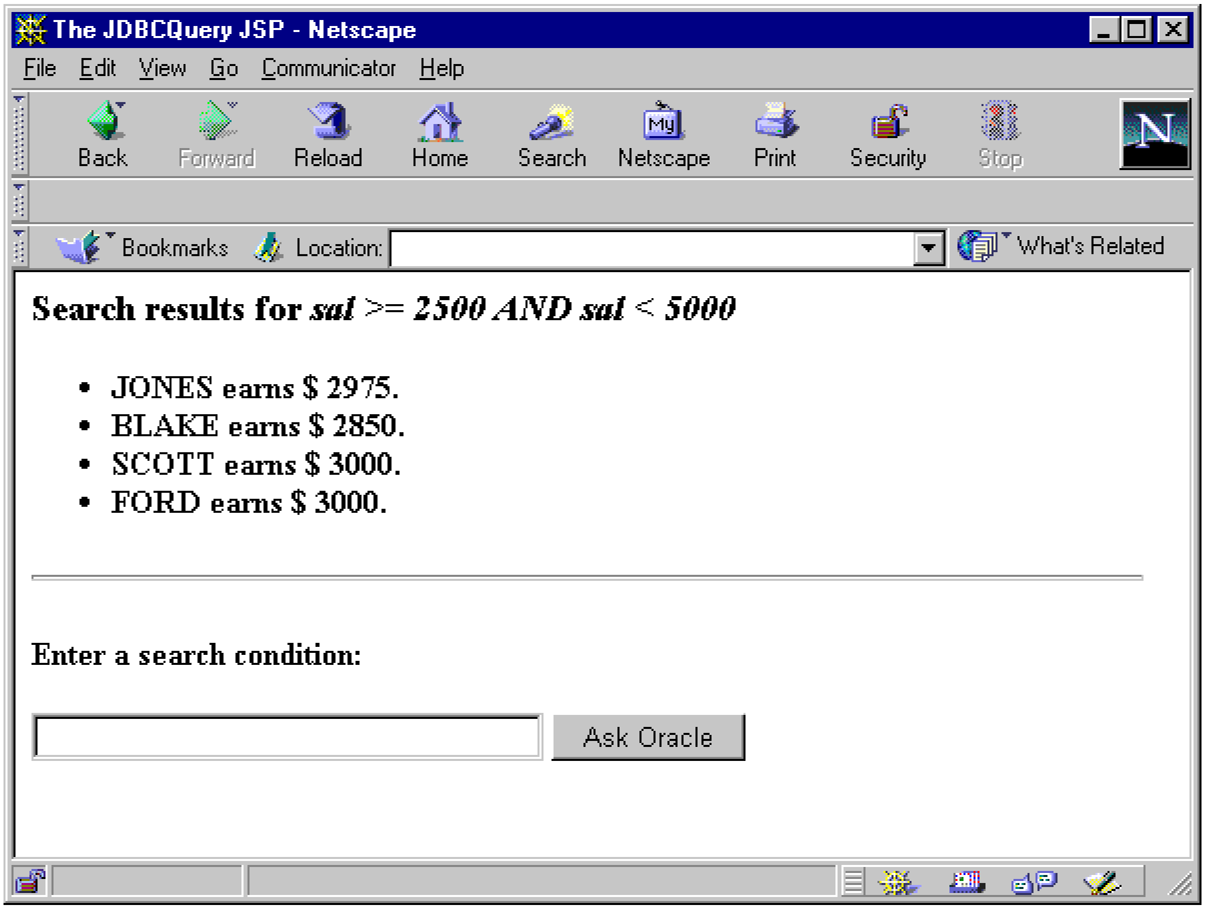
Notes:

- Another approach to this example would be to define the `runQuery()` method in `<%...%>` scriptlet syntax instead of `<%!...%>` declaration syntax.
- This example uses the JDBC OCI driver, which requires an Oracle client installation. If you want to run this sample, use an appropriate JDBC driver and connection string.

```
<%@ page language="java" import="java.sql.*" %>
<HTML>
<HEAD> <TITLE> The JDBCQuery JSP  </TITLE> </HEAD>
<BODY BGCOLOR="white">
<% String searchCondition = request.getParameter("cond");
   if (searchCondition != null) { %>
      <H3> Search results for  <I> <%= searchCondition %> </I> </H3>
      <B> <%= runQuery(searchCondition) %> </B> <HR><BR>
<% }  %>
<B>Enter a search condition:</B>
<FORM METHOD="get">
<INPUT TYPE="text" NAME="cond" SIZE=30>
<INPUT TYPE="submit" VALUE="Ask Oracle");
</FORM>
</BODY>
</HTML>
<%-- Declare and define the runQuery() method. --%>
<%! private String runQuery(String cond) throws SQLException {
     Connection conn = null;
     Statement stmt = null;
     ResultSet rset = null;
     try {
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
        conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                           "scott", "tiger");
        stmt = conn.createStatement();
        // dynamic query
        rset = stmt.executeQuery ("SELECT ename, sal FROM scott.emp "+
                           (cond.equals("") ? "" : "WHERE " + cond ));
       return (formatResult(rset));
     } catch (SQLException e) {
         return ("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
     } finally {
        if (rset!= null) rset.close();
        if (stmt!= null) stmt.close();
        if (conn!= null) conn.close();
    }
  }
  private String formatResult(ResultSet rset) throws SQLException {
    StringBuffer sb = new StringBuffer();
    if (!rset.next())
      sb.append("<P> No matching rows.<P>\n");
    else {  sb.append("<UL>");
            do {  sb.append("<LI>" + rset.getString(1) +
                            " earns $ " + rset.getInt(2) + ".</LI>\n");
            } while (rset.next());
           sb.append("</UL>");
    }
    return sb.toString();
  }
%>
```

The graphic below illustrates sample output for the following input:

```
sal >= 2500 AND sal < 5000
```


The JDBCQuery JSP - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location:
What's Related

Search results for *sal >= 2500 AND sal < 5000*

- JONES earns $ 2975.
- BLAKE earns $ 2850.
- SCOTT earns $ 3000.
- FORD earns $ 3000.

Enter a search condition:

4 Key Considerations

This chapter discusses important programming, configurational, and runtime considerations in developing a JSP application. The following topics are covered:

- General JSP Programming Strategies, Tips, and Traps
- Key JSP Configuration Issues
- Oracle JSP Runtime Page and Class Reloading

General JSP Programming Strategies, Tips, and Traps

This section discusses issues you should consider when programming JSP pages that will run in the Oracle JSP container, regardless of the particular target environment. The following assortment of topics are covered:

- JavaBeans Versus Scriptlets
- Use of JDBC Performance Enhancement Features
- Static Includes Versus Dynamic Includes
- When to Consider Creating and Using JSP Tag Libraries
- Use of a Central Checker Page
- Workarounds for Large Static Content in JSP Pages
- Method Variable Declarations Versus Member Variable Declarations
- Page Directive Characteristics
- JSP Preservation of White Space and Use with Binary Data
- Oracle XML Support

Note: In addition to being aware of what is discussed in this section, you should be aware of Oracle JSP translation and deployment issues and behavior. See Chapter 6, "JSP Translation and Deployment".

JavaBeans Versus Scriptlets

The section "Separation of Business Logic from Page Presentation: Calling JavaBeans" on page 1-5 describes a key advantage of JavaServer Pages technology: Java code containing the business logic and determining the dynamic content can be separated from the HTML code containing the request processing, presentation logic, and static content. This separation allows HTML experts to focus on presentation logic in the JSP page itself, while Java experts focus on business logic in JavaBeans that are called from the JSP page.

A typical JSP page will have only brief snippets of Java code, usually for Java functionality for request processing or presentation. The sample page in "JSP Starter Sample for Data Access" on page 3-19, although illustrative, is probably not an ideal design. Data access, such as in the `runQuery()` method in the sample, is usually more appropriate in a JavaBean. However, the `formatResult()` method in the sample, which formats the output, is more appropriate for the JSP page itself.

Use of JDBC Performance Enhancement Features

You can use the following performance enhancement features, supported through Oracle JDBC extensions, in JSP applications executed by the Oracle JSP container:

- caching database connections
- caching JDBC statements
- batching update statements
- prefetching rows during a query
- caching rowsets

Most of these performance features are supported by the Oracle `ConnBean` and `ConnCacheBean` data-access JavaBeans (but not by `DBBean`). These beans are described in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Database Connection Caching

Creating a new database connection is an expensive operation that you should avoid whenever possible. Instead, use a cache of database connections. A JSP application can get a logical connection from a pre-existing pool of physical connections, and return the connection to the pool when done.

You can create a connection pool at any one of the four JSP scopes—`application`, `session`, `page`, or `request`. It is most efficient to use the maximum possible scope—`application` scope if that is permitted by the Web server, or `session` scope if not.

The Oracle JDBC connection caching scheme, built upon standard connection pooling as specified in the JDBC 2.0 standard extensions, is implemented in the `ConnCacheBean` data-access JavaBean provided with Oracle9*i*. This is probably how most JSP developers will use connection caching. This bean is described in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

It is also possible to use the Oracle JDBC `OracleConnectionCacheImpl` class directly, as though it were a JavaBean, as in the following example (although all `OracleConnectionCacheImpl` functionality is available through `ConnCacheBean`).

```
<jsp:useBean id="occi" class="oracle.jdbc.pool.OracleConnectionCacheImpl"
             scope="session" />
```

The same properties are available in `OracleConnectionCacheImpl` as in `ConnCacheBean`. They can be set either through `jsp:setProperty` statements or directly through the class setter methods.

For information about the Oracle JDBC connection caching scheme and the `OracleConnectionCacheImpl` class, see the *Oracle9i JDBC Developer's Guide and Reference*.

JDBC Statement Caching

Statement caching, an Oracle JDBC extension, improves performance by caching executable statements that are used repeatedly within a single physical connection, such as in a loop or in a method that is called repeatedly. When a statement is cached, the statement does not have to be re-parsed, the statement object does not have to be recreated, and parameter size definitions do not have to be recalculated each time the statement is executed.

The Oracle JDBC statement caching scheme is implemented in the `ConnBean` and `ConnCacheBean` data-access JavaBeans that are provided with Oracle9*i*. Each of these beans has a `stmtCacheSize` property that can be set through a `jsp:setProperty` statement or the `setStmtCacheSize()` method of the bean. The beans are described in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Statement caching is also available directly through the Oracle JDBC `OracleConnection` and `OracleConnectionCacheImpl` classes. For information about the Oracle JDBC statement caching scheme and the `OracleConnection` and `OracleConnectionCacheImpl` classes, see the *Oracle9i JDBC Developer's Guide and Reference*.

Important: Statements can be cached only within a single physical connection. When you enable statement caching for a connection cache, statements can be cached across multiple logical connection objects from a single pooled connection object, but not across multiple pooled connection objects.

Update Batching

The Oracle JDBC update batching feature associates a batch value (limit) with each prepared statement object. With update batching, instead of the JDBC driver executing a prepared statement each time its "execute" method is called, the driver adds the statement to a batch of accumulated execution requests. The driver will pass all the operations to the database for execution once the batch value is reached. For example, if the batch value is 10, then each batch of ten operations will be sent to the database and processed in one trip.

The Oracle JSP container supports Oracle JDBC update batching directly, through the `executeBatch` property of the `ConnBean` data-access JavaBean. You can set this property through a `jsp:setProperty` statement or through the setter method of the bean. If you use `ConnCacheBean` instead, you can enable update batching through Oracle JDBC functionality in the connection and statement objects you create. These beans are described in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

For more information about Oracle JDBC update batching, see the *Oracle9i JDBC Developer's Guide and Reference*.

Row Prefetching

The Oracle JDBC row prefetching feature allows you to set the number of rows to prefetch into the client during each trip to the database while a result set is being populated during a query, reducing the number of round trips to the server.

The Oracle JSP container supports Oracle JDBC row prefetching directly, through the `preFetch` property of the `ConnBean` data-access JavaBean. You can set this property through a `jsp:setProperty` statement or through the setter method of the bean. If you use `ConnCacheBean` instead, you can enable row prefetching through Oracle JDBC functionality in the connection and statement objects you create. These beans are described in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

For more information about Oracle JDBC row prefetching, see the *Oracle9i JDBC Developer's Guide and Reference*.

Rowset Caching

A cached rowset provides a disconnected, serializable, and scrollable container for retrieved data. This feature is useful for small sets of data that do not change often, particularly when the client requires frequent or continued access to the information. By contrast, using a normal result set requires the underlying connection and other resources to be held. Be aware, however, that large cached rowsets consume a lot of memory on the client.

In Oracle9*i*, Oracle JDBC provides a cached rowset implementation. If you are using an Oracle JDBC driver, use code inside a JSP page to create and populate a cached rowset as follows:

```
CachedRowSet crs = new CachedRowSet();
crs.populate(rset); // rset is a previously created JDBC ResultSet object.
```

Once the rowset is populated, the connection and statement objects used in obtaining the original result set can be closed.

For more information about Oracle JDBC cached rowsets, see the *Oracle9i JDBC Developer's Guide and Reference*.

Static Includes Versus Dynamic Includes

The `include` directive, described in "Directives" on page 1-10, makes a copy of the included page and copies it into a JSP page (the "including page") during translation. This is known as a *static include* (or *translate-time include*) and uses the following syntax:

```
<%@ include file="/jsp/userinfopage.jsp" %>
```

The `jsp:include` action, described in "JSP Actions and the <jsp: > Tag Set" on page 1-18, dynamically includes output from the included page within the output of the including page, during runtime. This is known as a *dynamic include* (or *runtime include*) and uses the following syntax:

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

For those of you who are familiar with C syntax, a static include is comparable to a `#include` statement. A dynamic include is similar to a function call. They are both useful, but serve different purposes.

Note: Both static includes and dynamic includes can be used only between pages in the same servlet context.

Logistics of Static Includes

A static include increases the size of the generated code for the including JSP page, as though the text of the included page is physically copied into the including page during translation (at the point of the `include` directive). If a page is included multiple times within an including page, multiple copies are made.

A JSP page that is statically included does not need to stand as an independent, translatable entity. It simply consists of text that will be copied into the including page. The including page, with the included text copied in, must then be translatable. And, in fact, the including page does not have to be translatable prior to having the included page copied into it. A sequence of statically included pages can each be fragments unable to stand on their own.

Logistics of Dynamic Includes

A dynamic include does *not* significantly increase the size of the generated code for the including page, although method calls, such as to the request dispatcher, will be added. The dynamic include results in runtime processing being switched from the including page to the included page, as opposed to the text of the included page being physically copied into the including page.

A dynamic include *does* increase processing overhead, with the necessity of the additional call to the request dispatcher.

A page that is dynamically included must be an independent entity, able to be translated and executed on its own. Likewise, the including page must be independent as well, able to be translated and executed without the dynamic include.

Advantages, Disadvantages, and Typical Uses

Static includes affect page size; dynamic includes affect processing overhead. Static includes avoid the overhead of the request dispatcher that a dynamic include necessitates, but may be problematic where large files are involved. (There is a 64K size limit on the service method of the generated page implementation class—see "Workarounds for Large Static Content in JSP Pages" on page 4-10.)

Overuse of static includes can also make debugging your JSP pages difficult, making it harder to trace program execution. Avoid subtle interdependencies between your statically included pages.

Static includes are typically used to include small files whose content is used repeatedly in multiple JSP pages. For example:

- Statically include a logo or copyright message at the top or bottom of each page in your application.
- Statically include a page with declarations or directives (such as imports of Java classes) that are required in multiple pages.

- Statically include a central "status checker" page from each page of your application. (See "Use of a Central Checker Page" on page 4-9.)

Dynamic includes are useful for modular programming. You may have a page that sometimes executes on its own but sometimes is used to generate some of the output of other pages. Dynamically included pages can be reused in multiple including pages without increasing the size of the including pages.

When to Consider Creating and Using JSP Tag Libraries

Some situations dictate that the development team consider creating and using custom tags. In particular, consider the following situations:

- JSP pages would otherwise have to include a significant amount of Java logic regarding presentation and format of output.
- Special manipulation or redirection of JSP output is required.

Replacing Java Syntax

Because one cannot count on JSP developers being experienced in Java programming, they may not be ideal candidates for coding Java logic in the page—logic that dictates presentation and format of the JSP output, for example.

This is a situation where JSP tag libraries might be helpful. If many of your JSP pages will require such logic in generating their output, a tag library to replace Java logic would be a great convenience for JSP developers.

An example of this is the JML sample tag library provided with Oracle9*i*. This library, documented in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*, includes tags that support logic equivalent to Java loops and conditionals.

Manipulating or Redirecting JSP Output

Another common situation for custom tags is if special runtime processing of the response output is required. Perhaps the desired functionality requires an extra processing step or redirection of the output to somewhere other than the browser.

An example is to create a custom tag that you can place around a body of text whose output will be redirected into a log file instead of to a browser, such as in the following example (where `cust` is the prefix for the tag library and `log` is one of the library’s tags):

```
<cust:log>
   Today is <%= new java.util.Date() %>
   Text to log.
   More text to log.
   Still more text to log.
</cust:log>
```

See "Tag Handlers" on page 7-4 for information about processing of tag bodies.

Use of a Central Checker Page

For general management or monitoring of your JSP application, it may be useful to use a central "checker" page that you include from each page in your application. A central checker page could accomplish tasks such as the following during execution of each page:

- Check session status.
- Check login status (such as checking the cookie to see if a valid login has been accomplished).
- Check usage profile (if a logging mechanism has been implemented to tally events of interest, such as mouse clicks or page visits).

There could be many more uses as well.

As an example, consider a session checker class, `MySessionChecker`, that implements the `HttpSessionBindingListener` interface. (See "Standard Session Resource Management with HttpSessionBindingListener" on page 3-10.)

```
public class MySessionChecker implements HttpSessionBindingListener
{
   ...

   valueBound(HttpSessionBindingEvent event)
   {...}

   valueUnbound(HttpSessionBindingEvent event)
   {...}

   ...
}
```

You can create a checker JSP page, suppose `centralcheck.jsp`, that includes something like the following:

```
<jsp:useBean id="sessioncheck" class="MySessionChecker" scope="session" />
```

In any page that includes `centralcheck.jsp`, the servlet container will call the `valueUnbound()` method implemented in the `MySessionChecker` class as soon as `sessioncheck` goes out of scope (at the end of the session). Presumably this is to manage session resources. You could include `centralcheck.jsp` at the end of each JSP page in your application.

Workarounds for Large Static Content in JSP Pages

JSP pages with large amounts of static content (essentially, large amounts of HTML code without content that changes at runtime) may result in slow translation and execution.

There are two primary workarounds for this (either workaround will speed translation):

- Put the static HTML into a separate file and use a dynamic `include` command (`jsp:include`) to include its output in the JSP page output at runtime. See "JSP Actions and the <jsp: > Tag Set" on page 1-18 for information about the `jsp:include` command.

  Important: A static `<%@ include... %>` command would not work. It would result in the included file being included at translation time, with its code being effectively copied back into the including page. This would not solve the problem.

- Put the static HTML into a Java resource file.

  The Oracle JSP container will do this for you if you enable the JSP `external_resource` configuration parameter. This parameter is documented in "Oracle JSP Configuration Parameters" on page 9-7.

  For pre-translation, the `-extres` option of the `ojspc` command-line tool also offers this functionality.

  Note: Putting static HTML into a resource file may result in a larger memory footprint than the `jsp:include` workaround mentioned above, because the page implementation class must load the resource file whenever the class is loaded.

Another possible, though unlikely, problem with JSP pages that have large static content is that most (if not all) JVMs impose a 64K byte size limit on the code within any single method. Although `javac` would be able to compile it, the JVM would be unable to execute it. Depending on the implementation of the JSP translator, this may become an issue for a JSP page, because generated Java code from essentially the entire JSP page source file goes into the service method of the page implementation class. (Java code is generated to output the static HTML to the browser, and Java code from any scriptlets is copied directly.)

Another possible, though rare, scenario is for the Java scriptlets in a JSP page to be large enough to create a size limit problem in the service method. If there is enough Java code in a page to create a problem, however, then the code should be moved into JavaBeans.

Method Variable Declarations Versus Member Variable Declarations

In "Scripting Elements" on page 1-12, it is noted that JSP `<%! ... %>` declarations are used to declare member variables, while method variables must be declared in `<% ... %>` scriptlets.

Be careful to use the appropriate mechanism for each of your declarations, depending on how you want to use the variables:

- A variable that is declared in `<%! ... %>` JSP declaration syntax is declared at the class level in the page implementation class that is generated by the JSP translator.
- A variable that is declared in `<% ... %>` JSP scriptlet syntax is local to the service method of the page implementation class.

Consider the following example, `decltest.jsp`:

```
<HTML>
<BODY>
<% double f2=0.0; %>
<%! double f1=0.0; %>
Variable declaration test.
</BODY>
</HTML>
```

This results in something like the following code in the page implementation class:

```
package ...;
import ...;
```

```
public class decltest extends oracle.jsp.runtime.HttpJsp {
   ...

   // ** Begin Declarations
   double f1=0.0;                  // *** f1 declaration is generated here ***
   // ** End Declarations

   public void _jspService
                (HttpServletRequest request, HttpServletResponse response)
                throws IOException, ServletException {
      ...

      try {
         out.println( "<HTML>");
         out.println( "<BODY>");
         double f2=0.0;      // *** f2 declaration is generated here ***
         out.println( "");
         out.println( "");
         out.println( "Variable declaration test.");
         out.println( "</BODY>");
         out.println( "</HTML>");
         out.flush();
      }
      catch( Exception e) {
         try {
            if (out != null) out.clear();
         }
         catch( Exception clearException) {
         }
      finally {
         if (out != null) out.close();
      }
   }
}
```

Note: This code is provided for conceptual purposes only. Most of the class is deleted for simplicity, and the actual code of a page implementation class generated by the Oracle JSP translator would differ somewhat.

Page Directive Characteristics

This section discusses the following `page` directive characteristics:

- A `page` directive is static and takes effect during translation; you cannot specify parameter settings to be evaluated at runtime.
- Java `import` settings in `page` directives are cumulative within a JSP page.

Page Directives Are Static

A `page` directive is static; it is interpreted during translation. You cannot specify dynamic settings to be interpreted at runtime. Consider the following examples:

Example 1 The following `page` directive is *valid*.

```
<%@ page contentType="text/html; charset=EUCJIS" %>
```

Example 2 The following `page` directive is *not valid* and will result in an error. (`EUCJIS` is hard-coded here, but the example also holds true for any character set determined dynamically at runtime.)

```
<% String s="EUCJIS"; %>
<%@ page contentType="text/html; charset=<%=s%>" %>
```

For some `page` directive settings there are workarounds. Reconsidering Example 2, there is a `setContentType()` method that allows dynamic setting of the content type, as described in "Dynamic Content Type Settings" on page 8-4.

Page Directive Import Settings Are Cumulative

Java `import` settings in `page` directives within a JSP page are cumulative.

Within any single JSP page, the following two examples are equivalent:

```
<%@ page language="java" %>
<%@ page import="sqlj.runtime.ref.DefaultContext, java.sql.*" %>
```

or:

```
<%@ page language="java" %>
<%@ page import="sqlj.runtime.ref.DefaultContext" %>
<%@ page import="java.sql.*" %>
```

After the first `page` directive `import` setting, the `import` setting in the second `page` directive adds to the set of classes or packages to be imported, as opposed to replacing the classes or packages to be imported.

JSP Preservation of White Space and Use with Binary Data

The Oracle JSP container (and JavaServer Pages implementations in general) preserves source code white space, including carriage returns and linefeeds, in what is output to the browser. Insertion of such white space may not be what the developer intended, and typically makes JSP technology a poor choice for generating binary data.

White Space Examples

The following two JSP pages produce different HTML output, due to the use of carriage returns in the source code.

Example 1—No Carriage Returns

The following JSP page does *not* have carriage returns after the `Date()` and `getParameter()` calls. (The third and fourth lines, starting with the `Date()` call, actually comprise a single wraparound line of code.)

`nowhitsp.jsp`:

```
<HTML>
<BODY>
<%= new java.util.Date() %> <% String user=request.getParameter("user"); %> <%=
(user==null) ? "" : user %>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

This results in the following HTML output to the browser. (Note that there are no blank lines after the date.)

```
<HTML>
<BODY>
Tue May 30 20:07:04 PDT 2000
<B>Enter name:</B>
<FORM METHOD=get>
```

```
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

Example 2—Carriage Returns

The following JSP page *does* include carriage returns after the `Date()` and `getParameter()` calls.

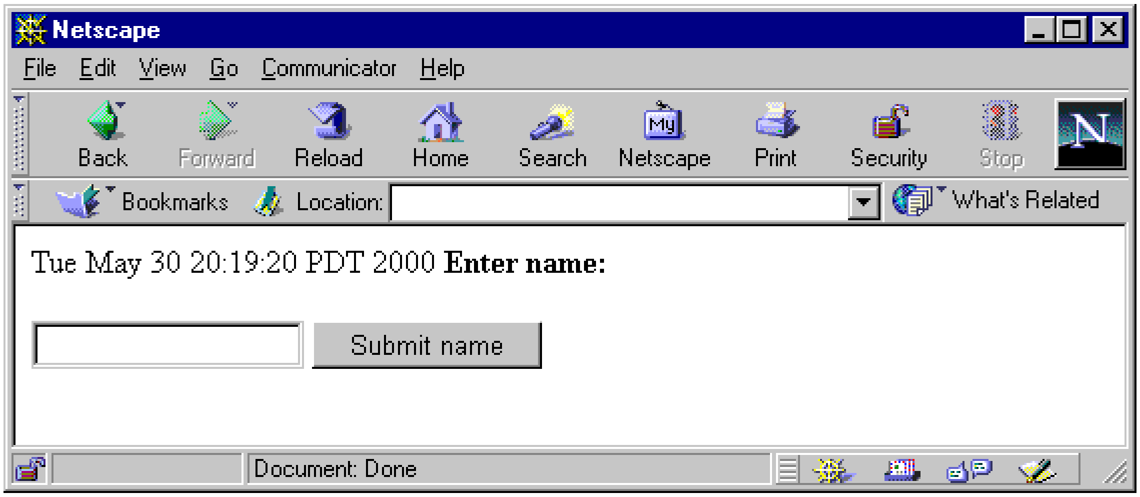
`whitesp.jsp`:

```
<HTML>
<BODY>
<%= new java.util.Date() %>
<% String user=request.getParameter("user"); %>
<%= (user==null) ? "" : user %>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

This results in the following HTML output to the browser.

```
<HTML>
<BODY>
Tue May 30 20:19:20 PDT 2000


<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

Note the two blank lines between the date and the "Enter name:" line. In this particular case the difference is not significant, because both examples produce the same appearance in the browser, as shown below. However, this discussion nevertheless demonstrates the general point about preservation of white space.


Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: What's Related

Tue May 30 20:19:20 PDT 2000 **Enter name:**

Submit name

Document: Done

Reasons to Avoid Binary Data in JSP Pages

For the following reasons, JSP pages are a poor choice for generating binary data. Generally you should use servlets instead.

- JSP implementations are not designed to handle binary data—there are no methods for writing raw bytes in the `JspWriter` object.
- During execution, the JSP container preserves whitespace. Whitespace is sometimes unwanted, making JSP pages a poor choice for generating binary output (a `.gif` file, for example) to the browser, or for other uses where whitespace is significant.

  Consider the following example:

  ```
  ...
  <% out.getOutputStream().write(...binary data...) %>
  <% out.getOutputStream().write(...more binary data...) %>
  ```

  In this case, the browser will receive an unwanted newline characters in the middle of the binary data or at the end, depending on the buffering of your output buffer. You can avoid this problem by not using a carriage return between the lines of code, but of course this is an undesirable programming style.

Trying to generate binary data in JSP pages largely misses the point of JSP technology anyway, which is intended to simplify the programming of dynamic textual content.

Oracle XML Support

This section describes the following Oracle support features for XML that may be useful in JSP pages:

- XML-Alternative Syntax
- OracleXMLQuery Class

For information about additional support for XML and XSL, refer to the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

XML-Alternative Syntax

JSP tags, such as `<%...%>` for scriptlets, `<%!...%>` for declarations, and `<%=...%>` for expressions, are not syntactically valid within an XML document. Sun Microsystems addressed this in the *JavaServer Pages Specification*, *Version 1.1* by defining equivalent JSP tags using syntax that is XML-compatible. This is implemented through a standard DTD that you can specify within a `jsp:root` start tag at the beginning of an XML document.

This functionality allows you, for example, to write XML-based JSP pages in an XML authoring tool.

The Oracle JSP container does not use this DTD directly or require you to use a `jsp:root` tag, but the Oracle JSP translator includes logic to recognize the alternative syntax specified in the standard DTD. Table 4–1 documents this syntax.

Table 4–1 XML-Alternative Syntax

Standard JSP Syntax	XML-Alternative JSP Syntax

`<%@` *`directive`* `...%>`
Such as:
`<%@ page ... %>`
`<%@ include ... %>`
	`<jsp:directive.`*`directive`* `... />`
	Such as:
	`<jsp:directive.page ... />`
	`<jsp:directive.include ... />`

`<%! ... %>` (declaration)	`<jsp:declaration>`
	`...declarations go here...`
	`</jsp:declaration>`

Table 4–1 XML-Alternative Syntax (Cont.)

Standard JSP Syntax	XML-Alternative JSP Syntax

`<%= ... %>` (expression)	`<jsp:expression>`
	`...expression goes here...`
	`</jsp:expression>`

`<% ... %>` (scriptlet)	`<jsp:scriptlet>`
	`...code fragment goes here...`
	`</jsp:scriptlet>`

JSP action tags, such as `jsp:useBean`, for the most part already use syntax that complies with XML. Changes due to quoting conventions or for request-time attribute expressions may be necessary, however.

OracleXMLQuery Class

The `oracle.xml.sql.query.OracleXMLQuery` class is provided with Oracle9*i* as part of the XML-SQL utility for XML functionality in database queries. This class requires file `xsu12.jar` (for JDK 1.2.x) or `xsu111.jar` (for JDK 1.1.x), both of which are provided with Oracle9*i*.

For information about the `OracleXMLQuery` class and other XML-SQL utility features, refer to the *Oracle9i XML Developer's Kits Guide* - *XDK*.

Key JSP Configuration Issues

This section covers important effects of how you set key `page` directive parameters and JSP configuration parameters. The discussion focuses on JSP page optimization, classpath issues, and class loader issues. The following topics are covered:

- Optimization of JSP Execution
- Classpath and Class Loader Issues

Optimization of JSP Execution

There are settings you can consider to optimize JSP performance, including the following:

- Unbuffering a JSP Page
- Not Checking for Retranslation
- Not Using an HTTP Session

Unbuffering a JSP Page

By default, a JSP page uses an area of memory known as a *page buffer*. This buffer (8KB by default) is required if the page uses dynamic globalization support content type settings, forwards, or error pages. If it does not use any of these features, you can disable the buffer in a `page` directive:

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

This will improve the performance of the page by reducing memory usage and saving an output step. Output goes straight to the browser instead of going through the buffer first.

Not Checking for Retranslation

When the Oracle JSP container executes a JSP page, by default it will check whether a page implementation class already exists, compare the `.class` file timestamp against the `.jsp` source file timestamp, and retranslate the page if the `.class` file is older.

If comparing timestamps is unnecessary (as is the case in a typical deployment environment, where source code will not change), you can avoid the timestamp comparison by setting the JSP `developer_mode` flag to `false`. The default setting is `true`. For information about how to set this flag in the JServ environment, see "Setting JSP Parameters in JServ" on page 9-18.

Note: This discussion is not relevant for pre-translation scenarios.

Not Using an HTTP Session

If a JSP page does not require an HTTP session (essentially, does not require storage or retrieval of session attributes), then you can avoid using a session through the following `page` directive:

```
<%@ page session="false" %>
```

This will improve the performance of the page by eliminating the overhead of session creation or retrieval.

Note that although servlets by default do *not* use a session, JSP pages by default *do* use a session.

Classpath and Class Loader Issues

The Oracle JSP container uses its own classpath, distinct from the Web server classpath, and by default uses its own class loader to load classes from this classpath. This has significant advantages and disadvantages.

The JSP classpath combines the following elements:

- the Oracle JSP default classpath
- additional classpaths you specify in the JSP `classpath` configuration parameter

If there are classes you want loaded by the Oracle JSP class loader instead of the system class loader, then use the `classpath` configuration parameter, or place the classes in the Oracle JSP default classpath. See "Advantages and Disadvantages of the Oracle JSP Class Loader" on page 4-22 for related discussion.

Oracle JSP Default Classpath

Oracle JSP defines standard locations on the Web server for locating `.class` files and `.jar` files for classes (such as JavaBeans) that it requires. The Oracle JSP container will find files in these locations without any Web server classpath configuration.

These locations are as follows and are relative to the application root:

```
/WEB-INF/classes
/WEB-INF/lib
/_pages
```

(The WEB-INF directories are not relevant in a servlet 2.0 environment, such as JServ.)

Important: If you want classes in the WEB-INF directories to be loaded by the system class loader instead of the Oracle JSP class loader, place the classes somewhere in the Web server classpath as well. The system class loader takes priority—any class that is placed in both classpaths will always be loaded by the system class loader.

The _pages directory is the default location for translated and compiled JSP pages (as output by the JSP translator).

The classes directory is for individual Java .class files. These classes should be stored in subdirectories under the classes directory, according to Java package naming conventions.

For example, consider a JavaBean called LottoBean whose code defines it to be in the oracle.jsp.sample.lottery package. The Oracle JSP container will look for LottoBean.class in the following location relative to the application root:

```
/WEB-INF/classes/oracle/jsp/sample/lottery/LottoBean.class
```

The lib directory is for .jar files. Because Java package structure is specified in the .jar file structure, the .jar files are all directly in the lib directory (not in subdirectories).

As an example, LottoBean.class might be stored in lottery.jar, located as follows relative to the application root:

```
/WEB-INF/lib/lottery.jar
```

The application root directory can be located in any of the following locations (as applicable, depending on your Web server and servlet environment), listed in the order they are searched:

- the Web server directory the application is mapped to

- the Web server document root directory
- the directory containing the `globals.jsa` file (where applicable, typically in a servlet 2.0 environment)

Notes:

- Some Web servers, particularly those supporting the servlet 2.0 specification, do not offer full application support such as complete servlet context functionality. In this case, or when application mapping is not used, the default application is the server itself, and the application root is the Web server document root.
- For older servlet environments, the `globals.jsa` file is an Oracle extension that can be used as an application marker to establish an application root. See "Oracle JSP Application and Session Support for JServ" on page 9-26.

Oracle JSP classpath Configuration Parameter

Use the JSP `classpath` configuration parameter to add to the Oracle JSP classpath.

For more information about this parameter, see "Oracle JSP Configuration Parameters" on page 9-7. For information about how to set this parameter in the JServ environment, see "Setting JSP Parameters in JServ" on page 9-18.

Advantages and Disadvantages of the Oracle JSP Class Loader

Using the Oracle JSP class loader results in the following advantages and disadvantages:

- limited access to classes loaded by the Oracle JSP class loader from classes loaded by any other class loader

  When a class is loaded by the Oracle JSP class loader , its definition exists in that class loader only. Classes loaded by the system class loader or any other class loader, including any servlets, would have only limited access. The classes loaded by another class loader could not cast the class loaded by the Oracle JSP class loader or call methods on it. This may be desirable or undesirable, depending on your situation.

- automatic class reloading

  By default, the Oracle JSP class loader will automatically reload a class in the Oracle JSP classpath whenever the class file or JAR file has been modified since it was last loaded. For a JSP page, for example, this can happen as a result of dynamic retranslation, which occurs by default if the `.jsp` source file for a page has a more recent timestamp than its corresponding page implementation `.class` file.

  This is usually only advantageous in a development environment. In a typical deployment environment, the source, class, and JAR files will not change, and it is inefficient to check them for changes.

  See "Dynamic Class Reloading" on page 4-25 for more information.

It follows that in a deployment environment, you will typically *not* want to use the Oracle JSP classpath. By default, the `classpath` parameter is empty.

Oracle JSP Runtime Page and Class Reloading

This section describes conditions under which the Oracle JSP container retranslates pages, reloads pages, and reloads classes during runtime.

Dynamic Page Retranslation

As a Web application is running, the Oracle JSP container by default will automatically retranslate and reload a JSP page whenever the page source is modified.

The JSP container checks whether the last-modified time of the page implementation class file, as indicated in the Oracle JSP in-memory cache, is older than the last-modified time of the JSP page source file.

You can avoid the overhead of the Oracle JSP container checking timestamps for retranslation by setting the JSP `developer_mode` flag to `false`. This is advantageous in a deployment environment, where source and class files will typically not change. For more information about this flag, see "Oracle JSP Configuration Parameters" on page 9-7. For how to set it in a JServ environment, see "Setting JSP Parameters in JServ" on page 9-18.

Notes:

- Because of the usage of in-memory values for the class file last-modified time, note that removing a page implementation class file from the file system will *not* cause the Oracle JSP container to retranslate the associated JSP page source. The JSP container will only retranslate when the JSP page source file timestamp changes.
- The class file will be regenerated when the cache is lost. This happens whenever a request is directed to this page after the server is restarted or after another page in this application has been retranslated.

Dynamic Page Reloading

The Oracle JSP container will automatically reload a JSP page (in other words, reload the generated page implementation class) in the following circumstances:

- the page is retranslated

  (See "Dynamic Page Retranslation" above.)

- a Java class that is called by the page and was loaded by the Oracle JSP class loader (and not the system class loader) is modified

  (See "Dynamic Class Reloading" below.)

- any page in the same application is reloaded

  A JSP pages is associated with the overall Web application within which it runs. (Even JSP pages not associated with a particular application are considered to be part of a "default application".)

  Whenever a JSP page is reloaded, all JSP pages in the application are reloaded.

Notes:

- The Oracle JSP container does *not* reload a page just because a statically included file has changed. (Statically included files, included through `<%@ include ... %>` syntax, are included during translation-time.)
- Page reloading and page retranslation are not the same thing. Reloading does not imply retranslation.

Dynamic Class Reloading

By default, before the Oracle JSP container dispatches a request that will execute a Java class that was loaded by the Oracle JSP class loader, it checks to see if the class file has been modified since it was first loaded. If the class has been modified, then the Oracle JSP class loader reloads it.

This applies only to classes in the Oracle JSP classpath, which includes the following:

- JAR files in the `/WEB-INF/lib` directory (servlet 2.2)
- `.class` files in the `/WEB-INF/classes` directory (servlet 2.2)

- classes in paths specified through the Oracle JSP `classpath` configuration parameter
- generated `.class` files in the `_pages` output directory

As mentioned in the preceding section, "Dynamic Page Reloading", reloading a class results in the dynamic reloading of JSP pages that reference that class.

Important:

- Remember that classes must be in the JSP classpath, not the system classpath, to be dynamically reloaded. If they are in the system classpath as well, the system class loader may take precedence in some circumstances, possibly interfering with JSP automatic-reloading functionality.
- Dynamic class reloading can be expensive in terms of CPU usage. You can disable this feature by setting the JSP `developer_mode` parameter to `false`. This is appropriate in deployment environments where classes are not expected to change.

For information about the `classpath` and `developer_mode` configuration parameters and how to set them in a JServ environment, see "Oracle JSP Configuration Parameters" on page 9-7 and "Setting JSP Parameters in JServ" on page 9-18.

5

Oracle-Specific Programming Extensions

This chapter discusses extended JSP functionality offered by Oracle that is not portable to other JSP environments. This consists of event-handling through the Oracle `JspScopeListener` mechanism and support for SQLJ, a standard syntax for embedding SQL statements directly into Java code. The chapter is organized as follows:

- Oracle JSP Event Handling with JspScopeListener
- Oracle JSP Support for Oracle SQLJ

Notes:

- For servlet 2.0 environments, the Oracle JSP container supports non-portable extensions through a mechanism called `globals.jsa` to support a Web application framework. "Oracle JSP Application and Session Support for JServ" on page 9-26 describes this mechanism.
- The Oracle JSP container also has extended (and non-portable) globalization support, which is described in "Oracle JSP Extended Support for Multibyte Parameter Encoding" on page 8-5.

Oracle JSP Event Handling with JspScopeListener

In standard servlet and JSP technology, only session-based events are supported. The Oracle JSP container extends this support through the `JspScopeListener` interface and `JspScopeEvent` class in the `oracle.jsp.event` package. This mechanism supports the four standard JSP scopes for event-handling for any Java objects used in a JSP application:

- `page`
- `request`
- `session`
- `application`

For Java objects that are used in your application, implement the `JspScopeListener` interface in the appropriate class, then attach objects of that class to a JSP scope using tags such as `jsp:useBean`.

When the end of a scope is reached, objects that implement `JspScopeListener` and have been attached to the scope will be so notified. The Oracle JSP container accomplishes this by sending a `JspScopeEvent` instance to such objects through the `outOfScope()` method specified in the `JspScopeListener` interface.

Properties of the `JspScopeEvent` object include the following:

- the scope that is ending (one of the constants `PAGE_SCOPE`, `REQUEST_SCOPE`, `SESSION_SCOPE`, or `APPLICATION_SCOPE`)
- the container object that is the repository for objects at this scope (one of the implicit objects `page`, `request`, `session`, or `application`)
- the name of the object that the notification pertains to (the name of the instance of the class that implements `JspScopeListener`)
- the JSP implicit `application` object

The Oracle JSP event listener mechanism significantly benefits developers who want to always free object resources that are of `page` or `request` scope, regardless of error conditions. It frees these developers from having to surround their page implementations with Java `try`/`catch`/`finally` blocks.

Oracle JSP Support for Oracle SQLJ

SQLJ is a standard syntax for embedding static SQL instructions directly in Java code, greatly simplifying database-access programming. The Oracle JSP container and its JSP translator support Oracle SQLJ, allowing you to use SQLJ syntax in JSP statements. SQLJ statements are indicated by the `#sql` token.

For general information about Oracle SQLJ programming features, syntax, and command-line options, see the *Oracle9i SQLJ Developer's Guide and Reference*.

SQLJ JSP Code Example

Following is a sample SQLJ JSP page. (The `page` directive imports classes that are typically required by SQLJ.)

```
<%@ page language="sqlj"
    import="sqlj.runtime.ref.DefaultContext,oracle.sqlj.runtime.Oracle" %>
<HTML>
<HEAD> <TITLE> The SQLJQuery JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">
<% String empno = request.getParameter("empno");
if (empno != null) { %>
<H3> Employee # <%=empno %> Details: </H3>
<%= runQuery(empno) %>
<HR><BR>
<% } %>
<B>Enter an employee number:</B>
<FORM METHOD="get">
<INPUT TYPE="text" NAME="empno" SIZE=10>
<INPUT TYPE="submit" VALUE="Ask Oracle");
</FORM>
</BODY>
</HTML>
<%!

private String runQuery(String empno) throws java.sql.SQLException {
   DefaultContext dctx = null;
   String ename = null; double sal = 0.0; String hireDate = null;
   StringBuffer sb = new StringBuffer();
   try {
       dctx = Oracle.getConnection("jdbc:oracle:oci8:@", "scott", "tiger");
       #sql [dctx] {
       select ename, sal, TO_CHAR(hiredate,'DD-MON-YYYY')
       INTO :ename, :sal, :hireDate
       FROM scott.emp WHERE UPPER(empno) = UPPER(:empno)
```

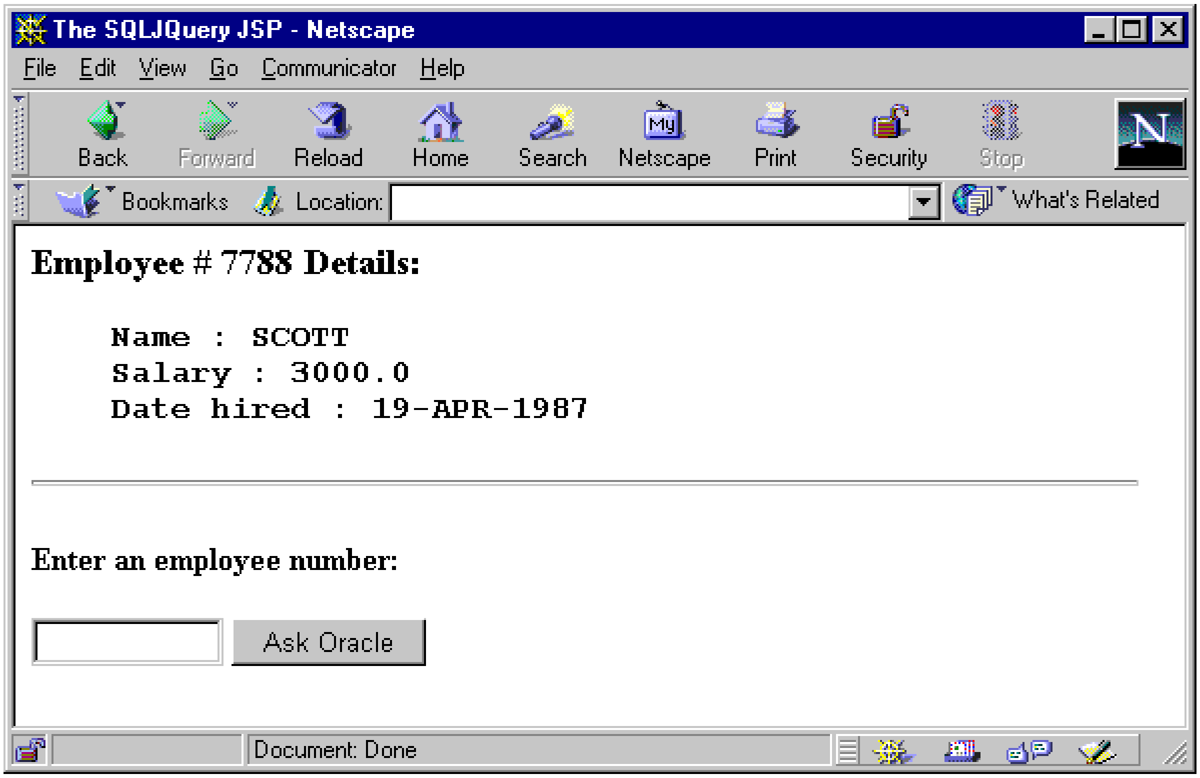
```
      };
      sb.append("<BLOCKQUOTE><BIG><B><PRE>\n");
      sb.append("Name : " + ename + "\n");
      sb.append("Salary : " + sal + "\n");
      sb.append("Date hired : " + hireDate);
      sb.append("</PRE></B></BIG></BLOCKQUOTE>");
      } catch (java.sql.SQLException e) {
        sb.append("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
      } finally {
      if (dctx!= null) dctx.close();
      }
  return sb.toString();
}

%>
```

This example uses the JDBC OCI driver, which requires an Oracle client installation. The `Oracle` class used in getting the connection is provided with Oracle SQLJ.

Notes:

- In case a JSP page is invoked multiple times in the same JVM, it is recommended that you always use an explicit connection context, such as `dctx` in the example, instead of the default connection context. (Note that `dctx` is a local method variable.)
- The Oracle JSP container requires Oracle SQLJ release 8.1.6.1 or higher.
- In future releases (and in Oracle9*i* Application Server release 2), the Oracle JSP container will support `language="sqlj"` in a `page` directive to trigger the Oracle SQLJ translator during JSP translation. For forward compatibility, it is recommended as a good programming practice that you begin using this directive.

Entering employee number 7788 for the schema used in the example results in the following output:


The SQLJQuery JSP - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location:
What's Related
Ask Oracle
Document: Done

Employee # 7788 Details:

```
Name  :  SCOTT
Salary : 3000.0
Date hired : 19-APR-1987
```

Enter an employee number:

Triggering the SQLJ Translator

You can trigger the Oracle JSP translator to invoke the Oracle SQLJ translator by using the file name extension `.sqljsp` for the JSP source file.

This results in the JSP translator generating a `.sqlj` file instead of a `.java` file. The Oracle SQLJ translator is then invoked to translate the `.sqlj` file into a `.java` file.

Using SQLJ results in additional output files; see "Generated Files and Locations (On-Demand Translation)" on page 6-6.

Important:

- To use Oracle SQLJ, you will have to install appropriate SQLJ JAR or ZIP files, depending on your environment, and add them to your classpath. See "Required and Optional Files for Oracle JSP" on page 9-2.
- Do not use the same base file name for a `.jsp` file and a `.sqljsp` file in the same application, because they would result in the same generated class name and `.java` file name.

Setting Oracle SQLJ Options

When you execute or pre-translate a SQLJ JSP page, you can specify desired Oracle SQLJ option settings. This is true both in on-demand translation scenarios and pre-translation scenarios, as follows:

- In an on-demand translation scenario, use the JSP `sqljcmd` configuration parameter. This parameter, in addition to allowing you to specify a particular SQLJ translator executable, allows you to set SQLJ command-line options.

  For information, see the `sqljcmd` description in "Oracle JSP Configuration Parameters" on page 9-7. For how to set configuration parameters in a JServ environment, see "Setting JSP Parameters in JServ" on page 9-18.

- In a pre-translation scenario with the `ojspc` pre-translation tool, use the `ojspc -S` option. This option allows you to set SQLJ command-line options.

  For information, see "Command-Line Syntax for ojspc" on page 6-17 and "Option Descriptions for ojspc" on page 6-18.

6 JSP Translation and Deployment

This chapter discusses operation of the Oracle JSP translator, then discusses the `ojspc` utility and situations where pre-translation is useful, followed by general discussion of a number of additional JSP deployment considerations.

The chapter is organized as follows:

Functionality of the Oracle JSP Translator

JSP translators generate standard Java code for a JSP page implementation class. This class is essentially a servlet class wrapped with features for JSP functionality.

This section discusses general functionality of the Oracle JSP translator, focusing on its behavior in on-demand translation scenarios. The following topics are covered:

- Generated Code Features
- Generated Package and Class Names (On-Demand Translation)
- Generated Files and Locations (On-Demand Translation)
- Sample Page Implementation Class Source

Important: Implementation details in this section regarding package and class naming, file and directory naming, output file locations, and generated code are for illustrative purposes. The precise details apply to Oracle JSP 1.1.x.x releases only and are subject to change from release to release.

Generated Code Features

This section discusses general features of the page implementation class code that is produced by the Oracle JSP translator in translating JSP source (`.jsp` and `.sqljsp` files).

Features of Page Implementation Class Code

When the Oracle JSP translator generates servlet code in the page implementation class, it automatically handles some of the standard programming overhead. For both the on-demand translation model and the pre-translation model, generated code automatically includes the following features:

- It extends a wrapper class (`oracle.jsp.runtime.HttpJsp`) provided by the Oracle JSP container that implements the standard `javax.servlet.jsp.HttpJspPage` interface (which extends the more generic `javax.servlet.jsp.JspPage` interface, which in turn extends the standard `javax.servlet.Servlet` interface).
- It implements the `_jspService()` method specified by the `HttpJspPage` interface. This method, often referred to generically as the "service" method, is the central method of the page implementation class. Code from any Java

scriptlets and expressions in the JSP page is incorporated into this method implementation.

- It includes code to request an HTTP session, unless your JSP source code specifically sets `session=false` (which can be done in a `page` directive).

Inner Class for Static Text

The service method, `_jspService()`, of the page implementation class includes print commands—`out.print()` calls on the implicit `out` object—to print any static text in the JSP page. The Oracle JSP translator, however, places the static text itself in an inner class within the page implementation class. The service method `out.print()` statements reference attributes of the inner class to print the text.

This inner class implementation results in an additional `.class` file when the page is translated and compiled. In a client-side pre-translation scenario, be aware this means there is an extra `.class` file to deploy.

The name of the inner class will always be based on the base name of the `.jsp` file or `.sqljsp` file. For `mypage.jsp`, for example, the inner class (and its `.class` file) will always include "mypage" in its name.

Note: The Oracle JSP translator can optionally place the static text in a Java resource file, which is advantageous for pages with large amounts of static text. (See "Workarounds for Large Static Content in JSP Pages" on page 4-10.) You can request this feature through the JSP `external_resource` configuration parameter for on-demand translation, or the `ojspc -extres` option for pre-translation.

Even when static text is placed in a resource file, the inner class is still produced, and its `.class` file must be deployed. (This is only noteworthy if you are in a client-side pre-translation scenario.)

General Conventions for Output Names

The Oracle JSP translator follows a consistent set of conventions in naming output classes, packages, files and directories. *However, this set of conventions and other implementation details may change from release to release.*

One fact that is *not* subject to change is that the base name of a JSP page will be included intact in output class and file names as long as it does not include special characters. For example, translating `MyPage123.jsp` will always result in the string "MyPage123" being part of the page implementation class name, Java source file name, and class file name.

In Oracle JSP 1.1.x.x releases (as well as some previous releases), the base name is preceded by an underscore ("_"). Translating `MyPage123.jsp` results in page implementation class `_MyPage123` in source file `_MyPage123.java`, which is compiled into `_MyPage123.class`.

Similarly, where path names are used in creating Java package names, each component of the path is preceded by an underscore. Translating `/jspdir/myapp/MyPage123.jsp`, for example, results in class `_MyPage123` being in the following package:

```
_jspdir._myapp
```

The package name is used in creating directories for output `.java` and `.class` files, so the underscores are also evident in output directory names. For example, in translating a JSP page in the directory `htdocs/test`, the Oracle JSP translator by default will create directory `htdocs/_pages/_test` for the page implementation class source.

Note: All output directories are created under the standard `_pages` directory by default, as described in "Generated Files and Locations (On-Demand Translation)" on page 6-6. You can change this behavior, however, through the `page_repository_root` configuration parameter, described in "Oracle JSP Configuration Parameters" on page 9-7, or the `ojspc -d` and `-srcdir` options, described in "Option Descriptions for ojspc" on page 6-18.

If special characters are included in a JSP page name or path name, the Oracle JSP translator takes steps to ensure that no characters that would be illegal in Java appear in the output class, package, and file names. For example, translating `My-name_foo12.jsp` results in `_My_2d_name__foo12` being the class name, in source file `_My_2d_name__foo12.java`. The hyphen is converted to a string of alpha-numeric characters. (An extra underscore is also inserted before "foo12".) In this case, you can only be assured that alphanumeric components of the JSP page name will be included intact in the output class and file names. For example, you could search for "My", "name", or "foo12".

These conventions are demonstrated in examples provided later in this section and later in this chapter.

Generated Package and Class Names (On-Demand Translation)

Although the Sun Microsystems *JavaServer Pages Specification, Version 1.1* defines a uniform process for parsing and translating JSP text, it does not describe how the generated classes should be named—that is up to each JSP implementation.

This section describes how the Oracle JSP container creates package and class names when it generates code during translation.

Note: For information about general conventions used by the Oracle JSP container in naming output classes, packages, files, and schema paths, see "General Conventions for Output Names" on page 6-3

Package Naming

In an on-demand translation scenario, the URL path that is specified when the user requests a JSP page—specifically, the path relative to the doc root or application root—determines the package name for the generated page implementation class. Each directory in the URL path represents a level of the package hierarchy.

It is important to note, however, that generated package names are *always* lowercase, regardless of the case in the URL.

Consider the following URL as an example:

```
http://host[:port]/HR/expenses/login.jsp
```

In Oracle JSP 1.1.x.x releases, this results in the following package specification in the generated code (implementation details are subject to change in future releases):

```
package _hr._expenses;
```

No package name is generated if the JSP page is at the doc root or application root directory, where the URL is as follows:

```
http://host[:port]/login.jsp
```

Class Naming

The base name of the `.jsp` file (or `.sqljsp` file) determines the class name in the generated code.

Consider the following URL example:

```
http://host[:port]/HR/expenses/UserLogin.jsp
```

In Oracle JSP 1.1.x.x releases, this yields the following class name in the generated code (implementation details are subject to change in future releases):

```
public class _UserLogin extends ...
```

Be aware that the case (lowercase / uppercase) that end users type in the URL must match the case of the actual `.jsp` or `.sqljsp` file name. For example, they can specify `UserLogin.jsp` if that is the actual file name, or `userlogin.jsp` if that is the actual file name, but not `userlogin.jsp` if `UserLogin.jsp` is the actual file name.

In Oracle JSP 1.1.x.x releases, the translator determines the case of the class name according to the case of the file name. For example:

- `UserLogin.jsp` results in class `_UserLogin`.
- `Userlogin.jsp` results in class `_Userlogin`.
- `userlogin.jsp` results in class `_userlogin`.

If you care about the case of the class name, then you must name the `.jsp` file or `.sqljsp` file accordingly. However, because the page implementation class is invisible to the end user, this is usually not a concern.

Generated Files and Locations (On-Demand Translation)

This section describes files that are generated by the Oracle JSP translator and where they are placed. For pre-translation scenarios, `ojspc` places files differently and has its own set of relevant options—see "Summary of ojspc Output Files, Locations, and Related Options" on page 6-25.

The following subsections mention several JSP configuration parameters. For more information about them, and about how to set them in a JServ environment, see "Oracle JSP Configuration Parameters" on page 9-7 and "Setting JSP Parameters in JServ" on page 9-18.

Note: For information about general conventions used by the Oracle JSP container in naming output classes, packages, files, and schema paths, see "General Conventions for Output Names" on page 6-3

Files Generated by the Oracle JSP Container

This section considers both regular JSP pages (`.jsp` files) and SQLJ JSP pages (`.sqljsp` files) in listing files that are generated by the Oracle JSP translator. For the file name examples, presume a file `Foo.jsp` or `Foo.sqljsp` is being translated.

Source files:

- A `.sqlj` file is produced by the Oracle JSP translator if the page is a SQLJ JSP page (for example, `_Foo.sqlj`).
- A `.java` file is produced for the page implementation class and inner class (for example, `_Foo.java`). It is produced either directly by the Oracle JSP translator from the `.jsp` file, or by the SQLJ translator from the `.sqlj` file if the page is a SQLJ JSP page. (The currently installed Oracle SQLJ translator is used by default, but you can specify an alternative translator or an alternative release of the Oracle SQLJ translator by using the JSP `sqljcmd` configuration parameter.)

Binary files:

- In the case of a SQLJ JSP page, if you use ISO standard SQLJ code generation (SQLJ `-codegen=iso` setting), one or more binary files are produced during SQLJ translation for SQLJ profiles. By default these are `.ser` Java resource files, but they will be `.class` files if you enable the SQLJ `-ser2class` option (through the JSP `sqljcmd` configuration parameter). The resource file or `.class` file has "Foo" as part of its name.

  Note: No profiles are produced if you use Oracle-specific code generation in SQLJ (`-codegen=oracle`). As of Oracle9i release 2, this is the SQLJ default mode.

- A `.class` file is produced by the Java compiler for the page implementation class. (The Java compiler is `javac` by default, but you can specify an alternative compiler using the JSP `javaccmd` configuration parameter.)
- An additional `.class` file is produced for the inner class of the page implementation class. This file will have "Foo" as part of its name.
- A `.res` Java resource file is optionally produced for the static page content (for example, `_Foo.res`) if the JSP `external_resource` configuration parameter is enabled.

Note: The exact names of generated files for the page implementation class may change in future releases, but will still have the same general form. The names would always include the base name (such as "Foo" in these examples), but may include slight variations beyond that.

Oracle JSP Translator Output File Locations

The Oracle JSP container uses the Web server document repository to generate or load translated JSP pages.

By default, the root directory is the Web server doc root directory (for JServ) or the servlet context root directory of the application the page belongs to.

You can specify an alternative root directory through the JSP `page_repository_root` configuration parameter.

In Oracle JSP 1.1.x.x releases, generated files are placed as follows (implementation details may change in future releases):

- If the `.jsp` (or `.sqljsp`) file is directly in the root directory, then the Oracle JSP container will place generated files into a default `_pages` subdirectory directly under the root directory.
- If the `.jsp` (or `.sqljsp`) file is in a subdirectory under the root directory, then a parallel directory structure is created under the `_pages` subdirectory for the generated files. Subdirectory names under the `_pages` directory are based on subdirectory names under the root directory.

  As an example, consider a JServ environment with an `htdocs` doc root directory. If a `.jsp` file is in the following directory:

  ```
  htdocs/subdir/test
  ```

  then generated files will be placed in the following directory:

  ```
  htdocs/_pages/_subdir/_test
  ```

Sample Page Implementation Class Source

This section uses an example to illustrate the information in the preceding sections.

Consider the following scenario:

- JSP page code is in the file `hello.jsp`.
- The page is executed in a JServ environment.
- The `hello.jsp` file is located in the following directory:

  `htdocs/test`

Important: Code generation details discussed here are according to the Oracle implementation of the JSP 1.1 specification. Details may change in the future, as the result of either changes in the specification or changes in how Oracle implements aspects that are not specified.

Sample Page Source: hello.jsp

Following is the JSP code in `hello.jsp`:

```
<HTML>
<HEAD><TITLE>The Hello User JSP</TITLE></HEAD>
<BODY>
<% String user=request.getParameter("user"); %>
<H3>Welcome <%= (user==null) ? "" : user %>!</H3>
<P><B> Today is <%= new java.util.Date() %>. Have a nice day! :-)</B></P>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

Sample: Generated Package and Class

Because `hello.jsp` is in the `test` subdirectory of the root directory (`htdocs`), Oracle JSP 1.1.x.x releases generate the following package name in the page implementation code.

```
package _test;
```

The Java class name is determined by the base name of the `.jsp` file (including case), so the following class definition is generated in the page implementation code:

```
public class _hello extends oracle.jsp.runtime.HttpJsp
{
   ...
}
```

(Because the page implementation class is invisible to the end user, the fact that its name does not adhere to Java capitalization conventions is generally not a concern.)

Sample: Generated Files

Because `hello.jsp` is located as follows:

```
htdocs/test/hello.jsp
```

Oracle JSP 1.1.x.x releases generate output files as follows (the page implementation class `.java` file and `.class` file, and the inner class `.class` file, respectively):

```
htdocs/_pages/_test/_hello.java
htdocs/_pages/_test/_hello.class
htdocs/_pages/_test/_hello$__jsp_StaticText.class
```

Note: These file names are based specifically on Oracle JSP 1.1.x.x implementations; the exact details may change in future releases. All file names will always include the base "hello", however.

Sample Page Implementation Code: _hello.java

Following is the generated page implementation class Java code (`_hello.java`), as generated by Oracle JSP 1.1.x.x releases:

```
package _test;

import oracle.jsp.runtime.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import java.beans.*;


public class _hello extends oracle.jsp.runtime.HttpJsp {

  public final String _globalsClassName = null;

  // ** Begin Declarations


  // ** End Declarations

  public void _jspService(HttpServletRequest request, HttpServletResponse
response) throws IOException, ServletException {

    /* set up the intrinsic variables using the pageContext goober:
    ** session = HttpSession
    ** application = ServletContext
    ** out = JspWriter
    ** page = this
    ** config = ServletConfig
    ** all session/app beans declared in globals.jsa
    */
    JspFactory factory = JspFactory.getDefaultFactory();
    PageContext pageContext = factory.getPageContext( this, request, response,
null, true, JspWriter.DEFAULT_BUFFER, true);
    // Note: this is not emitted if the session directive == false
    HttpSession session = pageContext.getSession();
    if (pageContext.getAttribute(OracleJspRuntime.JSP_REQUEST_REDIRECTED,
PageContext.REQUEST_SCOPE) != null) {
      pageContext.setAttribute(OracleJspRuntime.JSP_PAGE_DONTNOTIFY, "true",
PageContext.PAGE_SCOPE);
      factory.releasePageContext(pageContext);
      return;
}
    ServletContext application = pageContext.getServletContext();
    JspWriter out = pageContext.getOut();
    hello page = this;
    ServletConfig config = pageContext.getServletConfig();

    try {
      // global beans
      // end global beans
      out.print(__jsp_StaticText.text[0]);
      String user=request.getParameter("user");
      out.print(__jsp_StaticText.text[1]);
      out.print(  (user==null) ? "" : user );
      out.print(__jsp_StaticText.text[2]);
      out.print(  new java.util.Date() );
      out.print(__jsp_StaticText.text[3]);

      out.flush();

    }
    catch( Exception e) {
      try {
        if (out != null) out.clear();
      }
      catch( Exception clearException) {
      }
      pageContext.handlePageException( e);
    }
    finally {
      if (out != null) out.close();
      factory.releasePageContext(pageContext);
    }

  }
  private static class __jsp_StaticText {
    private static final char text[][]=new char[4][];
    static {
      text[0] =
      "<HTML>\r\n<HEAD><TITLE>The Welcome User
JSP</TITLE></HEAD>\r\n<BODY>\r\n".toCharArray();
      text[1] =
      "\r\n<H3>Welcome ".toCharArray();
      text[2] =
      "!</H3>\r\n<P><B> Today is ".toCharArray();
      text[3] =
      ". Have a nice day! :-)</B></P>\r\n<B>Enter name:</B>\r\n<FORM
METHOD=get>\r\n<INPUT TYPE=\"text\" NAME=\"user\" SIZE=15>\r\n<INPUT
TYPE=\"submit\" VALUE=\"Submit
name\">\r\n</FORM>\r\n</BODY>\r\n</HTML>".toCharArray();
    }
  }
}
```

JSP Pre-Translation and the ojspc Utility

This section describes the `ojspc` utility, provided with Oracle9*i* for pre-translation, and is organized as follows:

- General Use of ojspc for Pre-Translation
- Details of the ojspc Pre-Translation Tool

General Use of ojspc for Pre-Translation

You can use `ojspc` to pre-translate JSP pages in any environment, which may be useful in saving end users the translation overhead the first time a page is executed.

If you are pre-translating in some environment other than the target environment, specify the `ojspc -d` option to set an appropriate base directory for placement of generated binary files.

As an example, consider a JServ environment with the following JSP source file:

```
htdocs/test/foo.jsp
```

A user would invoke this with the following URL:

```
http://host[:port]/test/foo.jsp
```

During on-demand translation at execution time, the Oracle JSP translator would use a default base directory of `htdocs/_pages` for placement of generated binary files. Therefore, if you pre-translate, you should set `htdocs/_pages` as the base directory for binary output, such as in the following example (assume `%` is a UNIX prompt):

```
% cd htdocs
% ojspc -d _pages test/foo.jsp
```

The URL noted above specifies an application-relative path of `test/foo.jsp`, so at execution time the Oracle JSP container looks for the binary files in a `_test` subdirectory under the default `htdocs/_pages` directory. This subdirectory would be created automatically by `ojspc` if it is run as in the above example. At execution time, the Oracle JSP container would find the pre-translated binaries and would not have to perform translation, assuming that the source file was not altered after pre-translation. (By default, the page would be re-translated if the source file timestamp is later than the binary timestamp, assuming the source file is available and the `bypass_source` configuration parameter is not enabled.)

Note: Oracle JSP implementation details, such as use of an underscore ("_") in output directory names (as for `_test` above), are subject to change from release to release. This documentation applies specifically to Oracle JSP 1.1.x.x releases.

Details of the ojspc Pre-Translation Tool

The following topics are covered here:

- Overview of ojspc Functionality
- Option Summary Table for ojspc
- Command-Line Syntax for ojspc
- Option Descriptions for ojspc
- Summary of ojspc Output Files, Locations, and Related Options

Notes: There are other possible scenarios, such as in a middle-tier environment, for using `ojspc` to pre-translate JSP pages. See "General Use of ojspc for Pre-Translation" on page 6-13.

Overview of ojspc Functionality

For a simple JSP (not SQLJ JSP) page, default functionality for `ojspc` is as follows:

- It takes a `.jsp` file as an argument.
- It invokes the Oracle JSP translator to translate the `.jsp` file into Java page implementation class code, producing a `.java` file. The page implementation class includes an inner class for static page content.
- It invokes the Java compiler to compile the `.java` file, producing two `.class` files (one for the page implementation class itself and one for the inner class).

And following is the default `ojspc` functionality for a SQLJ JSP page:

- It takes a `.sqljsp` file as an argument instead of a `.jsp` file.
- It invokes the Oracle JSP translator to translate the `.sqljsp` file into a `.sqlj` file for the page implementation class (and inner class).
- It invokes the Oracle SQLJ translator to translate the `.sqlj` file. This produces a `.java` file for the page implementation class (and inner class). Also, if you use

ISO standard SQLJ code generation (SQLJ `-codegen=iso` setting), a SQLJ "profile" file is produced that is, by default, a `.ser` Java resource file.

Note: No profiles are produced if you use Oracle-specific SQLJ code generation (SQLJ `-codegen=oracle` setting). As of Oracle9*i* release 2, this is the default mode.

For information about SQLJ profiles and Oracle-specific code generation, see the *Oracle9i SQLJ Developer's Guide and Reference*.

- It invokes the Java compiler to compile the `.java` file, producing two `.class` files (one for the page implementation class itself and one for the inner class).

Under some circumstances (see the `-extres` option description below), `ojspc` settings direct the Oracle JSP translator to produce a `.res` Java resource file for static page content instead of putting this content into the inner class of the page implementation class. However, the inner class is still created and must still be deployed with the page implementation class.

Because `ojspc` invokes the Oracle JSP translator, `ojspc` output conventions are the same as for the Oracle JSP container in general, as applicable. For general information about Oracle JSP translator output, including generated code features, general conventions for output names, generated package and class names, and generated files and locations, see "Functionality of the Oracle JSP Translator" on page 6-2.

Note: The `ojspc` command-line tool is a front-end utility that invokes the `oracle.jsp.tool.Jspc` class.

Option Summary Table for ojspc

Table 6–1 describes the options supported by the `ojspc` pre-translation utility. These options are further discussed in "Option Descriptions for ojspc" on page 6-18.

The second column notes comparable or related JSP configuration parameters for on-demand translation environments (such as JServ).

Table 6–1 Options for ojspc Pre-Translation Utility

Option	Related JSP Configuration Parameters	Description	Default
-addclasspath	classpath (related, but with different functionality)	additional classpath entries for `javac`	empty (no additional path entries)
-appRoot	n/a	application root directory for application-relative static `include` directives from the page	current directory
-debug	emit_debuginfo	boolean to direct `ojspc` to generate a line map to the original `.jsp` file for debugging	false
-d	page_repository_root	location where `ojspc` should place generated binary files (`.class` and resource)	current directory
-extend	n/a	class for the generated page implementation class to extend	empty
-extres	external_resource	boolean to direct `ojspc` to generate an external resource file for static text from the `.jsp` file	false
-implement	n/a	interface for the generated page implementation class to implement	empty
-noCompile	javaccmd	boolean to direct `ojspc` *not* to compile the generated page implementation class	false
-packageName	n/a	package name for the generated page implementation class	empty (generate package names per `.jsp` file location)
-S-<sqlj option>	sqljcmd	`-S` prefix followed by an Oracle SQLJ option (for `.sqljsp` files)	empty
-srcdir	page_repository_root	location where `ojspc` should place generated source files (`.java` and `.sqlj`)	current directory

Table 6–1 Options for ojspc Pre-Translation Utility (Cont.)

Option	Related JSP Configuration Parameters	Description	Default
-verbose	n/a	boolean to direct `ojspc` to print status information as it executes	false
-version	n/a	boolean to direct `ojspc` to display the Oracle JSP version number	false

Command-Line Syntax for ojspc

Following is the general `ojspc` command-line syntax (assume `%` is a UNIX prompt):

```
% ojspc [option_settings] file_list
```

The file list can include `.jsp` files or `.sqljsp` files.

Be aware of the following syntax notes:

- If multiple `.jsp` files are translated, they all must use the same character set (either by default or through `page` directive `contentType` settings).
- Use spaces between file names in the file list.
- Use spaces as separators between option names and option values in the option list.
- Option names are not case sensitive, but option values usually are (such as package names, directory paths, class names, and interface names).
- Enable boolean options, which are disabled by default, by simply entering the option name on the command line. For example, type `-extres`, *not* `-extres true`.)

Following is an example:

```
% ojspc -d /myapp/mybindir -srcdir /myapp/mysrcdir -extres MyPage.sqljsp MyPage2.jsp
```

Option Descriptions for ojspc

This section describes the `ojspc` options in more detail.

-addclasspath (fully qualified path; `ojspc` default: empty)

Use this option to specify additional classpath entries for `javac` to use when compiling generated page implementation class source. Otherwise, `javac` uses only the system classpath.

Notes:

- In an on-demand translation scenario, the JSP `classpath` configuration parameter provides related, although different, functionality. See "Oracle JSP Configuration Parameters" on page 9-7.
- The `-addclasspath` setting is also used by the SQLJ translator for SQLJ JSP pages.

-appRoot (fully qualified path; `ojspc` default: current directory)

Use this option to specify an application root directory. The default is the current directory, from which `ojspc` was run.

The specified application root directory path is used as follows:

- It is used for static `include` directives in the page being translated. The specified directory path is prepended to any application-relative (context-relative) paths in the `include` directives of the translated page.
- It is used in determining the package of the page implementation class. The package will be based on the location of the file being translated relative to the application root directory. The package, in turn, determines the placement of output files. (See "Summary of ojspc Output Files, Locations, and Related Options" on page 6-25.)

This option is necessary, for example, so included files can still be found if you run `ojspc` from some other directory.

Consider the following example.

- You want to translate the following file:

  ```
  /abc/def/ghi/test.jsp
  ```

- You run `ojspc` from the current directory, `/abc`, as follows (assume `%` is a UNIX prompt):

  ```
  % cd /abc
  % ojspc def/ghi/test.jsp
  ```

- The `test.jsp` page has the following `include` directive:

  ```
  <%@ include file="/test2.jsp" %>
  ```

- The `test2.jsp` page is in the `/abc` directory, as follows:

  ```
  /abc/test2.jsp
  ```

This requires no `-appRoot` setting, because the default application root setting is the current directory, which is the `/abc` directory. The `include` directive uses the application-relative `/test2.jsp` syntax (note the beginning "/"), so the included page will be found as `/abc/test2.jsp`.

The package in this case is `_def._ghi`, based on the location of `test.jsp` relative to the current directory, from which `ojspc` was run (the current directory is the default application root). Output files are placed accordingly.

If, however, you run `ojspc` from some other directory, suppose `/home/mydir`, then you would need an `-appRoot` setting as in the following example:

```
% cd /home/mydir
% ojspc -appRoot /abc /abc/def/ghi/test.jsp
```

The package is still `_def._ghi`, based on the location of `test.jsp` relative to the specified application root directory.

> **Note:** It is typical for the specified application root directory to be some level of parent directory of the directory where the translated JSP page is located.

-d (fully qualified path; `ojspc` default: current directory)

Use this option to specify a base directory for `ojspc` placement of generated binary files—`.class` files and Java resource files. (The `.res` files produced for static content by the `-extres` option are Java resource files, as are `.ser` profile files produced by the SQLJ translator for SQLJ JSP pages.)

The specified path is taken simply as a file system path (not an application-relative or page-relative path).

Subdirectories under the specified directory are created automatically, as appropriate, depending on the package. See "Summary of ojspc Output Files, Locations, and Related Options" on page 6-25 for more information.

The default is to use the current directory (your current directory when you executed `ojspc`).

It is recommended that you use this option to place generated binary files into a clean directory so that you easily know what files have been produced.

Notes:

- In environments such as Windows NT that allow spaces in directory names, enclose the directory name in quotes.
- In an on-demand translation scenario, the JSP `page_repository_root` configuration parameter provides related functionality. See "Oracle JSP Configuration Parameters" on page 9-7.

-debug (boolean; `ojspc` default: `false`)

Enable this flag to instruct `ojspc` to generate a line map to the original `.jsp` file for debugging. Otherwise, line-mapping will be to the generated page implementation class.

This is useful for source-level JSP debugging, such as when using Oracle9*i* JDeveloper.

Note: In an on-demand translation scenario, the JSP `emit_debuginfo` configuration parameter provides the same functionality. See "Oracle JSP Configuration Parameters" on page 9-7.

-extend (fully qualified Java class name; `ojspc` default: empty)

Use this option to specify a Java class that the generated page implementation class will extend.

-extres (boolean; `ojspc` default: `false`)

Enable this flag to instruct `ojspc` to place generated static content (the Java print commands that output static HTML code) into a Java resource file instead of into an inner class of the generated page implementation class.

The resource file name is based on the JSP page name. For Oracle JSP 1.1.x.x releases, it will be the same core name as the JSP name (unless special characters are included in the JSP name), but with an underscore ("_") prefix and `.res` suffix. Translation of `MyPage.jsp`, for example, would create `_MyPage.res` in addition to normal output. The exact implementation for name generation may change in future releases, however.

The resource file is placed in the same directory as `.class` files.

If there is a lot of static content in a page, this technique will speed translation and may speed execution of the page. For more information, see "Workarounds for Large Static Content in JSP Pages" on page 4-10.

Notes:

- The inner class is still created and must still be deployed.
- In an on-demand translation scenario, the JSP `external_resource` configuration parameter provides the same functionality. See "Oracle JSP Configuration Parameters" on page 9-7.

-implement (fully qualified Java interface name; `ojspc` default: empty)

Use this option to specify a Java interface that the generated page implementation class will implement.

-noCompile (boolean; `ojspc` default: `false`)

Enable this flag to direct `ojspc` *not* to compile the generated page implementation class Java source. This allows you to compile it later with an alternative Java compiler.

Notes:

- In an on-demand translation scenario, the JSP `javaccmd` configuration parameter provides related functionality, allowing you to specify an alternative Java compiler directly. See "Oracle JSP Configuration Parameters" on page 9-7.
- For a SQLJ JSP page, enabling `-noCompile` does not prevent SQLJ translation, just Java compilation.

-packageName (fully qualified package name; `ojspc` default: per `.jsp` file location)

Use this option to specify a package name for the generated page implementation class, using Java "dot" syntax.

Without setting this option, the package name is determined according to the location of the `.jsp` file relative to the current directory (from which you ran `ojspc`).

Consider an example where you run `ojspc` from the `/myapproot` directory, while the `.jsp` file is in the `/myapproot/src/jspsrc` directory (assume `%` is a UNIX prompt):

```
% cd /myapproot
% ojspc -packageName myroot.mypackage src/jspsrc/Foo.jsp
```

This results in `myroot.mypackage` being used as the package name.

If this example did *not* use the `-packageName` option, Oracle JSP 1.1.x.x releases would use `_src._jspsrc` as the package name, by default. (Be aware that such implementation details are subject to change in future releases.)

-S-<sqlj option> <value> (`-S` followed by SQLJ option setting; `ojspc` default: empty)

For SQLJ JSP pages, use the `ojspc -S` option to pass an Oracle SQLJ option to the SQLJ translator. You can use multiple occurrences of `-S`, with one SQLJ option per occurrence.

Unlike when you run the SQLJ translator directly, use a space between a SQLJ option and its value (this is for consistency with other `ojspc` options).

For example (from a UNIX prompt):

```
% ojspc -S-codegen iso -d /myapproot/mybindir MyPage.jsp
```

This directs SQLJ to generate ISO standard code instead of the default Oracle-specific code.

Here is another example:

```
% ojspc -S-codegen iso -S-ser2class true -d /myapproot/mybindir MyPage.jsp
```

This again directs SQLJ to generate ISO standard code, and also enables the `-ser2class` option to convert the profile to a `.class` file.

Note: As the preceding example shows, you can use an explicit `true` setting in enabling a SQLJ boolean option through the `-S` option setting. This is in contrast to `ojspc` boolean options, which do *not* take an explicit `true` setting.

Note the following for particular Oracle SQLJ options:

- Do not use the SQLJ `-encoding` option; instead, use the `contentType` parameter in a `page` directive in the JSP page.
- Do not use the SQLJ `-classpath` option if you use the `ojspc -addclasspath` option.
- Do not use the SQLJ `-compile` option if you use the `ojspc -noCompile` option.
- Do not use the SQLJ `-d` option if you use the `ojspc -d` option.
- Do not use the SQLJ `-dir` option if you use the `ojspc -srcdir` option.

For information about Oracle SQLJ translator options, see the *Oracle9i SQLJ Developer's Guide and Reference.*

Note: In an on-demand translation scenario, the JSP `sqljcmd` configuration parameter provides related functionality, allowing you to specify an alternative SQLJ translator or specify SQLJ option settings. See "Oracle JSP Configuration Parameters" on page 9-7.

-srcdir (fully qualified path; `ojspc` default: current directory)

Use this option to specify a base directory location for `ojspc` placement of generated source files—`.sqlj` files (for SQLJ JSP pages) and `.java` files.

The specified path is taken simply as a file system path (not an application-relative or page-relative path).

Subdirectories under the specified directory are created automatically, as appropriate, depending on the package. See "Summary of ojspc Output Files, Locations, and Related Options" on page 6-25 for more information.

The default is to use the current directory (your current directory when you executed `ojspc`).

It is recommended that you use this option to place generated source files into a clean directory so that you conveniently know what files have been produced.

Notes:

- In environments such as Windows NT that allow spaces in directory names, enclose the directory name in quotes.
- In an on-demand translation scenario, the JSP `page_repository_root` configuration parameter provides related functionality. See "Oracle JSP Configuration Parameters" on page 9-7.

-verbose (boolean; `ojspc` default: `false`)

Enable this option to direct `ojspc` to report its translation steps as it executes.

The following example shows `-verbose` output for the translation of `myerror.jsp` (in this example, `ojspc` is run from the directory where `myerror.jsp` is located; assume `%` is a UNIX prompt):

```
% ojspc -verbose myerror.jsp
Translating file: myerror.jsp
1 JSP files translated successfully.
Compiling Java file: ./_myerror.java
```

-version (boolean; `ojspc` default: `false`)

Enable this option for `ojspc` to display the Oracle JSP version number and then exit.

Summary of ojspc Output Files, Locations, and Related Options

By default, `ojspc` generates the same set of files that are generated by the Oracle JSP translator in an on-demand translation scenario and places them in or under the current directory (from which `ojspc` was executed).

Here are the files:

- a `.sqlj` source file (SQLJ JSP pages only)
- a `.java` source file
- a `.class` file for the page implementation class
- a `.class` file for the inner class for static text
- a Java resource file (`.ser`) or, optionally, a `.class` file for the SQLJ profile (SQLJ JSP pages only)

  This assumes standard SQLJ code generation. Oracle-specific SQLJ code generation produces no profiles.
- optionally, a Java resource file (`.res`) for the static text of the page

For more information about files that are generated by the Oracle JSP translator, see "Generated Files and Locations (On-Demand Translation)" on page 6-6.

To summarize some of the commonly used options described under "Option Descriptions for ojspc" on page 6-18, you can use the following `ojspc` options to affect file generation and placement:

- `-appRoot` to specify an application root directory
- `-srcdir` to place source files in a specified alternative location
- `-d` to place binary files (`.class` files and Java resource files) in a specified alternative location
- `-noCompile` to *not* compile the generated page implementation class source (as a result of this, no `.class` files are produced)

  In the case of SQLJ JSP pages, translated `.java` files are still produced, but not compiled.
- `-extres` to put static text into a Java resource file
- `-S-ser2class` (SQLJ `-ser2class` option, for SQLJ JSP pages only, and for ISO standard SQLJ code generation only) to generate the SQLJ profile in a `.class` file instead of a `.ser` Java resource file

For output file placement, the directory structure underneath the current directory (or directories specified by the `-d` and `-srcdir` options, as applicable) is based on the package. The package is based on the location of the file being translated relative to the application root, which is either the current directory or the directory specified in the `-appRoot` option.

For example, presume you run `ojspc` as follows (presume `%` is a UNIX prompt):

```
% cd /abc
% ojspc def/ghi/test.jsp
```

Then the package is `_def._ghi` and output files will be placed in the directory `/abc/_def/_ghi`, where the `_def/_ghi` subdirectory structure is created as part of the process.

If you specify alternate output locations through the `-d` and `-srcdir` options, a `_def/_ghi` subdirectory structure is created under the specified directories.

Now presume `ojspc` is run from some other directory, as follows:

```
% cd /home/mydir
% ojspc -appRoot /abc /abc/def/ghi/test.jsp
```

The package is still `_def._ghi`, according to the location of `test.jsp` relative to the specified application root. Output files will be placed in the directory `/home/mydir/_def/_ghi` or in a `_def/_ghi` subdirectory under locations specified through the `-d` and `-srcdir` options. In either case, the `_def/_ghi` subdirectory structure is created as part of the process.

Notes: It is advisable that you run `ojspc` once for each directory of your JSP application, so files in different directories can be given different package names, as appropriate.

Additional JSP Deployment Considerations

This section covers a variety of general deployment considerations and scenarios, mostly independent of your target environment.

The following topics are covered:

- General JSP Pre-Translation Without Execution
- Deployment of Binary Files Only
- Deployment of JSP Pages with Oracle9i JDeveloper
- Doc Root for JServ

General JSP Pre-Translation Without Execution

In an on-demand translation environment, it is possible to specify JSP pre-translation only, without execution, by enabling the standard `jsp_precompile` request parameter when invoking the JSP page from the end user's browser.

Following is an example:

```
http://host[:port]/foo.jsp?jsp_precompile
```

Refer to the Sun Microsystems *JavaServer Pages Specification, Version 1.1*, for more information.

Deployment of Binary Files Only

If your JSP source is proprietary, you can avoid exposing the source by pre-translating JSP pages and deploying only the translated and compiled binary files. Pages that are pre-translated, either from previous execution in an on-demand translation scenario or by using `ojspc`, can be deployed to any environment that supports the Oracle JSP container. There are two aspects to this scenario:

- You must deploy the binary files appropriately.
- In the target environment, the Oracle JSP container must be configured properly to run pages when the `.jsp` (or `.sqljsp`) source is not available.

Deploying the Binary Files

After JSP pages have been translated, archive the directory structure and contents that are under the binary output directory, then copy the directory structure and contents to the target environment, as appropriate. For example:

- If you pre-translate with `ojspc`, you should specify a binary output directory with the `ojspc -d` option, then archive the directory structure under that specified directory.
- If you are archiving binary files produced during previous execution in a JServ (on-demand translation) environment, archive the output directory structure, typically under the default `htdocs/_pages` directory.

In the target environment, restore the archived directory structure under the appropriate directory, such as under the `htdocs/_pages` directory in a JServ environment.

Configuring the Oracle JSP Container for Execution with Binary Files Only

Set JSP configuration parameters as follows to execute JSP pages when the `.jsp` or `.sqljsp` source is unavailable:

- `bypass_source` to `true`
- `developer_mode` to `false`

Without these settings, the Oracle JSP container will always look for the `.jsp` or `.sqljsp` file to see if it has been modified more recently than the page implementation `.class` file, and abort with a "file not found" error if it cannot find the `.jsp` or `.sqljsp` file.

With these parameters set appropriately, the end user can invoke a page with the same URL that would be used if the source file were in place. For an example, consider a JServ environment—if the binary files for `foo.jsp` are in the `htdocs/_pages/_test` directory, then the page can be invoked with the following URL without `foo.jsp` being present:

http://*host*:[*port*]/test/foo.jsp

For how to set configuration parameters in a JServ environment, see "Setting JSP Parameters in JServ" on page 9-18.

Deployment of JSP Pages with Oracle9*i* JDeveloper

Oracle9*i* JDeveloper release 3.1 and higher includes a deployment option, "Web Application to Web Server", that was added specifically for JSP applications.

This option generates a deployment profile that specifies the following:

- a JAR file containing Business Components for Java (BC4J) classes required by the JSP application
- static HTML files required by the JSP application
- the path to the Web server

The developer can either deploy the application immediately upon creating the profile, or save the profile for later use.

Doc Root for JServ

JSP pages and servlets running in the JServ environment supplied with Oracle9*i*, which are routed through the Apache `mod_jserv` module provided with JServ, use the Apache doc root. This doc root (typically `htdocs`) is set in the `DocumentRoot` command of the Apache `httpd.conf` configuration file.

For JSP pages running in JServ, JSP pages as well as static files are located in or under the doc root.

Note: For an overview of the role of the Oracle HTTP Server and its `mod_jserv` module, see "Role of the Oracle HTTP Server" on page 2-3.

7 JSP Tag Libraries

This chapter discusses custom tag libraries, covering the basic framework that vendors can use to provide their own libraries, and concluding with a comparison of standard runtime tags versus vendor-specific compile-time tags. The chapter is organized as follows:

- Standard Tag Library Framework
- Compile-Time Tags

For complete information about the tag libraries provided with Oracle9*i* release 2, as summarized in "Overview of JSP Tag Libraries and JavaBeans Provided with Oracle9i" on page 2-11, see the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Note: The tag library framework is supported by the Oracle JSP container even in a JServ (servlet 2.0) environment. For full servlet 2.2 tag library support, however, you should use a servlet 2.2 or higher environment such as Oracle9*i*AS Containers for J2EE (preferably) or Tomcat.

Standard Tag Library Framework

Standard JavaServer Pages technology allows vendors to create custom JSP tag libraries.

A tag library defines a collection of custom actions. The tags can be used directly by developers in manually coding a JSP page, or automatically by Java development tools. A tag library must be portable between different JSP container implementations.

For information beyond what is provided here regarding tag libraries and the standard JavaServer Pages tag library framework, refer to the following resources.

- Sun Microsystems *JavaServer Pages Specification, Version 1.1*
- Sun Microsystems Javadoc for the `javax.servlet.jsp.tagext` package, at the following Web site:

```
http://java.sun.com/j2ee/j2sdkee/techdocs/api/javax/servlet/jsp/tagext/package-summary.html
```

Note: Do not use the `servlet.jar` file of the Tomcat 3.1 beta servlet/JSP implementation if you are using custom tags. The constructor signature was changed for the class `javax.servlet.jsp.tagext.TagAttributeInfo`, which will result in compilation errors. Instead, use the `servlet.jar` file that is provided with Oracle9*i* release 2 or the production version of Tomcat 3.1.

Overview of a Custom Tag Library Implementation

A custom tag library is made accessible to a JSP page through a `taglib` directive of the following general form:

```
<%@ taglib uri="URI" prefix="prefix" %>
```

Note the following:

- The tags of a library are defined in a *tag library description* file, as described in "Tag Library Description Files" on page 7-11.
- The URI in the `taglib` directive specifies where to find the tag library description file, as described in "The taglib Directive" on page 7-14. It is possible to use *URI shortcuts*, as described in "Use of web.xml for Tag Libraries" on page 7-12.

- The prefix in the `taglib` directive is a string of your choosing that you use in your JSP page with any tag from the library.

  Assume the `taglib` directive specifies a prefix `oracust`:

  ```
  <%@ taglib uri="URI" prefix="oracust" %>
  ```

  Further assume that there is a tag `mytag` in the library. You might use `mytag` as follows:

  ```
  <oracust:mytag attr1="...", attr2="..." />
  ```

  Using the `oracust` prefix informs the JSP translator that `mytag` is defined in the tag library description file that can be found at the URI specified in the above `taglib` directive.

- The entry for a tag in the tag library description file provides specifications about usage of the tag, including whether the tag uses attributes (as `mytag` does), and the names of those attributes.

- The semantics of a tag—the actions that occur as the result of using the tag—are defined in a *tag handler class*, as described in "Tag Handlers" on page 7-4. Each tag has its own tag handler class, and the class name is specified in the tag library description file.

- The tag library description file indicates whether a tag uses a body.

  As seen above, a tag without a body is used as in the following example:

  ```
  <oracust:mytag attr1="...", attr2="..." />
  ```

  By contrast, a tag with a body is used as in the following example:

  ```
  <oracust:mytag attr1="...", attr2="..." >
     ...body...
  </oracust:mytag>
  ```

- A custom tag action can create one or more server-side objects that are available for use by the tag itself or by other JSP scripting elements, such as scriptlets. These objects are referred to as *scripting variables*.

  Details regarding the scripting variables that a custom tag uses are defined in a *tag-extra-info* class. This is described in "Scripting Variables and Tag-Extra-Info Classes" on page 7-7.

A tag can create scripting variables with syntax such as in the following example, which creates the object `myobj`:

```
<oracust:mytag id="myobj" attr1="...", attr2="..." />
```

- The tag handler of a nested tag can access the tag handler of an outer tag, in case this is required for any of the processing or state management of the nested tag. See "Access to Outer Tag Handler Instances" on page 7-10.

The sections that follow provide more information about these topics.

Tag Handlers

A *tag handler* describes the semantics of the action that results from use of a custom tag. A tag handler is an instance of a Java class that implements one of two standard Java interfaces, depending on whether the tag processes a body of statements (between a start tag and an end tag).

Each tag has its own handler class. By convention, the name of the tag handler class for a tag `abc`, for example, is `AbcTag`.

The tag library description (TLD) file of a tag library specifies the name of the tag handler class for each tag in the library. (See "Tag Library Description Files" on page 7-11.)

A tag handler instance is a server-side object used at request time. It has properties that are set by the JSP container, including the page context object for the JSP page that uses the custom tag, and a parent tag handler object if the use of this custom tag is nested within an outer custom tag.

See "Sample Tag Handler Class: ExampleLoopTag.java" on page 7-16 for sample code of a tag handler class.

Note: The Sun Microsystems *JavaServer Pages Specification, Version 1.1* does not mandate whether multiple uses of the same custom tag within a JSP page should use the same tag handler instance or different tag handler instances—this implementation detail is left to the discretion of JSP vendors. The Oracle JSP container uses a separate tag handler instance for each use of a tag.

Custom Tag Body Processing

Custom tags, like standard JSP tags, may or may not have a body. And in the case of a custom tag, even when there is a body, it may not need special handling by the tag handler.

There are three possible situations:

- There is no body.

  In this case, there is just a single tag, as opposed to a start tag and end tag. Following is a general example:

  ```
  <oracust:abcdef attr1="...", attr2="..." />
  ```

- There is a body that does not need special handling by the tag handler.

  In this case, there is a start tag and end tag with a body of statements in between, but the tag handler does not have to process the body—body statements are passed through for normal JSP processing only. Following is a general example:

  ```
  <foo:if cond="<%= ... %>" >
  ...body executed if cond is true, but not processed by tag handler...
  </foo:if>
  ```

- There is a body that needs special handling by the tag handler.

  In this case also, there is a start tag and end tag with a body of statements in between; however, the tag handler must process the body.

  ```
  <oracust:ghijkl attr1="...", attr2="..." >
  ...body processed by tag handler...
  </oracust:ghijkl>
  ```

Integer Constants for Body Processing

The tag handling interfaces that are described in the following sections specify a `doStartTag()` method (further described below) that you must implement to return an appropriate `int` constant, depending on the situation. The possible return values are as follows:

- `SKIP_BODY` if there is no body or if evaluation and execution of the body should be skipped
- `EVAL_BODY_INCLUDE` if there is a body that does not require special handling by the tag handler

- `EVAL_BODY_TAG` if there is a body that requires special handling by the tag handler

Handlers for Tags That Do Not Process a Body

For a custom tag that does not have a body, or has a body that does not need special handling by the tag handler, the tag handler class implements the following standard interface:

- `javax.servlet.jsp.tagext.Tag`

The following standard support class implements the `Tag` interface and can be used as a base class:

- `javax.servlet.jsp.tagext.TagSupport`

The `Tag` interface specifies a `doStartTag()` method and a `doEndTag()` method. The tag developer provides code for these methods in the tag handler class, as appropriate, to be executed as the start tag and end tag, respectively, are encountered. The JSP page implementation class generated by the Oracle JSP translator includes appropriate calls to these methods.

Action processing—whatever you want the action tag to accomplish—is implemented in the `doStartTag()` method. The `doEndTag()` method would implement any appropriate post-processing. In the case of a tag without a body, essentially nothing happens between the execution of these two methods.

The `doStartTag()` method returns an integer value. For a tag handler class implementing the `Tag` interface (either directly or indirectly), this value must be either `SKIP_BODY` or `EVAL_BODY_INCLUDE` (described in "Integer Constants for Body Processing" above). `EVAL_BODY_TAG` is illegal for a tag handler class implementing the `Tag` interface.

Handlers for Tags That Process a Body

For a custom tag with a body that requires special handling by the tag handler, the tag handler class implements the following standard interface:

- `javax.servlet.jsp.tagext.BodyTag`

The following standard support class implements the `BodyTag` interface and can be used as a base class:

- `javax.servlet.jsp.tagext.BodyTagSupport`

The `BodyTag` interface specifies a `doInitBody()` method and a `doAfterBody()` method in addition to the `doStartTag()` and `doEndTag()` methods specified in the `Tag` interface.

Just as with tag handlers implementing the `Tag` interface (described in the preceding section, "Handlers for Tags That Do Not Process a Body"), the tag developer implements the `doStartTag()` method for action processing by the tag, and the `doEndTag()` method for any post-processing.

The `doStartTag()` method returns an integer value. For a tag handler class implementing the `BodyTag` interface (directly or indirectly), this value must be either `SKIP_BODY` or `EVAL_BODY_TAG` (described in "Integer Constants for Body Processing" on page 7-5). `EVAL_BODY_INCLUDE` is illegal for a tag handler class implementing the `BodyTag` interface.

In addition to implementing the `doStartTag()` and `doEndTag()` methods, the tag developer, as appropriate, provides code for the `doInitBody()` method, to be invoked before the body is evaluated, and the `doAfterBody()` method, to be invoked after each evaluation of the body. (The body could be evaluated multiple times, such as at the end of each iteration of a loop.) The JSP page implementation class generated by the Oracle JSP translator includes appropriate calls to all of these methods.

After the `doStartTag()` method is executed, the `doInitBody()` and `doAfterBody()` methods are executed if the `doStartTag()` method returned `EVAL_BODY_TAG`.

The `doEndTag()` method is executed after any body processing, when the end tag is encountered.

For custom tags that must process a body, the `javax.servlet.jsp.tagext.BodyContent` class is available for use. This is a subclass of `javax.servlet.jsp.JspWriter` that can be used to process body evaluations so that they can re-extracted later. The `BodyTag` interface includes a `setBodyContent()` method that can be used by the JSP container to give a `BodyContent` handle to a tag handler instance.

Scripting Variables and Tag-Extra-Info Classes

A custom tag action can create one or more server-side objects, known as *scripting variables*, that are available for use by the tag itself or by other scripting elements, such as scriptlets and other tags.

Details regarding scripting variables that a custom tag defines must be specified in a subclass of the standard `javax.servlet.jsp.tagext.TagExtraInfo` abstract class. This document refers to such a subclass as a *tag-extra-info class*.

The JSP container uses tag-extra-info instances during translation. (The tag library description file, specified in the `taglib` directive that imports the library into a JSP page, specifies the tag-extra-info class to use, if applicable, for any given tag.)

A tag-extra-info class has a `getVariableInfo()` method to retrieve names and types of the scripting variables that will be assigned during HTTP requests. The JSP translator calls this method during translation, passing it an instance of the standard `javax.servlet.jsp.tagext.TagData` class. The `TagData` instance specifies attribute values set in the JSP statement that uses the custom tag.

This section covers the following topics:

- Defining Scripting Variables
- Scripting Variable Scopes
- Tag-Extra-Info Classes and the getVariableInfo() Method

Defining Scripting Variables

Objects that are defined explicitly in a custom tag can be referenced in other actions through the page context object, using the object ID as a handle. Consider the following example:

```
<oracust:foo id="myobj" attr1="..." attr2="..." />
```

This statement results in the object `myobj` being available to any scripting elements between the tag and the end of the page. The `id` attribute is a translation-time attribute. The tag developer provides a tag-extra-info class that will be used by the JSP container. Among other things, the tag-extra-info class specifies what class to instantiate for the `myobj` object.

The JSP container enters `myobj` into the page context object, where it can later be obtained by other tags or scripting elements using syntax such as the following:

```
<oracust:bar ref="myobj" />
```

The `myobj` object is passed through the tag handler instances for `foo` and `bar`. All that is required is knowledge of the name of the object (`myobj`).

Important: Note that `id` and `ref` are merely sample attribute names; there are no special predefined semantics for these attributes. It is up to the tag handler to define attribute names and create and retrieve objects in the page context.

Scripting Variable Scopes

Specify the scope of a scripting variable in the tag-extra-info class of the tag that creates the variable. It can be one of the following `int` constants:

- `NESTED`—if the scripting variable is available between the start tag and end tag of the action that defines it
- `AT_BEGIN`—if the scripting variable is available from the start tag until the end of the page
- `AT_END`—if the scripting variable is available from the end tag until the end of the page

Tag-Extra-Info Classes and the getVariableInfo() Method

You must create a tag-extra-info class for any custom tag that creates scripting variables. The class describes the scripting variables and must be a subclass of the standard `javax.servlet.jsp.tagext.TagExtraInfo` abstract class.

The key method of the `TagExtraInfo` class is `getVariableInfo()`, which is called by the JSP translator and returns an array of instances of the standard `javax.servlet.jsp.tagext.VariableInfo` class (one array instance for each scripting variable the tag creates).

The tag-extra-info class constructs each `VariableInfo` instance with the following information regarding the scripting variable:

- its name
- its Java type (cannot be a primitive type)
- a boolean indicating whether it is a newly declared variable
- its scope

Important: In Oracle JSP 1.1.x.x releases, the `getVariableInfo()` method can return either a fully qualified class name (FQCN) or a partially qualified class name (PQCN) for the Java type of the scripting variable. FQCNs were required in previous releases, and are still preferred in order to avoid confusion in case there are duplicate class names between packages.

Note that primitive types are not supported.

See "Sample Tag-Extra-Info Class: ExampleLoopTagTEI.java" on page 7-17 for sample code of a tag-extra-info class.

Access to Outer Tag Handler Instances

Where nested custom tags are used, the tag handler instance of the nested tag has access to the tag handler instance of the outer tag, which may be useful in any processing and state management performed by the nested tag.

This functionality is supported through the static `findAncestorWithClass()` method of the `javax.servlet.jsp.tagext.TagSupport` class. Even though the outer tag handler instance is not named in the page context object, it is accessible because it is the closest enclosing instance of a given tag handler class.

Consider the following JSP code example:

```
<foo:bar1 attr="abc" >
   <foo:bar2 />
</foo:bar1>
```

Within the code of the `bar2` tag handler class (class `Bar2Tag`, by convention), you can have a statement such as the following:

```
Tag bar1tag = TagSupport.findAncestorWithClass(this, Bar1Tag.class);
```

The `findAncestorWithClass()` method takes the following as input:

- the `this` object that is the class handler instance from which `findAncestorWithClass()` was called (a `Bar2Tag` instance in the example)
- the name of the `bar1` tag handler class (presumed to be `Bar1Tag` in the example), as a `java.lang.Class` instance

The findAncestorWithClass() method returns an instance of the appropriate tag handler class, in this case Bar1Tag, as a javax.servlet.jsp.tagext.Tag instance.

It is useful for a Bar2Tag instance to have access to the outer Bar1Tag instance in case the Bar2Tag needs the value of a bar1 tag attribute or needs to call a method on the Bar1Tag instance.

Tag Library Description Files

A *tag library description* (TLD) file is an XML document that contains information about a tag library and about individual tags of the library. The name of a TLD file has the .tld extension.

A JSP container uses the TLD file in determining what action to take when it encounters a tag from the library.

A tag entry in the TLD file includes the following:

- name of the custom tag
- name of the corresponding tag handler class
- name of the corresponding tag-extra-info class (if applicable)
- information indicating how the tag body (if any) should be processed
- information about the attributes of the tag (the attributes that you specify whenever you use the custom tag)

Here is a sample TLD file entry for the tag myaction:

```
<tag>
  <name>myaction</name>
  <tagclass>examples.MyactionTag</tagclass>
  <teiclass>examples.MyactionTagExtraInfo</teiclass>
  <bodycontent>JSP</bodycontent>
  <info>
      Perform a server-side action (one mandatory attr; one optional)
  </info>
  <attribute>
    <name>attr1</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>attr2</name>
    <required>false</required>
  </attribute>
</tag>
```

According to this entry, the tag handler class is `MyactionTag` and the tag-extra-info class is `MyactionTagExtraInfo`. The attribute `attr1` is required; the attribute `attr2` is optional.

The `bodycontent` parameter indicates how the tag body (if any) should be processed. There are three valid values:

- A value of `empty` indicates that the tag uses no body.
- A value of `JSP` indicates that the tag body should be processed as JSP source and translated.
- A value of `tagdependent` indicates that the tag body should not be translated. Any text in the body is treated as static text.

The `taglib` directive in a JSP page informs the JSP container where to find the TLD file. (See "The taglib Directive" on page 7-14.)

For more information about tag library description files, see the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

Note: In the Tomcat 3.1 servlet/JSP implementation, the TLD file `bodycontent` parameter for a given tag is not read if the tag itself (in the JSP page) has no body. It is possible, therefore, to have an invalid `bodycontent` value in your TLD file (such as `none` instead of `empty`) without realizing it. Using the file in another JSP environment, such as the Oracle JSP container, would then result in errors.

Use of web.xml for Tag Libraries

The Sun Microsystems *Java Servlet Specification, Version 2.2* describes a standard deployment descriptor for servlets—the `web.xml` file. JSP pages can use this file in specifying the location of a JSP tag library description file.

For JSP tag libraries, the `web.xml` file can include a `taglib` element and two subelements:

- `taglib-uri`
- `taglib-location`

The `taglib-location` subelement indicates the application-relative location (by starting with "/") of the tag library description file.

The `taglib-uri` subelement indicates a "shortcut" URI to use in `taglib` directives in your JSP pages, with this URI being mapped to the TLD file location specified in the accompanying `taglib-location` subelement. (The term URI, *universal resource indicator*, is somewhat equivalent to the term URL, *universal resource locator*, but is more generic.)

Important: When a JSP application uses a `web.xml` file, you must deploy `web.xml` with the application. Treat it as a Java resource file.

Following is a sample `web.xml` entry for a tag library description file:

```
<taglib>
   <taglib-uri>/oracustomtags</taglib-uri>
   <taglib-location>/WEB-INF/oracustomtags/tlds/MyTLD.tld</taglib-location>
</taglib>
```

This makes `/oracustomtags` equivalent to `/WEB-INF/oracustomtags/tlds/MyTLD.tld` in `taglib` directives in your JSP pages. See "Using a Shortcut URI for the TLD File" below for an example.

See the Sun Microsystems *Java Servlet Specification, Version 2.2* and the Sun Microsystems *JavaServer Pages Specification, Version 1.1* for more information about the `web.xml` deployment descriptor and its use for tag library description files.

Notes:

- Do not use the sample `web.xml` file from the Tomcat 3.1 servlet/JSP implementation. It introduces new elements that will not pass the standard DTD XML validation.
- Do not use the term "urn" instead of "uri" in a `web.xml` file. Some JSP implementations allow this (such as Tomcat 3.1), but using "urn" will not pass the standard DTD XML validation.

The taglib Directive

Import a custom library into a JSP page using a `taglib` directive, as follows:

```
<%@ taglib uri="URI" prefix="prefix" %>
```

For the URI, you have the following options:

- Specify a shortcut URI, as defined in a `web.xml` file (see "Use of web.xml for Tag Libraries" above).
- Fully specify the tag library description (TLD) file name and location.

Using a Shortcut URI for the TLD File

Assume the following `web.xml` entry for a tag library defined in the tag library description file `MyTLD.tld`:

```
<taglib>
   <taglib-uri>/oracustomtags</taglib-uri>
   <taglib-location>/WEB-INF/oracustomtags/tlds/MyTLD.tld</taglib-location>
</taglib>
```

Given this example, the following directive in your JSP page results in the JSP container finding the `/oracustomtags` URI in `web.xml` and, therefore, finding the accompanying name and location of the tag library description file (`MyTLD.tld`):

```
<%@ taglib uri="/oracustomtags" prefix="oracust" %>
```

This statement allows you to use any of the tags of this custom tag library in a JSP page.

Fully Specifying the TLD File Name and Location

If you do not want your JSP application to depend on a `web.xml` file for its use of a tag library, `taglib` directives can fully specify the name and location of the tag library description file, as follows:

```
<%@ taglib uri="/WEB-INF/oracustomtags/tlds/MyTLD.tld" prefix="oracust" %>
```

The location is specified as an application-relative location (by starting with "/", as in this example). See "Requesting a JSP Page" on page 1-8 for related discussion of application-relative syntax.

Alternatively, you can specify a `.jar` file instead of a `.tld` file in the `taglib` directive, where the `.jar` file contains a tag library description file. The tag library description file must be located and named as follows when you create the JAR file (for servlet 2.2):

```
META-INF/taglib.tld
```

Then the `taglib` directive might be as follows, for example:

```
<%@ taglib uri="/WEB-INF/oracustomtags/tlds/MyTLD.jar" prefix="oracust" %>
```

End-to-End Example: Defining and Using a Custom Tag

This section provides an end-to-end example of the definition and use of a custom tag, `loop`, that is used to iterate through the tag body a specified number of times.

Included in the example are the following:

- JSP source for a page that uses the tag
- source code for the tag handler class
- source code for the tag-extra-info class
- the tag library description file

Note: Sample code here uses extended datatypes in the `oracle.jsp.jml` package. These types are described in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Sample JSP Page: exampletag.jsp

Following is a sample JSP page that uses the `loop` tag, specifying that the outer loop be executed five times and the inner loop three times:

```
exampletag.jsp
<%@ taglib prefix="foo" uri="/WEB-INF/exampletag.tld" %>
<% int num=5; %>
<br>
<pre>
<foo:loop index="i" count="<%=num%>">
body1here: i expr: <%=i%> i property: <jsp:getProperty name="i" property="value"
/>
  <foo:loop index="j" count="3">
  body2here: j expr: <%=j%>
  i property: <jsp:getProperty name="i" property="value" />
  j property: <jsp:getProperty name="j" property="value" />
  </foo:loop>
</foo:loop>
</pre>
```

Sample Tag Handler Class: ExampleLoopTag.java

This section provides source code for the tag handler class, `ExampleLoopTag`. Note the following:

- The `doStartTag()` method returns the integer constant `EVAL_BODY_TAG`, so that the tag body (essentially, the loop) is processed.
- After each pass through the loop, the `doAfterBody()` method increments the counter. It returns `EVAL_BODY_TAG` if there are more iterations left and `SKIP_BODY` after the last iteration.

Here is the code:

```
package examples;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.util.Hashtable;
import java.io.Writer;
import java.io.IOException;
import oracle.jsp.jml.JmlNumber;

public class ExampleLoopTag
    extends BodyTagSupport
{

    String index;
    int count;
    int i=0;
    JmlNumber ib=new JmlNumber();

    public void setIndex(String index)
    {
      this.index=index;
    }
    public void setCount(String count)
    {
      this.count=Integer.parseInt(count);
    }
    public int doStartTag() throws JspException {
        return EVAL_BODY_TAG;
    }

    public void doInitBody() throws JspException {
        pageContext.setAttribute(index, ib);
        i++;
        ib.setValue(i);
    }

    public int doAfterBody() throws JspException {
        try {
            if (i >= count) {
                bodyContent.writeOut(bodyContent.getEnclosingWriter());
                return SKIP_BODY;
            } else
                pageContext.setAttribute(index, ib);
            i++;
            ib.setValue(i);
            return EVAL_BODY_TAG;
        } catch (IOException ex) {
            throw new JspTagException(ex.toString());
        }
    }
}
```

Sample Tag-Extra-Info Class: ExampleLoopTagTEI.java

This section provides the source code for the tag-extra-info class that describes the scripting variable used by the `loop` tag.

A `VariableInfo` instance is constructed that specifies the following for the variable:

- The variable name is according to the `index` attribute.
- The variable is of the type `oracle.jsp.jml.JmlNumber` (this must be specified as a fully qualified class name).
- The variable is newly declared.
- The variable scope is `NESTED`.

In addition, the tag-extra-info class has an `isValid()` method that determines whether the count attribute is valid—it must be an integer.

```
package examples;
import javax.servlet.jsp.tagext.*;

public class ExampleLoopTagTEI extends TagExtraInfo {
    public VariableInfo[] getVariableInfo(TagData data) {
        return new VariableInfo[]
            {
                new VariableInfo(data.getAttributeString("index"),
                                 "oracle.jsp.jml.JmlNumber",
                                 true,
                                 VariableInfo.NESTED)
            };
    }

    public boolean isValid(TagData data)
    {
      String countStr=data.getAttributeString("count");
      if (countStr!=null)   // for request time case
      {
        try {
          int count=Integer.parseInt(countStr);
        }
        catch (NumberFormatException e)
        {
          return false;
        }
      }
      return true;
    }
}
```

Sample Tag Library Description File: exampletag.tld

This section presents the tag library description (TLD) file for the tag library. In this example, the library consists of only the one tag, `loop`.

This TLD file specifies the following for the `loop` tag:

- the tag handler class—`examples.ExampleLoopTag`
- the tag-extra-info class—`examples.ExampleLoopTagTEI`
- `bodycontent` specification of `JSP`

  This means the JSP translator should process and translate the body code.

- attributes `index` and `count`, both mandatory

  The `count` attribute can be a request-time JSP expression.

Here is the TLD file:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
        PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
        "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- a tag library descriptor -->

<taglib>
  <!-- after this the default space is
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>simple</shortname>
<!--
  there should be no <urn></urn> here
-->
  <info>
        A simple tag library for the examples
  </info>

  <!-- example tag -->
  <!-- for loop -->
  <tag>
    <name>loop</name>
    <tagclass>examples.ExampleLoopTag</tagclass>
    <teiclass>examples.ExampleLoopTagTEI</teiclass>
    <bodycontent>JSP</bodycontent>
    <info>for loop</info>
    <attribute>
        <name>index</name>
        <required>true</required>
    </attribute>
    <attribute>
        <name>count</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```

Compile-Time Tags

Standard tag libraries, as described in the Sun Microsystems *JavaServer Pages Specification, Version 1.1*, use a runtime support mechanism. They are typically portable, not requiring any particular JSP container.

It is also possible, however, for a vendor to support custom tags through vendor-specific functionality in their JSP translator. Such tags are not portable to other containers.

It is generally advisable to develop standard, portable tags that use the runtime mechanism, but there may be scenarios where tags using a compile-time mechanism are appropriate, as discussed in this section.

General Compile-Time Versus Runtime Considerations

The JSP 1.1 specification describes a runtime support mechanism for custom tag libraries. This mechanism, using an XML-style tag library description file to specify the tags, is covered in "Standard Tag Library Framework" on page 7-2.

Creating and using a tag library that adheres to this model assures that the library will be portable to any standard JSP environment.

There are, however, reasons to consider compile-time implementations:

- A compile-time implementation may produce more efficient code.
- A compile-time implementation allows the developer to catch errors during translation and compilation, instead of the end-user seeing them at runtime.

In the future, the Oracle JSP container may support a general framework for creating custom tag libraries with compile-time tag implementations. Such implementations would depend on the Oracle JSP translator, so would not be portable to other JSP environments.

Oracle JML Library: Compile-Time Versus Runtime

Oracle provides a portable tag library called the JSP Markup Language (JML) library. This library uses the standard JSP 1.1 runtime mechanism.

However, the JML tags are also supported through a compile-time mechanism. This is because the tags were first introduced with older JSP versions that preceded the JSP 1.1 specification, when the runtime mechanism was introduced. The compile-time tags are still supported for backward compatibility.

The general advantages and disadvantages of compile-time implementations apply to the Oracle JML tag library as well. There may be situations where it is advantageous to use the compile-time JML implementation as first introduced in older Oracle JSP versions. There are also a few additional tags in that implementation, and some additional expression syntax that is supported.

Both the runtime version and the compile-time version of the JML library are described in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

8 Oracle JSP Globalization Support

The Oracle JSP container provides standard globalization support (also known as National Language Support, or NLS) according to the Sun Microsystems *JavaServer Pages Specification, Version 1.1*, and also offers extended support for servlet environments that do not support multibyte parameter encoding.

Standard Java support for localized content depends on the use of Unicode 2.0 for uniform internal representation of text. Unicode is used as the base character set for conversion to alternative character sets.

This chapter describes key aspects of how the Oracle JSP container handles Oracle Globalization Support. The following topics are covered:

- Content Type Settings in the page Directive
- Dynamic Content Type Settings
- Oracle JSP Extended Support for Multibyte Parameter Encoding

---

Note: For detailed information about Oracle Globalization Support, see the *Oracle9i Database Globalization Support Guide*.

Content Type Settings in the page Directive

You can use the `page` directive `contentType` parameter to set the MIME type and to optionally set the character encoding for a JSP page. The MIME type applies to the HTTP response at runtime. The character encoding, if set, applies to both the page text during translation and the HTTP response at runtime.

Use the following syntax for the `page` directive:

```
<%@ page ... contentType="TYPE; charset=character_set" ... %>
```

or, to set the MIME type while using the default character set:

```
<%@ page ... contentType="TYPE" ... %>
```

`TYPE` is an IANA (Internet Assigned Numbers Authority) MIME type; `character_set` is an IANA character set. (When specifying a character set, the space after the semi-colon is optional.)

For example:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
```

or:

```
<%@ page language="java" contentType="text/html" %>
```

The default MIME type is `text/html`. The IANA maintains a registry of MIME types at the following site:

```
ftp://www.isi.edu/in-notes/iana/assignments/media-types/media-types
```

The default character encoding is `ISO-8859-1` (also known as Latin-1). The IANA maintains a registry of character encodings at the following site. Use the indicated "preferred MIME name" if one is listed:

```
http://www.iana.org/assignments/character-sets
```

(There is no JSP requirement to use an IANA character set as long as you use a character set that Java and the Web browser support, but the IANA site lists the most common character sets. Using the preferred MIME names they document is recommended.)

The parameters of a `page` directive are static. If a page discovers during execution that a different setting is necessary for the response, it can do one of the following:

- Use the servlet response object API to set the content type during execution, as described in "Dynamic Content Type Settings" on page 8-4.
- Forward the request to another JSP page or to a servlet.

Notes:

- The `page` directive that sets `contentType` should appear as early as possible in the JSP page.
- A JSP page written in a character set other than `ISO-8859-1` must set the appropriate character set in a `page` directive. It cannot be set dynamically because the page has to be aware of the setting during translation. Dynamic settings are for runtime only.
- The JSP 1.1 specification assumes that a JSP page is written in the same character set that it will use to deliver its content.
- This document, for simplicity, assumes the typical case that the page text, request parameters, and response parameters all use the same encoding (although other scenarios are technically possible). Request parameter encoding is controlled by the browser, although Netscape browsers and Internet Explorer follow the setting you specify for the response parameters.

Dynamic Content Type Settings

For situations where the appropriate content type for the HTTP response is not known until runtime, you can set it dynamically in the JSP page. The standard `javax.servlet.ServletResponse` interface specifies the following method for this purpose:

```
public void setContentType(java.lang.String contenttype)
```

The implicit `response` object of a JSP page is a `javax.servlet.http.HttpServletResponse` instance, where the `HttpServletResponse` interface extends the `ServletResponse` interface.

The `setContentType()` method input, like the `contentType` setting in a `page` directive, can include a MIME type only, or both a character set and a MIME type. For example:

```
response.setContentType("text/html; charset=UTF-8");
```

or:

```
response.setContentType("text/html");
```

As with a `page` directive, the default MIME type is `text/html` and the default character encoding is `ISO-8859-1`.

This method has no effect on interpreting the text of the JSP page during translation. If a particular character set is required during translation, that must be specified in a `page` directive, as described in "Content Type Settings in the page Directive" on page 8-2.

Be aware of the following important usage notes.

- The JSP page cannot be unbuffered if you are using the `setContentType()` method. It is buffered by default; do not set `buffer="none"` in a `page` directive.
- The `setContentType()` call must appear early in the page, before any output to the browser or any `jsp:include` command (which flushes the JSP buffer to the browser).
- In servlet 2.2 environments, the `response` object has a `setLocale()` method that sets a default character set based on the specified locale, overriding any previous character set. For example, the following method call results in a character set of `Shift_JIS`:

  ```
  response.setLocale(new Locale("ja", "JP"));
  ```

Oracle JSP Extended Support for Multibyte Parameter Encoding

Character encoding of request parameters is not well defined in the HTTP specification. Most servlet containers must interpret them using the servlet default encoding, `ISO-8859-1`.

For such environments, where the servlet container cannot encode multibyte request parameters and bean property settings, the Oracle JSP container offers extended support in two ways:

- through the `setReqCharacterEncoding()` method

or:

- through the `translate_params` configuration parameter

The setReqCharacterEncoding() Method

Oracle provides a `setReqCharacterEncoding()` method that is useful in case the default encoding for the servlet container is not appropriate. Use this method to specify the encoding of multibyte request parameters and bean property settings, such as for a `getParameter()` call in Java code or a `jsp:setProperty` statement to set a bean property in JSP code. If the default encoding is already appropriate, then it is not necessary to use this method, and in fact using it may create some performance overhead in your application.

The `setReqCharacterEncoding()` method is a static method in the `PublicUtil` class of the `oracle.jsp.util` package.

This method affects parameter names and values, specifically:

- request object `getParameter()` method output
- request object `getParameterValues()` method output
- request object `getParameterNames()` method output
- `jsp:setProperty` settings for bean property values

When invoking the method, input a request object and a string that specifies the desired encoding, as follows:

```
oracle.jsp.util.PublicUtil.setReqCharacterEncoding(myrequest, "EUC-JP");
```

Notes:

- Beginning with Oracle JSP 1.1.2.x releases, using the `setReqCharacterEncoding()` method is preferable to using the `translate_params` configuration parameter described in "The translate_params Configuration Parameter" below.
- The `setReqCharacterEncoding()` method is forward-compatible with the method `request.setCharacterEncoding(encoding)` of the servlet 2.3 API.

The translate_params Configuration Parameter

This section describes how to use the JSP `translate_params` configuration parameter for encoding of multibyte request parameters and bean property settings, such as for a `getParameter()` call in Java code or for a `jsp:setProperty` statement to set a bean property in JSP code.

Note that beginning with Oracle JSP 1.1.2.x releases, it is preferable to use the `PublicUtil.setReqCharacterEncoding()` method instead. See "The setReqCharacterEncoding() Method" above.

Also note that you should *not* enable `translate_params` in any of the following circumstances:

- when the servlet container properly handles multibyte parameter encoding itself

  Setting `translate_params` to `true` in this situation will cause incorrect results. As of this writing, however, it is known that JServ, JSWDK, and Tomcat all do *not* properly handle multibyte parameter encoding.

- when the request parameters use a different encoding from what is specified for the response in the JSP `page` directive or `setContentType()` method
- when code with workaround functionality equivalent to what `translate_params` accomplishes is already present in the JSP page

  See "Code Equivalent to the translate_params Configuration Parameter" on page 8-7.

Effect of translate_params in Overriding Non-Multibyte Servlet Containers

Setting `translate_params` to `true` overrides servlet containers that cannot encode multibyte request parameters and bean property settings. (For information about how to set JSP configuration parameters in a JServ environment, see "Setting JSP Parameters in JServ" on page 9-18.)

When this flag is enabled, the Oracle JSP container encodes the request parameters and bean property settings based on the character set of the `response` object, as indicated by the `response.getCharacterEncoding()` method.

The `translate_params` flag affects parameter names and values, specifically:

- request object `getParameter()` method output
- request object `getParameterValues()` method output
- request object `getParameterNames()` method output
- `jsp:setProperty` settings for bean property values

Code Equivalent to the translate_params Configuration Parameter

There may be situations where you do not want to or cannot use the `translate_params` configuration parameter. It is useful to be aware of equivalent functionality that can be implemented through scriptlet code in the JSP page, for example:

```
<%@ page contentType="text/html; charset=EUC-JP" %>
...
String paramName="XXYYZZ";       // where XXYYZZ is a multibyte string
paramName =
   new String(paramName.getBytes(response.getCharacterEncoding()), "ISO8859_1");
String paramValue = request.getParameter(paramName);
paramValue= new String(paramValue.getBytes("ISO8859_1"), "EUC-JP");
...
```

This code accomplishes the following:

- Sets `XXYYZZ` as the parameter name to search for. (Presume `XX`, `YY`, and `ZZ` are three Japanese characters.)
- Encodes the parameter name to `ISO-8859-1`, the servlet container character set, so that the servlet container can interpret it. (First a byte array is created for the parameter name, using the character encoding of the request object.)

- Gets the parameter value from the request object by looking for a match for the parameter name. (It is able to find a match because parameter names in the request object are also in `ISO-8859-1` encoding.)
- Encodes the parameter value to `EUC-JP` for further processing or output to the browser.

See the next two sections for a globalization sample that depends on `translate_params` being enabled, and one that contains the equivalent code so that it does not depend on the `translate_params` setting.

Globalization Sample Depending on translate_params

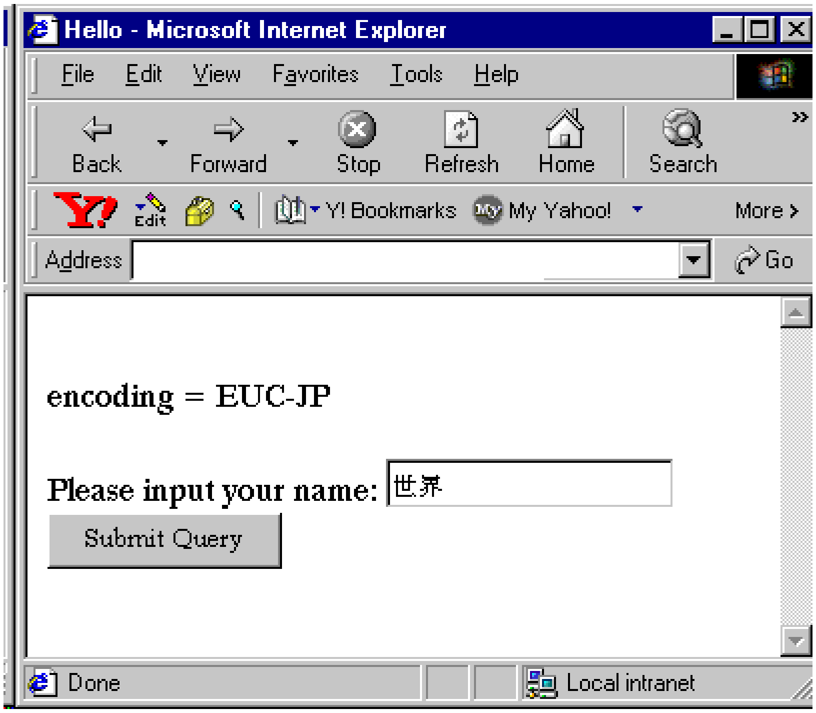
The following sample accepts a user name in Japanese characters and correctly outputs the name back to the browser. In a servlet environment that cannot encode multibyte request parameters, this sample depends on the JSP configuration setting of `translate_params=true`.

Presume `XXYY` is the parameter name (something equivalent to "user name" in Japanese) and `AABB` is the default value (also in Japanese).

(See the next section for a sample that has the code equivalent of the `translate_params` functionality, so does not depend on the `translate_params` setting.)

```
<%@ page contentType="text/html; charset=EUC-JP" %>

<HTML>
<HEAD>
<TITLE>Hello</TITLE></HEAD>
<BODY>
<%
   //charset is as specified in page directive (EUC-JP)
   String charset = response.getCharacterEncoding();
%>
   <BR> encoding = <%= charset %> <BR>

<%

String paramValue = request.getParameter("XXYY");

if (paramValue == null || paramValue.length() == 0) { %>
   <FORM METHOD="GET">
   Please input your name: <INPUT TYPE="TEXT" NAME="XXYY" value="AABB" size=20>
<BR>
   <INPUT TYPE="SUBMIT">
   </FORM>
<% }
else
{ %>
   <H1> Hello, <%= paramValue %> </H1>
<% } %>
</BODY>
</HTML>
```

Following is the sample input:

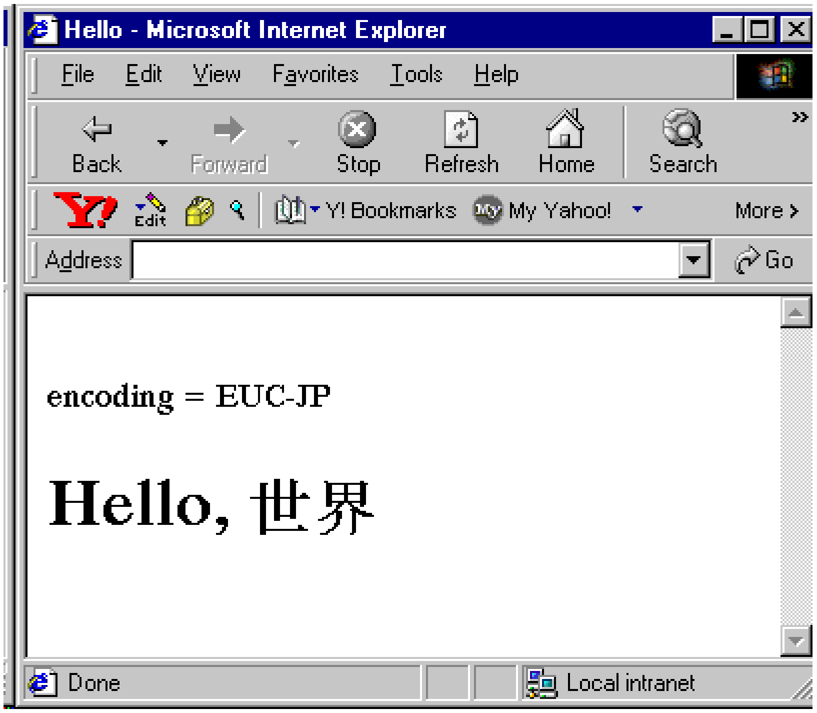

Hello - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Y! Bookmarks My Yahoo! More
Address Go
Done
Local intranet

encoding = EUC-JP

Please input your name: 世界

Submit Query

and the sample output:


Hello - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Y! Bookmarks My Yahoo! More
Address Go
Done
Local intranet

encoding = EUC-JP

Hello, 世界

Globalization Sample Not Depending on translate_params

The following sample, as with the preceding sample, accepts a user name in Japanese characters and correctly outputs the name back to the browser. This sample, however, has the code equivalent of `translate_params` functionality, so does not depend on the `translate_params` setting.

Important: If you use `translate_params`-equivalent code, do *not* also enable the `translate_params` flag. This would cause incorrect results.

Presume XXYY is the parameter name (something equivalent to "user name" in Japanese) and AABB is the default value (also in Japanese).

For an explanation of the critical code in this sample, see "Code Equivalent to the translate_params Configuration Parameter" on page 8-7.

```
<%@ page contentType="text/html; charset=EUC-JP" %>

<HTML>
<HEAD>
<TITLE>Hello</TITLE></HEAD>
<BODY>
<%
   //charset is as specified in page directive (EUC-JP)
   String charset = response.getCharacterEncoding();
%>
   <BR> encoding = <%= charset %> <BR>
<%
String paramName = "XXYY";

paramName = new String(paramName.getBytes(charset), "ISO8859_1");

String paramValue = request.getParameter(paramName);

if (paramValue == null || paramValue.length() == 0) { %>
   <FORM METHOD="GET">
   Please input your name: <INPUT TYPE="TEXT" NAME="XXYY" value="AABB" size=20>
<BR>
   <INPUT TYPE="SUBMIT">
   </FORM>
<% }
else
{
   paramValue= new String(paramValue.getBytes("ISO8859_1"), "EUC-JP"); %>
   <H1> Hello, <%= paramValue %> </H1>
<% } %>
</BODY>
</HTML>
```

9 Oracle JSP in Apache JServ

Oracle9*i* release 2 includes an Apache JServ servlet environment. For those who use this environment, there are special considerations relating to servlet and JSP usage, as with any servlet 2.0 environment.

The following topics are covered here:

Getting Started in a JServ Environment

This section provides information about configuring JServ to run JSP pages, covering the following topics:

- Required and Optional Files for Oracle JSP
- Adding Files to the JServ Web Server Classpath
- Mapping JSP File Name Extensions for JServ
- Oracle JSP Configuration Parameters
- Setting JSP Parameters in JServ

Required and Optional Files for Oracle JSP

This section summarizes JAR and ZIP files required in order to use the Oracle JSP container, as well as optional JAR and ZIP files to use Oracle JDBC and SQLJ functionality, JML or SQL custom tags, or custom data-access JavaBeans.

Required files must also be added to your classpath. (See "Adding Files to the JServ Web Server Classpath" on page 9-4.)

Note: The servlet library for your servlet environment must be installed on your system and included in the classpath in your Web server configuration file. This library contains the standard `javax.servlet.*` packages.

The following files are provided with Oracle9*i* release 2 and must be installed on your system:

- `ojsp.jar` (the Oracle JSP container)
- `xmlparserv2.jar` (for XML parsing—required for the `web.xml` deployment descriptor in a servlet 2.2 environment, and for any tag library descriptors)
- `servlet.jar` (standard servlet library, servlet 2.2 version)

In addition, if your JSP pages will use Oracle JSP Markup Language (JML) tags, SQL tags, or data-access JavaBeans, you will need the following files:

- `ojsputil.jar` (Oracle JSP utility library)
- `xsu12.jar`, for JDK 1.2.x, or `xsu111.jar`, for JDK 1.1.x (for XML functionality)

To run in a client environment, `xsu12.jar` or `xsu111.jar` is required only if you will use XML functionality in the data-access JavaBeans (such as getting a result set as an XML string). The `xsu12.jar` and `xsu111.jar` files are included with Oracle9*i*.

For Oracle data access, you will also need the following:

- Oracle JDBC class files (for any Oracle data access)
- Oracle SQLJ class files (if using SQLJ code in your JSP pages)

See "Files for JDBC (optional)" on page 9-4 and "Files for SQLJ (optional)" on page 9-4 for more information.

To use JDBC data sources or Enterprise JavaBeans, you will need the following:

- `jndi.jar`

(This file is required for some of the Oracle JSP demos.)

Servlet Library Notes Note that Oracle JSP 1.1.x.x releases require and supply the 2.2 version of the servlet library, which is where the standard `javax.servlet.*` packages are located. Your Web server environment likely requires and supplies a different servlet library version. You must be careful in your classpath to have the version for your Web server precede the version for the Oracle JSP container. "Adding Files to the JServ Web Server Classpath" on page 9-4 further discusses this.

Table 9–1 summarizes the servlet library versions. Do not confuse the Sun Microsystems JSWDK (JavaServer Web Developer's Kit) with the Sun Microsystems JSDK (Java Servlet Developer's Kit).

Table 9–1 Servlet Library Versions

Servlet Library Version	Library File Name	Provided with:
servlet 2.2	`servlet.jar`	Oracle JSP, Tomcat 3.1
servlet 2.1	`servlet.jar`	Sun JSWDK 1.0
servlet 2.0	`jsdk.jar`	Sun JSDK 2.0; also used with JServ

(For JServ, download `jsdk.jar` separately.)

Files for JDBC (optional) The following files are required if you will use Oracle JDBC for data access. (Be aware that Oracle SQLJ uses Oracle JDBC.)

- `ojdbc14.jar` or `.zip` (for JDK 1.4 environments)

or:

- `classes12.jar` or `.zip` (for JDK 1.2 or 1.3 environments)

or:

- `classes111.jar` or `.zip` (for JDK 1.1 environments)

Files for SQLJ (optional) The following files are necessary if your JSP pages use Oracle SQLJ for their data access:

- `translator.jar` or `.zip` (for the SQLJ translator, for JDK 1.2.x or 1.1.x)

as well as the appropriate SQLJ runtime:

- `runtime12.jar` or `.zip` (for JDK 1.2.x with Oracle9*i* JDBC)

or:

- `runtime12ee.jar` or `.zip` (for JDK 1.2.x enterprise edition with Oracle9*i* JDBC)

or:

- `runtime11.jar` or `.zip` (for JDK 1.1.x with Oracle9*i* JDBC)

or:

- `runtime.jar` or `.zip` (more general—for JDK 1.2.x or 1.1.x with any Oracle JDBC version)

or:

- `runtime-nonoracle.jar` or `.zip` (generic—for use with non-Oracle JDBC drivers and any JDK environment)

(The JDK 1.2.x enterprise edition allows data source support, in compliance with the ISO SQLJ specification.)

Adding Files to the JServ Web Server Classpath

To add files to the Web server classpath in a JServ environment, insert appropriate `wrapper.classpath` commands into the `jserv.properties` file in the JServ `conf` directory. Note that `jsdk.jar` should already be in the classpath. This file is from the Sun Microsystems JSDK 2.0 and provides servlet 2.0 versions of the javax.servlet.* packages that are required by JServ. Additionally, files for your JDK environment should already be in the classpath.

The following example (which happens to use UNIX directory paths) includes files for JSP, JDBC, and SQLJ. Replace [Oracle_Home] with your Oracle Home path.

```
# servlet 2.0 APIs (required by JServ, from Sun JSDK 2.0):
wrapper.classpath=jsdk2.0/lib/jsdk.jar
#
# servlet 2.2 APIs (required and provided by OC4J):
wrapper.classpath=[Oracle_Home]/ojsp/lib/servlet.jar
# JSP packages:
wrapper.classpath=[Oracle_Home]/ojsp/lib/ojsp.jar
wrapper.classpath=[Oracle_Home]/ojsp/lib/ojsputil.jar
# XML parser (used for servlet 2.2 web deployment descriptor):
wrapper.classpath=[Oracle_Home]/ojsp/lib/xmlparserv2.jar
# JDBC libraries for Oracle database access (JDK 1.2.x environment):
wrapper.classpath=[Oracle_Home]/ojsp/lib/classes12.zip
# SQLJ translator (optional):
wrapper.classpath=[Oracle_Home]/ojsp/lib/translator.zip
# SQLJ runtime (optional) (for JDK 1.2.x enterprise edition):
wrapper.classpath=[Oracle_Home]/ojsp/lib/runtime12.zip
```

Important:

- If servlet.jar (provided with Oracle JSP for servlet 2.2 versions of javax.servlet.* packages) is in your classpath in a JServ environment, jsdk.jar must precede it.
- You must also ensure that the Oracle JSP container can find javac (or an alternative Java compiler, according to your javaccmd configuration parameter setting). For javac in a JDK 1.1.x environment, the JDK classes.zip file must be in the Web server classpath. For javac in a JDK 1.2.x or later environment, the JDK tools.jar file must be in the Web server classpath.

Now consider an example where you have the following useBean command:

```
<jsp:useBean id="queryBean" class="mybeans.JDBCQueryBean" scope="session" />
```

You can add the following `wrapper.classpath` command to the `jserv.properties` file. (This example happens to be for a Windows NT environment.)

```
wrapper.classpath=D:\Apache\Apache1.3.9\beans\
```

And then `JDBCQueryBean.class` should be located as follows:

```
D:\Apache\Apache1.3.9\beans\mybeans\JDBCQueryBean.class
```

Mapping JSP File Name Extensions for JServ

In a JServ environment, mapping each JSP file name extension to `oracle.jsp.JspServlet`—the JSP front-end servlet for JServ—requires an `ApJServAction` command in either the `jserv.conf` file or the `mod_jserv.conf` file. These configuration files are in the JServ `conf` directory.

(In older versions, you must instead update the `httpd.conf` file in the Apache `conf` directory. In newer versions, the `jserv.conf` or `mod_jserv.conf` file is "included" into `httpd.conf` during execution—look at the `httpd.conf` file to see which one it includes.)

Following is an example (which happens to use UNIX syntax):

```
# Map file name extensions (.sqljsp and .SQLJSP are optional).
ApJServAction .jsp /servlets/oracle.jsp.JspServlet
ApJServAction .JSP /servlets/oracle.jsp.JspServlet
ApJServAction .sqljsp /servlets/oracle.jsp.JspServlet
ApJServAction .SQLJSP /servlets/oracle.jsp.JspServlet
```

The path you use in this command for `oracle.jsp.JspServlet` is not a literal directory path in the file system. The path to specify depends on your JServ servlet configuration—how the servlet zone is mounted, the name of the zone properties file, and the file system directory that is specified as the repository for the servlet. ("Servlet zone" is a JServ term that is similar conceptually to "servlet context".) Consult your JServ documentation for more information.

Important: With the above configurations, Oracle JSP will support page references that use either a `.jsp` file name extension or a `.JSP` file name extension, but the case in the reference must match the actual file name in a case-sensitive environment. If the file name is `file.jsp`, you can reference it that way, but not as `file.JSP`. If the file name is `file.JSP`, you can reference it that way, but not as `file.jsp`. (The same holds true for `.sqljsp` versus `.SQLJSP`.)

Oracle JSP Configuration Parameters

This section describes the configuration parameters supported by the Oracle `JspServlet`.

Configuration Parameters Summary Table

Table 9–2 summarizes the configuration parameters supported by Oracle `JspServlet`, the front-end of the Oracle JSP container. For each parameter, the table notes the following:

- whether it is used during page translation or page execution
- whether it is typically of interest in a development environment, deployment environment, or both
- any equivalent `ojspc` translation options for pages you are pretranslating

  (The `ojspc` utility does not use `JspServlet`.)

Be aware of the following:

- The parameters `debug_mode` and `send_error` are new with Oracle JSP 1.1.2.x releases.
- The parameter `alias_translation` is for use in the JServ environment only.
- The parameter `session_sharing` is for use with `globals.jsa` only (presumably in a servlet 2.0 environment such as JServ).

Notes:

- See "Details of the ojspc Pre-Translation Tool" on page 6-14 for a description of the `ojspc` options.
- The distinction between execution-time and translation-time is not particularly significant in a real-time translation environment, but may be of interest when pre-translating.

Table 9–2 Oracle JSP Configuration Parameters

Parameter	Related ojspc Options	Description	Default	Used in JSP Translation or Execution?	Used in Development or Deployment Environment?
alias_translation **(Apache-specific)**	n/a	boolean; `true` to work around JServ limitations in directory aliasing for JSP page references	false	execution	development and deployment
bypass_source	n/a	boolean; `true` for the Oracle JSP container to ignore `FileNotFound` exceptions on `.jsp` source; uses pre-translated and compiled code when source is not available	false	execution	deployment (also used by JDeveloper)
classpath	-addclasspath (related, but different functionality)	additional classpath entries for Oracle JSP class loading	null (no addl. path)	translation or execution	development and deployment
debug_mode	n/a	boolean; `true` for the Oracle JSP container to print the stack trace when a runtime exception occurs	true	execution	development
developer_mode	n/a	boolean; `false` to *not* check timestamps to see if page retranslation and class reloading is necessary when a page is requested	true	execution	development and deployment

Table 9–2 Oracle JSP Configuration Parameters (Cont.)

Parameter	Related ojspc Options	Description	Default	Used in JSP Translation or Execution?	Used in Development or Deployment Environment?
emit_debuginfo	-debug	boolean; `true` to generate a line map to the original `.jsp` file for debugging during development	false	translation	development
external_resource	-extres	boolean; `true` for the Oracle JSP container to place all static content of the page into a separate Java resource file during translation	false	translation	development and deployment
javaccmd	-noCompile	Java compiler command line—`javac` options, or alternative Java compiler run in a separate JVM (`null` for JDK `javac` with default options)	null	translation	development and deployment
page_repository_root	-srcdir -d	alternative root directory (fully qualified path) for the Oracle JSP container to use in loading and generating JSP pages	null (use default root)	translation or execution	development and deployment
send_error	n/a	boolean; `true` to output standard "404" messages for file-not-found, "500" messages for compilation errors (instead of outputting customized messages)	false	execution	deployment
session_sharing **(for use with globals.jsa in servlet 2.0 environments)**	n/a	boolean; for applications using `globals.jsa`, `true` for JSP session data to be propagated to underlying servlet session	true	execution	development and deployment

Table 9–2 Oracle JSP Configuration Parameters (Cont.)

Parameter	Related ojspc Options	Description	Default	Used in JSP Translation or Execution?	Used in Development or Deployment Environment?
sqljcmd	-S	SQLJ command line—`sqlj` options, or alternative SQLJ translator run in a separate JVM (`null` for the Oracle SQLJ version provided with Oracle9*i*, with default option settings)	null	translation	development and deployment
translate_params	n/a	boolean; `true` to override servlet containers that do not perform multibyte encoding	false	execution	development and deployment
unsafe_reload **(for development only)**	n/a	boolean; `true` to *not* restart the application and sessions whenever a JSP page is retranslated and reloaded	false	execution	development

Configuration Parameter Descriptions

This section describes the Oracle JSP configuration parameters in more detail.

alias_translation (boolean; default: `false`) **(Apache-specific)**

This parameter allows the Oracle JSP container to work around limitations in the way JServ handles directory aliasing. For information about the current limitations, see "Directory Alias Translation" on page 9-23.

You must set `alias_translation` to `true` for `httpd.conf` directory aliasing commands, such as the following example, to work properly in the JServ servlet environment:

```
Alias /icons/ "/apache/apache139/icons/"
```

bypass_source (boolean; default: `false`)

Normally, when a JSP page is requested, the Oracle JSP container will throw a `FileNotFound` exception if it cannot find the corresponding `.jsp` source file, even if it can find the page implementation class. (This is because, by default, the JSP container checks the page source to see if it has been modified since the page implementation class was generated.)

Set this parameter to `true` for the Oracle JSP container to proceed and execute the page implementation class even if it cannot find the page source.

If `bypass_source` is enabled, the JSP container will still check for retranslation if the source is available and is needed. One of the factors in determining whether it is needed is the setting of the `developer_mode` parameter.

Notes:

- The `bypass_source` option is useful in deployment environments that have the generated classes only, not the source. (For related discussion, see "Deployment of Binary Files Only" on page 6-27.)
- Oracle9*i* JDeveloper enables `bypass_source` so that you can translate and run a JSP page before you have saved the JSP source to a file.

---

classpath (fully qualified path; default: `null`)

Use this parameter to add classpath entries to the Oracle JSP default classpath for use during translation, compilation, or execution of JSP pages. For information about the Oracle JSP classpath and class loader, see "Classpath and Class Loader Issues" on page 4-20.

The exact syntax depends on your Web server environment and operating system. See "Setting JSP Parameters in JServ" on page 9-18 for some examples.

Overall, the Oracle JSP container loads classes from its own classpath (including entries from this `classpath` parameter), the system classpath, the Web server classpath, the page repository, and predefined locations relative to the root directory of the JSP application.

Be aware that classes loaded through the path specified in the `classpath` setting path are loaded by the JSP class loader, not the system class loader. During JSP execution, classes loaded by the JSP class loader cannot access (or be accessed by) classes loaded by the system class loader or any other class loader.

Notes:

- Oracle JSP runtime automatic class reloading applies only to classes in the Oracle JSP classpath. This includes paths specified through this `classpath` parameter. (See "Dynamic Class Reloading" on page 4-25 for information about this feature.)
- When you pre-translate pages, the `ojspc -addclasspath` option offers some related, though different, functionality. See "Option Descriptions for ojspc" on page 6-18.

debug_mode (boolean; default: `true`)

Use the default `true` setting of this flag to direct the Oracle JSP container to print a stack trace whenever a runtime exception occurs. Set it to `false` to disable this feature.

developer_mode (boolean; default: `true`)

Set this flag to `false` to instruct the Oracle JSP container to *not* routinely compare the timestamp of the page implementation class to the timestamp of the `.jsp` source file when a page is requested. With `developer_mode` set to `true`, the Oracle JSP container checks every time to see if the source has been modified since the page implementation class was generated. If that is the case, the JSP container retranslates the page. With `developer_mode` set to `false`, the JSP container will check only upon the initial request for the page or application. For subsequent requests, it will simply re-execute the generated page implementation class.

This flag also affects dynamic class reloading for JavaBeans and other support classes called by a JSP page. With `developer_mode` set to `true`, the Oracle JSP container checks to see if such classes have been modified since being loaded by the Oracle JSP class loader.

Oracle generally recommends setting `developer_mode` to `false`, particularly in a deployment environment where code is not likely to change and where performance is a significant issue.

Also see "Oracle JSP Runtime Page and Class Reloading" on page 4-24.

emit_debuginfo (boolean; default: `false`)

Set this flag to true to instruct the Oracle JSP container to generate a line map to the original `.jsp` file for debugging during development. Otherwise, lines will be mapped to the generated page implementation class.

Notes:

- Oracle9*i* JDeveloper enables `emit_debuginfo`.
- When you are pre-translating pages, the `ojspc -debug` option is equivalent. See "Option Descriptions for ojspc" on page 6-18.

---

external_resource (boolean; default: `false`)

Set this flag to true to instruct the Oracle JSP translator to place generated static content (the Java print commands that output static HTML code) into a Java resource file instead of into the service method of the generated page implementation class.

The resource file name is based on the JSP page name, with the `.res` suffix. With Oracle9*i*, translation of `MyPage.jsp`, for example, would create `_MyPage.res` in addition to normal output. The exact implementation may change in future releases, however.

The resource file is placed in the same directory as generated class files.

If there is a lot of static content in a page, this technique will speed translation and may speed execution of the page. In extreme cases, it may even prevent the service method from exceeding the 64K method size limit imposed by the Java VM. For more information, see "Workarounds for Large Static Content in JSP Pages" on page 4-10.

Note: When you are pre-translating pages, the `ojspc -extres` option is equivalent.

---

javaccmd (compiler executable; default: `null`)

This parameter is useful in any of the following circumstances:

- if you want to set `javac` command-line options (although default settings are typically sufficient)

- if you want to use a compiler other than `javac` (optionally including command-line options)
- if you want to run the Java compiler in a separate process from the Oracle JSP container

Specifying an alternative compiler results in the Oracle JSP container spawning that executable as a separate process in a separate JVM, instead of spawning the JDK default compiler within the same JVM in which the Oracle JSP container is running. You can fully specify the path for the executable, or specify only the executable and let the Oracle JSP container look for it in the system path.

Following is an example of a `javaccmd` setting to enable the `javac -verbose` option:

```
javaccmd=javac -verbose
```

The exact syntax depends on your servlet environment. See "Setting JSP Parameters in JServ" on page 9-18.

Notes:

- The specified Java compiler must be installed in the classpath and any front-end utility (if applicable) must be installed in the system path.
- When you are pre-translating pages, the `ojspc -noCompile` option allows similar functionality. It results in no compilation by `javac`, so you can compile the translated classes manually using your desired compiler. See "Option Descriptions for ojspc" on page 6-18.

page_repository_root (fully qualified directory path; default: `null`)

The Oracle JSP container uses the Web server document repository to generate or load translated JSP pages. By default, in an on-demand translation scenario, the root directory is the Web server doc root directory (for JServ) or the servlet context root directory of the application the page belongs to. JSP page source is in the root directory or some subdirectory. Generated files are written to a `_pages` subdirectory or some corresponding subdirectory.

Set the `page_repository_root` option to instruct the Oracle JSP container to use a different root directory.

For information about file locations relative to the root directory and `_pages` subdirectory, see "Oracle JSP Translator Output File Locations" on page 6-8.

Notes:

- The specified directory, `_pages` subdirectory, and any appropriate subdirectories under these are created automatically if they do not already exist.
- When you are pre-translating pages, the `ojspc` options `-srcdir` and `-d` provide related functionality. See "Option Descriptions for ojspc" on page 6-18.

send_error (boolean; default: `false`)

Set this flag to `true` to direct the Oracle JSP container to output generic "404" messages for file-not-found conditions, and generic "500" messages for compilation errors.

This is as opposed to outputting customized messages that provide more information (such as the name of the file not found). Some environments, such as JServ, do not allow output of a customized message if a "404" or "500" message is output.

session_sharing (boolean; default: `true`) **(for use with globals.jsa)**

When a `globals.jsa` file is used for an application, presumably in a servlet 2.0 environment, each JSP page uses a distinct JSP session wrapper attached to the single overall servlet session object provided by the servlet container.

In this situation, the default `true` setting of the `session_sharing` parameter results in JSP session data being propagated to the underlying servlet session. This allows servlets in the application to access the session data of JSP pages in the application.

If `session_sharing` is `false` (which parallels standard behavior in most JSP implementations), JSP session data is not propagated to the servlet session. As a result, application servlets would not be able to access JSP session data.

This parameter is meaningless if `globals.jsa` is not used. For information about `globals.jsa`, see "Oracle JSP Application and Session Support for JServ" on page 9-26.

sqljcmd (SQLJ translator executable and options; default: `null`)

This parameter is useful in any of the following circumstances:

- if you want to set one or more SQLJ command-line options

  (You can set multiple SQLJ options in the `sqljcmd` setting.)

- if you want to use a different SQLJ translator (or at least a different version) than the one provided with Oracle9*i*
- if you want to run SQLJ in a separate process from the Oracle JSP container

Specifying a SQLJ translator executable results in the JSP container spawning that executable as a separate process in a separate JVM, instead of spawning the default SQLJ translator within the same JVM in which the JSP container is running.

You can fully specify the path for the executable, or specify only the executable and let the JSP container look for it in the system path.

Following is an example of a `sqljcmd` setting to log into `scott/tiger` for online semantics-checking and to generate ISO standard SQLJ code:

```
sqljcmd=sqlj -user=scott/tiger -codegen=iso
```

(The exact syntax depends on your servlet environment. See "Setting JSP Parameters in JServ" on page 9-18.)

Notes:

- Appropriate SQLJ files must be in the classpath, and any front-end utility (such as `sqlj` in the example) must be in the system path. (For Oracle SQLJ, the `translator` ZIP or JAR file and the appropriate SQLJ runtime ZIP or JAR file must be in the classpath. See "Required and Optional Files for Oracle JSP" on page 9-2.)
- Presumably the great majority of JSP developers will use Oracle SQLJ (as opposed to some other SQLJ product) if they use SQLJ code in their JSP pages; however, this option is useful if you want to use a different Oracle SQLJ version (for example, one intended for use with Oracle JDBC 8.0.x/7.3.x drivers instead of Oracle9*i* drivers) or if you want to set SQLJ options.
- When you are pre-translating pages, the `ojspc -S` option provides related functionality. See "Option Descriptions for ojspc" on page 6-18.

---

translate_params (boolean; default: `false`)

Note: Beginning with Oracle JSP 1.1.2.x releases, it is preferable to use the `PublicUtil.setReqCharacterEncoding()` method instead of using the `translate_params` parameter. See "The setReqCharacterEncoding() Method" on page 8-5.

Set this flag to `true` to override servlet containers that do not encode multibyte (globalization support) request parameters or bean property settings. With this setting, the Oracle JSP container encodes request parameters and bean property settings. Otherwise, the JSP container returns the parameters from the servlet container unchanged.

For more information about the functionality and use of `translate_params`, including situations where it should not be used, see "Oracle JSP Extended Support for Multibyte Parameter Encoding" on page 8-5.

unsafe_reload (boolean; default: `false`) **(for development only)**

By default, the Oracle JSP container restarts the application and sessions whenever a JSP page is dynamically retranslated and reloaded (which occurs when the JSP translator finds a `.jsp` source file with a more recent timestamp than the corresponding page implementation class).

Set this parameter to `true` to instruct the JSP container *not* to restart the application after dynamic retranslations and reloads. This avoids having existing sessions become invalid.

For a given JSP page, this parameter has no effect after the initial request for the page if `developer_mode` is set to `false` (in which case the JSP container never retranslates after the initial request).

Important: This parameter is intended for developers only and is *not* recommended for deployment environments.

Setting JSP Parameters in JServ

Each Web application in a JServ environment has its own properties file, known as a *zone properties file*. In Apache terminology, a *zone* is essentially the same as a servlet context.

The name of the zone properties file depends on how you mount the zone. (See the JServ documentation for information about zones and mounting.)

To set JSP configuration parameters in a JServ environment, set the `JspServlet initArgs` property in the application zone properties file, as in the following example (which happens to use UNIX syntax):

```
servlet.oracle.jsp.JspServlet.initArgs=developer_mode=false,
sqljcmd=sqlj -user=scott/tiger -codegen=iso,classpath=/mydir/myapp.jar
```

(This is a single wraparound line.)

The servlet path, `servlet.oracle.jsp.JspServlet`, also depends on how you mount the zone. It does not represent a literal directory path.

Be aware of the following:

- The effects of multiple `initArgs` commands are cumulative and overriding. For example, consider the following two commands (in order):

  ```
  servlet.oracle.jsp.JspServlet.initArgs=foo1=val1,foo2=val2
  servlet.oracle.jsp.JspServlet.initArgs=foo1=val3
  ```

This combination is equivalent to the following single command:

```
servlet.oracle.jsp.JspServlet.initArgs=foo1=val3,foo2=val2
```

In the first two commands, the `val3` value overrides the `val1` value for `foo1`, but does not affect the `foo2` setting.

- Because `initArgs` parameters are comma-separated, there can be no commas within a parameter setting. Spaces and other special characters (such as "=" in this example) do not cause a problem, however.

Considerations for JServ Servlet Environments

There are special considerations in running the Oracle JSP container in JServ-based platforms, because this is a servlet 2.0 environment. The servlet 2.0 specification lacked support for some significant features that are available in servlet 2.1 and 2.2 environments.

For information about how to configure a JServ environment for the Oracle JSP container, see the following sections:

- "Adding Files to the JServ Web Server Classpath" on page 9-4
- "Mapping JSP File Name Extensions for JServ" on page 9-6
- "Setting JSP Parameters in JServ" on page 9-18

This section discusses the following Apache-specific considerations:

- Dynamic Includes and Forwards in JServ
- Application Framework for JServ
- JSP and Servlet Session Sharing
- Directory Alias Translation

Dynamic Includes and Forwards in JServ

JSP dynamic includes (the `jsp:include` action) and forwards (the `jsp:forward` action) rely on request dispatcher functionality that is present in servlet 2.1 and 2.2 environments but not in servlet 2.0 environments.

The Oracle JSP container, however, provides extended functionality to allow dynamic includes and forwards from one JSP page to another JSP page or to a static HTML file in JServ and other servlet 2.0 environments.

This Oracle JSP functionality for servlet 2.0 environments does not, however, allow dynamic forwards or includes to servlets. (Servlet execution is controlled by the JServ or other servlet container, not the JSP container.)

If you want to include or forward to a servlet in JServ, however, you can create a JSP page that acts as a wrapper for the servlet.

The following example shows a servlet, and a JSP page that acts as a wrapper for that servlet. In a JServ environment, you can effectively include or forward to the servlet by including or forwarding to the JSP wrapper page.

Servlet Code Presume that you want to include or forward to the following servlet:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class TestServlet extends HttpServlet {

    public void init(ServletConfig config) throws ServletException
    {
      super.init(config);
      System.out.println("initialized");
    }

    public void destroy()
    {
      System.out.println("destroyed");
    }

    public void service
        (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML><BODY>");
        out.println("TestServlet Testing");
        out.println("<H3>The local time is: "+ new java.util.Date());
        out.println("</BODY></HTML>");
    }
}
```

JSP Wrapper Page Code You can create the following JSP wrapper (`wrapper.jsp`) for the preceding servlet.

```
<%-- wrapper.jsp--wraps TestServlet for JSP include/forward --%>
<%@ page isThreadSafe="true" import="TestServlet" %>
<%!
  TestServlet s=null;
  public void jspInit() {
    s=new TestServlet();
    try {
      s.init(this.getServletConfig());
    } catch (ServletException se)
    {
      s=null;
    }
  }
  public void jspDestroy() {
    s.destroy();
  }
%>
<% s.service(request,response); %>
```

Including or forwarding to `wrapper.jsp` in a servlet 2.0 environment has the same effect as directly including or forwarding to `TestServlet` in a servlet 2.1 or 2.2 environment.

Note Regarding Dynamic Includes and Forwards

- Whether to set `isThreadSafe` to `true` or `false` in the wrapper JSP page depends on whether the original servlet is thread-safe.
- As an alternative to using a wrapper JSP page for this situation, you can add HTTP client code to the original JSP page (the one from which the `include` or `forward` is to occur). You can use an instance of the standard `java.net.URL` class to create an HTTP request from the original JSP page to the servlet. (Note that you cannot share session data or security credentials in this scenario.) Alternatively, you can use the `HTTPClient` class from Innovation GmbH. The Oracle JVM provides a modified version of this class that supports SSL, directly or through a proxy, when you use `https://` for the URL. (See `http://www.innovation.ch/java/HTTPClient` for general information about this class. Click "Getting Started" for information that includes how to replace the JDK HTTP client with the `HTTPClient` class.) Details of these alternatives are outside the scope of this document, however, and this approach is generally not recommended.

Application Framework for JServ

The servlet 2.0 specification does not provide the full servlet context framework for application support that is provided in later specifications.

For servlet 2.0 environments, including JServ, the Oracle JSP container supplies its own application framework using a file, `globals.jsa`, that you can use as an application marker.

For more information, see "Distinct Applications and Sessions Through globals.jsa" on page 9-27.

JSP and Servlet Session Sharing

To share HTTP session information between JSP pages and servlets in a JServ environment, you must configure your environment so that `oracle.jsp.JspServlet` (the servlet that acts as the front-end of the Oracle JSP container) is in the same zone as the servlet or servlets with which you want your JSP pages to share a session. Consult your Apache documentation for more information.

To verify proper zone setup, some browsers allow you to enable a warning for cookies. In an Apache environment, the cookie name includes the zone name.

Additionally, for applications that use a `globals.jsa` file, the JSP configuration parameter `session_sharing` should be set to `true` (the default) for JSP session data to be accessible to servlets. See these sections for related information:

- "Oracle JSP Application and Session Support for JServ" on page 9-26
- "Oracle JSP Configuration Parameters" on page 9-7
- "Setting JSP Parameters in JServ" on page 9-18

Directory Alias Translation

Apache supports directory aliasing by allowing you to create a "virtual directory" through an `Alias` command in the `httpd.conf` configuration file. This allows Web documents to be placed outside the default doc root directory.

Consider the following sample `httpd.conf` entry:

```
Alias /icons/ "/apache/apache139/icons/"
```

This command should result in `icons` being usable as an alias for the `/apache/apache139/icons/` path. In this way, for example, the file `/apache/apache139/icons/art.gif`, could be accessed by the following URL:

```
http://host[:port]/icons/art.gif
```

Currently, however, this functionality does not work properly for servlets and JSP pages, because the JServ `getRealPath()` method returns an incorrect value when processing a file under an alias directory.

Oracle provides an Apache-specific JSP configuration parameter, `alias_translation`, that works around this limitation when you set `alias_translation` to `true` (the default setting is `false`).

Be aware that setting `alias_translation=true` also results in the alias directory becoming the application root. Therefore, in a dynamic `include` or `forward` command where the target file name starts with "/", the expected target file location will be relative to the alias directory.

Consider the following example, which results in all JSP and HTML files under `/private/foo` being effectively under the application `/mytest`:

```
Alias /mytest/ "/private/foo/"
```

Also assume there is a JSP page located as follows:

```
/private/foo/xxx.jsp
```

The following dynamic `include` command will work, because `xxx.jsp` is directly below the aliased directory, `/private/foo`, which is effectively the application root:

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

JSP pages in other applications or in the general doc root cannot forward to or include JSP pages or HTML files under the `/mytest` application. It is only possible to forward to or include pages or HTML files within the same application (per the servlet 2.2 specification).

Notes:

- An implicit application is created for the Web server document root and each aliasing root.
- For information about how to set JSP configuration parameters in a JServ environment, see "Setting JSP Parameters in JServ" on page 9-18.

Also be aware that there are issues when two aliases begin with the same partial directory path. Consider the following two aliases as an example:

```
Alias /foo/bar1 "/path/to/my/dir/x/bar1"
Alias /foo/bar2 "/path/to/my/dir/y/bar2"
```

An initial request for `/foo/bar1/bar1.jsp` will work, but a subsequent request for `/foo/bar2/bar2.jsp` will incorrectly look in `/path/to/my/dir/x` for `bar2.jsp`, and will fail with a `FileNotFound` exception. This is due to further limitations with the JServ `getRealPath()` implementation, which returns incorrect information. There are two workarounds for this situation:

- Have only one alias, with real directories underneath:

  ```
  Alias /foo  "/path/to/my/dir"
  ```

  Here the `bar1` and `bar2` directories would physically exist as `/path/to/my/dir/bar1` and `/path/to/my/dir/bar2`, and there would not be a problem.

or:

- Have more than one alias, but do not have common directory names:

  ```
  Alias /foo/bar1  "/path/to/my/dir/x_bar1"
  Alias /foo/bar2  "/path/to/my/dir/y_bar2"
  ```

  Note that the physical directories do not have the same name as the alias directories (unlike the problematic example above, where alias directories and physical directories shared `bar1` and `bar2` in common).

Oracle JSP Application and Session Support for JServ

The Oracle JSP container defines a file, `globals.jsa`, as a mechanism for implementing the JSP specification in a servlet 2.0 environment. Web applications and servlet contexts were not fully defined in the servlet 2.0 specification.

This section discusses the `globals.jsa` mechanism and covers the following topics:

- Overview of globals.jsa Functionality
- Overview of globals.jsa Syntax and Semantics
- The globals.jsa Event Handlers
- Global Declarations and Directives

For sample applications, see "Samples Using globals.jsa for Servlet 2.0 Environments" on page 9-39.

Important: Use all lowercase for the `globals.jsa` file name. Mixed case works in a non-case-sensitive environment, but makes it difficult to diagnose resulting problems if you port the pages to a case-sensitive environment.

Overview of globals.jsa Functionality

Within any single Java virtual machine, you can use a `globals.jsa` file for each application (or, equivalently, for each servlet context). This file supports the concept of Web applications in the following areas:

- application deployment—through its role as an application location marker to define an application root
- distinct applications and sessions—through its use by the Oracle JSP container in providing distinct servlet context and session objects for each application
- application lifecycle management—through start and end events for sessions and applications

The `globals.jsa` file also provides a vehicle for global Java declarations and JSP directives across all JSP pages of an application.

Application Deployment through globals.jsa

To deploy an Oracle JSP application that does not incorporate servlets, copy the directory structure into the Web server and create a file called `globals.jsa` to place at the application root directory.

The `globals.jsa` file can be of zero size. The Oracle JSP container will locate it, and its presence in a directory defines that directory (as mapped from the URL virtual path) as the root directory of the application.

The JSP container also defines default locations for JSP application resources. For example, application beans and classes in the application-relative `/WEB-INF/classes` and `/WEB-INF/lib` directories (servlet 2.2 or higher) will automatically be loaded by the Oracle JSP classloader without the need for specific configuration.

Notes: For an application that *does* incorporate servlets, especially in a servlet environment preceding the servlet 2.2 specification, manual configuration is required as with any servlet deployment. For servlets in a servlet 2.2 or higher environment, you can include the necessary configuration in the standard `web.xml` deployment descriptor.

Distinct Applications and Sessions Through globals.jsa

The servlet 2.0 specification does not have a clearly defined concept of a Web application and there is no defined relationship between servlet contexts and applications, as there is in later servlet specifications. In a servlet 2.0 environment such as JServ, there is only one servlet context object per JVM. A servlet 2.0 environment also has only one session object.

The `globals.jsa` file, however, provides support for multiple applications and multiple sessions in a Web server, particularly for use in a servlet 2.0 environment.

Where a distinct servlet context object would not otherwise be available for each application, the presence of a `globals.jsa` file for an application allows the Oracle JSP container to provide the application with a distinct `ServletContext` object.

Additionally, where there would otherwise be only one session object (with either one servlet context or across multiple servlet contexts), the presence of a `globals.jsa` file allows the Oracle JSP container to provide a proxy `HttpSession` object to the application. This prevents the possibility of session variable-name collisions with other applications, although unfortunately it cannot protect application data from being inspected or modified by other applications. This is because `HttpSession` objects must rely on the underlying servlet session environment for some of their functionality.

Application and Session Lifecycle Management Through globals.jsa

An application must be notified when a significant state transition occurs. For example, applications often want to acquire resources when an HTTP session begins and release resources when the session ends, or restore or save persistent data when the application itself is started or terminated.

In standard servlet and JSP technology, however, only session-based events are supported.

For applications that use a `globals.jsa` file, the Oracle JSP container extends this functionality with the following four events:

- `session_OnStart`
- `session_OnEnd`
- `application_OnStart`
- `application_OnEnd`

You can write event handlers in the `globals.jsa` file for any of these events that the server should respond to.

The `session_OnStart` event and `session_OnEnd` event are triggered at the beginning and end of an HTTP session, respectively.

The `application_OnStart` event is triggered for any application by the first request for that application within any single JVM. The `application_OnEnd` event is triggered when the Oracle JSP container unloads an application.

For more information, see "The globals.jsa Event Handlers" on page 9-31.

Overview of globals.jsa Syntax and Semantics

This section is an overview of general syntax and semantics for a `globals.jsa` file.

Each event block in a `globals.jsa` file—a `session_OnStart` block, a `session_OnEnd` block, an `application_OnStart` block, or an `application_OnEnd` block—has an event start tag, an event end tag, and a body (everything between the start and end tags) that includes the event-handler code.

The following example shows this pattern:

```
<event:session_OnStart>
   <% This scriptlet contains the implementation of the event handler %>
</event:session_OnStart>
```

The body of an event block can contain any valid JSP tags—standard tags as well as tags defined in a custom tag library.

The scope of any JSP tag in an event block, however, is limited to only that block. For example, a bean that is declared in a `jsp:useBean` tag within one event block must be redeclared in any other event block that uses it. You can avoid this restriction, however, through the `globals.jsa` global declaration mechanism—see "Global Declarations and Directives" on page 9-36.

For details about each of the four event handlers, see "The globals.jsa Event Handlers" on page 9-31.

Important: Static text as used in a regular JSP page can reside in a `session_OnStart` block only. Event blocks for `session_OnEnd`, `application_OnStart`, and `application_OnEnd` can contain only Java scriptlets.

JSP implicit objects are available in `globals.jsa` event blocks as follows:

- The `application_OnStart` block has access to the `application` object.
- The `application_OnEnd` block has access to the `application` object.
- The `session_OnStart` block has access to the `application`, `session`, `request`, `response`, `page`, and `out` objects.
- The `session_OnEnd` block has access to the `application` and `session` objects.

Example of a Complete globals.jsa File This example shows you a complete `globals.jsa` file, using all four event handlers.

```
<event:application_OnStart>

   <%-- Initializes counts to zero --%>
   <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />
```

```
</event:application_OnStart>

<event:application_OnEnd>

   <%-- Acquire beans --%>
   <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <% application.log("The number of page hits were: " + pageCount.getValue() ); %>
   <% application.log("The number of client sessions were: " + sessionCount.getValue() ); %>

</event:application_OnEnd>

<event:session_OnStart>

   <%-- Acquire beans --%>
   <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <%
      sessionCount.setValue(sessionCount.getValue() + 1);
      activeSessions.setValue(activeSessions.getValue() + 1);
   %>
   <br>
   Starting session #: <%=sessionCount.getValue() %> <br>
   There are currently <b> <%= activeSessions.getValue() %> </b> active sessions <p>

</event:session_OnStart>

<event:session_OnEnd>

   <%-- Acquire beans --%>
   <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <%
      activeSessions.setValue(activeSessions.getValue() - 1);
   %>

</event:session_OnEnd>
```

The globals.jsa Event Handlers

This section provides details about each of the four `globals.jsa` event handlers.

application_OnStart

The `application_OnStart` block has the following general syntax:

```
<event:application_OnStart>
   <% This scriptlet contains the implementation of the event handler %>
</event:application_OnStart>
```

The body of the `application_OnStart` event handler is executed when the Oracle JSP container loads the first JSP page in the application. This usually occurs when the first HTTP request is made to any page in the application, from any client. Applications use this event to initialize application-wide resources, such as a database connection pool or data read from a persistent repository into application objects.

The event handler must contain only JSP tags (including custom tags) and white space—it cannot contain static text.

Errors that occur in this event handler but are not processed in the event-handler code are automatically trapped by the Oracle JSP container and logged using the servlet context of the application. Event handling then proceeds as if no error had occurred.

Example: application_OnStart The following `application_OnStart` example is from the "A globals.jsa Example for Application Events: lotto.jsp" on page 9-39. In this example, the generated lottery numbers for a particular user are cached for an entire day. If the user re-requests the picks, he or she gets the same set of numbers. The cache is recycled once a day, giving each user a new set of picks. To function as intended, the lotto application must make the cache persistent when the application is being shut down, and must refresh the cache when the application is reactivated.

The `application_OnStart` event handler reads the cache from the `lotto.che` file.

```
<event:application_OnStart>

<%
        Calendar today = Calendar.getInstance();
        application.setAttribute("today", today);
        try {
                FileInputStream fis = new FileInputStream
                            (application.getRealPath("/")+File.separator+"lotto.che");
                ObjectInputStream ois = new ObjectInputStream(fis);
                Calendar cacheDay = (Calendar) ois.readObject();
                if (cacheDay.get(Calendar.DAY_OF_YEAR) == today.get(Calendar.DAY_OF_YEAR)) {
                        cachedNumbers = (Hashtable) ois.readObject();
                        application.setAttribute("cachedNumbers", cachedNumbers);
                }
                ois.close();
        } catch (Exception theE) {
                // catch all -- can't use persistent data
        }
%>

</event:application_OnStart>
```

application_OnEnd

The `application_OnEnd` block has the following general syntax:

```
<event:application_OnEnd>
   <% This scriptlet contains the implementation of the event handler %>
</event:application_OnEnd>
```

The body of the `application_OnEnd` event handler is executed when the Oracle JSP container unloads the JSP application. Unloading occurs whenever a previously loaded page is reloaded after on-demand dynamic re-translation (unless the JSP `unsafe_reload` configuration parameter is enabled), or when the JSP container, which itself is a servlet, is terminated by having its `destroy()` method called by the underlying servlet container. Applications use the `application_OnEnd` event to clean up application level resources or to write application state to a persistent store.

The event handler must contain only JSP tags (including custom tags) and white space—it cannot contain static text.

Errors that occur in this event handler but are not processed in the event-handler code are automatically trapped by the Oracle JSP container and logged using the servlet context of the application. Event handling then proceeds as if no error had occurred.

Example: application_OnEnd The following `application_OnEnd` example is from the "A globals.jsa Example for Application Events: lotto.jsp" on page 9-39. In this event handler, the cache is written to file `lotto.che` before the application is terminated.

```
<event:application_OnEnd>

<%
        Calendar now = Calendar.getInstance();
        Calendar today = (Calendar) application.getAttribute("today");
        if (cachedNumbers.isEmpty() ||
                   now.get(Calendar.DAY_OF_YEAR) > today.get(Calendar.DAY_OF_YEAR)) {
                File f = new File(application.getRealPath("/")+File.separator+"lotto.che");
                if (f.exists()) f.delete();
                return;
        }

        try {
                FileOutputStream fos = new FileOutputStream
                            (application.getRealPath("/")+File.separator+"lotto.che");
                ObjectOutputStream oos = new ObjectOutputStream(fos);
                oos.writeObject(today);
                oos.writeObject(cachedNumbers);
                oos.close();
        } catch (Exception theE) {
                // catch all -- can't use persistent data
        }
%>

</event:application_OnEnd>
```

session_OnStart

The `session_OnStart` block has the following general syntax:

```
<event:session_OnStart>
   <% This scriptlet contains the implementation of the event handler %>
   Optional static text...
</event:session_OnStart>
```

The body of the `session_OnStart` event handler is executed when the Oracle JSP container creates a new session in response to a JSP page request. This occurs on a per client basis, whenever the first request is received for a session-enabled JSP page in an application.

Applications might use this event for the following purposes:

- to initialize resources tied to a particular client
- to control where a client starts in an application

Because the implicit `out` object is available to `session_OnStart`, this is the only `globals.jsa` event handler that can contain static text in addition to JSP tags.

The `session_OnStart` event handler is called before the code of the JSP page is executed. As a result, output from `session_OnStart` precedes any output from the page.

The `session_OnStart` event handler and the JSP page that triggered the event share the same `out` stream. The buffer size of this stream is controlled by the buffer size of the JSP page. The `session_OnStart` event handler does not automatically flush the stream to the browser—the stream is flushed according to general JSP rules. Headers can still be written in JSP pages that trigger the `session_OnStart` event.

Errors that occur in this event handler but are not processed in the event-handler code are automatically trapped by the Oracle JSP container and logged using the servlet context of the application. Event handling then proceeds as if no error had occurred.

Example: session_OnStart The following example makes sure that each new session starts on the initial page (`index.jsp`) of the application.

```
<event:session_OnStart>

   <% if (!page.equals("index.jsp")) { %>
         <jsp:forward page="index.jsp" />
   <% } %>

</event:session_OnStart>
```

session_OnEnd

The `session_OnEnd` block has the following general syntax:

```
<event:session_OnEnd>
   <% This scriptlet contains the implementation of the event handler %>
</event:session_OnEnd>
```

The body of the `session_OnEnd` event handler is executed when the Oracle JSP container invalidates an existing session. This occurs in either of the following circumstances:

- The application invalidates the session by calling the `session.invalidate()` method.
- The session expires ("times out") on the server.

Applications use this event to release client resources.

The event handler must contain only JSP tags (including tag library tags) and white space—it cannot contain static text.

Errors that occur in this event handler but are not processed in the event-handler code are automatically trapped by the Oracle JSP container and logged using the servlet context of the application. Event handling then proceeds as if no error had occurred.

Example: session_OnEnd The following example decrements the "active session" count when a session is terminated.

```
<event:session_OnEnd>

     <%-- Acquire beans --%>
     <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

     <%
         activeSessions.setValue(activeSessions.getValue() - 1);
     %>

 </event:session_OnEnd>
```

Global Declarations and Directives

In addition to holding event handlers, a `globals.jsa` file can be used to globally declare directives and objects for the JSP application. You can include JSP directives, JSP declarations, JSP comments, and JSP tags that have a `scope` parameter (such as `jsp:useBean`).

This section covers the following topics:

- Global JSP Directives
- Declarations in globals.jsa
- Global JavaBeans
- Structure of globals.jsa
- Global Declarations and Directives Example

Global JSP Directives

Directives used within a `globals.jsa` file serve a dual purpose:

- They declare the information that is required to process the `globals.jsa` file itself.
- They establish default values for succeeding pages.

A directive in a `globals.jsa` file becomes an implicit directive for all JSP pages in the application, although a `globals.jsa` directive can be overwritten for any particular page.

A `globals.jsa` directive is overwritten in a JSP page on an attribute-by-attribute basis. If a `globals.jsa` file has the following directive:

```
<%@ page import="java.util.*" bufferSize="10kb" %>
```

and a JSP page has the following directive:

```
<%@page bufferSize="20kb" %>
```

then this would be equivalent to the page having the following directive:

```
<%@ page import="java.util.*" bufferSize="20kb" %>
```

Declarations in globals.jsa

If you want to declare a method or data member to be shared across any of the event handlers in a `globals.jsa` file, use a JSP `<%! ... %>` declaration within the `globals.jsa` file.

Note that JSP pages in the application do not have access to these declarations, so you cannot use this mechanism to implement an application library. Declaration support is provided in the `globals.jsa` file for common functions to be shared across event handlers.

Global JavaBeans

Probably the most common elements declared in `globals.jsa` files are global objects. Objects declared in a `globals.jsa` file become part of the implicit object environment of the `globals.jsa` event handlers and all the JSP pages in the application.

An object declared in a `globals.jsa` file (such as by a `jsp:useBean` statement) does not need to be redeclared in any of the individual JSP pages of the application.

You can declare a global object using any JSP tag or extension that has a `scope` parameter, such as `jsp:useBean` or `jml:useVariable`. Globally declared objects must be of either `session` or `application` scope (not `page` or `request` scope).

Nested tags are supported. Thus, a `jsp:setProperty` command can be nested in a `jsp:useBean` declaration. (A translation error occurs if `jsp:setProperty` is used outside a `jsp:useBean` declaration.)

Structure of globals.jsa

When a global object is used in a `globals.jsa` event handler, the position of its declaration is important. Only those objects that are declared before a particular event handler are added as implicit objects to that event handler. For this reason, developers are advised to structure their `globals.jsa` file in the following sequence:

1. global directives
2. global objects
3. event handlers
4. `globals.jsa` declarations

Global Declarations and Directives Example

The sample `globals.jsa` file below accomplishes the following:

- It defines the JML tag library (in this case, the compile-time implementation) for the `globals.jsa` file, as well as for all subsequent pages.

  By including the `taglib` directive in the `globals.jsa` file, the directive does not have to be included in any of the individual JSP pages of the application.

- It declares three application variables for use by all pages (in the `jsp:useBean` statements).

For an additional example of using `globals.jsa` for global declarations, see "A globals.jsa Example for Global Declarations: index2.jsp" on page 9-45.

```
<%-- Directives at the top --%>

   <%@ taglib uri="oracle.jsp.parse.OpenJspRegisterLib" prefix="jml" %>

<%-- Declare global objects here --%>

   <%-- Initializes counts to zero --%>
   <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
   <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

<%-- Application lifecycle event handlers go here --%>

   <event:application_OnStart>
      <% This scriptlet contains the implementation of the event handler %>
   </event:application_OnStart>

   <event:application_OnEnd>
      <% This scriptlet contains the implementation of the event handler %>
   </event:application_OnEnd>

   <event:session_OnStart>
      <% This scriptlet contains the implementation of the event handler %>
   </event:session_OnStart>

   <event:session_OnEnd>
      <% This scriptlet contains the implementation of the event handler %>
   </event:session_OnEnd>

<%-- Declarations used by the event handlers go here --%>
```

Samples Using globals.jsa for Servlet 2.0 Environments

This section has examples of how the Oracle `globals.jsa` mechanism can be used in servlet 2.0 environments to provide an application framework and application-based and session-based event handling. The following examples are provided:

- A globals.jsa Example for Application Events: lotto.jsp
- A globals.jsa Example for Application and Session Events: index1.jsp
- A globals.jsa Example for Global Declarations: index2.jsp

For information about `globals.jsa` usage, see "Oracle JSP Application and Session Support for JServ" on page 9-26.

Note: The examples in this section base some of their functionality on application shutdown. Many servers do not allow an application to be shut down manually. In this case, `globals.jsa` cannot function as an application marker. However, you can cause the application to be automatically shut down and restarted (presuming `developer_mode` is set to `true`) by updating either the `lotto.jsp` source or the `globals.jsa` file. (The Oracle JSP container always terminates a running application before retranslating and reloading an active page.)

A globals.jsa Example for Application Events: lotto.jsp

This sample illustrates `globals.jsa` event handling through the `application_OnStart` and `application_OnEnd` event handlers. In this sample, numbers are cached on a per-user basis for the duration of the day. As a result, only one set of numbers is ever presented to a user for a given lottery drawing. In this sample, a user is identified by their IP address.

Code has been written for `application_OnStart` and `application_OnEnd` to make the cache persistent across application shutdowns. The sample writes the cached data to a file as it is being terminated and reads from the file as it is being restarted (presuming the server is restarted the same day that the cache was written).

globals.jsa File for lotto.jsp

```
<%@ page import="java.util.*, oracle.jsp.jml.*" %>

<jsp:useBean id = "cachedNumbers" class = "java.util.Hashtable" scope = "application" />

<event:application_OnStart>

<%
        Calendar today = Calendar.getInstance();
        application.setAttribute("today", today);
        try {
                FileInputStream fis = new FileInputStream
                            (application.getRealPath("/")+File.separator+"lotto.che");
                ObjectInputStream ois = new ObjectInputStream(fis);
                Calendar cacheDay = (Calendar) ois.readObject();
                if (cacheDay.get(Calendar.DAY_OF_YEAR) == today.get(Calendar.DAY_OF_YEAR)) {
                        cachedNumbers = (Hashtable) ois.readObject();
                        application.setAttribute("cachedNumbers", cachedNumbers);
                }
                ois.close();
        } catch (Exception theE) {
                // catch all -- can't use persistent data
        }
%>

</event:application_OnStart>

<event:application_OnEnd>

<%
        Calendar now = Calendar.getInstance();
        Calendar today = (Calendar) application.getAttribute("today");
        if (cachedNumbers.isEmpty() ||
                   now.get(Calendar.DAY_OF_YEAR) > today.get(Calendar.DAY_OF_YEAR)) {
                File f = new File(application.getRealPath("/")+File.separator+"lotto.che");
                if (f.exists()) f.delete();
                return;
        }

        try {
                FileOutputStream fos = new FileOutputStream
                            (application.getRealPath("/")+File.separator+"lotto.che");
                ObjectOutputStream oos = new ObjectOutputStream(fos);
                oos.writeObject(today);
                oos.writeObject(cachedNumbers);
                oos.close();
        } catch (Exception theE) {
                // catch all -- can't use persistent data
        }
%>

</event:application_OnEnd>
```

lotto.jsp Source

```
<%@ page session = "false" %>
<jsp:useBean id = "picker" class = "oracle.jsp.sample.lottery.LottoPicker" scope = "page" />

<HTML>
<HEAD><TITLE>Lotto Number Generator</TITLE></HEAD>
<BODY BACKGROUND="images/cream.jpg" BGCOLOR="#FFFFFF">
<H1 ALIGN="CENTER"></H1>

<BR>

<!-- <H1 ALIGN="CENTER"> IP: <%= request.getRemoteAddr() %> <BR> -->
<H1 ALIGN="CENTER">Your Specially Picked</H1>
<P ALIGN="CENTER"><IMG SRC="images/winningnumbers.gif" WIDTH="450" HEIGHT="69" ALIGN="BOTTOM"
BORDER="0"></P>
<P>

<P ALIGN="CENTER">
<TABLE ALIGN="CENTER" BORDER="0" CELLPADDING="0" CELLSPACING="0">
<TR>
<%
        int[] picks;
        String identity = request.getRemoteAddr();

        // Make sure its not tomorrow
        Calendar now = Calendar.getInstance();
        Calendar today = (Calendar) application.getAttribute("today");
        if (now.get(Calendar.DAY_OF_YEAR) > today.get(Calendar.DAY_OF_YEAR)) {
                System.out.println("New day....");
                cachedNumbers.clear();
                today = now;
                application.setAttribute("today", today);
        }

        synchronized (cachedNumbers) {
                if ((picks = (int []) cachedNumbers.get(identity)) == null) {
                        picks = picker.getPicks();
                        cachedNumbers.put(identity, picks);
                }
        }
        for (int i = 0; i < picks.length; i++) {
%>
     <TD>
     <IMG SRC="images/ball<%= picks[i] %>.gif" WIDTH="68" HEIGHT="76" ALIGN="BOTTOM" BORDER="0">
     </TD>

<%
     }
%>
</TR>
</TABLE>

</P>

<P ALIGN="CENTER"><BR>
<BR>
<IMG SRC="images/playrespon.gif" WIDTH="120" HEIGHT="73" ALIGN="BOTTOM" BORDER="0">

</BODY>
</HTML>
```

A globals.jsa Example for Application and Session Events: index1.jsp

This example uses a `globals.jsa` file to process applications and session lifecycle events. It counts the number of active sessions, the total number of sessions, and the total number of times the application page has been hit. Each of these values is maintained at the `application` scope. The application page (`index1.jsp`) updates the page hit count on each request. The `globals.jsa session_OnStart` event handler increments the number of active sessions and the total number of sessions. The `globals.jsa session_OnEnd` handler decrements the number of active sessions by one.

The page output is simple. When a new session starts, the session counters are output. The page counter is output on every request. The final tally of each value is output in the `globals.jsa application_OnEnd` event handler.

Note the following in this example:

- When the counter variables are updated, access must be synchronized, as these values are maintained at `application` scope.
- The count values use the `oracle.jsp.jml.JmlNumber` extended datatype, which simplifies the use of data values at application scope. For information about the JML extended datatypes, refer to the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

globals.jsa File for index1.jsp

```
<%@ taglib uri="oracle.jsp.parse.OpenJspRegisterLib" prefix="jml" %>

<event:application_OnStart>

      <%-- Initializes counts to zero --%>
      <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
      <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
       <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

      <%-- Consider storing pageCount persistently -- If you do read it here --%>

</event:application_OnStart>

<event:application_OnEnd>
      <%-- Acquire beans --%>
      <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
      <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />

      <% application.log("The number of page hits were: " + pageCount.getValue() ); %>
       <% application.log("The number of client sessions were: " + sessionCount.getValue() ); %>

      <%-- Consider storing pageCount persistently -- If you do write it here --%>

</event:application_OnEnd>

<event:session_OnStart>

      <%-- Acquire beans --%>
      <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
      <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />
      <%
        synchronized (sessionCount) {
                sessionCount.setValue(sessionCount.getValue() + 1);
      %>
                <br>
                Starting session #: <%= sessionCount.getValue() %> <br>
      <%
        }
      %>

      <%
        synchronized (activeSessions) {
                activeSessions.setValue(activeSessions.getValue() + 1);
      %>
                There are currently <b> <%= activeSessions.getValue() %> </b> active sessions <p>
      <%
        }
      %>

</event:session_OnStart>

<event:session_OnEnd>

      <%-- Acquire beans --%>
      <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

      <%
         synchronized (activeSessions) {
                activeSessions.setValue(activeSessions.getValue() - 1);
        }
      %>

</event:session_OnEnd>
```

index1.jsp Source

```
<%-- Acquire beans --%>
<jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />

<%
   synchronized(pageCount) {
        pageCount.setValue(pageCount.getValue() + 1);
   }
%>
```

```
This page has been accessed <b> <%= pageCount.getValue() %> </b>  times.
<p>
```

A globals.jsa Example for Global Declarations: index2.jsp

This example uses a `globals.jsa` file to declare variables globally. It is based on the event handler sample in "A globals.jsa Example for Application and Session Events: index1.jsp" on page 9-42, but differs in that the three application counter variables are declared globally. (In the original event-handler sample, by contrast, each event handler and the JSP page itself had to provide `jsp:useBean` statements to locally declare the beans they were accessing.)

Declaring the beans globally results in implicit declaration in all event handlers and the JSP page.

globals.jsa File for index2.jsp

```
<%-- globally declares variables and initializes them to zero --%>

<jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
<jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
<jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application" />

<event:application_OnStart>

      <%-- Consider storing pageCount persistently -- If you do read it here --%>

</event:application_OnStart>

<event:application_OnEnd>

      <% application.log("The number of page hits were: " + pageCount.getValue() ); %>
       <% application.log("The number of client sessions were: " + sessionCount.getValue() ); %>
      <%-- Consider storing pageCount persistently -- If you do write it here --%>

</event:application_OnEnd>

<event:session_OnStart>

      <%
         synchronized (sessionCount) {
                sessionCount.setValue(sessionCount.getValue() + 1);
      %>
```

```
              <br>
              Starting session #: <%= sessionCount.getValue() %> <br>

      <%
        }
      %>

      <%
        synchronized (activeSessions) {
                activeSessions.setValue(activeSessions.getValue() + 1);
      %>
                There are currently <b> <%= activeSessions.getValue() %> </b> active sessions <p>
      <%
        }
      %>

</event:session_OnStart>

<event:session_OnEnd>

      <%
         synchronized (activeSessions) {
                activeSessions.setValue(activeSessions.getValue() - 1);
        }
      %>

</event:session_OnEnd>
```

index2.jsp Source

```
<%-- pageCount declared in globals.jsa so active in all pages --%>

<%
   synchronized(pageCount) {
        pageCount.setValue(pageCount.getValue() + 1);
   }
%>

This page has been accessed <b> <%= pageCount.getValue() %> </b>  times.

<p>
```

A

Getting Started in Alternative Environments

This appendix provides information about configuring the Web server to run Oracle JSP and configuring Oracle JSP in alternative environments. The technical information focuses on the following environments:

- JSWDK (the Sun Microsystems JavaServer Web Developer's Kit)
- Tomcat (from Apache, in cooperation with Sun Microsystems)

This appendix includes the following topics:

- Configuration of Web Server and Servlet Environment for Oracle JSP
- Oracle JSP Configuration Parameter Settings

Note: For installation and configuration information for the JServ environment, provided with Oracle9*i* release 2, as well as for general configuration information and required files, see "Getting Started in a JServ Environment" on page 9-2. For information for an OC4J environment, refer to the *Oracle9iAS Containers for J2EE Support for JavaServer Pages Reference*.

Configuration of Web Server and Servlet Environment for Oracle JSP

Configuring your Web server to run the Oracle JSP container requires the following general steps:

1. Add JSP-related JAR and ZIP files to the Web server classpath.
2. Configure the Web server to map JSP file name extensions (`.jsp` and `.JSP` and, optionally, `.sqljsp` and `.SQLJSP`) to the Oracle `JspServlet`, which is the front-end of the Oracle JSP container.

These steps apply to any Web server environment, but the information in this section focuses on the Sun Microsystems JSWDK and Tomcat.

Note: Examples here are for a UNIX environment, but the basic information (such as directory names and file names) applies to other environments as well.

Adding Oracle JSP-Related JAR and ZIP Files to Web Server Classpath

You must update the Web server classpath to add JAR and ZIP files that are required by the Oracle JSP container, being careful to add them in the proper order. (In particular, you must be careful as to where you place the servlet 2.2 version of `servlet.jar` in the classpath, as described below.) This includes the following:

- `ojsp.jar`
- `xmlparserv2.jar`
- `servlet.jar` (servlet 2.2 version)

  Note that the `servlet.jar` supplied with Oracle9*i* release 2 is identical to the `servlet.jar` provided with Tomcat 3.1.
- `ojsputil.jar` (optional, for JML tags, SQL tags, and data-access JavaBeans)
- `xsu12.jar`, for JDK 1.2.x, or `xsu111.jar`, for JDK 1.1.x (optional, for XML functionality for JML tags, SQL tags, and data-access JavaBeans)
- additional optional ZIP and JAR files, as necessary, for JDBC and SQLJ

See "Required and Optional Files for Oracle JSP" on page 9-2 for additional information.

Important: You must also ensure that the Oracle JSP container can find `javac` (or an alternative Java compiler, according to your `javaccmd` configuration parameter setting). For `javac` in a JDK 1.1.x environment, the JDK `classes.zip` file must be in the Web server classpath. For `javac` in a JDK 1.2.x environment, the JDK `tools.jar` file must be in the Web server classpath.

Add Files to Classpath for the JSWDK Environment

Update the `startserver` script in the `jswdk-1.0` root directory to add files required by the Oracle JSP container to the `jspJars` environment variable. Append them to the last `.jar` file listed, using the appropriate directory syntax and separator character for your operating system, such as a colon (`":"`) for UNIX or a semi-colon (`";"`) for Windows NT. Here is an example:

```
jspJars=./lib/jspengine.jar:./lib/ojsp.jar:./lib/ojsputil.jar
```

This example (with UNIX syntax) assumes that the JAR files are in the `lib` subdirectory under the `jswdk-1.0` root directory.

Similarly, update the `startserver` script to specify any additional required files in the `miscJars` environment variable, such as in the following example:

```
miscJars=./lib/xml.jar:./lib/xmlparserv2.jar:./lib/servlet.jar
```

This example (with UNIX syntax) also assumes that the files are in the `lib` subdirectory under the `jswdk-1.0` root directory.

Important: In a JSWDK environment, the servlet 2.1 version of `servlet.jar` (provided with Sun JSWDK 1.0) must precede the servlet 2.2 version of `servlet.jar` (provided with Oracle9*i* release 2) in your classpath.

The servlet 2.1 version is typically in the `jsdkJars` environment variable. The overall classpath is formed through a combination of various `xxxJars` environment variables, including `jsdkJars`, `jspJars`, and `miscJars`. Examine the `startserver` script to verify that `miscJars` is added to the classpath *after* `jsdkJars`.

Add Files to Classpath for the Tomcat Environment

For Tomcat, the procedure for adding files to the classpath is more operating-system dependent than for the other servlet environments discussed here.

For a UNIX operating system, copy the JSP-related JAR and ZIP files to your `[TOMCAT_HOME]/lib` directory. This directory is automatically included in the Tomcat classpath.

For a Windows NT operating system, update the `tomcat.bat` file in the `[TOMCAT_HOME]\bin` directory to individually add each file to the `CLASSPATH` environment variable. The following example presumes that you have copied the files to the `[TOMCAT_HOME]\lib` directory:

```
set CLASSPATH=%CLASSPATH%;%TOMCAT_HOME%\lib\ojsp.jar;%TOMCAT_HOME%\lib\ojsputil.jar
```

The servlet 2.2 version of `servlet.jar` (the same version that is provided with Oracle9*i* release 2) is already included with Tomcat, so it needs no consideration.

Mapping JSP File Name Extensions to Oracle JspServlet

You must configure the Web server to be able to do the following:

- It must recognize appropriate file name extensions as JSP pages.

  Map `.jsp` and `.JSP`. Also map `.sqljsp` and `.SQLJSP` if your JSP pages use Oracle SQLJ.

- It must find and execute the servlet that begins processing JSP pages.

  For the Oracle JSP container, this is `oracle.jsp.JspServlet`, which you can think of as the front-end of the JSP container.

Important: With the above configurations, the Oracle JSP container will support page references that use either a `.jsp` file name extension or a `.JSP` file name extension, but the case in the reference must match the actual file name in a case-sensitive environment. If the file name is `file.jsp`, you can reference it that way, but not as `file.JSP`. If the file name is `file.JSP`, you can reference it that way, but not as `file.jsp`. (The same holds true for `.sqljsp` versus `.SQLJSP`.)

Map File Name Extensions for the JSWDK Environment

In a JSWDK environment, mapping each JSP file name extension to the Oracle `JspServlet` requires two steps.

1. The first step is to update the `mappings.properties` file in the `WEB-INF` directory *of each servlet context* to define JSP file name extensions. Do this with the following commands:

   ```
   # Map JSP file name extensions (.sqljsp and .SQLJSP are optional).
   .jsp=jsp
   .JSP=jsp
   .sqljsp=jsp
   .SQLJSP=jsp
   ```

2. The second step is to update the `servlet.properties` file in the `WEB-INF` directory *of each servlet context* to define the Oracle `JspServlet` as the servlet that begins JSP processing. In addition, be sure to comment out the previously defined mapping for the JSP reference implementation. Do this as follows:

   ```
   #jsp.code=com.sun.jsp.runtime.JspServlet (replacing this with Oracle)
   jsp.code=oracle.jsp.JspServlet
   ```

Map File Name Extensions for the Tomcat Environment

In a Tomcat environment, mapping each JSP file name extension to the Oracle `JspServlet` requires a single step. Update the servlet mapping section of the `web.xml` file as shown below.

Note: There is a global `web.xml` file in the `[TOMCAT_HOME]/conf` directory. To override any settings in this file for a particular application, update the `web.xml` file in the `WEB-INF` directory under the particular application root.

```
# Map file name extensions (.sqljsp and .SQLJSP are optional).

<servlet-mapping>
   <servlet-name>
      oracle.jsp.JspServlet
   </servlet-name>
   <url-pattern>
      *.jsp
   </url-pattern>
```

```
</servlet-mapping>

<servlet-mapping>
   <servlet-name>
      oracle.jsp.JspServlet
   </servlet-name>
   <url-pattern>
      *.JSP
   </url-pattern>
</servlet-mapping>

<servlet-mapping>
   <servlet-name>
      oracle.jsp.JspServlet
   </servlet-name>
   <url-pattern>
      *.sqljsp
   </url-pattern>
</servlet-mapping>

<servlet-mapping>
   <servlet-name>
      oracle.jsp.JspServlet
   </servlet-name>
   <url-pattern>
      *.SQLJSP
   </url-pattern>
</servlet-mapping>
```

You can optionally set an alias for the `oracle.jsp.JspServlet` class name, as follows:

```
<servlet>
   <servlet-name>
      ojsp
   </servlet-name>
   <servlet-class>
      oracle.jsp.JspServlet
   </servlet-class>
   ...
</servlet>
```

Setting this alias allows you to use "ojsp" instead of the class name for your other settings, as follows:

```
<servlet-mapping>
```

```
   <servlet-name>
      ojsp
   </servlet-name>
   <url-pattern>
      *.jsp
   </url-pattern>
</servlet-mapping>
```

Oracle JSP Configuration Parameter Settings

The Oracle JSP front-end servlet, `JspServlet`, supports a number of configuration parameters to control operation of the JSP container. These are described in "Oracle JSP Configuration Parameters" on page 9-7. They are set as servlet initialization parameters for `JspServlet`. How you accomplish this depends on the Web server and servlet environment you are using.

This section describes how to set them in the JSWDK and Tomcat servlet environments.

Setting Oracle JSP Parameters in JSWDK

To set JSP configuration parameters in a JSWDK environment, set the `jsp.initparams` property in the `servlet.properties` file in the `WEB-INF` directory of the application servlet context, as in the following example (which happens to use UNIX syntax):

```
jsp.initparams=developer_mode=false,classpath=/mydir/myapp.jar
```

Note: Because `initparams` parameters are comma-separated, there can be no commas within a parameter setting. Spaces and other special characters do not cause a problem, however.

Setting Oracle JSP Parameters in Tomcat

To set JSP configuration parameters in a Tomcat environment, add `init-param` entries in the `web.xml` file as shown below.

Note: There is a global `web.xml` file in the `[TOMCAT_HOME]/conf` directory. To override any settings in this file for a particular application, update the `web.xml` file in the `WEB-INF` directory under the particular application root.

```
<servlet>
   <init-param>
      <param-name>
         developer_mode
      </param-name>
      <param-value>
```

```
            true
         </param-value>
      </init-param>
      <init-param>
         <param-name>
            external_resource
         </param-name>
         <param-value>
            true
         </param-value>
      </init-param>
      <init-param>
         <param-name>
            javaccmd
         </param-name>
         <param-value>
            javac -verbose
         </param-value>
      </init-param>
   </servlet>
```

B
Third Party Licenses

This appendix includes the Third Party License for third party products included with Oracle9*i* Application Server and discussed in this document. Topics include:

- Apache HTTP Server
- Apache JServ

Apache HTTP Server

The Apache Software License

Apache JServ

Apache JServ Public License

Index