Oracle9*i*AS Containers for J2EE

Support for JavaServer Pages Developer's Guide

Release 2 (9.0.3)

August 2002

Part No. A97679-01

ORACLE

Oracle9*i*AS Containers for J2EE Support for JavaServer Pages Developer's Guide, Release 2 (9.0.3)

Part No. A97679-01

Primary Author: Brian Wright

Contributing Author: Michael Freedman

Contributors: Julie Basu, Alex Yiu, Sunil Kunisetty, Gael Stevens, Sumathi Gopalakrishnan, Ping Guo, Olga Peschansky, YaQing Wang, Song Lin, Helen Zhao, Hal Hildebrand, Jasen Minton, Ashok Banerjee, Matthieu Devin, Jose Alberto Fernandez, 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

Contents

2 Overview of the Oracle JSP Implementation

3 Getting Started

4 Basic Programming Considerations

5 JSP XML Support

6 Additional Considerations

7 JSP Translation and Deployment

8 JSP Tag Libraries

9 JSP Globalization Support

A Servlet and JSP Technical Background

B The Apache JServ Environment

C Third Party Licenses

Index

Send Us Your Comments

Oracle9*i*AS Containers for J2EE Support for JavaServer Pages Developer's Guide, Release 2 (9.0.3)
Part No. A97679-01

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: jpgreader_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 Oracle9*i*AS Containers for J2EE (OC4J) JSP container in Oracle9*i*AS release 2 (9.0.3) is a complete implementation of the Sun Microsystems *JavaServer Pages Specification, Version 1.2*

This preface contains these topics:

Note: The Sample Applications chapter available in previous releases has been removed. Applications that were listed there are available in the OC4J demos, from either of the following locations:

- the OC4J demo instance, included with the Oracle9*i*AS product
- the JSP download page on the Oracle Technology Network (requiring an OTN membership, which is free):

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

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 (technical background provided in Appendix A)
- 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 1.2 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 1.2 features, consult the Sun Microsystems *JavaServer Pages Specification, Version 1.2* or other appropriate reference materials.

The JSP 1.2 specification relies on a servlet 2.3 environment, and this document is geared largely toward such environments (also considering some JSP 1.1 backward compatibility issues). The OC4J JSP container has special features for earlier servlet environments, however, and there is special discussion of these features in Appendix B as they relate to servlet 2.0 environments, particularly Apache JServ, which is included with the Oracle9*i* Application Server.

For documentation of tag libraries and utilities that are provided with the OC4J product, please refer to the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference* (although an overview is provided here, in "Overview of Tag Libraries and Utilities Provided with OC4J" on page 2-13).

For a quick primer about getting started with JSP pages in OC4J, see the *Oracle9iAS Containers for J2EE User's Guide*.

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 JSP Overview"

This chapter highlights standard JSP 1.2 technology. It is not intended as a complete reference.

Chapter 2, "Overview of the Oracle JSP Implementation"

This chapter provides an overview of the JSP implementation provided with OC4J, including both portable and Oracle-specific value-added features.

Chapter 3, "Getting Started"

This contains information about required files for the OC4J JSP container, OC4J Web server configuration, and JSP configuration.

Chapter 4, "Basic Programming Considerations"

This chapter introduces basic JSP programming considerations, including JSP-servlet interaction and database access, and provides some examples.

Chapter 5, "JSP XML Support"

This chapter describes JavaServer Pages support for XML, primarily added with the JSP 1.2 specification. JSP XML syntax and the JSP XML view are described.

Chapter 6, "Additional Considerations"

This chapter discusses a variety of general programming, configuration, and runtime issues that the developer should be aware of. It also covers considerations specific to the OC4J environment.

Chapter 7, "JSP Translation and Deployment"

This chapter describes features of the Oracle9*i*AS JSP translator and Oracle `ojspc` pre-translation utility, and discusses general and OC4J-specific deployment considerations.

Chapter 8, "JSP Tag Libraries"

This chapter describes the standard JSP 1.2 framework for custom tag libraries. There is also discussion of OC4J extended features for tag library support, and vendor-specific compile-time tags.

Chapter 9, "JSP Globalization Support"

This chapter covers features for globalization support.

Appendix A, "Servlet and JSP Technical Background"

This appendix provides a brief background of servlet technology and introduces the standard JSP interfaces for translated pages.

Appendix B, "The Apache JServ Environment"

This appendix provides details for the JServ servlet 2.0 environment, including deployment, configuration, and special programming considerations.

Appendix C, "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.

Related Documentation

See the following additional OC4J documents 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 JSP Tag Libraries and Utilities Reference*

  This book provides conceptual information and detailed syntax and usage information for JSP 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 standards-based Java services supplied with OC4J, such as JTA, JNDI, JMS, JAAS/JAZN, 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.

Also available from the Oracle Java Platform group:

- *Oracle9i JDBC Developer's Guide and Reference*
- *Oracle9i SQLJ Developer's Guide and Reference*
- *Oracle9i JPublisher User's Guide*
- *Oracle9i Java Stored Procedures Developer's Guide*

The following documents are available from the Oracle9*i* Application Server group:

- *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*
- *Oracle9iAS Web Services Developer's Guide*
- *Oracle9i Application Server Migrating to Release 2 (9.0.3)*

The following are available from the JDeveloper group:

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

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

The following documents from the Oracle Server Technologies group are also of possible interest:

- *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*

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/`

If you already have a user name 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 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 document. 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.
`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, user names, 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.

Convention	Meaning	Example
`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, user names 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]`

Convention	Meaning	Example
. . .	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 JSP Overview

This chapter reviews standard features and functionality of JavaServer Pages technology, then concludes with a discussion of JSP execution models. For further general information, consult the Sun Microsystems *JavaServer Pages Specification, Version 1.2*.

JSP 1.2 functionality depends upon servlet 2.3 functionality. You can also refer to the Sun Microsystems *Java Servlet Specification, Version 2.3* for information.

For an overview of the JSP implementation in OC4J, see Chapter 2, "Overview of the Oracle JSP Implementation". Also note that Appendix A, "Servlet and JSP Technical Background", provides related background on standard servlet and JSP technology.

The following topics are covered here:

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

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, enables 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 (perhaps using SQLJ).

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

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.

Note: See the OC4J demos for some basic JSP sample applications.

What a JSP Page Looks Like

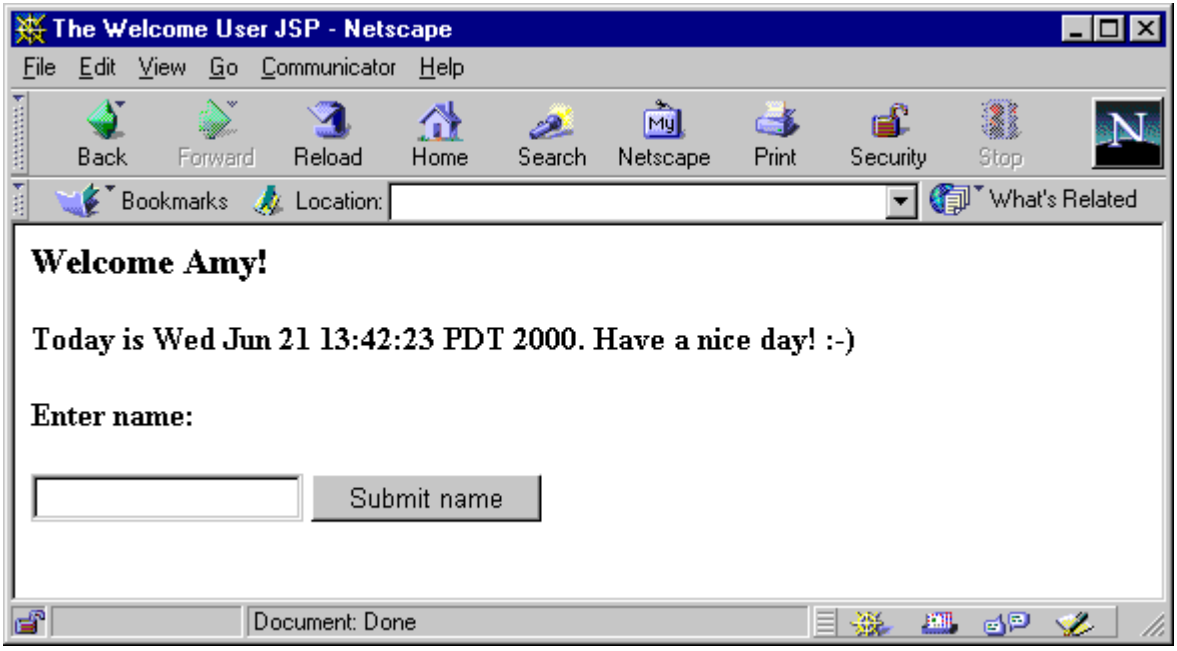
Here is an example of a simple JSP page. For an explanation of JSP syntax elements used here, see "Overview of JSP Syntax Elements" on page 1-7.

```
<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 traditional JSP page, Java elements are set off by tags such as <% and %>, as in the preceding example. (JSP XML syntax is different, as described in "Details of JSP XML Documents" on page 5-4.) 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":



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)
      }
   }
}
```

See "The Servlet Interface" on page A-3 for some background information about the standard `HttpServlet` abstract class, `HttpServletRequest` interface, and `HttpServletResponse` interface.

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.

Note: The list of packages imported into a JSP page by default changed in the OC4J 9.0.3 implementation. The default list was reduced to follow the JSP specification. See "Default Package Imports" on page 3-5 for more information. (Starting with 9.0.3, the preceding JSP example requires a configuration setting to import the `java.io` package.)

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 (covered in "Standard JSP Interfaces and Methods" on page A-12) 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 it is 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 (the person who 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 users see (the person who 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 "Standard Actions: JSP Tags" on page 1-16.

JSP Pages and Alternative Markup Languages

JavaServer Pages technology is typically used for dynamic HTML output, but the Sun Microsystems *JavaServer Pages Specification, Version 1.2* 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 JSP support for XML, refer to Chapter 5, "JSP XML Support" and to the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

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. There is also discussion of bean property conversions, and an introduction to custom tag libraries (used for custom actions). For more information, see the Sun Microsystems *JavaServer Pages Specification, Version 1.2*.

Note: This section describes traditional JSP syntax. For information about JSP XML syntax and JSP XML documents, see Chapter 5, "JSP XML Support".

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 1.2 specification supports the following directives:

- `page`—Use this directive to specify any of a number of page-dependent attributes, such as scripting language, content type and character encoding, a class to extend, packages to import, an error page to use, the JSP page output buffer size, and whether to automatically flush the buffer when it is full. For example:

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

or, to enable auto-flush and set the JSP page output buffer size to 20 KB:

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

or, to unbuffer the page:

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

Notes:

- The default buffer size is 8 KB.
- It is illegal to set `autoFlush="true"` when `buffer="none"`.
- A JSP page using an error page must be buffered. Forwarding to an error page (not outputting it to the browser) clears the buffer.
- In the Oracle JSP implementation, "`java`" is the default language setting. It is good programming practice to set it explicitly, however. You can also use a "`sqlj`" setting for SQLJ JSP pages.
- For information about using `page` directive attributes to set the content type and character set for the JSP page and response object, see "Content Type Settings in the page Directive" on page 9-2.

- `include`—Use this directive to specify a resource that contains text or code to be inserted into the JSP page when it is translated. For example:

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

  Specify either a page-relative or context-relative path to the resource. (See "Requesting a JSP Page" on page 1-28 for discussion of page-relative and context-relative paths.)

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 `jsp:include` takes effect at request-time instead of translation-time. See "Static Includes Versus Dynamic Includes" on page 6-3.
- The `include` directive can be used only between files in the same servlet context (application).
- See "JSP File Naming Conventions" on page 3-6 for information about naming conventions for included files.

- `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 includes a pointer to a *tag library descriptor* file and a prefix to distinguish use of tags from that library. For 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>
  ```

  JSP tag libraries and tag library descriptor files are introduced later in this chapter, in "Custom Tag Libraries" on page 1-24, and discussed in detail in Chapter 8, "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 6-8 for a comparison between the two.)

- *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 semicolon, and is contained within `<%=...%>` tags. For 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, can 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 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 code (one of which includes a JSP expression, which does *not* require a semicolon). 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 6-8.

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

Comments are contained within <%--....--%> syntax. For example:

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

Unlike HTML comments, JSP comments are not visible when users view the page source from their browsers.

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.

This section covers the following topics:

Explicit Objects

Explicit objects are typically JavaBean instances that are declared and created in `jsp:useBean` action statements. The `jsp:useBean` statement and other action statements are described in "Standard Actions: JSP Tags" on page 1-16, but here is an example:

```
<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" on page 1-15.

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

Implicit Objects

JSP technology makes available to any JSP page a set of *implicit objects*. These are Java objects that are created automatically by the JSP container 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 locations (for servlet 2.2 and servlet 2.3 classes, respectively):

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

`http://java.sun.com/products/servlet/2.3/javadoc/index.html`

- `page`

  This is an instance of the JSP page implementation class and is created when the page is translated. The page implementation class implements the interface `javax.servlet.jsp.HttpJspPage`. Note that `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 a class that implements 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 "Runtime Error Processing" on page 4-26.

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>
```

The `request` object, like the other implicit objects, is available automatically; it is not explicitly instantiated.

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, you can explicitly set the scope with the following syntax, as in the example in "Explicit Objects" on page 1-12:

scope="*scopevalue*"

There are four possible scopes:

- `scope="page"` (default scope)—The object is accessible only from within the JSP page where it was created. A page-scope object is stored in the implicit `pageContext` object. The `page` scope ends when the page stops executing.

  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. A request-scope object is stored in the implicit `request` object. The `request` scope ends at the conclusion of the HTTP request.

- `scope="session"`—The object is accessible from any JSP page that is sharing the same HTTP session as the JSP page that created the object. A session-scope object is stored in the implicit `session` object. The `session` scope ends when the HTTP session times out or is invalidated.

- `scope="application"`—The object is accessible from any JSP page that is used in the same Web application as the JSP page that created the object, within any single Java virtual machine. The concept is similar to that of a Java static variable. An application-scope object is stored in the implicit `application` servlet context object. The `application` scope ends when the application itself terminates, or when the JSP container or servlet container shuts down.

You can think of these four scopes as being in the following progression, from narrowest scope to broadest scope:

```
page < request < session < application
```

If you want to share an object between different pages in an application, such as when forwarding execution from one page to another, or including content from one page in another, you cannot use `page` scope for the shared object; in this case, there would be a separate object instance associated with each page. The narrowest scope you can use to share an object between pages is `request`. (For information about including and forwarding pages, see "Standard Actions: JSP Tags" below.)

Note: The `request`, `session`, and `application` scopes also apply to servlets.

Standard Actions: JSP Tags

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

For standard actions, there is a set of tags defined in the JSP specification. Although directives and scripting elements described earlier in this chapter are sufficient to code a JSP page, the standard tags described here provide additional functionality and convenience.

Here is the general tag syntax for JSP standard actions:

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

or, where there is no body:

```
<jsp:tag 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` tag accesses or creates an instance of a Java type, typically a JavaBean class, and associates the instance with a specified name, or ID. The instance is then available by that ID as a scripting variable of specified scope. Scripting variables are introduced in "Custom Tag Libraries" on page 1-24. Scopes are discussed in "JSP Objects and Scopes" on page 1-12.

  The key attributes are `class`, `type`, `id`, and `scope`. (There is also a less frequently used `beanName` attribute, discussed below.)

  Use the `id` attribute to specify the instance name. The JSP container will first search for an object by the specified ID, of the specified type, in the specified scope. If it does not exist, the container will attempt to create it.

  Intended use of the `class` attribute is to specify a class that can be instantiated, if necessary, by the JSP container. The class cannot be abstract and must have a no-argument constructor. Intended use of the `type` attribute is to specify a type that cannot be instantiated by the JSP container—either an interface, an abstract class, or a class without a no-argument constructor. You would use `type` in a situation where the instance will already exist, or where an instance of an instantiable class will be assigned to the type. There are three typical scenarios:

  – Use `type` and `id` to specify an instance that already exists in the target scope.

  – Use `class` and `id` to specify the name of an instance of the class, either an instance that already exists in the target scope, or an instance to be newly created by the JSP container.

  – Use `class`, `type`, and `id` to specify a class to instantiate and a type to assign the instance to. In this case, the class must be legally assignable to the type.

  Use the `scope` attribute to specify the scope of the instance—either `page` for the instance to be associated with the page context object, `request` for it to be associated with the HTTP request object, `session` for it to be associated with the HTTP session object, or `application` for it to be associated with the servlet context.

  As an alternative to using the `class` attribute, you can use the `beanName` attribute. In this case, you have the option of specifying a serializable resource instead of a class name. When you use the `beanName` attribute, the JSP container creates the instance by using the `instantiate()` method of the `java.beans.Beans` class.

Consider the following examples:

```
<jsp:useBean id="reqobj" type="mypkg.MyIntfc" scope="request" />
```

The preceding example uses a request-scope instance `reqobj` of type `MyIntfc`. Because `MyIntfc` is an interface and cannot be instantiated directly, `reqobj` would have to already exist.

```
<jsp:useBean id="pageobj" class="mybeans.PageBean" scope="page" />
```

The preceding example uses a page-scope instance `pageobj` of class `PageBean`, first creating it if necessary.

```
<jsp:useBean id="sessobj" class="mybeans.SessionBean"
             type="mypkg.MyIntfc scope="session" />
```

The preceding example creates an instance of class `SessionBean`, and assigns the instance to the variable `sessobj` of type `MyIntfc`.

- `jsp:setProperty`

  The `jsp:setProperty` tag sets one or more bean properties. The bean must have been previously specified in a `jsp:useBean` tag. 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 `jsp: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="*" />
```

When you use the `jsp:setProperty` tag, string input can be used to specify the value of a non-string property through conversions that happen behind the scenes. See "Bean Property Conversions from String Values" on page 1-22.

Important: For `property="*"`, the JSP 1.2 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 JSP translator can generate the corresponding `set`*`XXX`*`()` methods directly. In this case, introspection occurs only during translation. There will be no need to introspect the bean during runtime, which is more costly.

- `jsp:getProperty`

  The `jsp:getProperty` tag 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` tag. 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 `jsp:param` tags in conjunction with `jsp:include`, `jsp:forward`, and `jsp:plugin` tags (described below).

  Used with `jsp:forward` and `jsp:include` tags, a `jsp:param` tag optionally provides name/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" />
```

- `jsp:include`

  The `jsp:include` tag 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). For example:

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

  A "`true`" setting of the `flush` attribute results in the buffer being flushed to the browser when a `jsp:include` action is executed. The JSP 1.2 specification and the OC4J JSP container support either a "`true`" or "`false`" setting, with "`false`" being the default. (The JSP 1.1 specification supports only a "`true`" setting, with `flush` being a required attribute.)

  You can also have an action body with `jsp:param` tags, as shown in the following example:

  ```
  <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 example:

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

Notes:

- The `jsp:include` tag, 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 6-3.
- The `jsp:include` tag can be used only between pages in the same servlet context (application).

- `jsp:forward`

  The `jsp:forward` tag 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` tag; you cannot set `buffer="none"` in a `page` directive. 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` tags, as shown in the second of the following 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` tag can be used only between pages in the same servlet context.
- The `jsp:forward` tag 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` tag 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 code base, 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` tags between the `<jsp:params>` start-tag and the `</jsp:params>` end-tag to specify parameters to the applet or JavaBean. (Note that the `jsp:params` start-tag and end-tag are *not* necessary when using `jsp:param` in a `jsp:include` or `jsp:forward` action.)

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

  The following example, from the *Sun Microsystems JavaServer Pages Specification, Version 1.2*, 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` tag as well. Use of these parameters is according to the general HTML specification.

Bean Property Conversions from String Values

As noted earlier, when you use a JavaBean through a `jsp:useBean` tag in a JSP page, and then use a `jsp:setProperty` tag to set a bean property, string input can be used to specify the value of a non-string property through conversions that happen behind the scenes. There are two conversion scenarios:

- Typical Property Conversions
- Conversions for Property Types with Property Editors

Typical Property Conversions

For a bean property that does not have an associated property editor, Table 1–1 shows how conversion is accomplished when using a string value to set the property.

Table 1–1 Attribute Conversion Methods

Property Type	Conversion
boolean or Boolean	according to `valueOf(String)` method of `Boolean` class
byte or Byte	according to `valueOf(String)` method of `Byte` class
char or Character	according to `charAt(0)` method of `String` class (inputting an index value of 0)
double or Double	according to `valueOf(String)` method of `Double` class
int or Integer	according to `valueOf(String)` method of `Integer` class
float or Float	according to `valueOf(String)` method of `Float` class
long or Long	according to `valueOf(String)` method of `Long` class
short or Short	according to `valueOf(String)` method of `Short` class
Object	as if `String` constructor is called, using literal string input The `String` instance is returned as an `Object` instance.

Conversions for Property Types with Property Editors

A bean property can have an associated property editor, which is a class that implements the `java.beans.PropertyEditor` interface. Such classes can provide support for GUIs used in editing properties. Generally speaking, there are standard property editors for standard Java types, and there can be user-defined property editors for user-defined types. In the OC4J JSP implementation, however, only user-defined property editors are searched for. Default property editors of the `sun.beans.editors` package are not taken into account.

For information about property editors and how to associate a property editor with a type, you can refer to the Sun Microsystems *JavaBeans API Specification, Version 1.01*.

You can still use a string value to set a property that has an associated property editor, as specified in the JavaBeans specification. In this situation, the `setAsText(String text)` method specified in the `PropertyEditor` interface is used in converting from string input to a value of the appropriate type. (If the

`setAsText()` method throws an `IllegalArgumentException`, the conversion will fail.)

Custom Tag Libraries

In addition to the standard JSP tags discussed above, the JSP specification lets vendors define their own *tag libraries*, and lets vendors implement a framework that allows customers to define their own tag libraries as well.

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 when 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 1-7.

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

- tag library descriptor files

  A *tag library descriptor* (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.

- tag handlers

  A *tag handler* specifies the action of a custom tag and is an instance of a Java class that implements either the `Tag`, `IterationTag`, or `BodyTag` interface (as appropriate, depending on whether the tag has a body and whether the tag handler requires access to the body content) 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 are specified in the TLD file or in a subclass of the `TagExtraInfo` abstract class (in package `javax.servlet.jsp.tagext`). This document refers to a subclass of `TagExtraInfo` as a *tag-extra-info class*. The JSP container uses instances of these classes during translation.

- tag-library-validators

  A *tag-library-validator* class has logic to validate any JSP page that uses the tag library, according to specified constraints.

- event listeners

  You can use servlet 2.3 *event listeners* with a tag library. This functionality is offered as a convenient alternative to declaring listeners in the application `web.xml` file.

- use of `web.xml` for tag libraries

  The Sun Microsystems *Java Servlet Specification, Version 2.3* 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 descriptor 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 Chapter 8, "JSP Tag Libraries". For further information, see the Sun Microsystems *JavaServer Pages Specification, Version 1.2*.

For complete information about the tag libraries provided with OC4J, see the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

JSP Execution

This section provides a top-level look at how a JSP page 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* first appeared 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. Servlet containers are summarized in "Servlet Containers" on page A-3.

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

JSP Execution Models

There are two distinct execution models for JSP pages:

- In most implementations and situations, the JSP container translates pages *on demand* before triggering their execution; that is, at the time they are requested by the user.
- In some scenarios, however, the developer might want to translate the pages in advance and deploy them as working servlets. Command-line tools are available to translate the pages, load them, and "publish" them to make them available for execution. You can have the translation occur either on the client or in the server. When the end-user requests the JSP page, it is executed directly, with no translation necessary.

On-Demand Translation Model

It is typical to run JSP pages in an on-demand translation scenario. When a JSP page is requested from a Web server that incorporates the JSP container, a front-end

servlet is instantiated and invoked, assuming proper Web server configuration. This servlet can be thought of as the front-end of the JSP container. In OC4J, it is `oracle.jsp.runtimev2.JspServlet`.

`JspServlet` locates the JSP page, translates and compiles it if necessary (if the translated 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`. This is handled automatically during OC4J installation, as discussed in "JSP Container Setup" on page 3-8.

Pre-Translation Model

As an alternative to the typical on-demand scenario, developers may want to pre-translate their JSP pages before deploying them. This can offer the following advantages, for example:

- It can save time for the end-users when they first 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 or you have security concerns and you do not want to expose the code.

For more information, see "JSP Pre-Translation" on page 7-37 and "Deployment of Binary Files Only" on page 7-40.

Oracle 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 "The ojspc Pre-Translation Utility" on page 7-13.

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 extends a class (provided by the JSP container) that implements the `javax.servlet.jsp.HttpJspPage` interface (described in "Standard JSP Interfaces and Methods" on page A-12). 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 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 extended 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 Requesting a JSP Page

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

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

You can request it with a URL such as the following:

```
http://host[:port]/myapproot/dir1/HelloWorld.jsp
```

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.

Note: This is just a general example. By default in OC4J in Oracle9*i*AS release 2, the context path must start with "/j2ee" if you want processing to be routed to OC4J through the Oracle HTTP Server and `mod_oc4j`, as in a typical production environment. Oracle HTTP Server and mod_oc4j are introduced in "Role of the Oracle HTTP Server and mod_oc4j" on page 2-10. General servlet and JSP invocation are discussed in the *Oracle9iAS Containers for J2EE Servlet Developer's Guide*.

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" />
```

("Standard Actions: JSP Tags" on page 1-16 discusses the `jsp:include` and `jsp:forward` statements.)

2

Overview of the Oracle JSP Implementation

The JSP container provided with Oracle9*i*AS Containers for J2EE (OC4J) in the Oracle9*i* Application Server is a complete implementation of the Sun Microsystems *JavaServer Pages Specification, Version 1.2*. JSP 1.2 functionality depends upon servlet 2.3 functionality, and the OC4J servlet container is a complete implementation of the Sun Microsystems *Java Servlet Specification, Version 2.3*.

This chapter provides overviews of the Oracle9*i* Application Server, OC4J, the OC4J JSP implementation and features, and custom tag libraries and utilities that are also supplied (documented in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*).

The following topics are covered here:

- Overview of the Oracle9i Application Server and JSP Support
- Oracle9i JDeveloper JSP Support
- Overview of Oracle Value-Added Features

Overview of the Oracle9*i* Application Server and JSP Support

This section provides a brief overview of the Oracle9*i* Application Server, its J2EE environment, its JSP implementation, and its Web server:

Note: Users of earlier Oracle9*i*AS releases can refer to *Oracle9i Application Server Migrating to Release 2 (9.0.3)* for information about issues in migrating to Oracle9*i*AS release 2.

Overview of the Oracle9*i* Application Server

Oracle9*i* Application Server is a scalable, secure, middle-tier application server. It can be used to deliver Web content, host Web applications, connect to back-office applications, and make these services accessible to any client browser. Users can access information, perform business analysis, and run business applications on the Internet or corporate intranets or extranets. Major areas of functionality include business intelligence, e-business integration, J2EE Web services, performance and caching, portals, wireless, and management and security.

To deliver this range of content and services, the Oracle9*i* Application Server incorporates many components, including the Oracle HTTP Server, Oracle9*i*AS Web Cache, Oracle9*i*AS Web Services, Oracle9*i*AS Portal, Oracle9*i*AS Wireless, Oracle9*i*AS Forms Services and Reports Services (to support Oracle Forms-based applications and reports generation), Oracle9*i*AS Personalization, and various business logic runtime environments that support Enterprise JavaBeans, stored procedures, and Oracle Business Components for Java.

For its J2EE environment, Oracle9*i*AS provides the Oracle9*i*AS Containers for J2EE (OC4J), which includes the JSP container described in this manual, a servlet container, and an EJB container.

(In addition, Oracle9*i*AS includes an Apache JServ servlet environment, documented in Appendix B, "The Apache JServ Environment".)

Overview of OC4J

OC4J is a high-performance J2EE-compliant environment providing a scalable and reliable server infrastructure by supporting *clusters* and *load balancing*. (See the *Oracle9i Application Server Performance Guide* for information about clustering.) With Oracle9*i*AS release 2 (9.0.3), OC4J complies with the J2EE 1.3 specification.

OC4J General Features

Each OC4J instance runs in a single Java virtual machine. The JVM running an OC4J instance is referred to as a *node*. One or more nodes—typically about two to four—form an *island*. Multiple islands together form a cluster. For each OC4J cluster, there is a JVM for each OC4J instance, plus a JVM for the load balancer. (Other types of clusters are possible in Oracle9*i*AS as well, as discussed in the *Oracle9i Application Server Administrator's Guide*.)

In addition to load balancing, which improves performance by distributing requests among multiple servers, the clustering mechanism provides fault tolerance, which allows any particular server to redirect a client to another server in the event of failure.

Java applications built with any development tool can be deployed to OC4J, which supports standard EAR/WAR/JAR deployment. You can debug applications deployed to OC4J through standard Java profiling and debugging facilities.

In an Oracle9*i*AS environment, OC4J can be fully managed and configured using the HTML-based Oracle Enterprise Manager. This includes full support for managing clustering, configuration, and deployment.

OC4J Services

OC4J supports the following Java and J2EE services:

- J2EE Connector Architecture (JCA)—JCA defines a standard architecture for connecting J2EE platforms to heterogeneous enterprise information systems such as ERP systems, mainframe transaction processing, database systems, and legacy applications.
- Java Transaction API (JTA) and two-phase commits—JTA allows simultaneous updates to multiple resources in a single, coordinated transaction.
- Java Message Service (JMS) integration—This integration allows compatibility between the Oracle JMS implementation and those of other JMS providers.
- Java Naming and Directory Interface (JNDI)—JNDI associates names with resources for lookup purposes.

- Java Authentication and Authorization Service (JAAS)—The Oracle implementation of JAAS and the Java2 security model provides complete support for development and deployment of secure applications and for fine-grained authorization and access control.

OC4J Containers

OC4J supplies the following J2EE containers:

- a JSP container complying with the Sun JSP 1.2 specification

  The JSP bundle also supplies tag libraries to implement Web services, caching capabilities, SQL access, file access, and other features. For further overview of the JSP container provided with OC4J, see "Overview of the JSP Implementation in OC4J" on page 2-6.

- a servlet container complying with the Sun Microsystems servlet 2.3 specification (see below for key features)
- an EJB container complying with the Sun EJB 2.0 specification (see below for key features)

OC4J containers have been instrumented to support the Dynamic Monitoring Service (DMS) to provide runtime performance data. You can view this data through Enterprise Manager.

Note: Servlet 2.3 compliance is required in order to support JSP 1.2 compliance.

Key Servlet Container Features The OC4J servlet container supports stateful failover and cluster deployment in addition to the following key features:

- servlet filtering—This allows transformation of the content of an HTTP request or response, and modification of header information.
- application-level and session-level event listeners—This feature allows greater control over interaction with servlet context and HTTP session objects and, therefore, greater efficiency in managing resources that the application uses.
- integration with SSO and OID—This is through the Oracle JAAS implementation.

Key EJB Container Features The OC4J EJB container supports the following:

- session beans—A session bean is used for task-oriented requests. You can define a session bean as stateless or stateful. Stateless session beans cannot maintain state information across calls, while stateful session beans *can* maintain state across calls.
- entity beans—An entity bean represents data. It can use the container to maintain the data persistently, which is referred to as container-managed persistence (CMP), or it can use the bean implementation to manage the data, which is referred to as bean-managed persistence (BMP).
- message-driven beans (MDB)—A message-driven bean is used to receive JMS messages from a queue or topic. It can then invoke other EJBs to process the JMS message.

EJB support in OC4J also includes these features:

- clustering for session and entity beans
- enhanced entity bean concurrency models to support concurrent access from multiple clients
- extended locking models for entity beans (optimistic locking mode / pessimistic locking mode / read-only mode)
- Active Components for Java (AC4J), to provide a standards-based infrastructure for coordinating long-running business transactions

OC4J Standalone

In a production environment, it is typical to use OC4J inside a complete Oracle9*i*AS environment, including the Oracle HTTP Server (as described in "Role of the Oracle HTTP Server and mod_oc4j" on page 2-10), Oracle9*i*AS Web Cache, and Enterprise Manager.

For a development environment, OC4J is also available as a standalone component by downloading `OC4J_extended.zip` from the Oracle Technology Network (`http://otn.oracle.com`).

When using OC4J standalone, you can use its own HTTP Web listener through port 8888. For information about OC4J standalone, see the standalone version of the *Oracle9iAS Containers for J2EE User's Guide* (downloadable with OC4J_extended.zip) and the *Oracle9iAS Containers for J2EE Servlet Developer's Guide*.

Overview of the JSP Implementation in OC4J

The JSP container in OC4J is compliant with the Sun Microsystems JSP 1.2 specification.

In general, a JSP 1.2 environment requires a servlet 2.3 environment, such as the OC4J servlet container. The Oracle JSP implementation, however, also supports running on Apache JServ, a servlet 2.0 environment. To make this possible, the OC4J JSP container emulates required servlet features beyond the 2.0 specification.

For a variety of reasons, though, it is generally advisable to use the OC4J servlet 2.3 environment.

This section offers additional information on the following topics:

- History and Integration of JSP Containers
- JSP Front-End Servlet and Configuration
- OC4J JSP Features for JSP 1.2
- Configurable JSP Extensions in OC4J
- Portability Across Servlet Environments

History and Integration of JSP Containers

In Oracle9*i*AS release 1.0.2.2, the first release to include OC4J, there were two JSP containers: 1) a container developed by Oracle and known as "OracleJSP"; 2) a container licensed from Ironflare AB and known as the "Orion JSP container".

The OracleJSP container offered several advantages, including useful value-added features and enhancements such as for globalization and SQLJ support. The Orion container also offered advantages, including superior speed, but had disadvantages as well. It did not always exhibit standard behavior when compared to the JSP reference implementation (Tomcat), and its support for internationalization and globalization was not as complete.

Oracle9*i*AS release 2 (9.0.2) first integrated the OracleJSP and Orion containers into a single JSP container referred to in this manual as the "OC4J JSP container". This container offers the best features of both previous versions, runs efficiently as a servlet in the OC4J servlet container, and is integrated with other OC4J containers as well. The integrated container primarily consists of the OracleJSP translator and the Orion container runtime, running with a simplified dispatcher and the OC4J core runtime classes.

JSP Front-End Servlet and Configuration

The JSP container in OC4J uses the front-end servlet `oracle.jsp.runtimev2.JspServlet`. See "JSP Configuration in OC4J" on page 3-8.

For non-OC4J environments, including Apache JServ, use the old front-end servlet, `oracle.jsp.JspServlet`. See "Getting Started in a JServ Environment" on page B-2.

OC4J JSP Features for JSP 1.2

Beginning with the OC4J 9.0.3 implementation, the OC4J JSP container is fully compliant with the JSP 1.2 specification. Most of the new functionality is in the area of custom tag libraries. Here is a summary of new features:

- tag library features
  - There is a new tag handler interface that allows iteration through a tag body without having to maintain and access a body content object.
  - You can create a tag-library-validator class and associate it with a tag library. A validator instance will check any JSP page that uses the library, to verify that it meets whatever constraints you desire.
  - For convenience, you can declare servlet 2.3 event listeners in a tag library descriptor file instead of in the `web.xml` file. This enables you to more conveniently manage application and session resources associated with usage of the tag library.
  - You can package multiple tag libraries and their TLD files inside a single JAR file.

  See Chapter 8, "JSP Tag Libraries" for details about these features, and "Overview of Tag Library Changes Between the JSP 1.1 and 1.2 Specifications" on page 8-4 for a more detailed summary.
- XML features
  - The OC4J JSP container previously supported XML-alternative syntax, but this is now replaced with support according to the JSP 1.2 specification.
  - The OC4J JSP container generates an XML view of every translated page, which is a mapping to an XML document that describes the page. This view is available for use by tag-library-validator classes.

  See Chapter 5, "JSP XML Support" for information about these features.

- character encoding features

  OC4J JSP supports the `pageEncoding` attribute of the `page` directive. This enables you to specify a character encoding for the page source that is different than the character encoding for the response (specified in the `contentType` attribute).

  See "Content Type Settings" on page 9-2.

Configurable JSP Extensions in OC4J

In addition to JSP 1.2 compliance, the OC4J JSP container in Oracle9*i*AS release 2 (9.0.3) includes the following configurable features.

Also see "Overview of Oracle Value-Added Features" on page 2-13.

The following features are new in the OC4J 9.0.3 implementation:

- mode switch to avoid JSP translation errors if you have duplicate settings for the same directive attribute within a single JSP translation unit (OC4J or JServ)

  The JSP 1.2 specification mandates translation errors if you have duplicate settings for the same directive attribute within a single JSP translation unit (except for the `page` directive `import` attribute). These errors may be unwanted or inappropriate, for example, if a page and an included file both set an attribute to the same value (such as `language="java"`).

  In "JSP Configuration Parameters" on page 3-9, see the description of the `forgive_dup_dir_attr` parameter.

- separate mode switches for XML validation of `web.xml` file and TLD files (OC4J or JServ)

  Validation of `web.xml` is disabled by default but can be enabled. Validation of TLD files is enabled by default but can be disabled.

  In "JSP Configuration Parameters" on page 3-9, see the descriptions of the `xml_validate` and `no_tld_xml_validate` parameters.

- mode flag for extra imports (OC4J or JServ)

  Use this to automatically import certain Java packages beyond the JSP defaults.

  In "JSP Configuration Parameters" on page 3-9, see the description of the `extra_imports` parameter.

- "well-known" location for sharing tag libraries (OC4J or JServ)

  You can specify a directory where tag library JAR files can be placed for sharing across multiple Web applications.

  In "JSP Configuration Parameters" on page 3-9, see the description of the `well_known_taglic_loc` parameter.

- configurable JSP timeout

  You can specify a timeout value for JSP pages, after which a page is removed from memory if it has not been requested again. See "OC4J Configuration Parameters for JSP" on page 3-21.

The following features have been supported since the OC4J 9.0.2 implementation:

- mode switch for automatic page recompilation and class reloading (OC4J only)

  You have a choice of: 1) running JSP pages without any automatic reloading of classes or recompilation of JSP pages; 2) automatically reloading any classes that are used by the JSP page and have changed; or 3) automatically recompiling any JSP pages that have changed, as well as reloading any classes that have changed.

  In "JSP Configuration Parameters" on page 3-9, see the description of the `main_mode` parameter.

- tag handler instance pooling (OC4J or JServ)

  To save time in tag handler creation and garbage collection, you can optionally enable pooling of tag handler instances. They are pooled in `application` scope. You can use different settings in different pages, or even in different sections of the same page. See "Disabling or Enabling Runtime or Compile-Time Tag Handler Reuse" on page 8-38.

- output mode for null output (OC4J only)

  The OC4J configuration parameter `jsp-print-null` enables you to print an empty string instead of the default "null" string for null output from a JSP page.

  See "JSP Configuration in OC4J" on page 3-8.

- single-threaded-model JSP instance pooling (OC4J only)

  For single-threaded (non-thread-safe) JSP pages, page instances are pooled. There is no switch for this feature—it is always enabled.

Portability Across Servlet Environments

The JSP container is provided as a component of OC4J, but is portable to other environments. Because the OC4J JSP container itself emulates certain required servlet features, this portability extends to older servlet environments, in particular the Apache JServ servlet 2.0 environment. (Generally, a servlet 2.3 environment is required in order to support JSP 1.2 compliance.)

The servlet 2.0 specification was limited in that it provided only a single servlet context for each Java virtual machine, instead of a servlet context for each application. The OC4J JSP servlet emulation allows a full application framework in a servlet 2.0 environment, including providing applications with distinct `ServletContext` and `HttpSession` objects.

Because of this extended functionality, the OC4J JSP container is not limited by the underlying servlet environment.

In addition to JServ 1.1, the OC4J JSP container has been tested with Tomcat 3.1 (servlet 2.2) from the Apache Software Foundation, and JSWDK 1.0 (JavaServer Web Developer's Kit, servlet 2.1) from Sun Microsystems.

Role of the Oracle HTTP Server and mod_oc4j

Oracle HTTP Server, powered by the Apache Web server, is included with Oracle9*i* Application Server as the HTTP entry point for Web applications, particularly in a production environment. By default, it is the front-end for all OC4J processes—client requests go through Oracle HTTP Server first.

When the Oracle HTTP Server is used, dynamic content is delivered through various Apache *mod* components provided either by the Apache Software Foundation or by Oracle. Static content is typically delivered from the file system, which is more efficient in this case. 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.

Oracle9*i*AS supplies the `mod_oc4j` Apache mod, which is used for communication between the Oracle HTTP Server and OC4J. It routes requests from the Oracle HTTP Server to OC4J processes, and forwards responses from OC4J processes to Web clients.

Communication is through the Apache JServ protocol (AJP). AJP was chosen over HTTP because of a variety of AJP features allowing faster communication, including use of binary format and more efficient processing of message headers.

The following features are provided with `mod_oc4j`:

- load balancing capabilities across many back-end OC4J clusters
- stateless session routing of stateful servlets

  This is accomplished through enhanced use of cookies. Routing information is maintained in the cookie itself to ensure that stateful servlets are always routed to the same OC4J JVM.
- high availability

  A `mod_oc4j` module can restart an OC4J instance automatically, if necessary.

Notes:

- Oracle9*i*AS also includes `mod_jserv`, from Apache, for the JServ servlet environment. This feature is documented in Appendix B, "The Apache JServ Environment". Additional Apache mod components provided with Oracle9*i*AS are not relevant for JSP applications.
- It is possible to bypass the Oracle HTTP Server and access OC4J directly through its own Web listener, which is convenient for development or basic testing. And for OC4J standalone, Oracle HTTP Server is not available. For information about port configuration and default settings, see the *Oracle9iAS Containers for J2EE User's Guide*. For an introduction to OC4J standalone, see "OC4J Standalone" on page 2-5.

Oracle9*i* JDeveloper JSP Support

Visual Java programming tools now typically support JSP coding. In particular, Oracle9*i* JDeveloper supports JSP development and includes the following features:

- integration of the OC4J 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 "Application Deployment with Oracle9i JDeveloper" on page 7-36 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 the JDeveloper online help, or to the following site on the Oracle Technology Network:

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

For an overview of JSP tag libraries provided with JDeveloper, see the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Overview of Oracle Value-Added Features

OC4J value-added features for JSP pages can be grouped into three major categories:

- features implemented through custom tag libraries, custom JavaBeans, or custom classes that are generally portable to other JSP environments
- features that are Oracle-specific
- features supporting caching technologies

The rest of this section provides feature overviews in each of these areas, plus a brief summary of Oracle support for the JavaServer Pages Standard Tag Library (JSTL). JSTL support is summarized more fully in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Overview of Tag Libraries and Utilities Provided with OC4J

This section provides an overview of extended OC4J JSP features that are implemented through standards-compliant custom tag libraries, custom JavaBeans, and other classes. These features are fully documented in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*. Here is a summary list:

- extended types implemented as JavaBeans that can have a specified scope
- `JspScopeListener` for event-handling
- integration with XML and XSL
- data-access JavaBeans
- data-access tag library
- the JSP Markup Language (JML) custom tag library, which reduces the level of Java proficiency required for JSP development
- Oracle Personalization tag library
- Web services tag library
- additional utility tags and JavaBeans for uploading files, downloading files, sending e-mail from within an application, using EJBs, and using miscellaneous utilities

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, you cannot declare them 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, OC4J provides the `JmlBoolean`, `JmlNumber`, `JmlFPNumber`, and `JmlString` JavaBean classes in package `oracle.jsp.jml` to wrap the most common Java types.

JspScopeListener for Event-Handling

OC4J 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. You can integrate the Oracle `JspScopeListener` with `HttpSessionBindingListener` to handle session-based events, as well as page-based, request-based, and application-based events.

Tags Supporting XML

OC4J provides standard JSP XML support as prescribed by the JSP 1.2 specification and described in Chapter 5, "JSP XML Support". In addition, OC4J offers extended support through XML-related custom tags.

There are special tags to specify that all or part of a JSP page should be transformed through an XSL stylesheet before it is output. Input can be from the tag body or from an XML DOM object, and output can be to an XML DOM object to the

browser. You can use these tags multiple times in a single JSP page if you want to specify different style sheets for different portions of the page.

There is additional XML support as well:

- There is a utility tag to convert data from an input stream to an XML DOM object.
- There are several tags, for such things as caching and SQL operations, that now can take XML objects as input or send them as output.
- As of the JSP 1.2 specification, any compliant JSP container, including the OC4J JSP container, supports JSP XML syntax as an alternative to traditional syntax. See "Details of JSP XML Documents" on page 5-4.

> **Note:** The custom XML tag library provided with OC4J pre-dates the JavaServer Pages Standard Tag Library (JSTL) and has areas of duplicate functionality. Going forward, for standards compliance, it is advisable to use JSTL instead of the custom libraries as a general rule. Oracle is not desupporting the existing tags, however. For features in the custom library that are not yet available in JSTL, where there seems to be general usefulness, Oracle will try in the future to have the features adopted into the JSTL standard as appropriate.

Custom Data-Access JavaBeans

OC4J supplies a set of custom JavaBeans for database access. The following beans are provided in the `oracle.jsp.dbutil` package:

- `ConnBean`—Open a database connection. This bean also supports data sources and connection pooling.
- `ConnCacheBean`—Use the Oracle connection caching implementation for database connections.
- `DBBean`—Execute a database query.
- `CursorBean`—This bean provides general DML support for queries; `UPDATE`, `INSERT`, and `DELETE` statements; and stored procedure calls.

Custom Data-Access Tag Library

OC4J provides a custom SQL tag library for database access. The following tags are provided:

- `dbOpen`—Open a database connection. This tag also supports data sources and connection pooling.
- `dbClose`—Close a database connection.
- `dbQuery`—Execute a database query.
- `dbCloseQuery`—Close the cursor for a query.
- `dbNextRow`—Process the rows of a result set.
- `dbExecute`—Execute any SQL statement (DML or DDL).
- `dbSetParam`—Set a parameter to bind into a `dbQuery` or `dbExecute` tag.
- `dbSetCookie`—Set a cookie.

> **Note:** The custom SQL tag library provided with OC4J pre-dates the JavaServer Pages Standard Tag Library (JSTL) and has areas of duplicate functionality. Going forward, for standards compliance, it is advisable to use JSTL instead of the custom libraries as a general rule. Oracle is not desupporting the existing tags, however. For features in the custom library that are not yet available in JSTL, where there seems to be general usefulness, Oracle will try in the future to have the features adopted into the JSTL standard as appropriate.

JSP Markup Language (JML) Custom Tag Library

Although the Sun Microsystems *JavaServer Pages Specification, Version 1.2* 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.

OC4J provides custom tags as an alternative—the JSP Markup Language (JML). The Oracle JML 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, flow control, 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>
```

Notes:

- The custom JML tag library provided with OC4J pre-dates the JavaServer Pages Standard Tag Library (JSTL) and has areas of duplicate functionality. Going forward, for standards compliance, it is advisable to use JSTL instead of the custom libraries as a general rule. Oracle is not desupporting the existing tags, however. For features in the custom library that are not yet available in JSTL, where there seems to be general usefulness, Oracle will try in the future to have the features adopted into the JSTL standard as appropriate.
- Oracle JSP implementations 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.

Oracle9*i*AS Personalization Tag Library

Web site personalization is a mechanism to personalize recommendations to users of a site, based on behavioral and demographic data. Recommendations are made in real-time, during a user's Web session. User behavior is saved to a database repository for use in building models for predictions of future user behavior.

Oracle9*i*AS Personalization uses data mining algorithms in the Oracle database to choose the most relevant content available for a user. Recommendations are calculated by an Oracle9*i*AS Personalization recommendation engine, using typically large amounts of data regarding past and current user behavior. This is superior to other approaches that rely on common-sense heuristics and require manual definition of rules in the system.

The Oracle9*i*AS Personalization tag library brings this functionality to a wide audience of JSP developers for use in HTML, XML, or JavaScript pages. The tag interface is layered on top of the lower level Java API of the recommendation engine.

Web Services Tag Library

The Web services tag library provided with OC4J enables developers to conveniently create JSP pages for Web service client applications. The implementation uses a SOAP-based, RPC-style mechanism. A client application would access a Web Services Definition Language (WSDL) document, then use the WSDL information to access the operations of a Web service.

The tag library uses Oracle9*i*AS Web Services and the Oracle implementation of the dynamic invocation API, described in the *Oracle9iAS Web Services Developer's Guide*. When a client application acquires a WSDL document at runtime, the dynamic invocation API is the vehicle for invoking any SOAP operation described in the WSDL document.

JSP Utility Tags

OC4J provides utility tags to accomplish the following from within Web applications:

- sending e-mail messages
- uploading and downloading files
- using EJBs
- using miscellaneous utilities

For sending e-mail messages, you can use the `sendMail` tag or the `oracle.jsp.webutil.email.SendMailBean` JavaBean.

For uploading files, you can use the `httpUpload` tag or the `oracle.jsp.webutil.fileaccess.HttpUploadBean` JavaBean. For downloading, there is the `httpDownload` tag or the `HttpDownloadBean` JavaBean.

For using EJBs, there are tags to create a home instance, create an EJB instance, and iterate through a collection of EJBs.

There are also utility tags for displaying a date, displaying an amount of money in the appropriate currency, displaying a number, iterating through a collection, evaluating and including the tag body depending on whether the user belongs to a specified role, and displaying the last modification date of the current file.

Overview of Oracle-Specific Features

This section provides an overview of Oracle-specific programming extensions supported by the OC4J JSP container:

- support for SQLJ, a standard syntax for embedding SQL statements directly into Java code
- *global includes*, a mechanism to automatically statically include a file or files in multiple pages
- Dynamic Monitoring Service support for performance measurements
- enhanced application framework and globalization support for servlet 2.0 environments

SQLJ Support

Dynamic server pages commonly include data extracted from databases. JSP developers typically 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 OC4J JSP container supports SQLJ programming in JSP scriptlets.

SQLJ statements are indicated by the `#sql` token. You can trigger the JSP translator to invoke the Oracle SQLJ translator by using the file name extension `.sqljsp` for the JSP source code file, or by specifying `language="sqlj"` in a `page` directive.

For more information, see "JSP Support for Oracle SQLJ" on page 4-15.

Global Includes

The OC4J JSP container provides a feature called *global includes*. You can use this feature to specify one or more files to statically include into JSP pages in (or under) a specified directory, through virtual JSP `include` directives. During translation, the JSP container looks for a configuration file, `/WEB-INF/ojsp-global-include.xml`, that specifies the included files and the directories for the pages.

This enhancement is particularly useful in migrating applications that had used `globals.jsa` or `translate_params` functionality in previous Oracle JSP releases. For more information, see "Oracle JSP Global Includes" on page 7-9.

Support for Dynamic Monitoring Service

The Dynamic Monitoring Service (DMS) adds performance-monitoring features to a number of Oracle9*i*AS components, including OC4J. The goal of DMS is to provide information about runtime behavior through built-in performance measurements, so that users can diagnose, analyze, and debug any performance problems. DMS provides this information in a package that can be used at any time, including during live deployment. Data are published through HTTP and can be viewed with a browser.

The OC4J JSP container supports DMS features, calculating relevant statistics and providing information to DMS servlets such as the spy servlet and monitoring agent. Statistics include the following (using averages, maximums, and minimums, as applicable):

- processing time of HTTP request
- processing time of JSP service method
- number of JSP instances created or available
- number of active JSP instances

(Counts of JSP instances are applicable only for single-threaded situations, where `isThreadSafe` is set to `false` in a `page` directive.)

Standard configuration for these servlets is in the OC4J `global-web-application.xml` configuration file. Use the Enterprise Manager to access DMS, display DMS information, and, as appropriate, alter DMS configuration.

Also see the *Oracle9i Application Server Performance Guide*, which contains precise definitions of the JSP metrics and instructions for viewing and analyzing them.

Enhanced Servlet 2.0 Support

OC4J supports special features for the servlet 2.0 JServ environment. It is highly advisable to migrate to the OC4J servlet 2.3 environment as soon as practical, but in the meantime, be aware of the following:

- an enhanced application framework for servlet 2.0 environments

  See "JSP Application and Session Support for JServ" on page B-32.

- extended globalization support for servlet 2.0 environments

  See "Multibyte Parameter Encoding in JServ" on page B-24.

The referenced sections include information for migration to OC4J.

Overview of Tags and API for Caching Support

Faced with Web performance challenges, e-businesses must invest in more cost-effective technologies and services to improve the performance of their Internet sites. *Web caching*, the caching of both static and dynamic Web content, is a key technology in this area. Benefits of Web caching include performance, scalability, high availability, cost savings, and network traffic reduction.

OC4J provides the following support for Web caching technologies:

- the JESI tag set for Edge Side Includes (ESI), an XML-style markup language that allows dynamic content assembly away from the Web server

  The Oracle9*i*AS Web Cache provides an ESI engine.

- a tag set and servlet API for the Web Object Cache, an application-level cache that is embedded and maintained within a Java Web application

  The Web Object Cache uses the Oracle9*i* Application Server Java Object Cache as its default repository.

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

Support for the JavaServer Pages Standard Tag Library

With Oracle9*i*AS release 2 (9.0.3), the OC4J JSP product supports the JavaServer Pages Standard Tag Library (JSTL), as specified in the Sun Microsystems *JavaServer Pages Standard Tag Library, Version 1.0* specification.

JSTL is intended as a convenience for JSP page authors who are not familiar or not comfortable with scripting languages such as Java. Historically, scriptlets have been used in JSP pages to process dynamic data. With JSTL, the intent is for JSTL tag usage to replace the need for scriptlets.

Key JSTL features include the following:

- JSTL expression language (EL)

  The expression language further simplifies the code required to access and manipulate application data, making it possible to avoid request-time attributes as well as scriptlets.

- core tags for expression language support, conditional logic and flow control, iterator actions, and accessing URL-based resources

- tags for XML processing, flow control, and XSLT transformations

- SQL tags for database access
- tags for i18n-capable internationalization and formatting

  (The term "i18n" refers to an internationalization standard.)

Tag support is broken into four JSTL sublibraries according to these functional areas.

For a more complete summary of JSTL support, you can refer to the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*. For complete information about JSTL, refer to the specification at the following location:

`http://www.jcp.org/aboutJava/communityprocess/first/jsr052/index.html`

Note: The custom JML, XML, and SQL tag libraries provided with OC4J pre-date JSTL and have areas of duplicate functionality. Going forward, for standards compliance, it is advisable to use JSTL instead of the custom libraries as a general rule. Oracle is not desupporting the existing tags, however. For features in the custom libraries that are not yet available in JSTL, where there seems to be general usefulness, Oracle will try in the future to have the features adopted into the JSTL standard as appropriate.

3

Getting Started

This chapter covers basic issues in your JSP environment, including key support files, key OC4J configuration files, and configuration of the JSP container. It also discusses initial considerations such as application root functionality, classpath functionality, security issues, and file naming conventions.

Before getting started, it is assumed that you can do the following on your system:

- run Java
- run a Java compiler (typically the standard `javac`)
- run an HTTP servlet

The following topics are covered here:

Notes: 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.

Some Initial Considerations

This section discusses some initial considerations you should be aware of before you begin coding or using JSP pages:

Application Root Functionality

The servlet 2.2 and 2.3 specifications provide for each Web 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 Web application. This is the *application root*. Each Web application has its own application root. For a Web application in a servlet 2.2 or 2.3 environment, servlets, JSP pages, and static files such as HTML files are all based out of this application root. (By contrast, in servlet 2.0 environments the application root for servlets and JSP pages is distinct from the doc root for static files.)

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. The *servlet path* is defined in the application `web.xml` file. The `<servlet>` element within `web.xml` associates a servlet class with a servlet name. The `<servlet-mapping>` element within `web.xml` associates a URL pattern with a named servlet. When a servlet is executed, the servlet container will compare a specified URL pattern with known servlet paths, and pick the servlet path that matches. See the *Oracle9iAS Containers for J2EE Servlet Developer's Guide* for more information.

For example, consider an application with the application root `/home/dir/mybankapp/mybankwebapp`, which is mapped to the context path `/mybank`. Further assume the application includes a servlet whose servlet path is `loginservlet`. You can invoke this servlet 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/mybankapp/mybankwebapp/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/dir/defaultapp/defaultwebapp`, and a servlet with the servlet path `myservlet` uses the default context. Its URL would be as follows:

```
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/dir/defaultapp/defaultwebapp/dir2/def.html`:

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

Classpath Functionality

The JSP container uses standard locations on the Web server to look for translated JSP pages, as well as `.class` files and `.jar` files for any required classes (such as JavaBeans). The container will find files in these locations without any Web server classpath configuration, and has the ability to automatically reload classes in these locations, depending on configuration settings.

The locations for dependency classes are as follows and are relative to the application root:

```
/WEB-INF/classes/...
/WEB-INF/lib
```

The location for JSP page implementation classes (translated pages) is as follows:

```
.../_pages/...
```

The `/WEB-INF/classes` directory is for individual Java `.class` files. You should store these classes 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

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 `_pages` directory is under the J2EE home directory in OC4J. In Oracle9*i*AS, OC4J directory paths are configurable; in OC4J standalone, by default it would be as follows:

```
[Oracle_Home]/j2ee/home/application-deployments/app-name/web-app-name/temp
```

The *app-name* is determined through an `<application>` element in the OC4J `server.xml` file; the *web-app-name*, which corresponds to the WAR file name, is mapped to the *app-name* through a `<web-app>` element in the OC4J `default-web-site.xml` file (or other Web site XML file). See the *Oracle9iAS Containers for J2EE User's Guide* and the *Oracle9iAS Containers for J2EE Servlet Developer's Guide* for more information.

Generated page implementation classes for translated JSP pages are placed in subdirectories under the `_pages` directory according to the locations of the original `.jsp` files. See "Generated Files and Locations" on page 7-6 for information.

Important: Implementation details, such as the location of the `_pages` directory, are subject to change in future releases.

JSP Security Considerations

With respect to application security, be aware of the following:

- Verify that the `debug_mode` parameter has its default `false` setting if you want to suppress the display of the physical file path when nonexistent JSP files are requested. This parameter is described in "JSP Configuration in OC4J" on page 3-8.
- There are additional considerations for JServ environments. See "JSP Security Considerations in JServ" on page B-24.

Default Package Imports

Beginning with Oracle9*i*AS release 2 (9.0.3), the OC4J JSP container by default imports the following packages into any JSP page, in accordance with the JSP specification. No `page` directive `import` settings are required:

```
javax.servlet.*
javax.servlet.http.*
javax.servlet.jsp.*
```

In previous releases, the following packages were also imported by default:

```
java.io.*
java.util.*
java.lang.reflect.*
java.beans.*
```

The default list of packages to import was reduced to minimize the chance of a conflict between any unqualified class name you might use and a class by the same name in any of the imported packages.

However, this might result in migration problems for applications you have used with previous versions of OC4J. Such applications might no longer compile successfully. If you need imports beyond the default list, you have two choices:

- Specify additional package names or fully qualified class names in one or more `page` directive `import` settings. For more information, see the `page` directive under "Directives" on page 1-7, and see "Page Directive import Settings Are Cumulative" on page 6-12.

  For multiple pages, you can accomplish this through *global includes* functionality. See "Oracle JSP Global Includes" on page 7-9.

- Specify additional package names or fully qualified class names through the JSP `extra_imports` configuration parameter, or by using the `ojspc -extraImports` option for pre-translation. Syntax varies between OC4J configuration parameter settings, JServ configuration parameter settings, and `ojspc` option settings, so refer to the following as appropriate:

  – "JSP Configuration Parameters" on page 3-9

  – "Option Descriptions for ojspc" on page 7-20

  – "JSP Configuration Parameters for JServ" on page B-4

JSP File Naming Conventions

The file name extension `.jsp` for JSP pages is required by the Sun Microsystems *Java Servlet Specification, Version 2.3*. The servlet 2.3 specification does not, however, distinguish between complete, translatable pages and page fragments, such as files brought in through an `include` directive (as described in "Directives" on page 1-7).

The JSP 1.2 specification recommends the following:

- Use the `.jsp` extension for top-level pages—pages that are translatable on their own.
- Do *not* use `.jsp` for page fragments brought in through `include` directives. No particular extension is mandated for such files, but `.jspf` or `.jsf` is recommended.

Key Support Files Provided with OC4J

This section summarizes JAR and ZIP files that are used by the JSP container or JSP applications. These files are installed on your system and into your classpath with OC4J.

> **Note:** Some of the `.jar` files here, such as for JDBC and SQLJ, also have `.zip` alternatives.

- `ojsp.jar`—classes for the JSP container
- `ojsputil.jar`—classes for tag libraries and utilities provided with OC4J
- `xmlparserv2.jar`—for XML parsing; required for the `web.xml` deployment descriptor and any tag library descriptor files and XML-related tag functionality
- `xsu12.jar` / `xsu111.jar`—for XML functionality on the client (for JDK 1.2.x or higher, or 1.1.x, respectively)
- `ojdbc14.jar` / `classes12.jar` / `classes111.jar`—for the Oracle JDBC drivers (for JDK 1.4, 1.2 or higher, or 1.1, respectively)
- `translator.jar`—for the Oracle SQLJ translator
- `runtime12.jar` / `runtime12ee.jar` / `runtime11.jar` / `runtime.jar` / `runtime-nonoracle.jar`—for the Oracle SQLJ runtime (respectively: for JDK 1.2.x or higher with Oracle9*i* JDBC, JDK 1.2.x or higher enterprise edition with Oracle9*i* JDBC, JDK 1.1.x with Oracle9*i* JDBC, any 1.1.x or higher JDK with any Oracle JDBC version, or any JDK environment with non-Oracle JDBC drivers)
- `jndi.jar`—for JNDI service for lookup of resources such as JDBC data sources and Enterprise JavaBeans
- `jta.jar`—for the Java Transaction API

There are also files relating to particular areas, such as particular tag libraries. These include the following:

- `mail.jar`—for e-mail functionality within applications (standard `javax.mail` package)
- `activation.jar`—Java activation files for e-mail functionality
- `cache.jar`—for the Oracle9*i* Application Server Java Object Cache (which is the default back-end repository for the OC4J Web Object Cache)

JSP Configuration in OC4J

This section covers the following topics regarding configuration of the JSP environment:

- JSP Container Setup
- JSP Configuration Parameters
- OC4J Configuration Parameters for JSP

Notes:

- Discussion of OC4J configuration files and configuration parameters, and how to update them manually, generally assumes an OC4J standalone environment. This is typical during development. For information about JSP configuration through Oracle Enterprise Manager in an Oracle9*i*AS environment, such as for production deployment, see "JSP Configuration in Oracle Enterprise Manager" on page 3-25.
- For non-OC4J environments, including JServ, use the old `oracle.jsp.JspServlet` front-end servlet instead of the `oracle.jsp.runtimev2.JspServlet` version. See "Getting Started in a JServ Environment" on page B-2.

JSP Container Setup

The JSP container is appropriately preconfigured in OC4J. The following settings appear in the OC4J `global-web-application.xml` file to map the name of the front-end JSP servlet, and to map the appropriate file name extensions for JSP pages:

```
<orion-web-app ... >
   ...
   <web-app>
       ...
      <servlet>
         <servlet-name>jsp</servlet-name>
         <servlet-class>oracle.jsp.runtimev2.JspServlet</servlet-class>
         ...
            init_params
         ...
      </servlet>
```

```
      ...
      <servlet-mapping>
         <servlet-name>jsp</servlet-name>
         <url-pattern>/*.jsp</url-pattern>
      </servlet-mapping>
      <servlet-mapping>
          <servlet-name>jsp</servlet-name>
          <url-pattern>/*.JSP</url-pattern>
       </servlet-mapping>
       <servlet-mapping>
           <servlet-name>jsp</servlet-name>
           <url-pattern>/*.sqljsp</url-pattern>
       </servlet-mapping>
      <servlet-mapping>
           <servlet-name>jsp</servlet-name>
           <url-pattern>/*.SQLJSP</url-pattern>
      </servlet-mapping>

      ...
   </web-app>
   ...
</orion-web-app>
```

See the *Oracle9iAS Containers for J2EE Servlet Developer's Guide* for more information about the `global-web-application.xml` file.

JSP Configuration Parameters

The JSP front-end servlet in OC4J, `oracle.jsp.runtimev2.JspServlet`, supports a number of configuration parameters to control JSP operation. This section describes those parameters. There is a summary table, followed by more complete descriptions, and documentation of how to set them in the OC4J `global-web-application.xml` or `orion-web.xml` file.

JSP Configuration Parameter Summary Table

Table 3–1 summarizes the configuration parameters supported by `JspServlet`. For each parameter, the table notes any equivalent `ojspc` translation options for pages you are pre-translating, and whether the parameter is for runtime or compile-time use.

Notes: See "The ojspc Pre-Translation Utility" on page 7-13 for a description of the `ojspc` options.

Table 3–1 JSP Configuration Parameters, OC4J Environment

Parameter	Related ojspc Options	Description	Default	Runtime / Compile-Time
check_page_scope	(n / a)	Set this boolean to `true` to enable page-scope checking by `JspScopeListener` (OC4J only).	false	runtime
debug_mode	(n / a)	Set this boolean to `true` to print the stack trace when a runtime exception occurs.	false	runtime
emit_debuginfo	-debug	Set this boolean to `true` to generate a line map to the original `.jsp` file for debugging (for development).	false	compile-time
external_resource	-extres	Set this boolean to `true` to place all static content of the page into a separate Java resource file during translation.	false	compile-time
extra_imports	-extraImports	Use this to add imports beyond the JSP defaults.	null	compile-time
forgive_dup_dir_attr	-forgiveDupDirAttr	Set this boolean to `true` to avoid JSP 1.2 translation errors if you have duplicate settings for the same directive attribute within a single JSP translation unit.	false	compile-time
javaccmd	-noCompile	Use this if you want to specify a `javac` command line, or if you want to specify an alternative Java compiler, optionally with command-line settings (for development). If you specify an alternative compiler, it will be spawned in a separate JVM. Use a `null` setting for the JDK `javac` with default settings.	null	compile-time

Table 3–1 JSP Configuration Parameters, OC4J Environment (Cont.)

Parameter	Related ojspc Options	Description	Default	Runtime / Compile-Time
main_mode	(n/a)	This determines whether classes are automatically reloaded or JSP pages are automatically recompiled, in case of changes. Possible settings are `justrun`, `reload`, and `recompile`.	recompile	runtime
no_tld_xml_validate	-noTldXmlValidate	Set this boolean to `true` to *not* perform XML validation of TLD files. By default, validation of TLD files is performed.	false	compile-time
old_include_from_top	-oldIncludeFromTop	Set this boolean to `true` for page locations in nested `include` directives to be relative to the top-level page, for backward compatibility with behavior prior to Oracle9*i*AS release 2.	false	compile-time
precompile_check	(n/a)	Set this boolean to `true` to check the HTTP request for a standard `jsp_precompile` setting.	false	runtime
reduce_tag_code	-reduceTagCode	Set this boolean to `true` for further reduction in the size of generated code for custom tag usage.	false	compile-time
req_time_introspection	-reqTimeIntrospection	Set this boolean to `true` to enable request-time JavaBean introspection whenever compile-time introspection is not possible.	false	compile-time
sqljcmd	-S	Use this if you want to specify a SQLJ command line, or if you want to specify an alternative SQLJ translator, optionally with command-line settings (for development). If you specify an alternative translator, it will be spawned in a separate JVM. Use a `null` setting for the Oracle SQLJ version provided with Oracle9*i*AS, with its default option settings.	null	compile-time

Table 3–1 JSP Configuration Parameters, OC4J Environment (Cont.)

Parameter	Related ojspc Options	Description	Default	Runtime / Compile-Time
static_text_in_chars	-staticTextInChars	Set this boolean to `true` to instruct the JSP translator to generate static text in JSP pages as characters instead of bytes.	false	compile-time
tags_reuse_default	(n/a)	This specifies the mode for JSP tag handler reuse: `runtime` for the runtime model, `compiletime` or `compiletime_with_release` for the compile-time model, or `none` to disable tag handler reuse.	runtime	either
well_known_taglib_loc	(n/a)	This specifies a directory where tag library JAR files can be placed for sharing across multiple Web applications. The default location is `j2ee/home/jsp/lib/taglib/` under the `[Oracle_Home]` directory.	(See description column.)	compile-time
xml_validate	-xmlValidate	Set this boolean to `true` to perform XML validation of the `web.xml` file. By default, validation of `web.xml` is *not* performed.	false	compile-time

JSP Configuration Parameter Descriptions

This section describes the JSP configuration parameters for OC4J in more detail.

check_page_scope (boolean; default: `false`)

For OC4J environments, set this parameter to `true` to enable Oracle-specific page-scope checking by the `JspScopeListener` utility. It is `false` by default for performance reasons.

This parameter is not relevant for non-OC4J environments. For JServ, Oracle-specific page-scope checking is always enabled. For other environments, the Oracle-specific implementation is not used and you must use the `checkPageScope` custom tag for `JspScopeListener` page-scope functionality. See "JspScopeListener for Event-Handling" on page 2-14 for a brief overview of this utility. See *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference* for detailed information.

debug_mode (boolean; default: `false`)

Use the default `true` setting to print a stack trace whenever a runtime exception occurs. A `false` setting disables this feature.

Important: When `debug_mode` is `false` and a file is not found, the full path of the missing file is *not* displayed. This is an important security consideration if you want to suppress the display of the physical file path when non-existent JSP files are requested.

---

emit_debuginfo (boolean; default: `false`)

During development, set this flag to `true` to instruct the JSP translator to generate a line map to the original `.jsp` file for debugging. Otherwise, lines will be mapped to the generated page implementation class `.java` file.

Notes:

- Oracle9*i* JDeveloper enables `emit_debuginfo`.
- For pre-translating pages, the `ojspc -debug` option is equivalent.

---

external_resource (boolean; default: `false`)

Set this flag to `true` to instruct the 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*AS release 2, translation of `MyPage.jsp`, for example, would create `_MyPage.res` in addition to normal output. (The exact implementation might change in future releases.)

The translator places the resource file into 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. For more information, see "Workarounds for Large Static Content in JSP Pages" on page 6-7.

Note: For pre-translating pages, the `ojspc -extres` option is equivalent.

---

extra_imports (import list; default: `null`)

As described in "Default Package Imports" on page 3-5, as of Oracle9*i*AS release 2 (9.0.3), the OC4J JSP container has a smaller default list of packages that are imported into each JSP page. This is in accordance with the JSP specification. You can avoid updating your code, however, by specifying package names or fully qualified class names for any additional imports through the `extra_imports` configuration parameter. See "Setting JSP Configuration Parameters in OC4J" on page 3-20 for general syntax, and be aware that the names can be either comma-delimited or space-delimited. Either of the following is okay, for example:

```
<init-param>
   <param-name>extra_imports</param-name>
   <param-value>java.util.* java.beans.*</param-value>
</init-param>
```

or:

```
<init-param>
   <param-name>extra_imports</param-name>
   <param-value>java.util.*,java.beans.*</param-value>
</init-param>
```

Note:

- For pre-translating pages, the `ojspc -extraImports` option is equivalent.
- As an alternative to using `extra_imports`, you can use global includes. See "Oracle JSP Global Includes" on page 7-9.

---

forgive_dup_dir_attr (forgive duplicate directive attributes; default: `false`)

Set this boolean to `true` to avoid translation errors in JSP 1.2 (or higher) if you have duplicate settings for the same directive attribute within a single JSP translation unit (a JSP page plus anything it includes through `include` directives).

The JSP 1.2 specification directs that a JSP container must verify that directive attributes, with the exception of the `page` directive `import` attribute, are not set

more than once each within a single JSP translation unit. See "Duplicate Settings of Page Directive Attributes Are Disallowed" on page 6-11 for more information.

The JSP 1.1 specification does *not* specify such a limitation. OC4J offers the `forgive_dup_dir_attr` parameter for backward compatibility.

Note: For pre-translating pages, the `ojspc -forgiveDupDirAttr` option is equivalent.

---

javaccmd (compiler executable and options; default: `null`)

This parameter is useful during development 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 JSP container

Specifying an alternative compiler results in that executable being spawned as a separate process in a separate JVM, instead of within the same JVM as the JSP container. 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.

For example, set `javaccmd` to the value `javac -verbose` to run the compiler in verbose mode.

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.
- For 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 any desired compiler.

---

main_mode (mode switch for reloading or recompilation; default: `recompile`)

This is a flag to direct the mode of operation of the JSP container, particularly for automatic recompilation of JSP pages and reloading of Java classes that have changed.

Here are the supported settings:

- `justrun`—The runtime dispatcher will not perform any timestamp checking, so there is no recompilation of JSP pages or reloading of Java classes. This mode is the most efficient mode for a deployment environment, where code will not change.
- `reload`—The dispatcher will check if any classes have been modified since loading, including translated JSP pages, JavaBeans invoked from pages, and any other dependency classes.
- `recompile` (default)—The dispatcher will check the timestamp of the JSP page, retranslate it and reload it if has been modified since loading, and execute all `reload` functionality as well.

no_tld_xml_validate (disabling of XML validation of TLD files; default: `false`)

Set this to `true` to disable XML validation of the tag library descriptor (TLD) files of the application. By default, validation of TLD files is performed.

See "Overview of TLD File Validation and Features" on page 8-8 for related information.

Note: For pre-translating pages, the `ojspc -noTldXmlValidate` option is equivalent.

---

old_include_from_top (backward compatibility for `include`; default: `false`)

This is for backward compatibility with Oracle JSP versions prior to Oracle9*i*AS release 2, for functionality of `include` directives. If set to `true`, page locations in nested `include` directives are relative to the top-level page. If set to `false`, page locations are relative to the immediate parent page. This complies with the JSP 1.2 specification.

Note: For pre-translating pages, the `ojspc -oldIncludeFromTop` option is equivalent.

---

precompile_check (`jsp_precompile` checking; default: `false`)

Set this to `true` to check the HTTP request for a standard `jsp_precompile` setting. If `precompile_check` is `true` and the request enables `jsp_precompile`, then the JSP page will be pre-translated only, without execution. Setting `precompile_check` to `false` improves performance and ignores any `jsp_precompile` setting in the request.

For more information about `jsp_precompile`, see "Standard JSP Pre-Translation without Execution" on page 7-40, and the Sun Microsystems *JavaServer Pages Specification, Version 1.2*.

reduce_tag_code (flag for size reduction of custom tag code; default: `false`)

The Oracle JSP implementation reduces the size of generated code for custom tag usage, but setting `reduce_tag_code` to `true` results in even further size reduction. There may be performance consequences regarding tag handler reuse, however. See "Tag Handler Code Generation" on page 8-40.

Note: For pre-translating pages, the `ojspc -reduceTagCode` option is equivalent.

---

req_time_introspection (flag for request-time introspection; default: `false`)

A `true` setting enables request-time JavaBean introspection whenever compile-time introspection is not possible. When compile-time introspection is possible and succeeds, this parameter is ignored and there is no request-time introspection.

As an example of a scenario for use of request-time introspection, assume a tag handler returns a generic `java.lang.Object` instance in `VariableInfo` of the tag-extra-info class during translation and compilation, but actually generates more specific objects during request-time (runtime). In this case, if `req_time_introspection` is enabled, the JSP container will delay introspection until request-time. (See "Scripting Variables, Declarations, and Tag-Extra-Info Classes" on page 8-41 for information about use of `VariableInfo`.)

Note: For pre-translating pages, the `ojspc -reqTimeIntrospection` option is equivalent.

---

sqljcmd (SQLJ translator executable and options; default: `null`)

This parameter is useful during development in any of the following circumstances:

- if you want to set one or more SQLJ command-line options
- if you want to use a different SQLJ translator, or at least a different version, than the one provided with OC4J
- if you want to run SQLJ in a separate process from the JSP container

Specifying a SQLJ translator executable results in its being spawned as a separate process in a separate JVM, instead of within the same JVM as the JSP container.

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.

For example, to run SQLJ with online semantics checking as user `scott/tiger`, and to generate ISO standard SQLJ code, set `sqljcmd` to the following value:

```
sqljcmd=sqlj -user=scott/tiger -codegen=iso
```

Appropriate SQLJ libraries 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 "Key Support Files Provided with OC4J" on page 3-7.

Notes: For pre-translating pages, the `ojspc -S` option provides related functionality.

---

static_text_in_chars (flag to generate static text as characters; default: `false`)

A `true` setting directs the JSP translator to generate static text in JSP pages as characters, instead of bytes. Enable this flag if your application requires the ability to change the character encoding dynamically during runtime, such as in the following example:

```
<% response.setContentType("text/html; charset=UTF-8"); %>
```

(See "Dynamic Content Type Settings" on page 9-5 for related information.)

The `false` default setting improves performance in outputting static text blocks.

Note: For pre-translating pages, the `ojspc -staticTextInChars` option is equivalent.

---

tags_reuse_default (setting for tag handler reuse; default: `runtime`)

Use this parameter to specify the mode of tag handler reuse (tag handler instance pooling), as follows:

- Use the setting `none` to disable tag handler reuse. (You can override this in any particular JSP page by setting the JSP page context attribute `oracle.jsp.tags.reuse` to a value of `true`.)
- Use the default setting `runtime` to enable the runtime model of tag handler reuse. (You can override this in any particular JSP page by setting the JSP page context attribute `oracle.jsp.tags.reuse` to a value of `false`.)
- Use the setting `compiletime` to enable the compile-time model of tag handler reuse in its basic mode.
- Use the setting `compiletime_with_release` to enable the compile-time model of tag handler reuse in its "with release" mode, where the tag handler `release()` method is called between usages of a given tag handler within a given page.

Note: For backward compatibility, a setting of `true` is also supported and is equivalent to `runtime`, and a setting of `false` is supported and is equivalent to `none`.

See "Disabling or Enabling Runtime or Compile-Time Tag Handler Reuse" on page 8-38 for more information about tag handler reuse.

well_known_taglib_loc (location for shared tag libraries; default: see description)

This specifies a directory where tag library JAR files can be placed for sharing across multiple Web applications. The default value is the following:

```
j2ee/home/jsp/lib/taglib/
```

This is under `[Oracle_Home]` if `[Oracle_Home]` is defined. If `[Oracle_Home]` is not defined, then this default location is under the current directory.

Important: Additional steps are also required for tag library sharing to work. See "Oracle Extension for Tag Library Sharing" on page 8-20.

---

xml_validate (XML validation of `web.xml` file; default: `false`)

Set this to `true` to enable XML validation of the application `web.xml` file. Because the Tomcat reference implementation does not perform XML validation, `xml_validate` is `false` by default.

Note: For pre-translating pages, the `ojspc -xmlValidate` option is equivalent.

Setting JSP Configuration Parameters in OC4J

In an OC4J standalone development environment, you can set JSP configuration parameters in `global-web-application.xml`, `web.xml`, or `orion-web.xml`, inside the `<servlet>` element for the JSP front-end servlet. In the portion of `global-web-application.xml` shown in "JSP Container Setup" on page 3-8, the settings would go where the *`init_params`* place holder appears. (In an Oracle9iAS production environment, you should use Enterprise Manager for configuration. See "JSP Configuration in Oracle Enterprise Manager" on page 3-25.)

The following example lists `<servlet>` element and subelement settings for the JSP front-end servlet. This sample enables the `precompile_check` flag, sets the `main_mode` flag to run without checking timestamps, and runs the Java compiler in verbose mode.

```
<servlet>
   <servlet-name>jsp</servlet-name>
   <servlet-class>oracle.jsp.runtimev2.JspServlet</servlet-class>
   <init-param>
      <param-name>precompile_check</param-name>
      <param-value>true</param-value>
   </init-param>
   <init-param>
      <param-name>main_mode</param-name>
      <param-value>justrun</param-value>
   </init-param>
   <init-param>
      <param-name>javaccmd</param-name>
      <param-value>javac -verbose</param-value>
   </init-param>
</servlet>
```

You can override any settings in the `global-web-application.xml` file with settings in the `web.xml` file for a particular application, and you can make deployment-specific overrides of `web.xml` settings through settings in

orion-web.xml. For information about global-web-application.xml and orion-web.xml, see the *Oracle9iAS Containers for J2EE Servlet Developer's Guide*.

OC4J Configuration Parameters for JSP

There are also OC4J configuration parameters—as opposed to parameters for the JspServlet front-end servlet of the JSP container—which affect JSP pages. This section documents JSP-related attributes of the root <orion-web-app> element of the OC4J global-web-application.xml file or orion-web.xml file. For more information about these files, see the *Oracle9iAS Containers for J2EE Servlet Developer's Guide*.

JSP-Related OC4J Configuration Parameter Descriptions

The following <orion-web-app> attributes, in the OC4J global-web-application.xml file or orion-web.xml file, affect JSP performance and functionality:

- jsp-print-null: Set this flag to "false" to print an empty string instead of the string "null" for null output from a JSP page. The default is "true".
- jsp-timeout: Specify an integer value, in seconds, after which any JSP page will be removed from memory if it has not been requested. This frees up resources in situations where some pages are called infrequently. The default value is 0, for no timeout.

> **Note:** The autoreload-jsp-pages and autoreload-jsp-beans attributes of the <orion-web-app> element are not supported by the OC4J JSP container in Oracle9*i*AS release 2. You can use the JSP main_mode configuration parameter, described in "JSP Configuration Parameter Descriptions" on page 3-12, for equivalent functionality.

Setting JSP-Related OC4J Configuration Parameters

To set configuration values that would apply to all applications in an OC4J instance, use the <orion-web-app> element of the OC4J global-web-application.xml file. To set configuration values for a particular application deployment, overriding settings in global-web-application.xml, use the <orion-web-app> element of the deployment-specific orion-web.xml file.

Here is an example:

```
<orion-web-app ... jsp-print-null="false" ... >
...
</orion-web-app>
```

Note that the `<orion-web-app>` element has numerous attributes and subelements. For a complete discussion, see the *Oracle9iAS Containers for J2EE Servlet Developer's Guide*.

Note: This discussion assumes an OC4J standalone development environment. In an Oracle9*i*AS production environment, you generally must use Enterprise Manager for configuration. However, as of Oracle9*i*AS release 2 (9.0.3), `jsp-print-null` and `jsp-timeout` are not yet supported by the Enterprise Manager JSP Properties Page. Given this fact, any settings must be made through the Enterprise Manager Web Module Advanced Properties Page (which enables you to edit `orion-web.xml`), or directly in the appropriate XML file. In the latter case, you must then run the `dcmctl` utility, which is a command-line alternative to Enterprise Manager. See "OC4J Deployment Features" on page 7-34.

Key OC4J Configuration Files

Be aware of the following key configuration files in the OC4J environment.

Global files for all OC4J applications, in the OC4J configuration files directory:

- `server.xml`—This has an overall `<application-server>` element, with an `<application>` subelement for each J2EE application. Each `<application>` subelement specifies the name of the application and the name and location of its EAR deployment file. The `<application-server>` element specifies the name of the general application source directory, where EAR files are placed for deployment and extracted, and the application deployment directory, where OC4J-specific configuration files are generated. Additionally, there is a `<web-site>` element for the default Web site, and you can add a `<web-site>` element for each additional Web site you want to have on the server.
- `default-web-site.xml` (or `http-web-site.xml` for OC4J standalone, or other Web site XML file as applicable)—This includes a `<web-app>` element for each Web application for the default Web site, mapping the application name to the "Web application name". The Web application name corresponds to the WAR deployment file name. Additional Web site XML files, as specified for additional Web sites in the `server.xml` file, have the same functionality.
- `global-web-application.xml`—This is a global configuration file for OC4J Web applications. It establishes default configurations and includes setup and configuration of the JSP front-end servlet, `JspServlet`.
- `application.xml`—This is another parent configuration file for OC4J applications.
- `data-sources.xml`—This specifies data sources for database connections.

(In Oracle9*i*AS, OC4J directory paths are configurable; in OC4J standalone, the configuration files directory is `j2ee/home/config` by default.)

In addition to the global `application.xml` file, there is a standard `application.xml` file, and optionally an `orion-application.xml` file, for each application. These files are in the application EAR file.

Also, in an application WAR file, which is inside the application EAR file, there is a standard `web.xml` file and optionally an `orion-web.xml` file. These are for application-specific and deployment-specific configuration settings, overriding `global-web-application.xml` settings or providing additional settings as appropriate. The `global-web-application.xml` and `orion-web.xml` files support the same elements, which is a superset of those supported by the `web.xml` file.

If the `orion-application.xml` and `orion-web.xml` files are not present in the archive files, they will be generated during initial deployment, according to settings in the `global-web-application.xml` file.

For additional information, see "Overview of EAR/WAR Deployment" on page 7-34. For complete information about the use of these files, see the *Oracle9iAS Containers for J2EE User's Guide* and the *Oracle9iAS Containers for J2EE Servlet Developer's Guide*.

JSP Configuration in Oracle Enterprise Manager

In an Oracle9*i*AS environment, such as for production deployment, you should perform most OC4J configuration through Enterprise Manager. This includes configuration of the front-end JSP servlet for the OC4J JSP container. The following graphic shows the Enterprise Manager JSP Properties Page for an OC4J instance.

(For information beyond what is described here about using Enterprise Manager to configure OC4J, see the *Oracle9iAS Containers for J2EE User's Guide*. For general information about using Enterprise Manager to manage your Oracle9*i*AS environment, see the *Oracle9i Application Server Administrator's Guide*.)

You can drill down to this page as follows:

1. From the Oracle9*i*AS Application Server Instance Home Page (the main page you reach when you first access Enterprise Manager), select the name of an OC4J instance in the System Components table. Enterprise Manager displays the OC4J Home Page for the OC4J instance.
2. From the OC4J Home Page, select **JSP Container Properties** under Instance Properties in the Administration section of the page.

Configuration Parameters Supported by the JSP Properties Page

Table 3–2 shows the correspondence between JSP container properties shown in the Enterprise Manager JSP Properties Page, and configuration parameters of the JSP container front-end servlet as described in "JSP Configuration Parameters" on page 3-9. See that section for the meanings of the settings.

Possible settings are shown with defaults in bold. Note that Enterprise Manager defaults are appropriate for a production environment, so are not necessarily the same as defaults otherwise, which are appropriate for a development environment.

Table 3–2 Enterprise Manager Properties, JSP Configuration Parameters

Enterprise Manager JSP Container Property	Possible Settings	JSP Configuration Parameter	Possible Settings
Debug Mode	**No** Yes	debug_mode	**false** true
External Resource for Static Content	No **Yes**	external_resource	**false** true
Generate Static Text as Bytes	No **Yes**	static_text_in_chars	**false** true
Tags Reuse Default	No **Yes**	tags_reuse_default	none **runtime**
Reduce Code Size for Custom Tags	**No** Yes	reduce_tag_code	**false** true
Emit Debug Info	**No** Yes	emit_debuginfo	**false** true
When a JSP Changes	**Recompile JSP** Reload Classes Do Nothing	main_mode	**recompile** reload justrun
Precompile Check	**No** Yes	precompile_check	**false** true

Table 3–2 Enterprise Manager Properties, JSP Configuration Parameters (Cont.)

Enterprise Manager JSP Container Property	Possible Settings	JSP Configuration Parameter	Possible Settings
Validate XML	**No** Yes	xml_validate	**false** true
SQLJ Command	command string (null by default)	sqljcmd	command string (null by default)
Alternate Java Compiler	command string (null by default)	javaccmd	command string (null by default)

Notes:

- In Oracle9*i*AS release 2 (9.0.3), Enterprise Manager supports only runtime (not compile-time) tag handler reuse. In other words, `tags_reuse_default` settings of `compiletime` or `compiletime_with_release` are not yet directly supported through Enterprise Manager.
- The Enterprise Manager JSP container property "Generate Static Text as Bytes" corresponds to the JSP configuration parameter `static_text_in_chars`, but with opposite orientation. Their defaults are equivalent.

Configuration Parameters Not Supported by the JSP Properties Page

For Enterprise Manager in Oracle9*i*AS release 2 (9.0.3), the following configuration parameters are not yet supported through the JSP Properties Page:

- JSP front-end servlet parameters: `check_page_scope`, `extra_imports`, `forgive_dup_dir_attr`, `no_tld_xml_validate`, `old_include_from_top`, `req_time_introspection`, and `well_known_taglib_loc`.
- JSP-related attributes of the `<orion-web-app>` element in `global-web-application.xml` or `orion-web.xml`: `jsp-print-null` and `jsp-timeout`.

Instead, you must update them in `orion-web.xml` or other appropriate XML file (such as `web.xml` or `global-web-application.xml`). Edit `orion-web.xml` or `global-web-application.xml` through the Enterprise Manager Web Module

Advanced Properties Page, as described in the *Oracle9iAS Containers for J2EE Servlet Developer's Guide*. Also see "Setting JSP Configuration Parameters in OC4J" on page 3-20 and "Setting JSP-Related OC4J Configuration Parameters" on page 3-21 for related information.

Note: If you update an XML configuration file manually in Oracle9*i*AS, you must then run the `dcmctl` utility, which is a command-line alternative to Enterprise Manager. See "OC4J Deployment Features" on page 7-34.

4

Basic Programming Considerations

This chapter discusses basic programming considerations for JSP pages, including JSP-servlet interaction and database access, with examples provided.

The following topics are included:

- JSP-Servlet Interaction
- JSP Data-Access Support and Features
- JSP Resource Management
- Runtime Error Processing

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 6-15.

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 or higher environment, such as OC4J (servlet 2.3). Appropriate reference is made to other sections of this document for related considerations for Apache 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 "Standard Actions: JSP Tags" on page 1-16.) 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 B-19.

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).

You can use a `jsp:param` tag 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 "Standard Actions: JSP Tags" on page 1-16.

Alternatively, you can pass data between a JSP page and a servlet through a JavaBean of appropriate scope 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 4-4.

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` tags. The `include()` method only temporarily transfers control; execution returns to the invoking servlet afterward.

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

Note: 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.

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.

Note: You can use the mechanisms discussed in this section 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 Data-Access Support and Features

This section discusses OC4J JSP and Oracle features to consider when accessing data, covering the following topics:

- Introduction to JSP Support for Data Access
- JSP Data-Access Sample Using JDBC
- Use of JDBC Performance Enhancement Features
- EJB Calls from JSP Pages
- JSP Support for Oracle SQLJ
- OracleXMLQuery Class

Note: For information about additional OC4J JSP data-access features—portable JavaBeans and tags for SQL functionality—see the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Introduction to JSP Support for Data Access

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

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 or Enterprise JavaBean). It is assumed that you are already at least somewhat familiar with JDBC basics, but for information about Oracle JDBC you can refer to the *Oracle9i JDBC Developer's Guide and Reference*.

The OC4J JSP container also supports EJB calls as well as SQLJ (embedded SQL in Java).

Additionally, there are SQL tags in the JavaServer Pages Standard Tag Library (JSTL), and JavaBeans and custom SQL tags supplied with OC4J. These are all documented in the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

JSP Data-Access Sample Using JDBC

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:

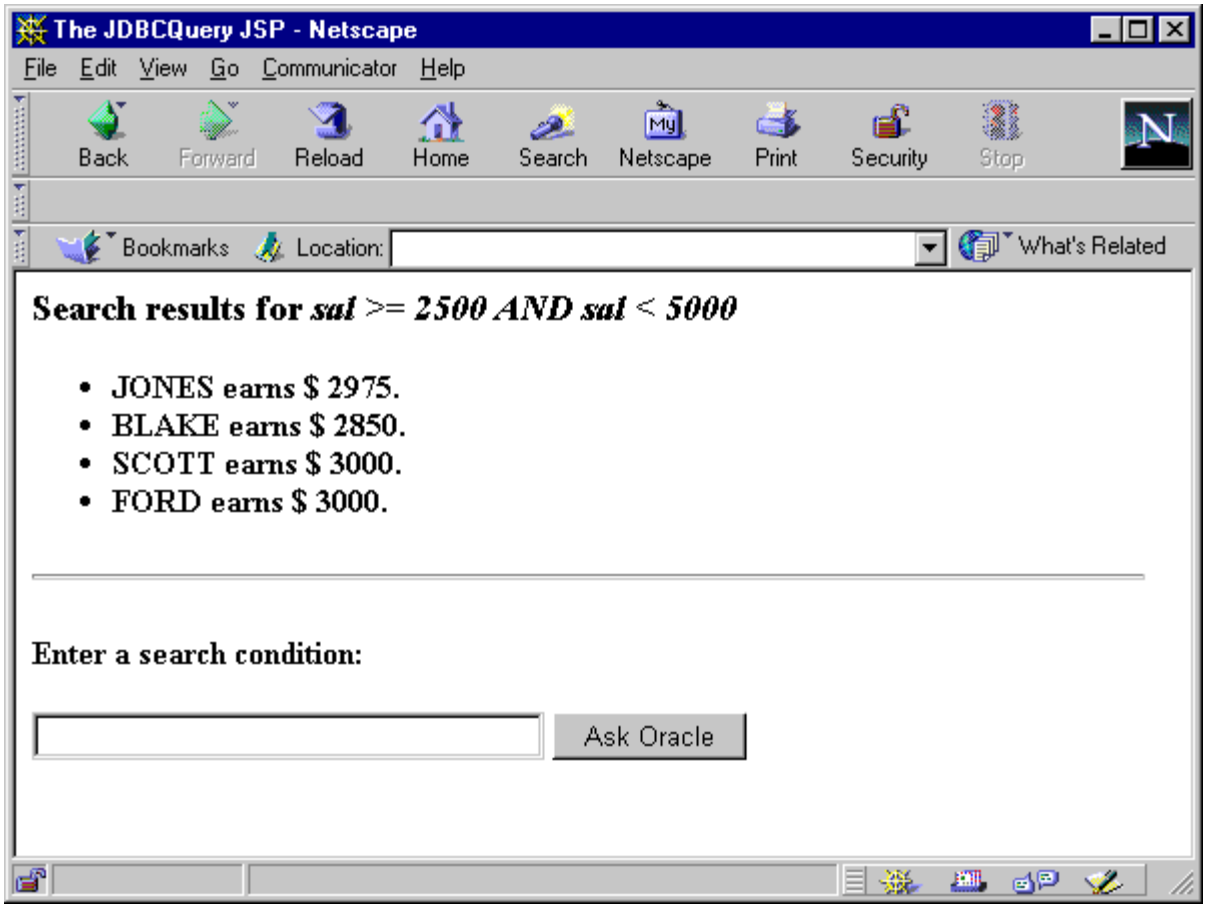
- Another approach to this example would be to define the `runQuery()` method in `<%...%>` scriptlet syntax instead of `<%!...%>` declaration syntax.
- This example uses the JDBC OCI driver, which requires an Oracle client installation. If you want to run this sample, use an appropriate JDBC driver and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



Use of JDBC Performance Enhancement Features

JSP applications in OC4J can use features for the following performance enhancements, supported through Oracle JDBC extension:

- 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 OC4J. Alternatively, you can use standard data-source connection pooling functionality, which is supported by the `ConnBean` data-access JavaBean. These beans are 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` tags or directly through the class setter methods.

Refer to the OC4J demos for examples of using `OracleConnectionCacheImpl` directly. 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 re-created, 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 OC4J. Each of these beans has a `stmtCacheSize` property that can be set through a `jsp:setProperty` tag or the bean `setStmtCacheSize()` method. 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 execution 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.

OC4J 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` tag 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

For the population of query result sets, the Oracle JDBC row prefetching feature enables you to determine the number of rows to prefetch into the client during each trip to the database. This reduces the number of round-trips to the server.

OC4J 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` tag 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*.

EJB Calls from JSP Pages

JSP pages can call EJBs to perform additional processing or data access. A typical application design uses JavaServer Pages as a front-end for the initial processing of client requests, with Enterprise JavaBeans being called to perform the work that involves reading from and writing to data sources. This section provides an overview of EJB usage, covering the following topics:

- Overview of Configuration and Deployment for EJBs
- Code Steps and Approaches for EJB Calls
- Use of the OC4J EJB Tag Library

See the OC4J demos for a complete example incorporating JSP pages and EJBs.

Overview of Configuration and Deployment for EJBs

The configuration and deployment steps for calling EJBs from JSP pages are similar to the steps for calling EJBs from servlets, which are described in the *Oracle9iAS Containers for J2EE Servlet Developer's Guide*. These steps include the following:

- Define an `<ejb-ref>` element in the application `web.xml` file for each EJB called from a JSP page.
- Create an `ejb-jar.xml` deployment descriptor that contains an `<enterprise-beans>` element with appropriate subelements, such as `<session>` or `<entity>`, that specify the types of EJBs. Within these subelements, specify the name, class name, and other details for each called EJB.
- Package the `ejb-jar.xml` file in the EJB archive. Deployment requirements are very similar to the requirements for servlets.

Code Steps and Approaches for EJB Calls

The key steps required for a JSP page to invoke an EJB are the following:

1. Import the EJB package for the bean home and remote interfaces into each JSP page that makes EJB calls. (In a JSP page, use a `page` directive for this.)
2. Use JNDI to look up the EJB home interface.
3. Create the EJB remote object from the home.
4. Invoke business methods on the remote object.

Because you can use almost any servlet code in a JSP page in the form of a scriptlet, one straightforward way to call EJBs from a JSP page is to use the same code in a

scriptlet that you would use in a servlet. This is one way to accomplish steps 2, 3, and 4.

Alternatively, you can use tags from the EJB tag library provided with OC4J (as described in the next section, "Use of the OC4J EJB Tag Library"). These tags simplify the coding. Essentially, they allow you to treat Enterprise JavaBeans similarly to regular JavaBeans, which are commonly used in JSP pages.

Use of the OC4J EJB Tag Library

Refer the preceding section, "Code Steps and Approaches for EJB Calls". As in that section, import the appropriate package in a `page` directive. Then use the OC4J EJB tags as follows:

- Use a `taglib` directive to specify the tag prefix and the tag library descriptor (TLD) file that you will use.
- For step 2 of the code steps, use an EJB `useHome` tag.
- For step 3 of the code steps, you can use an EJB `createBean` tag inside an EJB `useBean` tag.
- For step 4 of the code steps, the EJB `iterate` tag enables you to apply business methods to each member of a collection of EJB objects, usually returned by a `find` method.

For more information about the EJB tag library, including detailed tag syntax, see the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Deployment requirements are the same for the tag library approach as for the scriptlet code approach. As with any tag library, the TLD file and the library support files (tag handler classes and tag-extra-info classes) must be made accessible to your application.

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 OC4J JSP container supports Oracle SQLJ, allowing you to use SQLJ syntax in JSP statements. SQLJ statements are indicated by the `#sql` token. Oracle SQLJ database access is typically through the Oracle JDBC drivers.

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

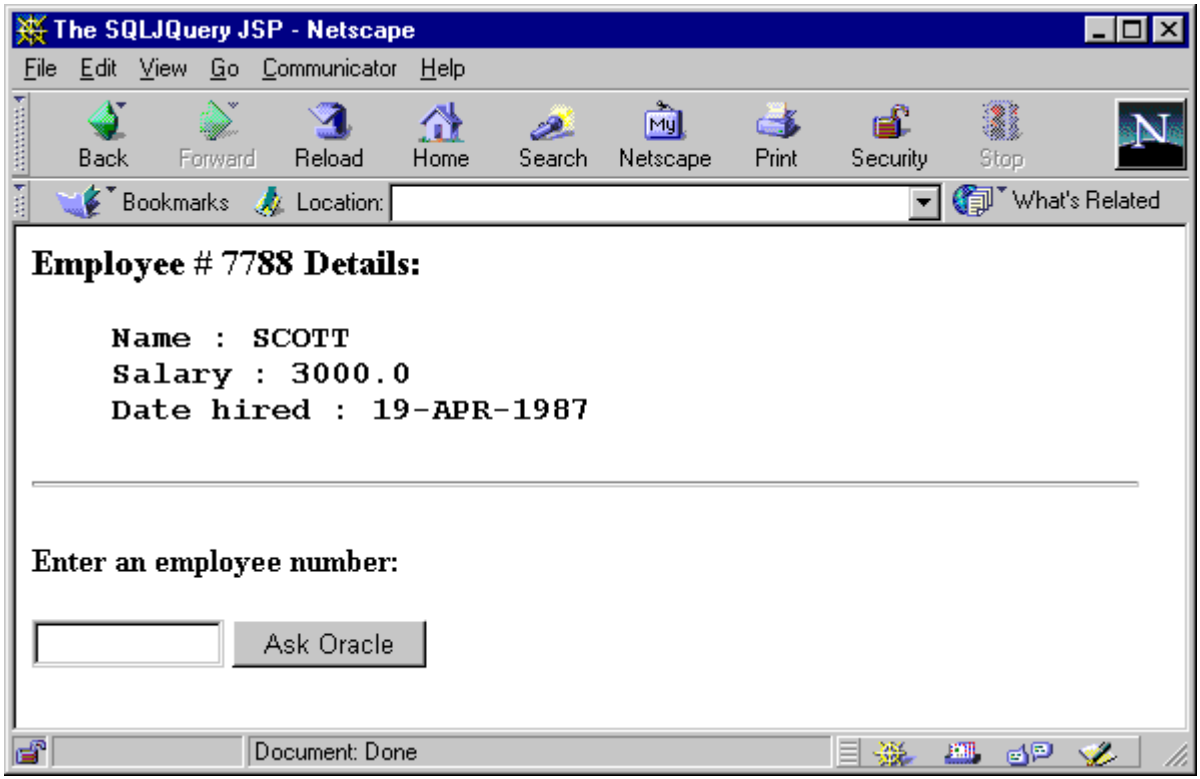
Following is a sample SQLJ JSP page. The page directive imports classes that are typically required by SQLJ.

```
<%@ 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.

Entering employee number 7788 results in output such as the following:



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.)
- If you use Oracle SQLJ, the OC4J JSP container requires SQLJ release 8.1.6.1 or higher.

For further examples of using SQLJ in JSP pages, refer to the OC4J demos.

Triggering the SQLJ Translator

You can trigger the OC4J JSP translator to invoke the Oracle SQLJ translator in one of two ways:

- by using the file name extension `.sqljsp` for the JSP source file

or:

- by specifying `language="sqlj"` in a `page` directive

Either of these 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 page 7-6.

Important:

- To use Oracle SQLJ, you must install the SQLJ JAR / ZIP files that are appropriate for your environment, and add them to your classpath. See "Key Support Files Provided with OC4J" on page 3-7.
- Do not use the same base file name for a `.jsp` file and a `.sqljsp` file in the same application, because this would result in duplicate generated class names and `.java` file names.

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, enables you to set SQLJ command-line options.

  For information about `sqljcmd`, see "JSP Configuration Parameters" on page 3-9.

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

  For information, see "Command-Line Syntax for ojspc" on page 7-20 and "Option Descriptions for ojspc" on page 7-20.

OracleXMLQuery Class

The `oracle.xml.sql.query.OracleXMLQuery` class is part of the Oracle9*i* XML-SQL utility for XML functionality in database queries. This class requires file `xsu12.jar` (or `xsu111.jar` for JDK 1.1.x), which is also required for XML functionality in some of the custom tags and JavaBeans provided with OC4J. This file is provided with Oracle9*i* and Oracle9*i*AS.

For a JSP sample using `OracleXMLQuery`, refer to the OC4J demos.

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

JSP Resource Management

This section discusses standard features and Oracle value-added features for resource management:

- Standard Session Resource Management: HttpSessionBindingListener (servlet 2.2 or higher environments)
- Overview of Oracle Value-Added Features for Resource Management (`JspScopeListener` for servlet 2.3 environments, `globals.jsa` for servlet 2.0 environments)

Standard Session Resource Management: 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-scope resources. Through this mechanism, a session-scope 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 Data-Access Sample Using JDBC" on page 4-8 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, and 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).

"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 4-23), 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-scope 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

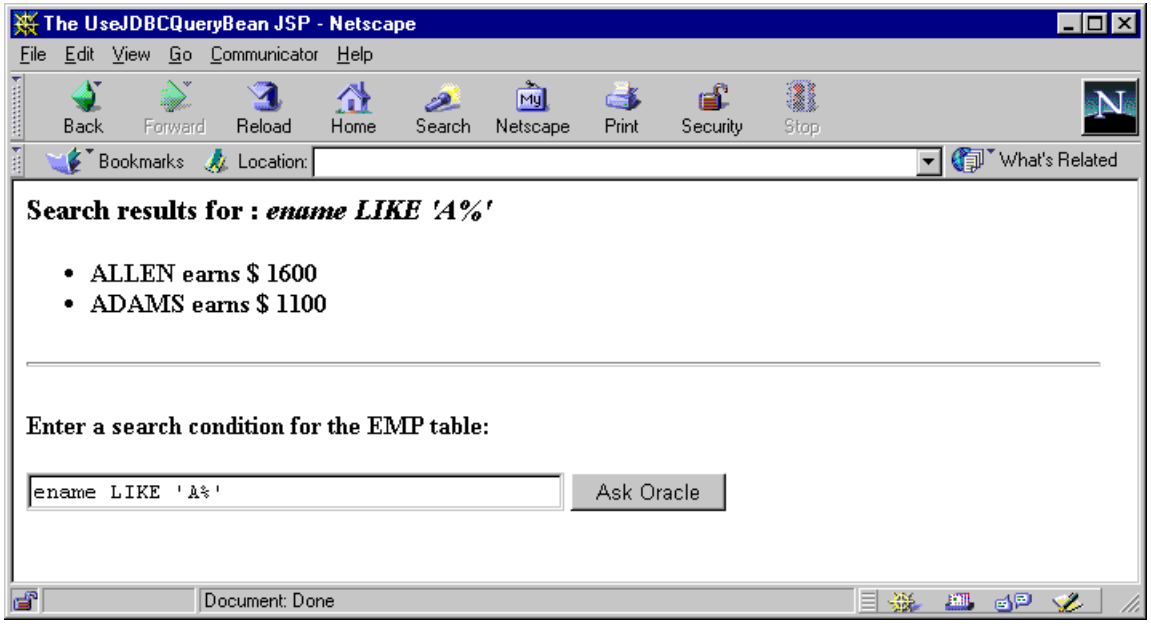
The following JSP page uses the `JDBCQueryBean` JavaBean defined in "JDBCQueryBean JavaBean Code" above, invoking the bean with `session` scope. It uses `JDBCQueryBean` to display employee names that match a search condition entered by the user.

`JDBCQueryBean` gets the search condition through the `jsp:setProperty` tag 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 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:



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 Value-Added Features for Resource Management

OC4J JSP provides the following features for managing application and session resources as well as page and request resources:

- `JspScopeListener` interface—This is for managing application-scope, session-scope, request-scope, or page-scope resources in a servlet 2.3 environment such as OC4J.

  This mechanism adheres to servlet and JSP standards in supporting objects of `page`, `request`, `session`, or `application` scope. To create a class that supports session scope as well as other scopes, you can integrate `JspScopeListener` with `HttpSessionBindingListener` by having the class implement both interfaces. For `page` scope in OC4J or JServ environments, you also have the option of using an Oracle-specific runtime implementation.

  For information about configuration and how to integrate with `HttpSessionBindingListener`, see the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

- `globals.jsa` file—This is for start and end events for application-scope and session-scope objects in a servlet 2.0 environment, such as JServ.

  See "The globals.jsa Event-Handlers" on page B-37 for information.

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 error processing mechanisms and provides an elementary example.

Servlet and JSP Runtime Error Mechanisms

This section describes servlet 2.3 and JSP 1.2 mechanisms for handling runtime exceptions, including the use of JSP error pages.

General Servlet Runtime Error Mechanism

Any runtime error encountered during execution of a JSP page is handled through 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 that you do not catch in the JSP page will result in forwarding of the request and uncaught exception, a `java.lang.Throwable` instance, to an error resource. This is the preferred way to handle JSP errors. In this case, the exception instance describing the error is stored in the `request` object through a `setAttribute()` call, using `javax.servlet.jsp.jspException` as the name.

You can specify the URL of an error resource by setting the `errorPage` attribute in a `page` directive in the originating JSP page. (For an overview of JSP directives, including the `page` directive, see "Directives" on page 1-7.)

In a servlet 2.2 or higher environment, you can also specify a default error page in the `web.xml` deployment descriptor with instructions such as the following:

```
<error-page>
   <error-code>404</error-code>
   <location>/error404.html</location>
</error-page>
```

See the Sun Microsystems *Java Servlet Specification, Version 2.3* for more information about default error resources.

JSP Error Pages

You have the option of using another JSP page as the error resource for runtime exceptions from an originating JSP page. A JSP error page must have a `page` directive setting the `isErrorPage="true"`. An error page defined in this way takes precedence over an error page declared in the `web.xml` file.

The `java.lang.Throwable` instance describing the error is accessible in the error page through the JSP implicit `exception` object. Only an error page can access this object. For information about JSP implicit objects, including the `exception` object, see "Implicit Objects" on page 1-13.

Be aware that if an originating JSP page has a `page` directive with `autoFlush="true"` (the default setting), and the contents of the `JspWriter` object from that page have already been flushed to the response output stream, then any further attempt to forward an uncaught exception to any error page might not be able to clear the response. Some of the response may have already been received by the browser.

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

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.

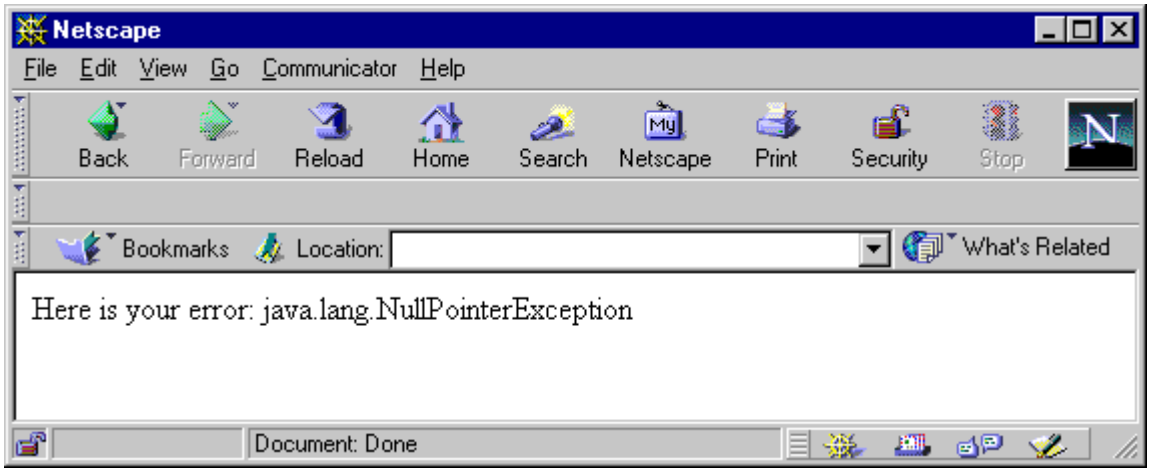
Code for nullpointer.jsp

```
<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:



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 a `jsp:forward`, the output from the "forward-to" page *replaces* the output from the "forward-from" page.

5

JSP XML Support

Because of additional support for XML in the JSP 1.2 specification, JavaServer Pages can increasingly be seen as an effective model for producing XML documents. With these enhancements, JSP technology becomes more complementary to XML technology and more accessible to XML tools. Another benefit of JSP XML support is that page validation becomes more powerful and comprehensive.

This chapter describes JavaServer Pages support for XML. This includes support for XML-style equivalents to JSP syntactical elements, and the concept of the "XML view" of a JSP page. These features were added in the JSP 1.2 specification, although the JSP 1.1 specification included optional support for JSP XML syntax and defined the syntax.

The chapter includes the following topics:

- JSP XML Documents and JSP XML View: Overview and Comparison
- Details of JSP XML Documents
- Details of the JSP XML View

For information about additional JSP support for XML and XSL, furnished in OC4J through custom tags, refer to the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

For general information about XML, refer to the XML specification at the following Web site:

`http://www.w3.org/XML/`

JSP XML Documents and JSP XML View: Overview and Comparison

Traditional JSP constructs, such as `<%@ page...>` directives, `<%@ include... >` directives, `<%...%>` for scriptlets, `<%!...%>` for declarations, and `<%=...%>` for expressions, are not syntactically valid within an XML document. Sun Microsystems first addressed this issue in the *JavaServer Pages Specification, Version 1.1* by defining equivalent JSP syntax that is XML-compatible. In JSP 1.1, however, support for this syntax by a JSP container is optional.

The *JavaServer Pages Specification, Version 1.2* offers more complete support for XML-compatible JSP syntax, adding features and requiring support by compliant JSP containers.

Note: The OC4J JSP container, as documented in previous releases of this manual, supported the optional XML-alternative syntax of the JSP 1.1 specification. The JSP container now replaces this implementation with full XML support as prescribed by the JSP 1.2 specification. The JSP 1.1 syntax itself remains unchanged, but there are now additional aspects of JSP XML support, as described in this chapter.

In addition, under the JSP 1.1 specification, you could intermix traditional syntax and XML-alternative syntax within a page. This is *not* true in a JSP 1.2 environment.

The term *JSP XML document* (called *JSP document* in the JSP 1.2 specification) refers to a JSP page that uses this XML-compatible syntax. The syntax includes, among other things, a root element and elements that serve as alternatives to JSP directives, declarations, expressions, and scriptlets. (Standard tag actions and custom tag actions already follow XML conventions.) See "Details of JSP XML Documents" on page 5-4 for details.

A JSP XML document is well formed in pure XML syntax, and is namespace-aware. It uses XML namespaces to specify the JSP XML core syntax and the syntaxes of any custom tag libraries used. A traditional JSP page, by contrast, is typically *not* an XML document.

A JSP XML document has the same file name extension as a traditional JSP page, `.jsp`. However, it is recognizable by the JSP container as an XML document because of its root element, `<jsp:root>`. Additionally, the semantic model for JSP XML documents is the same as for traditional pages. A JSP XML document dictates the same set of actions and results as a traditional page with equivalent syntax.

Processing of white space follows XSLT conventions. Once the nodes of a JSP XML document have been identified, textual nodes that have only white space are dropped from the document, except within `<jsp:text>` elements for template data. The content of `<jsp:text>` elements is kept exactly as is.

Note: *Template data* consists of any text that is not interpreted by the JSP translator.

In a JSP 1.2 environment, a JSP XML document can be processed directly by the JSP container. You can also use a JSP XML document with XML development tools or other XML tools, which will become increasingly important as such tools become more popular and prevalent.

Another key feature of XML support in the JSP 1.2 specification is the *JSP XML view*. The specification defines this as "the mapping between a JSP page, written in either XML syntax or traditional syntax, and an XML document describing it". The JSP container generates it during translation.

In the case of a JSP XML document, the JSP XML view is similar to the page source. One difference is that the XML view is expanded according to any `include` directives. Another (optional) difference, for JSP containers that support it, is that ID attributes for improved error reporting are added to all XML elements.

In the case of a traditional JSP page, the JSP container performs a series of transformations to create the XML view from the page. See "Details of the JSP XML View" on page 5-15 for details.

The key function of the JSP XML view is its use for page validation. Beginning with the JSP 1.2 specification, any tag library can have a `<validator>` element in its TLD file to specify a class that can perform validation. Such classes are referred to as *tag-library-validator* (TLV) classes. The purpose of a TLV class is to validate any JSP page that uses the tag library, verifying that the page adheres to any desired constraints that you have implemented. A validator class uses the JSP XML view as the source for its validation.

In summary, you can optionally use JSP XML syntax to create a JSP page that is XML-compatible. The JSP XML view, in contrast, is a function of the JSP container, for use in page validation.

Details of JSP XML Documents

This section describes the syntax of JSP XML documents in further detail. For a complete description, refer to the Sun Microsystems *JavaServer Pages Specification, Version 1.2*.

Important: You cannot intermix JSP traditional syntax and JSP XML syntax in a single file. You can, however, make use of both syntaxes together in a single translation unit through the use of `include` directives. For example, a traditional JSP page can include a JSP XML document.

Note: A JSP XML document does not use a `DOCTYPE` statement.

JSP XML syntax includes the following:

- a root element, `<jsp:root ...>`, which includes a namespace specification for the JSP XML core syntax, and namespace specifications for any custom tag libraries that are used
- JSP directive elements (for `page` and `include` directives)

Note: A separate mechanism, through `xmlns` attributes of the root element, is equivalent to the use of `taglib` directives. "JSP XML root Element and JSP XML Namespaces" on page 5-7 describes this.

- JSP declaration elements
- JSP expression elements
- JSP scriptlet elements
- JSP standard action elements
- JSP custom action elements
- a text element, `<jsp:text ... >`, for template (static) data
- other XML elements, if desired, pertaining to template data

This section describes each of these types of elements, followed by an example comparing a traditional JSP page to the equivalent JSP XML document.

Summary Table of JSP XML Syntax

Table 5–1 summarizes JSP XML syntax, comparing it to JSP traditional syntax as applicable.

Table 5–1 JSP XML Syntax Versus JSP Traditional Syntax

JSP XML Syntax	Corresponding JSP Traditional Syntax

Root element:
`<jsp:root`
`xmlns:jsp=...`
`xmlns:`*`xxx`*` =...`
`...`
`version=...`
`/>`
 	
The root element indicates the standard JSP XML namespace, XML namespaces for any custom tag libraries, and a JSP version number (required). See "JSP XML root Element and JSP XML Namespaces" on page 5-7.
	The `xmlns` settings for tag libraries are equivalent to JSP `taglib` directives.

JSP `page` directive element: `<jsp:directive.page ... />` See "JSP XML Directive Elements" on page 5-8.	`<%@ page ... %>`
JSP `include` directive element: `<jsp:directive.include ... />` See "JSP XML Directive Elements" on page 5-8.	`<%@ include ... %>`

JSP declaration element:
`<jsp:declaration>`
`declaration`
`</jsp:declaration>`
 	
See "JSP XML Declaration, Expression, and Scriptlet Elements" on page 5-9.
	`<%! declaration %>`

Table 5–1 JSP XML Syntax Versus JSP Traditional Syntax (Cont.)

JSP XML Syntax	Corresponding JSP Traditional Syntax

JSP expression element:
`<jsp:expression>`
`  expression`
`</jsp:expression>`
 	
See "JSP XML Declaration, Expression, and Scriptlet Elements" on page 5-9.
	`<%= expression %>`

JSP scriptlet element:
`<jsp:scriptlet>`
`  code fragment`
`</jsp:scriptlet>`
 	
See "JSP XML Declaration, Expression, and Scriptlet Elements" on page 5-9.
	`<% code fragment %>`

JSP standard action (such as `jsp:include` or `jsp:forward`) See "JSP XML Standard Action and Custom Action Elements" on page 5-10.	JSP standard action The traditional standard action syntax is already XML-compatible.
JSP custom action (any custom tag) See "JSP XML Standard Action and Custom Action Elements" on page 5-10.	JSP custom action The traditional custom action syntax is already XML-compatible.
JSP request-time attribute expression (within a standard or custom action): `<foo:bar attr="%=expr%" />` See "JSP XML Standard Action and Custom Action Elements" on page 5-10.	`<foo:bar attr="<%=expr%>" />`

Text element:
`<jsp:text>`
`...`
`</jsp:text>`
 	
This is for template data. See "JSP XML Text Elements and Other Elements" on page 5-10.
	Template data

Table 5–1 JSP XML Syntax Versus JSP Traditional Syntax (Cont.)

JSP XML Syntax	Corresponding JSP Traditional Syntax
Other XML elements. These may appear anywhere a `<jsp:text>` element may appear. See "JSP XML Text Elements and Other Elements" on page 5-10.	Template data

JSP XML root Element and JSP XML Namespaces

The `<jsp:root>` element has three primary functions:

- It establishes the document as a JSP XML document, instructing the JSP container to treat it accordingly.
- It identifies, through `xmlns` attribute settings, required XML namespaces for the JSP XML core syntax and any custom tag libraries.
- It specifies a JSP version number (required).

There is always one `xmlns` attribute to identify the namespace for the core JSP XML syntax:

```
xmlns:jsp="http://java.sun.com/JSP/Page"
```

This `xmlns:jsp` setting enables the use of standard elements defined in the JSP 1.2 specification.

You must also include an `xmlns` attribute for each custom tag library you use, specifying the tag library prefix and namespace—that is, pointing to the corresponding TLD file for use in validating your tag usage. These `xmlns` settings are equivalent to `taglib` directives in a traditional JSP page.

You can use either a URN or a URI to point to the TLD file. The JSP 1.2 specification provides the following example, for tag library prefixes `eg` and `temp`:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:eg="http://java.apache.org/tomcat/examples-taglib"
          xmlns:temp="urn:jsptld:/WEB-INF/tlds/my.tld"
          version="1.2"
>

...body of document...

</jsp:root>
```

A URN indicates an application-relative path and must be of the form "`urn:jsptld:`*`path`*", where the path is specified in the same way as the `uri` attribute in a `taglib` directive. See "Overview: Specifying a Tag Library with the taglib Directive" on page 8-16.

A URI can be a complete URL, or it can be according to mapping in the `<taglib>` element of the `web.xml` file or the `<uri>` element of a TLD file. See "Use of web.xml for Tag Libraries" on page 8-21 and "Packaging and Accessing Multiple Tag Libraries in a JAR File" on page 8-18.

Also note the `version` attribute in the example. This is a required attribute, specifying the JSP version that the page uses (1.2 or higher).

JSP XML Directive Elements

There are JSP XML elements that are equivalent to `page` and `include` directives. (The `taglib` directives are replaced by `xmlns` settings in the `<jsp:root>` element, as the preceding section, "JSP XML root Element and JSP XML Namespaces", describes.)

Transforming a `page` or `include` directive to the equivalent JSP XML element is straightforward, as shown in the following examples.

Example: page Directive The following `page` directive:

```
<%@ page language="sqlj"
    import="sqlj.runtime.ref.DefaultContext,oracle.sqlj.runtime.Oracle" %>
```

is equivalent to the following JSP XML element:

```
<jsp:directive.page language="sqlj"
    import="sqlj.runtime.ref.DefaultContext,oracle.sqlj.runtime.Oracle" />
```

Example: include Directive The following `include` directive:

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

is equivalent to the following JSP XML element:

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

Note: The XML view of a page does not contain `include` elements, because statically included segments are copied directly into the view.

JSP XML Declaration, Expression, and Scriptlet Elements

There are JSP XML elements that are equivalent to JSP declarations, expressions, and scriptlets.

Transforming any of these constructs to the equivalent JSP XML element is straightforward, as shown in the following examples.

Example: JSP Declaration The following JSP declaration:

```
<%! public String func(int myint) { if (myint<10) return("..."); } %>
```

is equivalent to the following JSP XML element:

```
<jsp:declaration>
  <![CDATA[ public String func(int myint) { if (myint<10) return("..."); } ]]>
</jsp:declaration>
```

The XML `CDATA` (character data) designation is used because the declaration includes a "<" character, which has special meaning to an XML parser. (If you use an XML editor to create your JSP XML pages, this would presumably be handled automatically.) Alternatively, you could write the following, using the "`<`" escape character instead of "<":

```
<jsp:declaration>
   public String func(int myint) { if (myint &lt; 10) return("..."); }
</jsp:declaration>
```

Example: JSP Expression The following JSP expression:

```
<%= (user==null) ? "" : user %>
```

is equivalent to the following JSP XML element:

```
<jsp:expression> (user==null) ? "" : user </jsp:expression>
```

Example: JSP Scriptlet The following JSP scriptlet:

```
<% if (pageBean.getNewName().equals("")) { %>
   ...
```

is equivalent to the following JSP XML element:

```
<jsp:scriptlet> if (pageBean.getNewName().equals("")) { </jsp:scriptlet>
   ...
```

JSP XML Standard Action and Custom Action Elements

Traditional syntax for JSP standard actions (such as `jsp:include`, `jsp:forward`, and `jsp:useBean`) and custom actions is already XML-compatible. In using standard actions or custom actions in JSP XML syntax, however, be aware of the following issues.

- A standard action or custom action element with an attribute that can accept a request-time expression value can take that value through the following syntax:

  "%=*expression*%"

  Note that there are no angle brackets, "<" and ">", around this syntax and that white space around *`expression`* is not necessary. Evaluation of *`expression`*, after any applicable quoting as in any XML document, is the same as for any JSP request-time expression.

- Any quoting must be according to the XML specification.
- You can introduce template data through `<jsp:text>` elements or through chosen XML elements that are neither standard nor custom. See "JSP XML Text Elements and Other Elements", which follows.

JSP XML Text Elements and Other Elements

A `<jsp:text>` element denotes template data in a JSP XML document:

```
<jsp:text>
   ...template data...
</jsp:text>
```

When a JSP container encounters a `<jsp:text>` element, it passes the contents to the current JSP `out` object (similar to the processing of an XSLT `<xsl:text>` element).

The JSP 1.2 specification also allows, wherever a `<jsp:text>` element can appear, the use of arbitrary elements (neither standard action elements nor custom action elements) for template data. These arbitrary elements are processed in the same way as `<jsp:text>` elements, with content being sent to the current JSP `out` object.

The following example is from the Sun Microsystems *JavaServer Pages Specification, Version 1.2*.

Example: Other JSP XML Elements Consider the following JSP XML document source text:

```
<hello><jsp:scriptlet>int i=3;</jsp:scriptlet>
<hi>
<jsp:text> hi you all
</jsp:text><jsp:expression>i</jsp:expression>
</hi>
</hello>
```

This source text results in the following output from the JSP container:

```
<hello> <hi> hi you all
3 </hi></hello>
```

(Note how the white space is treated.)

Sample Comparison: Traditional JSP Page Versus JSP XML Document

This section shows two versions of a JSP page—one in traditional syntax and one in XML syntax.

For information about deploying and running this example, refer to the following Web site:

```
http://otn.oracle.com/tech/java/oc4j/htdocs/how-to-jsp-xmlview.html
```

(You must register for an Oracle Technology Network membership, but it is free of charge.)

Sample Traditional JSP Page Here is the sample page in traditional syntax:

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

<jsp:useBean id = "picker" class = "oracle.jsp.sample.lottery.LottoPicker"
             scope = "page" />
```

```
<% picker.setIdentity(request.getRemoteAddr() ); %>

<HTML>
<HEAD>
 <TITLE>Lotto Number Generator</TITLE>
</HEAD>

<BODY BACKGROUND="images/cream.jpg" BGCOLOR="#FFFFFF">

<H1 ALIGN="CENTER"></H1>

<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 = picker.getPicks();
     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>
```

Sample JSP XML Document Here is the same page in XML syntax:

```
<jsp:root
   xmlns:jsp="http://java.sun.com/JSP/Page"
         version="1.2">

<jsp:directive.page session = "false" contentType="text/html"/>

<jsp:useBean id = "picker" class = "oracle.jsp.sample.lottery.LottoPicker"
             scope = "page" />
<jsp:scriptlet>picker.setIdentity(request.getRemoteAddr() ); </jsp:scriptlet>
<jsp:text><![CDATA[<HTML>

<HEAD>
 <TITLE>Lotto Number Generator</TITLE>
</HEAD>

<BODY BACKGROUND='../basic/lottery/images/cream.jpg' BGCOLOR='#FFFFFF'>

<H1 ALIGN='CENTER'></H1>

<BR>

<H1 ALIGN='CENTER'>Your Specially Picked</H1>
<P ALIGN='CENTER'><IMG SRC='../basic/lottery/images/winningnumbers.gif'
   WIDTH='450' HEIGHT='69' ALIGN='BOTTOM' BORDER='0'></P>

<P ALIGN='CENTER'>
<TABLE ALIGN='CENTER' BORDER='0' CELLPADDING='0' CELLSPACING='0'>
<TR>]]></jsp:text>
<jsp:scriptlet>
     int [] picks = picker.getPicks();
     for (int i = 0; i &lt; picks.length; i++)
  {
</jsp:scriptlet>
<jsp:text><![CDATA[<TD>
     <IMG SRC='../basic/lottery/images/ball]]>
</jsp:text>
<jsp:expression>picks[i]</jsp:expression>
<jsp:text>
   <![CDATA[.gif' WIDTH='68' HEIGHT='76' ALIGN='BOTTOM' BORDER='0'>
```

```
</TD>]]></jsp:text>
<jsp:scriptlet>
     }
</jsp:scriptlet>
<jsp:text><![CDATA[</TR>
</TABLE>
</P>
<P ALIGN='CENTER'><BR>
<BR>
<IMG SRC='../basic/lottery/images/playrespon.gif' WIDTH='120' HEIGHT='73'
ALIGN='BOTTOM' BORDER='0'>
</BODY>
</HTML>]]></jsp:text>
</jsp:root>
```

Details of the JSP XML View

When a container that complies with JSP 1.2 translates a JSP page, it creates an XML version, known as the *XML view*, of the parsing result. The JSP 1.2 specification defines the XML view as being a mapping of a JSP page (either a traditional page or a JSP XML document) into an XML document that describes it. The XML view can be used by tag-library-validator classes in validating the page. (See "Validation and Tag-Library-Validator Classes" on page 8-46.) The XML view of a page looks mostly like the page as you would write it yourself if you were using JSP XML syntax, with a couple of key differences, as described shortly.

This section covers the following topics:

- Transformation from a JSP Page to the XML View
- The jsp:id Attribute for Error Reporting During Validation
- Example: Transformation from Traditional JSP Page to XML View

Refer to the Sun Microsystems *JavaServer Pages Specification, Version 1.2* for further details.

Transformation from a JSP Page to the XML View

When translating a JSP page, the JSP container executes the following transformations in creating the XML view, both for traditional JSP pages and for JSP XML documents:

- The container expands the XML view to include files brought in through `include` directives.
- A JSP container that supports the optional `jsp:id` attribute (for improved error reporting) inserts that attribute into each XML element in the page. See "The jsp:id Attribute for Error Reporting During Validation" on page 5-16.

For a JSP XML document, these points constitute the key differences between the XML view and the original page.

The JSP container executes the following additional transformations for traditional JSP pages:

- It adds the `<jsp:root>` element, with the standard `xmlns` attribute setting for JSP XML syntax, and the `version` attribute for the JSP version. See "JSP XML root Element and JSP XML Namespaces" on page 5-7.

- It converts each `taglib` directive into an additional `xmlns` attribute in the `<jsp:root>` element. See "JSP XML root Element and JSP XML Namespaces" on page 5-7.
- It converts each `page` directive into the equivalent element in JSP XML syntax. See "JSP XML Directive Elements" on page 5-8.
- It converts each declaration, expression, and scriptlet into the equivalent element in JSP XML syntax. See "JSP XML Declaration, Expression, and Scriptlet Elements" on page 5-9.
- It converts request-time expressions into XML syntax. See "JSP XML Standard Action and Custom Action Elements" on page 5-10.
- It creates `<jsp:text>` elements for template data. See "JSP XML Text Elements and Other Elements" on page 5-10.
- It converts JSP quotations into XML quotations.
- It ignores JSP comments: `<%-- comment --%>`. They do not appear in the XML view.

Notes:

- The XML view has no `DOCTYPE` statement.
- No "other XML elements", as described in "JSP XML Text Elements and Other Elements" on page 5-10, appear in the XML view. Only `<jsp:text>` elements are used for template data.

The jsp:id Attribute for Error Reporting During Validation

The JSP 1.2 specification describes an optional `jsp:id` attribute that the JSP container can add to each XML element in the XML view. A container does *not* have to support this feature to comply with JSP 1.2, but the OC4J JSP container does support it.

The `jsp:id` attributes, if present, are used by tag-library-validator classes during page validation. The purpose of these attributes is to provide improved error reporting, possibly helping developers pinpoint where errors occur (depending on how the JSP container implements `jsp:id` support).

The `jsp:id` attribute values must be generated by the container in a way that ensures that each value, or ID, is unique across all elements in the XML view.

A tag-library-validator object can use these IDs in the `ValidationMessage` objects that it returns. (See "Validation and Tag-Library-Validator Classes" on page 8-46 for background information about TLV classes.)

In the OC4J JSP implementation, when a `ValidationMessage` object with IDs is returned, each ID is transformed to reflect the tag name and source location of the matching element.

Example: Transformation from Traditional JSP Page to XML View

This example shows traditional page source from one of the OC4J JSP demo applications, followed by the XML view of the page as generated by the OC4J JSP translator. (The demo displays the Oracle JSP version number and configuration parameter values.)

Traditional JSP Page Here is the traditional JSP page:

```
<HTML>
    <HEAD>
        <TITLE>OJSP Information </TITLE>
    </HEAD>
    <BODY BGCOLOR="#FFFFFF">
         OJSP Version:<BR>
            <%= application.getAttribute("oracle.jsp.versionNumber") %>
         <BR>
         OJSP Init Parameters:<BR>
         <%
         for (Enumeration paraNames = config.getInitParameterNames();
              paraNames.hasMoreElements() ;) {
             String paraName = (String)paraNames.nextElement();
         %>
         <%=paraName%> = <%=config.getInitParameter(paraName)%>
         <BR>
         <%
         }
         %>
    </BODY>
</HTML>
```

XML View of JSP Page Here is the corresponding XML view:

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" jsp:id="0" version="1.2">
   <jsp:text jsp:id="1"><![CDATA[   <HTML>
       <HEAD>
           <TITLE>OJSP Information </TITLE>
```

```
        </HEAD>
        <BODY BGCOLOR="#FFFFFF">
             OJSP Version:<BR>]]></jsp:text>
   <jsp:expression jsp:id="2">
      <![CDATA[ application.getAttribute("oracle.jsp.versionNumber") ]]>
   </jsp:expression>
   <jsp:text jsp:id="3"><![CDATA[
             <BR>
             OJSP Init Parameters:<BR>
             ]]>
   </jsp:text>
   <jsp:scriptlet jsp:id="4"><![CDATA[
             for (Enumeration paraNames = config.getInitParameterNames();
                  paraNames.hasMoreElements() ;) {
                 String paraName = (String)paraNames.nextElement();
             ]]></jsp:scriptlet>
   <jsp:text jsp:id="5"><![CDATA[
             ]]></jsp:text>
   <jsp:expression jsp:id="6"><![CDATA[paraName]]></jsp:expression>
   <jsp:text jsp:id="7"><![CDATA[ = ]]></jsp:text>
   <jsp:expression jsp:id="8">
      <![CDATA[config.getInitParameter(paraName)]]>
   </jsp:expression>
   <jsp:text jsp:id="9"><![CDATA[
             <BR>
             ]]></jsp:text>
   <jsp:scriptlet jsp:id="10"><![CDATA[
             }
             ]]></jsp:scriptlet>
   <jsp:text jsp:id="11"><![CDATA[
        </BODY>
  </HTML>

]]></jsp:text>
</jsp:root>
```

6

Additional Considerations

This chapter discusses an assortment of programming and runtime considerations in developing and executing JSP applications. The following categories are covered:

- JSP Programming Strategies, Tips, and Traps
- JSP Runtime Considerations and Optimization

JSP Programming Strategies, Tips, and Traps

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

- JavaBeans Versus Scriptlets
- 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

Note: In addition to being aware of what is discussed in this section, you should be aware of JSP translation and deployment issues and behavior. See Chapter 7, "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, 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 Data-Access Sample Using JDBC" on page 4-8, 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.

Static Includes Versus Dynamic Includes

The `include` directive, described in "Directives" on page 1-7, 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` tag, described in "Standard Actions: JSP Tags" on page 1-16, 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 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: You can use static includes and dynamic includes 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 is not required to be 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 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. (The service method of the generated page implementation class has a 64 KB size limit—see "Workarounds for Large Static Content in JSP Pages" on page 6-7.)

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 6-6.)

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.

Note: OC4J offers *global includes* as a convenient way to statically include a file into multiple pages. See "Oracle JSP Global Includes" on page 7-9.

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.
- You want to provide convenient JSP programming access to functionality that would otherwise require the use of a Java API.
- 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 tag library provided with OC4J. 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.

Providing Convenient JSP Programming Access to API Features

Instead of having Web application programmers rely on Java APIs from servlets or JSP scriptlets to use product functionality or extensions, you can provide a tag library. A tag library can make the programmer's task much more convenient, with appropriate API calls being handled automatically by the tag handlers.

For example, tags as well as JavaBeans are provided with OC4J for e-mail and file access functionality. There is also a tag library as well as a Java API provided with the OC4J Web Object Cache. Similarly, while Oracle9*i*AS Personalization provides a Java API, OC4J also provides a tag library that you can use instead if you want to program a personalization application.

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 tags of the library):

```
<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 8-25 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: HttpSessionBindingListener" on page 4-20.)

```
public class MySessionChecker implements HttpSessionBindingListener
{
   ...
   valueBound(HttpSessionBindingEvent event)
   {...}

   valueUnbound(HttpSessionBindingEvent event)
   {...}
   ...
}
```

You can create a checker 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.

Note: OC4J offers "global includes" as a convenient way to statically include a file into multiple pages. See "Oracle JSP Global Includes" on page 7-9.

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 of which will speed translation):

- Put the static HTML into a separate file and use a dynamic include (`jsp:include`) to include its output in the JSP page output at runtime. See "Standard Actions: JSP Tags" on page 1-16 for information about the `jsp:include` tag.

  Important: A static `include` directive 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 JSP translator will do this for you if you enable the `external_resource` configuration parameter. This parameter is documented in "JSP Configuration Parameter Descriptions" on page 3-12.

For pre-translation, the `-extres` option of the `ojspc` tool also offer 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 64 KB 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.)

Similarly, it is possible 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 1-9, 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. In this case, if declaring an object instance, the object can be accessed simultaneously from multiple requests. Therefore, the object must be thread-safe, unless `isThreadSafe="false"` is declared in a `page` directive.
- A variable that is declared in <% ... %> JSP scriptlet syntax is local to the service method of the page implementation class. Each time the method is called, a separate instance of the variable or object is created, so there is no need for thread safety.

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 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.
- Beginning with the JSP 1.2 specification, duplicate settings of directive attributes are disallowed. In particular, this pertains to the `page` directive, although the `page` directive `import` attribute is exempt from this limitation.
- Java `import` settings in `page` directives are cumulative within a JSP page or translation unit.

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 the second example, there is a `setContentType()` method that allows dynamic setting of the content type, as described in "Dynamic Content Type Settings" on page 9-5.

Duplicate Settings of Page Directive Attributes Are Disallowed

The JSP 1.2 specification states that a JSP container must verify that directive attributes, with the exception of the `page` directive `import` attribute, are not set more than once each within a single JSP translation unit (a JSP page plus anything it includes through `include` directives). In JSP 1.2, this effectively applies to `page` directives only, but in future JSP versions there might be additional relevant directives.

For backward compatibility to the JSP 1.1 standard, where duplicate settings of directive attributes are allowed, OC4J provides the `forgive_dup_dir_attr` configuration parameter. See "JSP Configuration Parameters" on page 3-9. You might have previously coded a page with multiple included segments that all set the `page` directive `language` attribute to "`java`", for example.

For clarity, be aware of the following points.

- The JSP 1.2 specification allows multiple `page` directives, as long as they set different attributes.

  The following are okay:

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

  or:

  ```
  ------------------------------
  <%@ page buffer="10kb" %>
  <%@ include file="b.jsp" %>
  ------------------------------
  ------------------------------
  b.jsp
  <%@ page session="false" %>
  ------------------------------
  ```

  The following are *not* okay:

  ```
  <%@ page buffer="none" %>
  <%@ page buffer="10kb" %>
  ```

  or:

  ```
  <%@ page buffer="none" buffer="10kb" %>
  ```

or:

```
-----------------------------
<%@ page buffer="10kb" %>
<%@ include file="b.jsp" %>
-----------------------------
-----------------------------
b.jsp
<%@ page buffer="3kb" %>
-----------------------------
```

- A translation unit consists of a JSP page plus anything it includes through `include` directives, but *not* pages it includes through `jsp:include` tags. Pages included through `jsp:include` tags are dynamically included at runtime, not statically included during translation. See "Static Includes Versus Dynamic Includes" on page 6-3 for more information.

  Therefore, the following is okay:

```
-----------------------------
<%@ page buffer="10kb" %>
<jsp:include page="b.jsp" />
-----------------------------
-----------------------------
b.jsp
<%@ page buffer="3kb" %>
-----------------------------
```

- As noted in the opening paragraph above, the `page` directive `import` attribute is exempt from the limitation against duplicate attribute settings. See the next section, "Page Directive import Settings Are Cumulative".

Page Directive import Settings Are Cumulative

The `page` directive `import` attribute is exempt from JSP 1.2 limitations on duplicate directive attributes. Java `import` settings in `page` directives within a JSP page or translation unit (a JSP page plus anything included through `include` directives) are cumulative.

Within any single JSP page or translation unit, 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

JSP containers generally preserve 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 form 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 code 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.

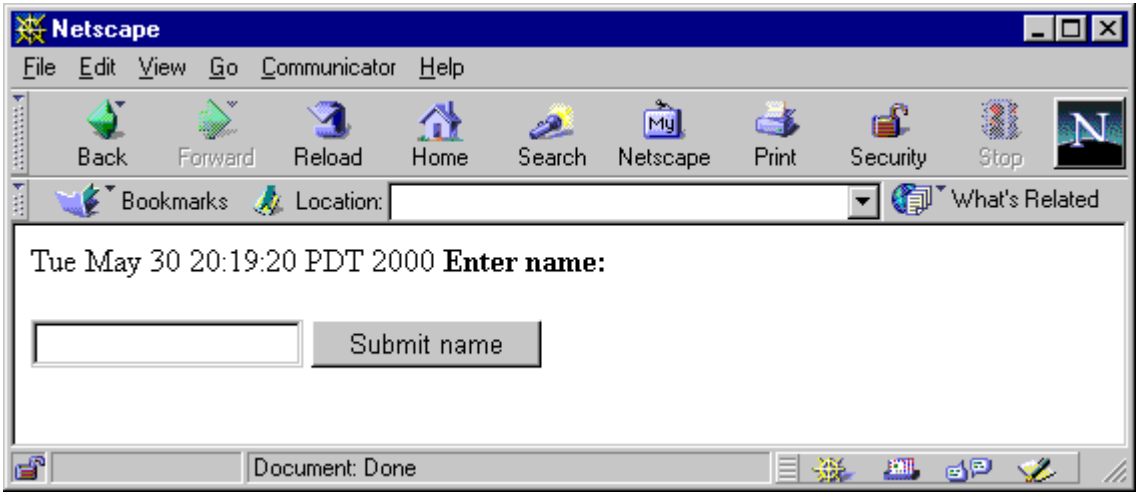
`whitesp.jsp`:

```
<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 code 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.



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 in the `JspWriter` class for writing raw bytes.
- During execution, the JSP container preserves white space. White space 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 white space 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 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.

JSP Runtime Considerations and Optimization

This section describes some of the JSP runtime functionality, particularly regarding dynamic page retranslation and class reloading, and points out some considerations for optimizing execution. The following topics are covered:

- Dynamic Page Retranslation and Class Reloading
- Optimization Considerations

Dynamic Page Retranslation and Class Reloading

By default, particularly for use in development environments where code is in flux, the JSP container has the following behavior during page execution.

- For each page being executed, the container checks whether a page implementation class already exists, compares the `.class` file timestamp against the `.jsp` source file timestamp, and retranslates the page if the `.class` file is older (indicating that the page has been modified since the page implementation class was loaded).
- For any request that will execute a Java class that was loaded by the JSP class loader, the container checks to see if the class file has been modified since it was last loaded. If the class has been modified, then the JSP class loader reloads it. This applies to class files in the following locations:
  - under the `/WEB-INF/classes` directory
  - in JAR files in the `/WEB-INF/lib` directory
  - under the `_pages` output directory (generated page implementation classes)

  See "Classpath Functionality" on page 3-3 for related information.
- The container reloads a JSP page (in other words, reloads the generated page implementation class) in the following circumstances:
  - the page is retranslated
  - a Java class that is called by the page and was loaded by the JSP class loader (not the system class loader) is modified
  - any page in the same application is reloaded

In a typical production environment, where source code will not change, comparing timestamps is unnecessary. In this case, you can avoid all timestamp comparisons

and any possible retranslation and reloading by setting the JSP `main_mode` flag to `justrun`. This will optimize program execution.

If you want to reload modified class files but not retranslate modified JSP pages, you can set `main_mode` to `reload`.

For more information about the `main_mode` flag, see "JSP Configuration Parameters" on page 3-9.

Notes:

- This discussion is not relevant for pre-translation scenarios.
- Because of the usage of in-memory values for class file last-modified times, removing a page implementation class file from the file system will *not* by itself cause retranslation of the associated JSP page source.
- The page implementation class file will be regenerated when the memory 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.
- In OC4J, if a statically included page is updated (that is, a page included through an `include` directive), the page that includes it will be automatically retranslated the next time it is invoked. (This is not true in JServ.)

Optimization Considerations

This section describes additional settings you can consider to optimize JSP performance.

Unbuffering a JSP Page

By default, a JSP page uses an area of memory known as a *page buffer*. This buffer (8 KB 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 the output step of copying the buffer.

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 direct that no session be used. Specify this with a `page` directive such as the following:

```
<%@ 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. For background information, see "Servlet Sessions" on page A-4.

7

JSP Translation and Deployment

This chapter discusses operation of the OC4J 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 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 JSP translator, focusing on its behavior in on-demand translation scenarios such as in OC4J in the Oracle9*i* Application Server. The following topics are covered:

- Features of Generated Code
- General Conventions for Output Names
- Generated Package and Class Names
- Generated Files and Locations
- Oracle JSP Global Includes

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 exact details are subject to change from release to release.

Features of Generated Code

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

Features of Page Implementation Class Code

When the 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 provided by the 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, expressions, and JSP tags 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"` in a `page` directive.

For introductory information about key JSP and servlet classes and interfaces, see Appendix A, "Servlet and JSP Technical Background".

Inner Class for Static Text

The service method, `_jspService()`, of the page implementation class includes print statements—`out.print()` or equivalent calls on the implicit `out` object—to print any static text in the JSP page. The 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 OC4J 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 6-7.) You can request this feature through the JSP `external_resource` configuration parameter for on-demand translation, or the `ojspc -extres` flag for pre-translation.

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

General Conventions for Output Names

The 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, however, 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 Oracle9*i*AS release 2, the base name is preceded by an underscore ("_"). Translating `MyPage123.jsp` results in the page implementation class `_MyPage123` in the 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 a directory such as `webapp/test`, the JSP translator by default will create a directory such as `webappdeployment/_pages/_test` for the page implementation class source. All output directories are created under the standard `_pages` directory, as described in "Generated Files and Locations" on page 7-6.

If you include special characters in a JSP page name or path name, the JSP translator takes steps to ensure that no illegal Java characters 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 chapter.

Generated Package and Class Names

Although the Sun Microsystems *JavaServer Pages Specification, Version 1.2* 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 OC4J JSP translator creates package and class names when it generates code during translation.

Note: For information about general conventions that the OC4J JSP translator uses in naming output classes, packages, and files, see "General Conventions for Output Names" on page 7-4.

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 the current OC4J JSP implementation, this results in the following package specification in the generated code:

```
package _hr._expenses;
```

(Implementation details are subject to change in future releases.)

No package name is generated if the JSP page is at the 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 the current OC4J JSP implementation, this yields the following class name in the generated code:

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

(Implementation details are subject to change in future releases.)

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.

Currently, the translator determines the case of the class name according to the case of the file name. For example:

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

This section describes files that are generated by the 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 7-32.

Wherever JSP configuration parameters are mentioned, see "JSP Configuration Parameters" on page 3-9 for more information.

Note: For information about general conventions used in naming output classes, packages, and files, see "General Conventions for Output Names" on page 7-4.

Files Generated by the JSP Translator

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

Source files:

- A `.sqlj` file (for example, `_Foo.sqlj`) is produced by the OC4J JSP translator if the page is a SQLJ JSP page.
- A `.java` file (for example, `_Foo.java`) is produced for the page implementation class and inner class. It is produced either directly by the 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 by using the `sqljcmd` JSP configuration parameter.

Binary files:

- In the case of a SQLJ JSP page with ISO code generation, 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 `sqljcmd` configuration parameter. The resource file or `.class` file has "Foo" as part of its name. (The default SQLJ code generation is `-codegen=oracle`, for Oracle-specific code. No profiles are generated for Oracle-specific code. For ISO code, use `-codegen=iso`.)
- A `.class` file is produced by the Java compiler for the page implementation class. The Java compiler is the JDK `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; in the current implementation it would be `_Foo$__jsp_StaticText.class`.
- A `.res` Java resource file (for example, `_Foo.res`) is optionally produced for the static page content if the `external_resource` JSP configuration parameter is enabled.

Note: The exact names of generated files for the page implementation class might 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 variations beyond that.

JSP Translator Output File Locations

The JSP translator places generated output files under a base `temp/_pages` directory, as in the following example:

```
[Oracle_Home]/j2ee/home/app-deployment/app-name/web-app-name/temp/_pages/...
```

Note the following, and refer to "Key OC4J Configuration Files" on page 3-23 for related information about the noted configuration files:

- The *app-deployment* directory is the OC4J deployment directory, specified in the OC4J `server.xml` file. It is typically the `application-deployments` directory.
- Also, *app-name* is the application name, according to an `<application>` element in `server.xml`.
- And *web-app-name* is the corresponding "Web application name", mapped to the application name in a `<web-app>` element in the OC4J `default-web-site.xml` file (or `http-web-site.xml` for OC4J standalone, or other Web site XML file as appropriate).

The path under the `_pages` directory depends on the path of the `.jsp` file under the application root directory.

As an example, consider the page `welcome.jsp` in the `examples/jsp` subdirectory under the OC4J standalone default Web application directory. The path would be as follows:

```
[Oracle_Home]/j2ee/home/default-web-app/examples/jsp/welcome.jsp
```

Assuming the default application deployment directory, the JSP translator would place the output files (`_welcome.java`, `_welcome.class`, and `_welcome$__jsp_StaticText.class` for the page implementation class inner class) in the following directory:

```
[Oracle_Home]/j2ee/home/application-deployments/default/defaultWebApp/temp/_pages/_examples/_jsp
```

Note the following for OC4J standalone. (Directories are configurable through Oracle Enterprise Manager in an Oracle9*i*AS environment.)

- The `application-deployments` directory is the OC4J default deployment directory.
- Also, `default` is the OC4J default application name and `defaultWebApp` is the default Web application name, both used for JSP pages placed in the `default-web-app` directory.
- Because the `.jsp` source file is in an `examples/jsp` subdirectory under the application root directory, the JSP translator generates `_examples._jsp` as the package name, and places the output files into an `_examples/_jsp` subdirectory under the `_pages` directory.

Important: Implementation details, such as the location of generated output files and use of "_" in output file names, are subject to change in future releases.

Oracle JSP Global Includes

The OC4J JSP container provides a feature called *global includes*. You can use this feature to specify one or more files to statically include into JSP pages in (or under) a specified directory, through virtual JSP `include` directives. During translation, the JSP container looks for a configuration file, `/WEB-INF/ojsp-global-include.xml`, that specifies the included files and the directories for the pages.

This enhancement is particularly convenient for migrating applications that used `globals.jsa` or `translate_params` functionality in previous Oracle JSP releases.

Globally included files can be used for the following, for example:

- global bean declarations (formerly supported through `globals.jsa`)
- common page headers or footers
- `translate_params` equivalent code (typically for a JServ environment)

The ojsp-global-include.xml File

The `ojsp-global-include.xml` file specifies the names of files to include, whether they should be included at the tops or bottoms of JSP pages, and the locations of JSP pages to which the global includes should apply. This section describes the elements of `ojsp-global-include.xml`.

<ojsp-global-include>

This is the root element of the `ojsp-global-include.xml` file. It has no attributes.

Subelements:

`<include>`

<include ... >

Use this subelement of `<ojsp-global-include>` to specify a file to be included, and whether it should be included at the top or bottom of JSP pages.

Subelements:

`<into>`

Attributes:

- `file`: Specify the file to be included, such as `"/header.html"` or `"/WEB-INF/globalbeandeclarations.jsph"`. The file name setting must start with a slash (`"/"`). In other words, it must be context-relative, not page-relative.
- `position`: Specify whether the file is to be included at the top or bottom of JSP pages. Supported values are "`top`" (default) and "`bottom`".

<into ... >

Use this subelement of `<include>` to specify a location (a directory, and possibly subdirectories) of JSP pages into which the specified file is to be included. This element has no subelements.

Attributes:

- `directory`: Specify a directory. Any JSP pages in this directory, and optionally its subdirectories, will statically include the file specified in the `file` attribute of the `<include>` element. The `directory` setting must start with a slash (`"/"`), such as `"/dir1"`. The setting can also include a slash after the directory

name, such as "/dir1/", or a slash will be appended internally during translation.

- subdir: Use this to specify whether JSP pages in all subdirectories of the directory should also have the file statically include. Supported values are "true" (default) and "false".

Global Include Examples

This section provides examples of global includes.

Example: Header/Footer Assume the following ojsp-global-include.xml file:

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE ojsp-global-include SYSTEM 'ojsp-global-include.dtd'>

<ojsp-global-include>
  <include file="/header.html">
     <into directory="/dir1" />
  </include>
  <include file="/footer1.html" position="bottom">
     <into directory="/dir1" subdir="false" />
     <into directory="/dir1/part1/" subdir="false" />
  </include>
  <include file="/footer2.html" position="bottom">
     <into directory="/dir1/part2/" subdir="false" />
  </include>
</ojsp-global-include>
```

This example accomplishes three objectives:

- The header.html file is included at the top of any JSP page in or under the dir1 directory. The result would be the same as if each .jsp file in or under this directory had the following include directive at the top of the page:

  ```
  <%@ include file="/header.html" %>
  ```

- The footer1.html file is included at the bottom of any JSP page in the dir1 directory or its part1 subdirectory. The result would be the same as if each .jsp file in those directories had the following include directive at the bottom of the page:

  ```
  <%@ include file="/footer1.html" %>
  ```

- The `footer2.html` file is included at the bottom of any JSP page in the `part2` subdirectory of `dir1`. The result would be the same as if each `.jsp` file in that directory had the following `include` directive at the bottom of the page:

  ```
  <%@ include file="/footer2.html" %>
  ```

  Note: If multiple header or multiple footer files are included into a single JSP page, the order of inclusion is according to the order of `<include>` elements in the `ojsp-global-include.xml` file.

Example: translate_params Equivalent Code Assume the following `ojsp-global-include.xml` file:

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE ojsp-global-include SYSTEM 'ojsp-global-include.dtd'>

<ojsp-global-include>
  <include file="/WEB-INF/nls/params.jsf">
     <into directory="/" />
  </include>
</ojsp-global-include>
```

And assume `params.jsf` contains the following:

```
<% request.setCharacterEncoding(response.getCharacterEncoding()); %>
```

The `params.jsf` file (essentially, the `setCharacterEncoding()` method call) is included at the top of any JSP page in or under the application root directory. In other words, it is included in any JSP page in the application. The result would be the same as if each `.jsp` file in or under this directory had the following `include` directive at the top of the page:

```
<%@ include file="/WEB-INF/nls/parms.jsf" %>
```

Also see "Migration Away from translate_params" on page B-30.

The ojspc Pre-Translation Utility

This section describes the `ojspc` utility, provided with OC4J for pre-translation of JSP pages. For consideration of pre-translation scenarios, see "JSP Pre-Translation" on page 7-37 and "Deployment of Binary Files Only" on page 7-40.

The following topics are covered here:

Important: To use `ojspc`, you must be using a Sun Microsystems JDK (version 1.1.8 or higher) and you must have `tools.jar` (for JDK 2.0 or higher) or `classes.zip` (for JDK 1.1.8) in your classpath.

Overview of Basic ojspc Functionality

For a simple JSP (not SQLJ JSP) page, default functionality for `ojspc` is as follows:

- It takes a JSP file (typically `.jsp`), either directly as an argument or from an archive file taken as an argument.
- It invokes the 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.

Following is the default `ojspc` functionality for a SQLJ JSP page:

- It takes a SQLJ JSP file (a `.sqljsp` file or a JSP file with `language="sqlj"` in a `page` directive), either directly as an argument or from an archive file taken as an argument.

- It invokes the JSP translator to translate the SQLJ JSP page 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, for ISO code generation, translation produces a SQLJ "profile" file that is, by default, a `.ser` Java resource file.

  For information about SQLJ profiles and Oracle-specific code generation, see the *Oracle9i SQLJ Developer's Guide and Reference*.

Note: The default SQLJ code generation setting is `-codegen=oracle`, for Oracle-specific code (with no profiles). For ISO code generation, specify `-codegen=iso`.

- 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 (as noted in the `-extres` option description), `ojspc` options direct the 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 JSP translator, `ojspc` output conventions are the same as for the translator in general, as applicable. For general information about 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 JSP Translator" on page 7-2.

Note: The `ojspc` command-line tool is a front-end utility that invokes the `oracle.jsp.tool.Jspc` class.

Overview of ojspc Batch Pre-Translation

Prior to Oracle9*i*AS release 2 (9.0.3), `ojspc` accepted only JSP files (or SQLJ JSP files) for translation. Now, however, it can also accept archive files—JAR, WAR, EAR, or ZIP files—for *batch pre-translation*.

> **Note:** The `ojspc` utility does not depend on the file name extension to determine whether a file is an archive file. It makes the determination by examining the internal file structure.

When the name of an archive file appears on the `ojspc` command line, `ojspc` by default executes the following steps:

1. Opens the archive file.
2. Translates and compiles all `.jsp` and `.sqljsp` files in the archive file.
3. Adds the resulting `.class` files, and any Java resource files, into the archive file (and discards `.java` and `.sqlj` files that were created in the process). The `.class` and resource files are added with directory paths such that upon extraction, they will be located in the same directory as would be the case if the original JSP files were translated after extraction.

> **Note:** The actual mechanics are that the original archive file is extracted into a temporary storage area (recursively if there are nested archive files), a temporary archive file is created, contents of the original archive file are copied into the temporary file, output `.class` and resource files from pre-translation are added to the temporary file, the original archive file is deleted, and the temporary file is given the name of the original file. (The original archive file is extracted in its entirety to ensure successful compilation of the translated pages.)

There are `ojspc` settings for additional functionality, as follows:

- You can use the `-batchMask` option to specify file name extensions for pre-translation. Whatever you specify is in addition to the defaults, which are `*.jsp` and `*.sqljsp`.
- You can use the `-output` option to specify a new archive file name. In this case, all contents of the original archive file are copied into the specified archive file, then the output `.class` files (and any resource files) from pre-translation are added to the specified file. The original archive file is unaltered, and you would use the new file instead of the original file for deployment.
- You can use the `-deleteSource` option if you do not want the JSP source files to appear in the resulting archive file. If you use `-deleteSource` without

using the `-output` option, then the contents of the original archive file are overwritten so that no pre-translated JSP source files are included. If you use both `-deleteSource` and `-output`, then the new archive file is created without any pre-translated JSP source files. The `-deleteSource` option applies to all JSP files that are pre-translated—`*.jsp` and `*.sqljsp` files plus files with extensions specified in the `-batchMask` setting.

For examples of these options, see the descriptions of those options under "Option Descriptions for ojspc" on page 7-20.

Option Summary Table for ojspc

Table 7–1 summarizes the options supported by the `ojspc` pre-translation utility. These options are further discussed in "Option Descriptions for ojspc" on page 7-20.

The second column notes comparable or related JSP configuration parameters for on-demand translation environments, such as OC4J.

Note: For a JServ environment, use the `ojspc_jserv` command instead of the `ojspc` command. See "Using ojspc for JServ" on page B-16. Be aware that the `-staticTextInChars` option is not relevant for JServ, so is not supported by `ojspc_jserv`.

Table 7–1 Options for ojspc Pre-Translation Utility

Option	Related JSP Configuration Parameters	Description	Default
-addclasspath	(none)	Specify additional classpath entries for `javac`.	empty (no additional path entries)
-appRoot	(none)	Specify the application root directory for application-relative static `include` directives from the page.	current directory
-batchMask	(none)	For batch pre-translation, optionally specify additional file name extensions for pre-translation.	`*.jsp`, `*.sqljsp`

Table 7–1 Options for ojspc Pre-Translation Utility (Cont.)

Option	Related JSP Configuration Parameters	Description	Default
-d	(none)	Specify the location where `ojspc` should place generated binary files (`.class` and resource). Do not use this option for batch pre-translation.	current directory
-debug	emit_debuginfo	Enabling this flag directs `ojspc` to generate a line map to the original `.jsp` file for debugging.	`false`
-deleteSource	(none)	For batch pre-translation, enabling this flag directs that JSP source files should be removed from (or not copied to) the resulting archive file.	`false`
-extend	(none)	Specify the class for the generated page implementation class to extend. Do not use this option for batch pre-translation.	empty
-extraImports	extra_imports	Use this to add imports beyond the JSP defaults.	empty
-extres	external_resource	Enabling this flag directs `ojspc` to generate an external resource file for static text from the `.jsp` file.	`false`
-forgiveDupDirAttr	forgive_dup_dir_attr	Enable this flag in order to avoid JSP 1.2 translation errors if you have duplicate settings for the same directive attribute within a single JSP translation unit.	`false`
-help (or -h)	(none)	Enabling this flag directs `ojspc` to display usage information.	`false`

Table 7–1 Options for ojspc Pre-Translation Utility (Cont.)

Option	Related JSP Configuration Parameters	Description	Default
-implement	(none)	Specify an interface for the generated page implementation class to implement. Do not use this option for batch pre-translation.	empty
-noCompile	javaccmd	Enabling this flag directs `ojspc` to *not* compile the generated page implementation class.	`false`
-noTldXmlValidate	no_tld_xml_validate	Enable this flag in order to *disable* XML validation of TLD files. By default, validation of TLD files is performed.	`false`
-oldIncludeFromTop	old_include_from_top	Enable this flag in order to specify that page locations in nested `include` directives are relative to the top-level page, for backward compatibility with Oracle JSP behavior prior to Oracle9*i*AS release 2.	`false`
-output	(none)	For batch pre-translation, optionally specify the name of the output archive file.	original archive file
-packageName	(none)	Specify the package name for the generated page implementation class.	empty (generate package names according to `.jsp` file location)
-reduceTagCode	reduce_tag_code	Enable this flag in order to specify further reduction in the size of generated code for custom tag usage.	`false`

Table 7–1 Options for ojspc Pre-Translation Utility (Cont.)

Option	Related JSP Configuration Parameters	Description	Default
-reqTimeIntrospection	req_time_introspection	Enable this flag in order to allow request-time JavaBean introspection whenever compile-time introspection is not possible.	`false`
-S-<*sqlj_option*>	sqljcmd	Use the `-S` prefix followed by an Oracle SQLJ option (for SQLJ JSP pages).	empty
-srcdir	(none)	Specify the location where `ojspc` should place generated source files (`.java` and `.sqlj`). Do not use this option for batch pre-translation.	current directory
-staticTextInChars	static_text_in_chars	Enable this flag in order to instruct the JSP translator to generate static text in JSP pages as characters instead of bytes.	`false`
-verbose	(none)	Enabling this flag directs `ojspc` to print status information as it executes.	`false`
-version	(none)	Enabling this flag directs `ojspc` to display the JSP version number.	`false`
-xmlValidate	xml_validate	Enable this flag in order to perform XML validation of the `web.xml` file. By default, validation of `web.xml` is *not* performed.	`false`

Command-Line Syntax for ojspc

Following is the general `ojspc` command-line syntax (where `%` is the system prompt):

```
% ojspc file_list [option_settings]
```

The file list can include JSP files, including SQLJ JSP files, or archive files (JAR, WAR, EAR, or ZIP 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 settings).
- Use spaces between file names in the file list.
- Use spaces as delimiters 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 (flags), which are disabled by default, by simply typing the option name in the command line. For example, type `-extres`, *not* `-extres true`.

Here are two examples:

```
% ojspc MyPage.sqljsp MyPage2.jsp -d /myapp/mybindir -srcdir /myapp/mysrcdir -extres

% ojspc myapp.war -deleteSource
```

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.

> **Note:** 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 your current directory when you ran `ojspc`.

The specified application root directory path is used as follows:

- 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.

- 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 7-32.)

This option is necessary, for example, so that 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 (where % 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 example 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 when you ran `ojspc`. (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 /abc/def/ghi/test.jsp -appRoot /abc
```

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.

-batchMask (additional file name extensions; `ojspc` default: `"*.jsp,*.sqljsp"`)

For batch pre-translation, you can use this option to specify file name extensions for pre-translation. By default, `.jsp` and `.sqljsp` files are pre-translated. Extensions specified through the `-batchMask` option are in addition to the default extensions.

Place quotes around the list of file name extensions, and use commas or semicolons as delimiters within the list. White space before or after a file name extension is ignored.

The following examples (where % is the system prompt) result in the same action, given that `.jsp` and `.sqljsp` files are pre-translated anyway, and that commas are equivalent to semicolons as delimiters):

```
% ojspc myapp.war -batchMask "*.jspf,*.jsph,*.jsp,*.sqljsp"

% ojspc myapp.zip -batchMask "*.jspf; *.jsph"
```

Notes:

- File name extensions specified in this option are *not* case-sensitive.
- There is no support for specifying directory paths or Java packages in the `-batchMask` setting.

-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 with SQLJ ISO code generation.)

The specified path is taken 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 7-32 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:

- Do not use `-d` for batch pre-translation.
- In environments such as Windows NT that allow spaces in directory names, enclose the directory name in quotes.

-debug (boolean; `ojspc` default: `false`)

Enabling this flag instructs `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 flag is useful for source-level JSP debugging, such as when you use Oracle9*i* JDeveloper.

Note: In an on-demand translation scenario, the JSP `emit_debuginfo` configuration parameter provides the same functionality.

-deleteSource (boolean; `ojspc` default: `false`)

For batch pre-translation, enable this flag if you do not want JSP source files that were pre-translated to appear in the resulting archive file. This is all `.jsp` and `.sqljsp` files by default, plus files with name extensions specified in the `-batchMask` option.

If you do not use the `-output` option—that is, if the original archive file is being updated and is also the output archive file—this means that the contents of the archive file are overwritten to remove any JSP files that are pre-translated. If you do use the `-output` option, this means that any JSP files that are pre-translated will not be copied to the specified output archive file. (The original archive file is unaltered.)

Note: As in any situation where JSP source files are not deployed, after you have used `-deleteSource` the target JSP runtime environment must be configured to operate properly without having source files available. See "Configuring the OC4J JSP Container for Execution with Binary Files Only" on page 7-41.

-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.

Note: Do not use `-extend` for batch pre-translation.

-extraImports (import list; `ojspc` default: empty)

As described in "Default Package Imports" on page 3-5, as of Oracle9*i*AS release 2 (9.0.3) the OC4J JSP container has a smaller default list of packages that are imported into each JSP page. This is in accordance with the JSP specification. You can avoid updating your code, however, by specifying package names or fully qualified class names for any additional imports through the `-extraImports` option. Be aware that the names must be comma-delimited, with no spaces, as in the following example:

```
% ojspc foo.jsp -extraImports java.util.*,java.io.*
```

Note:

- In an on-demand translation scenario, the JSP `extra_imports` configuration parameter provides the same functionality.
- As an alternative to using `-extraImports`, you can use global includes. See "Oracle JSP Global Includes" on page 7-9.

-extres (boolean; `ojspc` default: `false`)

Enabling this flag instructs `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. In the current OC4J JSP implementation, 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 might 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 6-7.

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.

-forgiveDupDirAttr (boolean; `ojspc` default: `false`)

Enabling this flag avoids translation errors in JSP 1.2 (or higher) if you have duplicate settings for the same directive attribute within a single JSP translation unit (a JSP page plus anything it includes through `include` directives).

The JSP 1.2 specification directs that a JSP container must verify that directive attributes, with the exception of the `page` directive `import` attribute, are not set

more than once each within a single JSP translation unit. See "Duplicate Settings of Page Directive Attributes Are Disallowed" on page 6-11 for more information.

The JSP 1.1 specification does *not* specify such a limitation. OC4J offers the `-forgiveDupDirAttr` option for backward compatibility.

Note: In an on-demand translation scenario, the JSP `forgive_dup_dir_attr` configuration parameter provides the same functionality.

-help (boolean; `ojspc` default: `false`)

Use this option to have `ojspc` display usage information and then exit. As a shortcut, `-h` is also accepted.

-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.

Note: Do not use `-implement` for batch pre-translation.

-noCompile (boolean; `ojspc` default: `false`)

Enabling this flag directs `ojspc` to *not* compile the generated page implementation class Java source. This is in case you want to compile it later for some reason, such as with an alternative Java compiler.

Notes:

- In an on-demand translation scenario, the JSP `javaccmd` configuration parameter provides related functionality. It enables you to specify a complete Java compiler command line, optionally using an alternative compiler.
- For a SQLJ JSP page, enabling `-noCompile` does not prevent SQLJ translation, just Java compilation.

-noTldXmlValidate (boolean; `ojspc` default: `false`)

Enable this flag if you do *not* want XML validation of tag library descriptor (TLD) files of the application. By default, validation of TLD files is performed.

See "Overview of TLD File Validation and Features" on page 8-8 for related information.

Note: In an on-demand translation scenario, the JSP `no_tld_xml_validate` configuration parameter provides the same functionality.

-oldIncludeFromTop (boolean; `ojspc` default: `false`)

This is for backward compatibility with Oracle JSP versions prior to Oracle9*i*AS release 2, for functionality of `include` directives. If you enable this flag, page locations in nested `include` directives are relative to the top-level page. Otherwise, page locations are relative to the immediate parent page. This complies with the JSP 1.2 specification.

Note: In an on-demand translation scenario, the JSP `old_include_from_top` configuration parameter provides the same functionality.

-output (archive file name; `ojspc` default: none)

For batch pre-translation, use the `-output` option if you want to specify a new archive file for output. In this case, all contents of the original archive file are copied into the specified archive file, then the output `.class` files (and any resource files) from pre-translation are added to the specified file. The original archive file is unaltered, and you would use the new file instead of the original file for deployment.

Without the `-output` option, the original archive file is updated to add output `.class` (and resource) files; no new archive file is created.

Here is an example of `-output` usage:

```
% ojspc myapp.war -output myappout.war
```

-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 your current directory when 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 (where `%` is a UNIX prompt):

```
% cd /myapproot
% ojspc src/jspsrc/Foo.jsp -packageName myroot.mypackage
```

This results in `myroot.mypackage` being used as the package name.

If this example did *not* use the `-packageName` option, the JSP translator (in its current implementation) would use `_src._jspsrc` as the package name, by default. (Be aware that such implementation details are subject to change in future releases.)

-reduceTagCode (boolean; `ojspc` default: `false`)

The Oracle JSP implementation reduces the size of generated code for custom tag usage, but enabling this flag results in even further size reduction. There may be performance consequences regarding tag handler reuse, however. See "Tag Handler Code Generation" on page 8-40.

Note: In an on-demand translation scenario, the JSP `reduce_tag_code` configuration parameter provides the same functionality.

-reqTimeIntrospection (boolean; `ojspc` default: `false`)

Enabling this flag allows request-time JavaBean introspection whenever compile-time introspection is not possible. When compile-time introspection is possible and succeeds, this parameter is ignored and there is no request-time introspection.

As a sample scenario for request-time introspection, assume a tag handler returns a generic `java.lang.Object` instance in the `VariableInfo` instance of the tag-extra-info class during translation and compilation, but actually generates more specific objects during request-time (runtime). In this case, if

`req_time_introspection` is enabled, the JSP container will delay introspection until request-time. (See "Scripting Variables, Declarations, and Tag-Extra-Info Classes" on page 8-41 for information about use of `VariableInfo`.)

> **Note:** In an on-demand translation scenario, the JSP `req_time_introspection` configuration parameter provides the same functionality.

-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`, where each specifies one SQLJ option setting.

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 (where % is a UNIX prompt):

```
% ojspc MyPage.jsp -S-codegen iso -d /myapproot/mybindir
```

This directs SQLJ to generate ISO standard code instead of the default Oracle-specific code.

Here is another example:

```
% ojspc MyPage.jsp -S-codegen iso -S-ser2class true -d /myapproot/mybindir
```

This again directs SQLJ to generate ISO standard code, and also enables the `-ser2class` option in order to convert the profile to a `.class` file.

> **Note:** As the preceding example shows, you must 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` or `pageEncoding` attribute 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. It enables you to enter a complete SQLJ command line, with desired option settings and optionally an alternative SQLJ compiler.

-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 7-32 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:

- Do not use `-srcdir` for batch pre-translation.
- In environments such as Windows NT that allow spaces in directory names, enclose the directory name in quotes.

-staticTextInChars (boolean; `ojspc` default: `false`)

Enabling this flag directs the JSP translator to generate static text in JSP pages as characters instead of bytes. The default setting is `false`, which improves performance in outputting static text blocks.

Enable this flag if your application requires the ability to change the character encoding dynamically during runtime, such as in the following example:

```
<% response.setContentType("text/html; charset=UTF-8"); %>
```

> **Note:** In an on-demand translation scenario, the JSP `static_text_in_chars` configuration parameter provides the same functionality.

-verbose (boolean; `ojspc` default: `false`)

Enabling this flag directs `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 myerror.jsp -verbose
Translating file: myerror.jsp
1 JSP files translated successfully.
Compiling Java file: ./_myerror.java
```

-version (boolean; `ojspc` default: `false`)

Use this option to have `ojspc` display the JSP version number and then exit.

-xmlValidate (boolean; `ojspc` default: `false`)

Enable this flag if you want XML validation of the application `web.xml` file. Because the Tomcat JSP reference implementation does not perform XML validation, this flag is disabled by default.

> **Note:** In an on-demand translation scenario, the JSP `xml_validate` configuration parameter provides the same functionality.

Summary of ojspc Output Files, Locations, and Related Options

By default, `ojspc` generates the same set of files that are generated by the JSP translator in an on-demand translation scenario, and (not considering batch pre-translation) places them in or under your current directory when you ran `ojspc`.

Here are the files:

- for SQLJ JSP pages—a `.sqlj` source file (for batch pre-translation, discarded after compilation)
- a `.java` source file (for batch pre-translation, discarded after compilation)
- a `.class` file for the page implementation class
- a `.class` file for the inner class for static text
- for SQLJ JSP pages using ISO code generation—a Java resource file (`.ser`), or optionally a `.class` file, for the SQLJ profile
- optionally, a Java resource file (`.res`) for the static text of the page

For more information about files that are generated by the JSP translator, see "Generated Files and Locations" on page 7-6.

To summarize some of the commonly used options described under "Option Descriptions for ojspc" on page 7-20, 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 location (not relevant for batch pre-translation)
- `-d` to place binary files (`.class` files and Java resource files) in a specified location (not relevant for batch pre-translation)
- `-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 (not considering batch pre-translation), 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, suppose you run `ojspc` as follows (where % 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 that you run `ojspc` from some other directory, as follows:

```
% cd /home/mydir
% ojspc /abc/def/ghi/test.jsp -appRoot /abc
```

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.

Note: 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.

JSP Deployment Considerations

This section covers general deployment considerations and scenarios, mostly independent of your target environment.

It discusses the following topics:

- Overview of EAR/WAR Deployment
- Application Deployment with Oracle9i JDeveloper
- JSP Pre-Translation
- Deployment of Binary Files Only

Overview of EAR/WAR Deployment

This section provides an overview of OC4J deployment features and standard WAR deployment features.

See *Oracle9iAS Containers for J2EE User's Guide* for detailed information about deployment to OC4J in an Oracle9*i*AS environment.

OC4J Deployment Features

In OC4J, deploy each application through a standard EAR (Enterprise archive) file. Specify the name of the application and the name and location of the EAR file through an `<application>` element in the OC4J `server.xml` file. (This file is in the OC4J configuration files directory. In Oracle9*i*AS, directory paths are configurable; in OC4J standalone, the configuration files directory is `j2ee/home/config` by default.)

For production, use Enterprise Manager for deployment. Enterprise Manager is recommended for managing OC4J and other components of Oracle9*i*AS in a production environment. Refer to the *Oracle9i Application Server Administrator's Guide* and *Oracle Enterprise Manager Administrator's Guide* for information.

OC4J also supports the `admin.jar` tool for deployment, typically in an OC4J standalone development environment. This modifies `server.xml` and other configuration files for you, based on settings you specify to the tool. Or you can modify the configuration files manually (not generally recommended). Note that if you modify configuration files in Oracle9*i*AS without going through Enterprise Manager, you must run the `dcmctl` tool, using its `updateConfig` command, to inform Oracle9*i*AS Distributed Configuration Management (DCM) of the updates. (This does not apply in an OC4J standalone mode, where OC4J is being run apart from Oracle9*i*AS.)

Here is the `dcmctl` command:

```
dcmctl updateConfig -ct oc4j
```

The `dcmctl` tool is documented in the *Oracle9i Application Server Administrator's Guide*.

The EAR file includes the following:

- a standard `application.xml` configuration file, in `/META-INF`
- optionally, an `orion-application.xml` configuration file, in `/META-INF`
- a standard WAR (Web archive) file

The WAR file includes the following:

- a standard `web.xml` configuration file, in `/WEB-INF`

  In the `web.xml` file for any particular application, you can override global settings for individual configuration parameters or for the definition of the JSP servlet (`oracle.jsp.runtimev2.JspServlet` by default). Each application uses its own instance of the JSP servlet.
- optionally, an `orion-web.xml` configuration file, in `/WEB-INF`
- classes necessary to run the application (servlets, JavaBeans, and so on), under `WEB-INF/classes` and in JAR files in `WEB-INF/lib`
- JSP pages and static HTML files

The EAR file goes in the OC4J applications directory, which is specified in the `application-directory` setting in the `<application-server>` element of the `server.xml` file (for example, `j2ee/home/applications`). This would be the same directory as is specified for the EAR file location in the `<application>` element in `server.xml`.

Through the OC4J auto-deployment feature, a new EAR file in the applications directory (as specified in `server.xml`) is detected automatically and hierarchically extracted.

See the *Oracle9iAS Containers for J2EE User's Guide* for more information about deployment to Oracle9*i*AS. See the standalone version of this document for information about `admin.jar` (which has additional uses as well). Also see "Key OC4J Configuration Files" on page 3-23 for a summary of important configuration files in OC4J.

Standard WAR Deployment

The Sun Microsystems *JavaServer Pages Specification, Version 1.1* (and higher) supports the packaging and deployment of Web applications, including JavaServer Pages, according to version 2.2 and higher of the Sun Microsystems *Java Servlet Specification*.

In typical JSP 1.2 implementations, you can deploy JSP pages through the WAR mechanism, creating WAR files through the JAR utility. The JSP pages can be delivered in source form and are deployed along with any required support classes and static HTML files.

According to the servlet specification, versions 2.2 and higher, a Web application includes a deployment descriptor file—`web.xml`—that contains information about the JSP pages and other components of the application. The `web.xml` file must be included in the WAR file.

The servlet specification also defines an XML DTD for `web.xml` deployment descriptors and specifies exactly how a servlet container must deploy a Web application to conform to the deployment descriptor.

Through these logistics, a WAR file is the best way to ensure that a Web application is deployed into any standard servlet environment exactly as the developer intends.

Deployment configurations in the `web.xml` deployment descriptor include mappings between servlet paths and the JSP pages and servlets that will be invoked. You can specify many additional features in `web.xml` as well, such as timeout values for sessions, mappings of file name extensions to MIME types, and mappings of error codes to JSP error pages.

For more information about standard WAR deployment, see the Sun Microsystems *Java Servlet Specification, Version 2.2* or *Version 2.3*.

Application Deployment with Oracle9*i* JDeveloper

Oracle9*i* JDeveloper supports many types of deployment profiles, including simple archive, J2EE application (EAR file), J2EE EJB module (EJB JAR file), J2EE Web module (WAR file), J2EE client module (client JAR file), tag library for JSP 1.2 (tag library JAR file), business components EJB session bean profile, business components CORBA server for VisiBroker, and business components archive profile.

When creating a Business Components for Java (BC4J) Web application using Oracle9*i* JDeveloper, a J2EE Web module deployment archive is generated, containing both the BC4J and the Web application files.

The JDeveloper deployment wizards create all the necessary code to deploy business components as a J2EE Web module. Typically, a JSP client accesses the BC4J application in a J2EE Web Module configuration. The JSP client can also use data tags, data Web beans, or UIX tags to access the business components. (See the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference* for an overview of the BC4J and UIX tag libraries.)

A J2EE Web module is packaged as a WAR file that contains one or more Web components (servlets and JSP pages) and `web.xml`, the deployment descriptor file.

JDeveloper lets you create the deployment profile containing the Web components and the `web.xml` file, and packages them into a standard J2EE EAR file for deployment. JDeveloper takes the resulting EAR file and deploys it to one or more Oracle9*i*AS instances.

For information about JDeveloper, refer to the JDeveloper online help, or to the following site on the Oracle Technology Network:

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

JSP Pre-Translation

JSP pages are typically used in an on-demand scenario, where pages are translated as they are invoked, in a sequence that is invisible to the user. Another approach is to pre-translate JSP pages, which offers at least two advantages:

- It saves end users the translation overhead the first time a page is invoked.
- It ensures that the developer or deployer, instead of end users, will see any translation or compilation errors.

You also might want to pre-translate pages so that you can deploy binary files only, as discussed in "Deployment of Binary Files Only" on page 7-40.

OC4J users can employ the Oracle `ojspc` utility for pre-translation, either specifying individual files, or specifying archive files (JAR, WAR, EAR, or ZIP) for batch pre-translation. There is also a standard `jsp_precompile` mechanism. These topics are covered in the following subsections:

- Techniques for Page Pre-Translation with ojspc
- Batch Pre-Translation with ojspc
- Standard JSP Pre-Translation without Execution

Also see "The ojspc Pre-Translation Utility" on page 7-13 for detailed information about this utility.

Techniques for Page Pre-Translation with ojspc

When you pre-translate with `ojspc` (not considering batch pre-translation), use the `-d` option to set an appropriate output base directory for placement of generated binary files.

Consider the example in "JSP Translator Output File Locations" on page 7-8, where the JSP page is located in the `examples/jsp` subdirectory under the OC4J standalone default Web application directory:

```
[Oracle_Home]/j2ee/home/default-web-app/examples/jsp/welcome.jsp
```

A user would invoke this with a URL such as the following:

```
http://host[:port]/examples/jsp/welcome.jsp
```

(This is just a general example and does not consider OC4J default configuration for the context path.)

In an on-demand translation scenario for this page, as explained in the example, the JSP translator would by default use the following base directory for placement of generated binary files:

```
[Oracle_Home]/j2ee/home/application-deployments/default/defaultWebApp/temp/_pages
```

When you pre-translate, set your current directory to the application root directory, then in `ojspc` set the `_pages` directory as the output base directory. This results in the appropriate package name and file hierarchy. Continuing the example (where % is a UNIX prompt):

```
% cd [Oracle_Home]/j2ee/home/default-web-app
% ojspc examples/jsp/welcome.jsp
-d [Oracle_Home]/j2ee/home/application-deployments/default/defaultWebApp/temp/_pages
```

(This assumes you specify the appropriate `Oracle_Home` directory.) The `ojspc` command is a single wraparound command, translating `examples/jsp/welcome.jsp` and specifying the `_pages` directory as the base output directory.

The URL noted above specifies an application-relative path of `examples/jsp/welcome.jsp`, so at execution time the JSP container looks for the binary files in an `_examples/_jsp` subdirectory under the `_pages` directory. This

subdirectory would be created automatically by `ojspc` if it is run as in the above example.

At execution time, the JSP container would find the pre-translated binaries and would not have to perform translation, assuming that either the source file was not altered after pre-translation, or the JSP `main_mode` flag is set to `justrun`.

Note: OC4J JSP implementation details, such as use of underscores ("_") in output directory names, are subject to change from release to release. This documentation applies specifically to Oracle9*i*AS release 2.

Batch Pre-Translation with ojspc

Beginning with the OC4J 9.0.3 implementation, there are `ojspc` features for batch pre-translation of JSP files in archive files (JAR, WAR, EAR, or ZIP files). When you specify an archive file on the `ojspc` command line, by default all `.jsp` and `.sqljsp` files in the contents will be pre-translated, and the archive file will be updated to include the output `.class` files and any Java resource files (but not `.java` or `.sqlj` files). You would then deploy the resulting archive file.

In addition to this basic functionality, you can use `ojspc` options to specify any of the following:

- Use the `-batchMask` option to specify file name extensions for pre-translation in addition to the default extensions `*.jsp` and `*.sqljsp`.
- Use the `-output` option to specify a new archive file for output. After pre-translation, this file will have copies of all the contents of the original archive file, as well as the output `.class` files (and any resource files) resulting from pre-translation. The original archive file will be unaltered.
- Use the `-deleteSource` option to specify that pre-translated JSP files will not be included in the resulting archive file. If you use the `-output` option, the JSP files will not appear in the specified output archive file. If you do not use the `-output` option, the JSP files will be removed from the original archive file. This applies to whatever JSP files are pre-translated (`.jsp` and `.sqljsp` files plus any files according to the `-batchMask` setting).

Standard JSP Pre-Translation without Execution

It is also possible to specify JSP pre-translation, without execution, when you invoke the page in the normal way. Accomplish this as follows:

1. Enable the JSP `precompile_check` configuration parameter. (See "JSP Configuration Parameters" on page 3-9.)
2. Enable the standard `jsp_precompile` request parameter when invoking the JSP page from the browser.

Following is an example of using `jsp_precompile`:

```
http://host[:port]/foo.jsp?jsp_precompile=true
```

or:

```
http://host[:port]/foo.jsp?jsp_precompile
```

(The "`=true`" is optional.)

Refer to the Sun Microsystems *JavaServer Pages Specification, Version 1.2*, for more information about this mode of operation.

Deployment of Binary Files Only

You can avoid exposing your JSP source, for proprietary or security reasons, by pre-translating the 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 standard J2EE environment. This involves two steps:

1. You must archive and deploy the binary files appropriately.
2. In the target environment, the JSP container must be configured to run pages without the JSP source being available.

Archiving and Deploying the Binary Files

You must take steps to create and archive the binary files in an appropriate hierarchy.

- If you pre-translate with `ojspc`, you must first set your current directory to the application root directory. After running `ojspc`, archive the output files using the `ojspc` output directory as the base directory for the archive. See "The ojspc Pre-Translation Utility" on page 7-13 for general information about this utility.

- If you are archiving binary files produced during previous execution in an on-demand translation environment, then archive the output directory structure, typically under the `_pages` directory.

In the target environment, place the archive JAR file in the `/WEB-INF/lib` directory; or restore the archived directory structure under the appropriate directory, typically under the `_pages` directory.

Configuring the OC4J JSP Container for Execution with Binary Files Only

If you have deployed binary files to an OC4J environment, set the JSP configuration parameter `main_mode` to the value `justrun` or `reload` to execute JSP pages without the original source.

Without this setting, the JSP translator will always look for the JSP source file to see if it has been modified more recently than the page implementation `.class` file, and terminate with a "file not found" error if it cannot find the source file.

With `main_mode` 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 how to set configuration parameters in the OC4J environment, see "Setting JSP Configuration Parameters in OC4J" on page 3-20.

8

JSP Tag Libraries

This chapter discusses custom tag libraries, covering the basic framework that vendors can use to provide their own libraries. There is also discussion of Oracle extensions, and a comparison of standard runtime tags versus vendor-specific compile-time tags. The chapter is organized as follows:

The chapter offers a detailed overview of standard tag library functionality. For complete information, refer to the Sun Microsystems *JavaServer Pages Specification, Version 1.2* (or higher). For information about the tag libraries provided with OC4J, see the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Custom tag syntax largely follows XML conventions. For general information about XML, you can find the specification at the following Web site:

`http://www.w3.org/XML/`

Overview: Tag Library Framework

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.

This section provides an overview of the JSP tag library framework, as well as a summary of new tag library features in the JSP 1.2 specification.

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 1.2*
- Sun Microsystems Javadoc for the `javax.servlet.jsp.tagext` package, at the following Web site:

`http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/jsp/tagext/package-summary.html`

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 points about implementation and usage of a tag library.

- The tags of a library are defined in a *tag library descriptor* (TLD) file, as "Tag Library Descriptor Files" on page 8-8 describes.
- The URI in the `taglib` directive is a pointer to the TLD file, as "Overview: Specifying a Tag Library with the taglib Directive" on page 8-16 discusses. It is possible to use *URI shortcuts*, as "Use of web.xml for Tag Libraries" on page 8-21 explains.
- 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 that 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 TLD file that can be found through the URI specified in the above `taglib` directive.

- The entry for a tag in the TLD file provides specifications about use 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 "Tag Handlers" on page 8-25 describes. Each tag has its own tag handler class, and the class name is specified in the TLD file.
- A tag attribute can be of any standard Java type or an object type—either the generic `java.lang.Object` or a user-defined type.

  You typically set an attribute of a standard Java type as a string value. The appropriate conversion is handled automatically.

  You can also set an attribute of type `Object` with a string value—the string is converted to an `Object` instance and passed in to the corresponding setter method in the tag handler instance. This feature complies with the JSP 1.2 specification.

  An attribute of a user-defined type must be set using a request-time expression that returns an instance of the type.
- The TLD file indicates whether a tag uses a body.

  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 known as *scripting variables*.

You can declare a scripting variable through a `<variable>` element in the TLD file or through a *tag-extra-info* class. See "Scripting Variables, Declarations, and Tag-Extra-Info Classes" on page 8-41 for more information.

A tag can create and use scripting variables with syntax such as in the following example, which creates the object `myobj`:

```
<oracust:mytag id="myobj" attr1="...", attr2="..." />
```

- The TLD file can optionally declare a *tag-library-validator* class for use with the tag library. This class would have logic to validate any JSP page that uses the tag library, according to specified constraints. See "Validation and Tag-Library-Validator Classes" on page 8-46.
- The TLD file can optionally declare one or more *event listeners* for use with the tag library. This functionality is offered as a convenient alternative to declaring listeners in the application `web.xml` file. See "Tag Library Event Listeners" on page 8-50.
- 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 8-37.

The remainder of this chapter provides details about these topics.

Note: The OC4J JSP container supports tag library features for the JServ environment as well, with the exception of tag library event listeners.

Overview of Tag Library Changes Between the JSP 1.1 and 1.2 Specifications

The JSP 1.2 specification adds features for improved tag library support in the following areas:

- tag library descriptor features

  New features are outlined in the next section, "Summary of TLD File Changes Between the JSP 1.1 and 1.2 Specifications". "Tag Library Descriptor Files" on page 8-8 describes TLD features in detail.

- support for multiple tag libraries and their TLD files in a single JAR file

  According to the JSP 1.1 specification, you cannot have multiple TLD files packaged in a single JAR file. The JSP 1.2 specification adds support for this, however. See "Tag Handlers" on page 8-25.

- tag handler features

  New features are summarized in "Summary of Tag Handler Changes Between the JSP 1.1 and 1.2 Specifications" on page 8-7. Tag handler features are described in detail in "Tag Handlers" on page 8-25.

- tag library validators

  This feature is new in JSP 1.2. See "Validation and Tag-Library-Validator Classes" on page 8-46.

- tag library event listeners

  This feature is also new in JSP 1.2. See "Tag Library Event Listeners" on page 8-50.

- support for tag attributes of type `Object`

  JSP 1.2 adds support for tag attributes of type `java.lang.Object`. The OC4J JSP container supports this feature, as described in the previous section, "Overview of a Custom Tag Library Implementation".

Important: In Oracle9*i*AS release 2 (9.0.3), the OC4J JSP container, by default, expects JSP 1.1—*not* JSP 1.2—tag syntax and usage. To use JSP 1.2 features described in the following sections, specify the JSP 1.2 TLD DTD, as shown in "Overview of TLD File Validation and Features" on page 8-8.

Summary of TLD File Changes Between the JSP 1.1 and 1.2 Specifications

The following list is a summary of features in TLD syntax and functionality that were introduced in the JSP 1.2 specification. These changes were not available until Oracle9*i*AS release 2 (9.0.3). "Tag Library Descriptor Files" on page 8-8 includes information about these features.

- the `<validator>` element and its subelements, allowing you to declare a tag-library-validator class for the tag library
- the `<listener>` element and its subelement, allowing you to declare event listeners for the tag library
- the `<variable>` subelement, and its own subelements, under the `<tag>` element, allowing you to declare scripting variables directly through the TLD
- the `<type>` subelement under the `<attribute>` subelement of the `<tag>` element, for noting the datatype of the attribute

- the <display-name>, <large-icon>, and <small-icon> elements, and also subelements of the same name under the <tag> element, for use by authoring tools
- renamed elements since the JSP 1.1 specification, as follows:
  - The <info> element, and the subelement of the same name under the <tag> element, were renamed to <description>.
  - The <tlibversion> element was changed to <tlib-version>.
  - The <jspversion> element was changed to <jsp-version>.
  - The <shortname> element was changed to <short-name>.
  - The <tagclass>, <teiclass>, and <bodycontent> subelements under the <tag> element were changed to <tag-class>, <tei-class>, and <body-content>.

Notes:

- The OC4J JSP container, beginning with Oracle9*i*AS release 2 (9.0.3), enables XML validation of TLD files separately from validation of the `web.xml` file. Validation of TLD files is enabled by default; validation of `web.xml` is disabled by default. (See "JSP Configuration Parameters" on page 3-9 for information about the `no_tld_xml_validate` and `xml_validate` parameters.) In Oracle9*i*AS release 2 (9.0.2) and prior, TLD files and `web.xml` were all validated through the `xml_validate` parameter, which was disabled by default.
- OC4J provides a sample XSL template that you can use with a standard XSLT program, such as `oraxsl`, to convert a JSP 1.1-compliant TLD file into one that is JSP 1.2-compliant. This template is located in the `misc` directory under the OC4J demos. See `ojspdemos/misc/index.html` for instructions.

Summary of Tag Handler Changes Between the JSP 1.1 and 1.2 Specifications

The JSP 1.1 specification has two interfaces that can be implemented by tag handlers—`Tag`, for tags without bodies, and `BodyTag`, for tags with bodies. The JSP 1.2 specification adds the `IterationTag` interface, for tags that call for iteration through a tag body, but do not require access to the tag body content through a body content object. `IterationTag` extends `Tag` and is extended by `BodyTag`.

Also in JSP 1.2, the `int` constant `EVAL_BODY_TAG`, which indicates that there is a tag body to be processed, is deprecated and replaced by `EVAL_BODY_AGAIN` and `EVAL_BODY_BUFFERED`. `EVAL_BODY_AGAIN` is used with tags that iterate through a tag body, to specify that iteration should continue. `EVAL_BODY_BUFFERED` is used with tags that require access to body content, to direct that a `BodyContent` object be created.

The JSP 1.2 specification also adds the `TryCatchFinally` interface, which any tag handler can implement for improved data integrity and resource management when exceptions occur.

The JSP 1.2 changes were not available until Oracle9*i*AS release 2 (9.0.3). "Tag Handlers" on page 8-25 includes information about these new features.

Tag Library Descriptor Files

A *tag library descriptor* (TLD) file is an XML-style document that contains information about a tag library and 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. The `taglib` directive in a JSP page informs the JSP container where to find the TLD file. (See "Overview: Specifying a Tag Library with the taglib Directive" on page 8-16.)

This section provides an overview and general information about TLD file syntax and usage, referring ahead to other sections as appropriate for more information about related topics. This section covers the following topics:

- Overview of TLD File Validation and Features
- Use of the tag Element
- Other Key Elements and Their Subelements: validator and listener

For complete information, refer to the Sun Microsystems *JavaServer Pages Specification, Version 1.2*.

See "Example: Using the IterationTag Interface and a Tag-Extra-Info Class" on page 8-57 for a sample TLD file.

Note: By default, the OC4J JSP container performs XML validation of TLD files. To disable this, set the `no_tld_xml_validate` JSP configuration parameter to `true`. (See "JSP Configuration Parameters" on page 3-9 for more information.) For pre-translation, use the `ojspc -noTldXmlValidate` option. (See "Option Descriptions for ojspc" on page 7-20.)

Overview of TLD File Validation and Features

The OC4J JSP container uses the `DOCTYPE` declaration of a TLD file to determine which TLD DTD version to validate against, unless TLD validation has been disabled. By default, as of Oracle9*i*AS release 2 (9.0.3), the JSP container assumes the JSP 1.1 TLD DTD. To use the JSP 1.2 TLD DTD, list the following as the system ID (DTD location):

```
http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd
```

Here is an example:

```
<!DOCTYPE taglib
        PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
        "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
```

When TLD validation is enabled, the XML parser must be able to reference the appropriate DTD, which it can do with the above `DOCTYPE` declaration for JSP 1.2. (TLD validation is enabled if the JSP `no_tld_xml_validate` parameter has its default `false` setting, or, for pre-translation, if the `ojspc -noTldXmlValidation` flag is not used.)

Note: According to the JSP 1.2 specification, use an absolute URL to specify the system ID. If a TLD file does not use a public external `DOCTYPE` declaration with an absolute URL, the OC4J JSP container will assume that the JSP 1.1 TLD DTD is intended.

A TLD file provides definitions for the tag library as a whole as well as for each individual tag. For each tag, it defines the tag name, its attributes (if any), its scripting variables (if any), and the name of the class that handles tag semantics. See "Use of the tag Element" on page 8-10.

For the library as a whole, TLD definitions can include a tag-library-validator class and event listeners. See "Other Key Elements and Their Subelements: validator and listener" on page 8-15.

A TLD file also provides additional definitions for the library as a whole, as follows.

Note: The `<tag>`, `<validator>`, and `<listener>` elements and the elements listed below are top-level subelements under the `<taglib>` root element of the TLD file.

- The required `<tlib-version>` element specifies the version number of the tag library (whatever version number you want to give it).
- The required `<jsp-version>` element specifies the JSP version upon which this tag library depends (such as 1.2).
- The `<uri>` element can specify a string value that uniquely identifies this tag library. In particular, this is useful in situations where multiple tag libraries and

their TLD files are packaged in a single JAR file. See "Packaging and Accessing Multiple Tag Libraries in a JAR File" on page 8-18.

- The required <short-name> element specifies a convenient default name for the library, for possible use by authoring tools. You could also use the short name as a preferred tag prefix for the library, for use in the taglib directive.
- There are also additional elements that you can use, typically for authoring tools—the <display-name> element for a display name of the tag library, and the <large-icon> and <small-icon> elements for the file names (.jpg or .gif) of a large icon, a small icon, or both. Icon file locations are relative to the TLD file.
- The <description> element can provide a description of the tag library.

Note: Several descriptive elements were added to the JSP 1.2 TLD DTD. In addition to the <description> element directly under the root <taglib> element, there are <description> subelements under the <tag>, <variable>, and <attribute> elements. There is also an <example> subelement under the <tag> element. These subelements can provide information for end-users of the tag library. In particular, a TLD can be processed, such as through an XSLT stylesheet, to provide end-user documentation from the material in the descriptive elements. This information can be displayed in the help windows of tools such as Oracle9*i* JDeveloper, for example.

Use of the tag Element

Each tag of a tag library is specified in a <tag> element, under the root <taglib> element of the TLD file. There must be at least one <tag> element in a TLD file. This section describes its usage and subelements.

Subelements of the tag Element

The subelements of a <tag> element define a tag, as follows:

- The required <name> subelement specifies the name of the tag.
- The required <tag-class> subelement specifies the name of the corresponding tag handler class. See "Tag Handlers" on page 8-25 for information about tag handler classes.

- The <body-content> subelement indicates how the tag body (if any) should be processed. See the example and accompanying discussion in "Sample tag Element and Use of Its body-content Subelement" on page 8-12.
- Each <variable> subelement (if any), with its further subelements, defines a scripting variable. See "Scripting Variables, Declarations, and Tag-Extra-Info Classes" on page 8-41 for information about scripting variables. The <variable> element is for relatively uncomplicated situations, where the logic for the scripting variable does not require a tag-extra-info class. The variable name is specified through either the <name-given> subelement, to specify the name directly, or the <name-from-attribute> subelement, to specify the name of a tag attribute that specifies the variable name. There is also a <variable-class> subelement to specify the class of the variable, a <scope> subelement to specify the scope of the variable, and a <declare> subelement to specify whether the variable is to be newly defined. See "Variable Declaration Through TLD variable Elements" on page 8-42 for more information. Another subelement under <variable> is an optional <description> element.
- Each <tei-class> subelement (if any) specifies the name of a tag-extra-info class that defines a scripting variable. This is for situations where declaring the variable through a <variable> element is not sufficient. See "Variable Declaration Through Tag-Extra-Info Classes" on page 8-44 for more information.
- Each <attribute> subelement (if any), with its further subelements, provides information about an attribute of the tag—a parameter that you can specify when you use the custom tag. Subelements of <attribute> include the <name> element to specify the attribute name, the <type> element to optionally note the Java type of the attribute value, the <required> element to specify whether the attribute is required (default false), and the <rtexprvalue> element to specify whether the attribute can accept runtime expressions as values (default false). See the example and accompanying discussion below. Another subelement under <attribute> is an optional <description> element.

Notes: As of Oracle9*i*AS release 2 (9.0.3), the OC4J JSP container ignores the `<type>` element. It is for informational use only, for anyone examining the TLD file. Additionally, note the following:

- For literal attribute values, where `<rtexprvalue>` specifies `false`, the `<type>` value (if any) should always be `java.lang.String`.
- When `<rtexprvalue>` specifies `true`, then the type of the tag handler property corresponding to this tag attribute determines what you should specify for the `<type>` value (if any).

- As with the tag library as a whole, each tag can have its own `<display-name>`, `<large-icon>`, and `<small-icon>` subelements for use by authoring tools.
- The `<description>` subelement can provide a description of the tag.
- The `<example>` subelement can provide an example of how to use the tag.

Notes:

- A custom tag name must qualify as an `NMTOKEN` according to the XML specification. For example, it cannot start with a numeric character.
- Attribute names must follow naming conventions for XML attributes, and their setter methods in tag handler classes must follow the JavaBeans specification.

Sample tag Element and Use of Its body-content Subelement

Here is a sample TLD file entry for a tag `myaction`:

```
<tag>
  <name>myaction</name>
  <tag-class>examples.MyactionTag</tag-class>
  <tei-class>examples.MyactionTagExtraInfo</tei-class>
  <body-content>JSP</body-content>
  <attribute>
    <name>attr1</name>
    <required>true</required>
```

```
  </attribute>
  <attribute>
    <name>attr2</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </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 and can take a runtime expression as its value.

The `<body-content>` element indicates how the tag body (if any) should be processed. There are three choices:

- A value of `empty` indicates that the tag uses no body. In this case, the OC4J JSP translator will return an exception if there is a tag body.
- A value of `JSP` (the default) indicates that the tag body should be processed as JSP source code and translated.
- A value of `tagdependent` indicates that the tag body should not be translated. Any text in the body is treated as template data.

Consider the following example:

```
<foo:bar>
   <%=blah%>
</foo:bar>
```

If the `bar` tag has a `<body-content>` value of `JSP`, then the body is processed by the JSP translator, and the expression is evaluated. With a `<body-content>` value of `tagdependent`, the JSP translator does not process the body. In this case, the characters "<", "%", "=", and ">" have no special meaning—they are treated as literal characters, along with the rest of the body, and are part of the JSP `out` object passed straight through to the tag handler .

There are additional considerations for JSP XML documents. In this case, because the document is parsed by the XML parser, it is not appropriate to implement support for a value of `tagdependent`. This value is essentially meaningless in a JSP XML document.

One reason for this is that in XML, there is already a convenient mechanism for escaping body content—using the `CDATA` token. But beyond that, there are many scenarios where it would actually be undesirable to pass content straight through as

a tagdependent implementation would do. Consider an example using a tag for SQL queries, comparing the following traditional syntax:

```
<foosql:query ... >
   select ... where salary > 1000
</foosql:query>
```

to the following JSP XML syntax:

```
<foosql:query ... >
   <![CDATA[select ... where salary > 1000]]>
</foosql:query>
```

In the traditional syntax, a <body-content> value of tagdependent would result in the query statement being passed straight through to the JSP out object, presumably the desired result.

In the XML syntax, the CDATA token (or, alternatively, a ">" escape character) is required, because otherwise the character ">" has special meaning to the XML parser.

In this example, if an implementation of tagdependent were used, the entire body would be passed through to the out object:

```
<![CDATA[select ... where salary > 1000]]>
```

But presumably, the information that should really be passed through is only the SQL query itself:

```
select ... where salary > 1000
```

This is what would happen by processing the body through a <body-content> value of JSP, and using the CDATA token for the XML parser. This is more appropriate behavior than what would happen with a tagdependent implementation.

See "Details of JSP XML Documents" on page 5-4 for more information about JSP XML syntax.

Other Key Elements and Their Subelements: validator and listener

The TLD <validator> and <listener> elements are new in the JSP 1.2 specification.

A <validator> element and its subelements specify information about a *tag-library-validator* (TLV) class that can validate JSP pages that use this tag library. The <validator> element has three subelements: <validator-class>, <description>, and <init-param>. The <init-param> subelement has the same functionality as <init-param> subelements within <servlet> elements in the web.xml file. It has <param-name> and <param-value> subelements to specify each parameter. See "Validation and Tag-Library-Validator Classes" on page 8-46 for more information.

A <listener> element and its <listener-class> subelement specify an event listener for use with the tag library, such as in creating and destroying resource pools used by the library. See "Tag Library Event Listeners" on page 8-50 for more information.

Tag Library and TLD Setup and Access

This section discusses the packaging, placement, and access of tag libraries and their TLD files. It covers the following topics:

Overview: Specifying a Tag Library with the taglib Directive

This section summarizes the use of `taglib` directives, discussing original functionality under the JSP 1.1 specification and new functionality under the JSP 1.2 specification.

Import a custom library into a JSP page by using a `taglib` directive, of the following general form:

```
<%@ taglib uri="URI" prefix="prefix" %>
```

The `prefix` setting specifies a string of characters that stipulates when tags from this library are being used. For example, if `mytag` is in a library that has a specified prefix of `oracust`, use `mytag` as follows:

```
<oracust:mytag attr1="..." attr2="..." >
...
</oracust:mytag>
```

> **Note:** Prefixes must follow the naming conventions of the XML namespaces specification.

Under the JSP 1.1 specification, the `uri` setting can indicate a file location as in either of the following scenarios, either directly or through a "shortcut" URI:

- It can indicate the physical location, within a WAR file structure, of the TLD file that defines the desired tag library.

- It can indicate the physical location of the JAR file that contains the components and TLD file of the desired tag library. Under the JSP 1.1 specification, there can be only one tag library and only one TLD file in the JAR file.

See "Specifying a Tag Library by Physical Location" on page 8-17 for more information.

Under the JSP 1.2 specification, the `uri` setting can still indicate the physical location of a TLD file or the location of a JAR file containing one tag library and its TLD file, but it can also be used as follows:

- It can specify one of multiple tag libraries packaged in a single JAR file, by specifying a value that matches the `<uri>` element value in one of the TLD files in the JAR file. In this case, the `uri` setting is intended to be a unique key, not a pointer to a physical location.

As under JSP 1.1, you can also use a shortcut URI.

See "Packaging and Accessing Multiple Tag Libraries in a JAR File" on page 8-18 for more information. For information about shortcut URIs, see "Use of web.xml for Tag Libraries" on page 8-21.

Specifying a Tag Library by Physical Location

As first defined in the JSP 1.1 specification, the `taglib` directive of a JSP page can fully specify the name and physical location, within a WAR file structure, of the TLD file that defines a particular tag library, as in the following example:

```
<%@ taglib uri="/WEB-INF/oracustomtags/tlds/mytld.tld" prefix="oracust" %>
```

Specify the location as application-relative, by starting with "`/`" as in this example. See "Requesting a JSP Page" on page 1-28 for discussion of application-relative syntax.

Be aware that the TLD file should be in the `WEB-INF` directory or a subdirectory.

Alternatively, as also defined since JSP 1.1, the `taglib` directive can specify the name and application-relative physical location of a JAR file instead of a TLD file, where the JAR file contains a single tag library and the TLD file that defines it. In this scenario, the TLD file must be located and named as follows in the JAR file:

```
META-INF/taglib.tld
```

You must place the JAR file in the `/WEB-INF/lib` directory.

Here is an example of a `taglib` directive that specifies a tag library JAR file:

```
<%@ taglib uri="/WEB-INF/lib/mytaglib.jar" prefix="oracust" %>
```

Also see "Packaging and Accessing Multiple Tag Libraries in a JAR File", following, which describes a scenario that is newly supported by the JSP 1.2 specification.

Note: In either scenario discussed in this section, the `taglib` directive can specify a "shortcut" URI that corresponds to the complete URI value according to settings in the `web.xml` file. See "Use of web.xml for Tag Libraries" on page 8-21.

Packaging and Accessing Multiple Tag Libraries in a JAR File

The preceding section, "Specifying a Tag Library by Physical Location", discusses the JSP 1.1 scenarios of using a `taglib` directive to specify a TLD file by physical location, or to specify a JAR file that contains a single tag library and its TLD file.

In addition to these scenarios, the JSP 1.2 specification allows the packaging of multiple tag libraries, and the TLD files that define them, in a single JAR file. Inside the JAR file, these TLD files must be located under the `/META-INF` directory or a subdirectory.

While a single TLD file in a JAR file is packaged as `/META-INF/taglib.tld` (although this is no longer s strict requirement under JSP 1.2), a JAR file with multiple TLD files must use unique names or subdirectories. Here are a couple of possibilities, for example, for packaging three TLD files in a JAR file:

```
/META-INF/abctags.tld
/META-INF/deftags.tld
/META-INF/ghitags.tld
```

or:

```
/META-INF/abc/taglib.tld
/META-INF/def/taglib.tld
/META-INF/ghi/taglib.tld
```

In each TLD file, there is a `<uri>` element under the root `<taglib>` element. Use this feature as follows:

- The `<uri>` element must specify a value that is to be matched by the `uri` setting of a `taglib` directive in any JSP page that wants to use the corresponding tag library.

- Each <uri> value must be unique across all <uri> values in all TLD files on the server.

The value of the <uri> element can be arbitrary—it is simply used as a key and does not indicate a physical location. By convention, however, its value is of the form of a physical location, such as in the following example:

```
<uri>http://www.mycompany.com/j2ee/jsp/tld/myproduct/mytags.tld</uri>
```

A <uri> value must follow the XML namespace convention.

A JAR file with multiple TLD files must be placed in the /WEB-INF/lib directory or in the OC4J "well-known" URI location described in "Oracle Extension for Tag Library Sharing" on page 8-20. During translation, the JSP container searches these two locations for JAR files, searches each JAR file for TLD files, and accesses each TLD file to find its <uri> element.

Notes:

- A <uri> element and the corresponding taglib directive can specify a "shortcut" URI setting. This corresponds to settings in the web.xml file, as "Use of web.xml for Tag Libraries" on page 8-21 explains.
- A JSP 1.2-compliant JSP container, such as the OC4J JSP container, supports the multiple TLD file packaging mechanism for JSP 1.1 TLD files as well as JSP 1.2 TLD files.

Example: URI Settings for Multiple Tag Libraries in a JAR File Consider a JAR file, myapptags.jar, that includes the following TLD files:

```
/META-INF/mytaglib1.tld
/META-INF/mytaglib2.tld
```

Assume that mytaglib1.tld specifies the following:

```
<taglib>
   <tlib-version>1.0</tlib-version>
   <jsp-version>1.2</jsp-version>
   <short-name>shorty</short-name>
   <uri>http://www.foo.com/jsp/mytaglib1</uri>
   <description>example TLD</description>
   <tag>
      <name>mytag1</name>
```

```
      ...
   </tag>
   ...
</taglib>
```

To use `mytag1` or any other tag defined in `mytaglib1.tld`, a JSP page could have the following `taglib` directive:

```
<%@ taglib uri="http://www.foo.com/jsp/mytaglib1" prefix="myprefix1" %>
```

URI values in this scenario (multiple tag libraries in a single JAR file) are used as keywords only. They can be arbitrary.

For a more complete example, see "Example: Multiple Tag Libraries and TLD Files in a JAR File" on page 8-22.

Oracle Extension for Tag Library Sharing

As an extension of standard JSP "well-known URI" functionality described in the JSP 1.2 specification, the OC4J JSP container supports the use of a shared tag library directory where you can place tag library JAR files to be shared across multiple Web applications.

The directory location is according to the setting of the OC4J JSP `well_known_taglib_loc` configuration parameter, with the specified location being under `[Oracle_Home]` if `[Oracle_Home]` is defined, or under the current directory (from which the OC4J process was started) if `[Oracle_Home]` is not defined. The default value of `well_known_taglib_loc` is as follows:

```
j2ee/home/jsp/lib/taglib/
```

Also see "JSP Configuration Parameters" on page 3-9 for a description of the `well_known_taglib_loc` parameter.

The shared directory must be added to the server-wide classpath by specifying it as a library path element. The default location is set in the `application.xml` file in the OC4J configuration files directory (`j2ee/home/config` by default in OC4J standalone); you can alter the setting there as desired. See the *Oracle9iAS Containers for J2EE User's Guide* for information about `application.xml`.

TLD files to be shared across a set of applications must be placed in a JAR file. There can be multiple JAR files in the well-known location. Each tag library will be uniquely identified through the `<uri>` element in its TLD file. Also see "Packaging and Accessing Multiple Tag Libraries in a JAR File" on page 8-18.

Use of web.xml for Tag Libraries

Versions 2.2 and higher of the Sun Microsystems *Java Servlet Specification* describe a standard deployment descriptor for servlets—the `web.xml` file. JSP pages can use this file in specifying the location or URI identifier of a JSP TLD file.

For JSP tag libraries, the `web.xml` file can include `<taglib>` elements and two subelements:

- `<taglib-uri>`
- `<taglib-location>`

For the scenario of an individual TLD file, or the scenario of a JAR file that contains a single tag library and its TLD file, the `<taglib-location>` subelement indicates the application-relative physical location (by starting with "`/`") of the TLD file or tag library JAR file. See "Specifying a Tag Library by Physical Location" on page 8-17 for related information.

For the scenario of a JAR file that contains multiple tag libraries and their TLD files, a `<taglib-location>` subelement indicates the unique identifier of a tag library. In this case, the `<taglib-location>` value actually indicates a key, not a location, and corresponds to the `<uri>` value in the TLD file of the desired tag library. See "Packaging and Accessing Multiple Tag Libraries in a JAR File" on page 8-18 for related information.

The `<taglib-uri>` subelement indicates a *shortcut URI* to use in `taglib` directives in your JSP pages, with this URI being mapped to the physical location or URI identifier specified in the accompanying `<taglib-location>` subelement.

Following is a sample `web.xml` entry for a TLD file:

```
<taglib>
   <taglib-uri>/oracustomtags</taglib-uri>
   <taglib-location>/WEB-INF/oracustomtags/tlds/mytld.tld</taglib-location>
</taglib>
```

This entry makes `/oracustomtags` equivalent to `/WEB-INF/oracustomtags/tlds/mytld.tld` in `taglib` directives in your JSP pages.

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 TLD file (`mytld.tld`):

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

This statement enables you to use any of the tags of this custom tag library in a JSP page.

See the Sun Microsystems *Java Servlet Specification, Version 2.3*, and the Sun Microsystems *JavaServer Pages Specification, Version 1.2*, for more information about the `web.xml` deployment descriptor.

Important: Using the `<taglib>` element in `web.xml` is required in the case of a TLD file that is located in the JSP shared tag library directory and has `<listener>` elements. This is the only way that the TLD file can be found and accessed in order to activate its listeners. See "Oracle Extension for Tag Library Sharing" on page 8-20 and "Tag Library Event Listeners" on page 8-50.

Example: Multiple Tag Libraries and TLD Files in a JAR File

This example presents key aspects of tag library packaging for some of the Oracle JSP demo applications. This is a situation where multiple tag libraries are packaged in a single JAR file. The JAR file includes tag handler classes, tag-library-validator classes, and TLD files for multiple libraries. The following shows the contents and structure of the JAR file:

```
examples/BasicTagParent.class
examples/ExampleLoopTag.class
examples/BasicTagChild.class
examples/BasicTagTLV.class
examples/TagElemFilter.class
examples/XMLViewTagTLV.class
examples/TagFilter.class
examples/XMLViewTag.class
META-INF/xmlview.tld
META-INF/exampletag.tld
META-INF/basic.tld
META-INF/MANIFEST.MF
```

Key TLD File Entries for Multiple-Library Example

This section illustrates the `<uri>` elements of the TLD files.

The `basic.tld` file includes the following:

```
<taglib>

  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>basic</short-name>
  <uri>http://xmlns.oracle.com/j2ee/jsp/tld/demos/basic.tld</uri>

  ...

</taglib>
```

The `exampletag.tld` file includes the following:

```
<taglib xmlns="http://java.sun.com/JSP/TagLibraryDescriptor">

  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>example</short-name>
  <uri>http://xmlns.oracle.com/j2ee/jsp/tld/demos/exampletag.tld</uri>

  ...

</taglib>
```

The `xmlview.tld` file includes the following:

```
<taglib>

  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>demo</short-name>
  <uri>http://xmlns.oracle.com/j2ee/jsp/tld/demos/xmlview.tld</uri>

  ...

</taglib>
```

Key web.xml File Entries for Multiple-Library Example

This section shows the <taglib> elements of the web.xml deployment descriptor, which map the full URI values (as seen in the <uri> elements of the TLD files in the previous section) to shortcut URI values used in the JSP pages that access these libraries.

```
...

<taglib>
  <taglib-uri>/oraloop</taglib-uri>
  <taglib-location>http://xmlns.oracle.com/j2ee/jsp/tld/demos/exampletag.tld
  </taglib-location>
</taglib>
<taglib>
  <taglib-uri>/orabasic</taglib-uri>
  <taglib-location>http://xmlns.oracle.com/j2ee/jsp/tld/demos/basic.tld
  </taglib-location>
</taglib>
<taglib>
  <taglib-uri>/oraxmlview</taglib-uri>
  <taglib-location>http://xmlns.oracle.com/j2ee/jsp/tld/demos/xmlview.tld
  </taglib-location>
</taglib>

...
```

JSP Page taglib Directives for Multiple-Library Example

This section shows taglib directives from the JSP pages of the demos, which reference the shortcut URI values defined in the web.xml elements listed in the preceding section.

The page basic1.jsp includes the following directive:

```
<%@ taglib prefix="basic" uri="/orabasic" %>
```

The page exampletag.jsp includes the following directive:

```
<%@ taglib prefix="example" uri="/oraloop" %>
```

The page xmlview.jsp includes the following directive:

```
<%@ taglib prefix="demo" uri="/oraxmlview" %>
```

Tag Handlers

This section describes *tag handlers*, which define the semantics of actions that result from the use of custom tags. It includes the following topics:

Overview of Tag Handlers

A tag handler is an instance of a Java class that directly or indirectly implements the standard `javax.servlet.jsp.tagext.Tag` interface. Depending on whether there is a tag body and how that body is to be processed, the tag handler implements one of the following interfaces, in the `javax.servlet.jsp.tagext` package:

- `Tag`—This interface defines the basic methods for all tag processing, but does not include tag body processing.
- `IterationTag`—This interface extends `Tag` and is for iterating through a tag body.
- `BodyTag`—This interface extends `IterationTag` and is for accessing the tag body content itself.

A tag handler class might implement one of these interfaces directly, or might extend a class (such as one of the support classes provided by Sun Microsystems) that implements one of them.

Each custom tag has its own handler class. By convention, the name of the tag handler class for a tag `abc`, for example, is `AbcTag`.

The TLD file of a tag library specifies the name of the tag handler class for each tag in the library. See "Tag Library Descriptor Files" on page 8-8.

A tag handler instance is typically created by the JSP page implementation instance, by use of a zero-argument constructor, and is a server-side object used at request-time. The tag handler 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 tag is nested within an outer tag. A tag handler, as applicable, supports parameter-passing, evaluation of the tag body, and access to other objects in the JSP page, including other tag handlers.

"Example: Using the IterationTag Interface and a Tag-Extra-Info Class" on page 8-57 includes code for a sample tag handler class.

Note: The Sun Microsystems *JavaServer Pages Specification, Version 1.2* does not mandate whether multiple uses of the same custom tag within a JSP page should use the same tag handler instance or different instances—this is left to the discretion of JSP vendors. See "OC4J JSP Tag Handler Features" on page 8-38 for information about the Oracle implementation.

Attribute Handling, Conversions from String Values

A tag handler class has an underlying property for each attribute of the custom tag. These properties are somewhat like JavaBean properties, with at least a setter method.

Recall that there are two approaches in setting a tag attribute:

- The first approach is where the attribute is a non-request-time attribute, set using a string literal value:

  ```
  nrtattr="string"
  ```

  For a non-request-time attribute, if the underlying tag handler property is not of type `String`, the JSP container will try to convert the string value to a value of the appropriate type.

  Because tag attributes correspond to bean-like properties, their processing, such as for these type conversions from string values, is similar to that of bean properties. See "Bean Property Conversions from String Values" on page 1-22.

- The second approach is where the attribute is a request-time attribute that is set using a request-time expression:

  ```
  rtattr="<%=expression%>"
  ```

For request-time attributes, there is no conversion—a request-time expression can be assigned to the attribute, and to its corresponding tag handler property, for any property type. This would apply to a tag attribute whose type is user-defined, for example.

Custom Tag Processing, with or without Tag Bodies

Custom tags, as with standard JSP tags, may or may not have a body. In the case of a custom tag, even when there is a body, its content may not have to be accessed by the tag handler.

There are four scenarios:

1. There is no body.

   In this case you need only a single tag, not a start-tag and end-tag. Following is a general example:

   ```
   <oracust:mytag attr1="...", attr2="..." />
   ```

   This is equivalent to the following, which is also permissible:

   ```
   <oracust:mytag attr1="...", attr2="..." ></oracust:abcdef>
   ```

   In this case, the tag handler should implement the `Tag` interface.

   The `<body-content>` setting for this tag in the TLD file should be `empty`.

2. There is a body; access of the body content by the tag handler is not required; the body is executed no more than once.

   In this case, there is a start-tag and an end-tag with a body of statements in between, but the tag handler does not process the body—body statements are passed through for normal JSP processing only. Following is a general example of this scenario:

   ```
   <foo:if cond="<%= ... %>" >
   ...body executed if cond is true, but body content not accessed by tag
   handler...
   </foo:if>
   ```

   In this case, the tag handler should implement the `Tag` interface.

   The `<body-content>` setting for this tag in the TLD file should be `JSP` (the default) or `tagdependent`, depending on whether the body content should be translated or treated as template data, respectively.

3. There is a body; access of the body content by the tag handler is not required; the body is executed multiple times (iterated).

   This is the same as the second scenario, except there is iterative processing of the tag body.

   ```
   <foo:myiteratetag ... >
   ...body executed multiple times, according to attribute or other settings,
   but body content not accessed by tag handler...
   </foo:myiteratetag>
   ```

   In this case, the tag handler should implement the `IterationTag` interface.

   The `<body-content>` setting for this tag in the TLD file should be `JSP` (the default) or `tagdependent`, depending on whether the body content should be translated or treated as template data, respectively.

4. There is a body that must be processed by the tag handler.

   Again, there is a start-tag and an end-tag with a body of statements in between; however, the tag handler must access the body content.

   ```
   <oracust:mybodytag attr1="...", attr2="..." >
   ...body accessed and processed by tag handler...
   </oracust:mybodytag>
   ```

   In this case, the tag handler should implement the `BodyTag` interface.

   The `<body-content>` setting for this tag in the TLD file should be `JSP` (the default) or `tagdependent`, depending on whether the body content should be translated or treated as template data, respectively.

Notes:

- In the first scenario, where there is no body, the action is known as an *empty action*. In the second, third, and fourth scenarios, where there is a body, the action is known as a *non-empty action*.
- In the first, second, and third scenarios, where no body content processing is required by the tag handler, the handler is known as a *simple tag handler.*
- For additional information about the `<body-content>` element, see "Use of the tag Element" on page 8-10.

Summary of Integer Constants for Body Processing

The tag handler interfaces that are described in the following sections specify methods that you must implement, as applicable, to return appropriate `int` constants, depending on the situation.

The possible return values from the `doStartTag()` method, which is defined in the `Tag` interface and inherited by the `IterationTag` and `BodyTag` interfaces, are as follows:

- `SKIP_BODY`—Use this value if there is no body or if evaluation of the body should be skipped.
- `EVAL_BODY_INCLUDE`—Use this value to evaluate the body and pass it through to the current JSP `out` object. There is no special processing of the body content; no body content object is created.
- `EVAL_BODY_BUFFERED` (for `BodyTag` classes only)—Use this value to create a `BodyContent` object for the content of the tag body, used for evaluation and processing of the content.
- `EVAL_BODY_TAG`—**This is deprecated** (formerly used if there is a body that requires special processing by the tag handler). Use `EVAL_BODY_AGAIN` or `EVAL_BODY_BUFFERED`, which both have the same `int` value as `EVAL_BODY_TAG`.

The possible return values from the `doAfterBody()` method, defined in the `IterationTag` interface and inherited by the `BodyTag` interface, are as follows:

- `SKIP_BODY`— Use this value to skip evaluation of the body or, when iterating through the body, to stop iterating.
- `EVAL_BODY_AGAIN`—Use this value to continue iterating through the body.

The possible return values from the `doEndTag()` method, defined in the `Tag` interface and inherited by the `IterationTag` and `BodyTag` interfaces, are as follows:

- `SKIP_PAGE`—Use this value to skip the rest of the page after the tag. This completes the request.
- `EVAL_PAGE`—Use this value to evaluate the remainder of the page after the tag.

Simple Tag Handlers without Iteration

For a custom tag that does not have a body, or has a body whose content does not require access and special processing by the tag handler, the tag handler is referred to as a *simple tag handler*. The tag handler class can implement the following standard interface:

- `javax.servlet.jsp.tagext.Tag`

However, if there is a tag body that is to be iterated, then the tag handler should implement the `IterationTag` interface instead—see "Simple Tag Handlers with Iteration" on page 8-31.

The standard `javax.servlet.jsp.tagext.TagSupport` class implements the `Tag` interface, but also implements the `IterationTag` interface. Because of this, it is inefficient to use the `TagSupport` class for a tag that does not iterate through the tag body. This is especially important to consider when migrating code from a JSP 1.1 environment to a JSP 1.2 environment, in case you created tag handlers that extended `TagSupport` under JSP 1.1. For simple tag handlers not requiring body iteration, it is best to implement the `Tag` interface from scratch.

The `Tag` interface defines methods for the following key functions:

- Set up the JSP page context object (`pageContext` property).
- Set or get the parent tag handler—the handler for the closest enclosing tag, if applicable (`parent` property).
- Set up the tag attributes.
- Conditionally process the tag body, as appropriate, according to the return value of the `doStartTag()` method. (See immediately following.)
- Conditionally process the remainder of the JSP page after the tag, as appropriate, according to the return value of the `doEndTag()` method. (See immediately following.)
- Release state information.

For complete information, see the Sun Microsystems `Tag` interface Javadoc at:

`http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/jsp/tagext/Tag.html`

In particular, the `Tag` interface specifies the following key methods:

- `doStartTag()`
- `doEndTag()`

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 JSP translator includes appropriate calls to these methods.

Implement action processing—whatever you want the action tag to accomplish—in the `doStartTag()` method. The `doEndTag()` method implements any appropriate post-processing. In the case of a tag without a body, essentially nothing happens between the execution of these two methods.

The `Tag` interface also specifies getter and setter methods for the `pageContext` and `parent` properties. The JSP page implementation instance invokes the `setPageContext()` and `setParent()` methods before invoking the `doStartTag()` and `doEndTag()` methods.

The `doStartTag()` method returns an `int` value. For a tag handler class implementing the `Tag` interface, this value is one of the following:

- `SKIP_BODY`—Do not evaluate the body, if any. This is the only option if the TLD file specifies a `<body-content>` setting of `empty` for the tag associated with this handler.
- `EVAL_BODY_INCLUDE`—Evaluate the body and pass it through to the current JSP `out` object.

The `doEndTag()` method also returns an `int` value, one of the following:

- `SKIP_PAGE`—Skip the rest of the page after the tag. If the request was originally from another page, from which the current page was forwarded to or included, only the remainder of the current page evaluation is skipped.
- `EVAL_PAGE`—Evaluate the remainder of the page after the tag.

Simple Tag Handlers with Iteration

For a custom tag that has a body that does not require access and special processing by the tag handler, but does require repeated reevaluation such as for iteration, the tag handler class can implement the following standard interface:

- `javax.servlet.jsp.tagext.IterationTag`

The `IterationTag` interface extends the `Tag` interface. A class that implements the `IterationTag` interface is still known as a simple tag handler.

The following standard support class implements the `IterationTag` interface, as well as the `java.io.Serializable` interface, and can be used as a base class:

- `javax.servlet.jsp.tagext.TagSupport`

In addition to implementing appropriate methods from the `Tag` and `IterationTag` interfaces, the `TagSupport` class includes a convenience method, `findAncestorWithClass()`, that calls the `getParent()` method defined in the `Tag` interface.

Note: It is *not* advisable to extend the `TagSupport` class if your tag handler does not have to support body iteration. Because `TagSupport` implements the `IterationTag` interface, there is looping logic that would be unnecessary. In addition to being generally inefficient, this increases the likelihood of methods exceeding a Java 64K size limit.

The `IterationTag` interface inherits basic tag-handling functionality, including the `doStartTag()` and `doEndTag()` methods, from the `Tag` interface. See "Simple Tag Handlers without Iteration" on page 8-30.

The `IterationTag` interface also defines the following additional key method:

- `doAfterBody()`

This method is called after each evaluation of the tag body, to see if the body should be evaluated again. It returns one of the following `int` values:

- `SKIP_BODY`—Stop iterating; do not reevaluate the tag body. Call `doEndTag()` instead. The `SKIP_BODY` setting is also used when the body is not to be evaluated in the first place, and is the only option if the TLD file specifies a `<body-content>` setting of `empty` for the tag associated with this handler.
- `EVAL_BODY_AGAIN`—Continue iterating; reevaluate the tag body. After the body is evaluated, the `doAfterBody()` method is called again.

Notes:

- In the JSP 1.1 specification, the `doAfterBody()` method is defined in the `BodyTag` interface. Moving this method definition to the `IterationTag` interface in JSP 1.2 allows a simple iteration tag handler to avoid the overhead of maintaining a `BodyContent` object.
- For a complete example of `IterationTag` usage, see "Example: Using the IterationTag Interface" on page 8-53.

Tag Handlers That Access Body Content

For a custom tag with body content that the tag handler must be able to access, the tag handler class can implement the following standard interface:

- `javax.servlet.jsp.tagext.BodyTag`

The following standard support class implements the `BodyTag` interface, as well as the `java.io.Serializable` interface, and can be used as a base class:

- `javax.servlet.jsp.tagext.BodyTagSupport`

This class implements appropriate methods from the `Tag`, `IterationTag`, and `BodyTag` interfaces.

Note: Do not use the `BodyTag` interface (or `BodyTagSupport` class) if your tag handler does not actually require access to the body content. This would result in the needless overhead of creating and maintaining a `BodyContent` object. Depending on whether iteration through the body is required, use the `Tag` interface or the `IterationTag` interface (or `TagSupport` class) instead.

BodyTag Features

The `BodyTag` interface inherits basic tag-handling functionality from the `Tag` interface, including the `doStartTag()` and `doEndTag()` methods and their defined return values. It also inherits functionality from the `IterationTag` interface, including the `doAfterBody()` method and its defined return values. See "Simple Tag Handlers without Iteration" on page 8-30 and "Simple Tag Handlers with Iteration" on page 8-31.

Along with its inherited features, the BodyTag interface adds functionality to capture execution results from the tag body. Evaluation of a tag body is encapsulated in an instance of the javax.servlet.jsp.tagext.BodyContent class. The page implementation object creates this instance as appropriate. See "BodyContent Objects" on page 8-35.

As with the Tag interface, the doStartTag() method specified in the BodyTag interface supports int return values of SKIP_BODY and EVAL_BODY_INCLUDE. For BodyTag, this method also supports an int return value of EVAL_BODY_BUFFERED. To summarize the meanings:

- SKIP_BODY—Do not evaluate the body.
- EVAL_BODY_INCLUDE—Evaluate the body and pass it through to the JSP out object without the body content being made available to the tag handler. (This is essentially the same behavior as in an EVAL_BODY_INCLUDE scenario with a tag handler that implements the IterationTag interface.)
- EVAL_BODY_BUFFERED—Create a BodyContent object for processing of the tag body content.

The BodyTag interface also adds definitions for the following methods:

- setBodyContent()—Set the bodyContent property (a BodyContent instance) of the tag handler.
- doInitBody()—Prepare to evaluate the tag body.

If the doStartTag() method returns EVAL_BODY_BUFFERED, the JSP page implementation instance executes the following steps, in order:

1. It creates a BodyContent instance.
2. It calls the setBodyContent() method of the tag handler, to pass the BodyContent instance to the tag handler.
3. It calls the doInitBody() method of the tag handler to perform initialization, if any, related to the BodyContent instance.

These steps occur before the tag body is evaluated. While the body is evaluated, the JSP out object will be bound to the BodyContent object.

After each evaluation of the body, as for tag handlers implementing the IterationTag interface, the page implementation instance calls the tag handler doAfterBody() method. This involves the following possible return values:

- SKIP_BODY—Stop iterating; do not reevaluate the tag body. Call doEndTag() instead. The JSP out object is restored from the page context.

- EVAL_BODY_AGAIN—Continue iterating; reevaluate the tag body. When the body is evaluated, it is passed through to the current JSP `out` object. After the body is evaluated, the `doAfterBody()` method is called again.

Once evaluation of the body is complete, for however many iterations are appropriate, the page implementation instance invokes the tag handler `doEndTag()` method.

BodyContent Objects

For tag handlers implementing the `BodyTag` interface, evaluation results from the tag body are made accessible to the tag handler through an instance of the `javax.servlet.jsp.tagext.BodyContent` class. This class extends the `javax.servlet.jsp.JspWriter` class.

A `BodyContent` instance is created through the `pushBody()` method of the JSP page context.

The `BodyContent` class, in addition to inheriting `JspWriter` features, adds methods to accomplish the following:

- Return its contents as a `java.io.Reader` object (`getReader()` method).
- Write its contents into a `java.io.Writer` object (`writeOut()` method).
- Convert its contents into a `String` object (`getString()` method).
- Clear its contents (`clearBody()` method).

Typical uses for a `BodyContent` object include the following:

- Convert its contents into a `String` instance and then use the string as a value for an operation.
- Write its contents into the JSP `out` object that was active as of when the start-tag was encountered.

TryCatchFinally Interface

For data integrity and resource management when exceptions occur during tag processing, the JSP 1.2 specification adds the `javax.servlet.jsp.tagext.TryCatchFinally` interface. Implementing this interface in your tag handlers is particularly useful for tags that must handle errors, and for ensuring the proper release of resources.

The `TryCatchFinally` interface specifies the following methods:

- `void doCatch(java.lang.Throwable throw)`

  This method can be invoked on a tag handler when a `Throwable` error occurs during evaluation of a tag body or during a call to the `doStartTag()`, `doEndTag()`, `doAfterBody()`, or `doInitBody()` method. The `Throwable` object that was encountered is taken as input by the `doCatch()` method. This method would *not* be invoked if the `Throwable` error occurs during a call to a setter method.

  The `doCatch()` method may throw an exception (the original `Throwable` exception or a new exception) to be propagated through an error chain.

- `void doFinally()`

  This method is invoked regardless of whether a `Throwable` error, as discussed for the `doCatch()` method, occurs. It would *not* be invoked, however, if a `Throwable` error occurs during a call to a setter method.

  The `doFinally()` method should not throw an exception.

Following is a typical `TryCatchFinally` invocation (from the Sun Microsystems *JavaServer Pages Specification*, *Version 1.2*):

```
h = get a Tag();  // get a tag handler, perhaps from pool

h.setPageContext(pc);  // initialize as desired
h.setParent(null);
h.setFoo("foo");

// tag invocation protocol; see Tag.java
try {
  h.doStartTag()...
  ....
  h.doEndTag()...
} catch (Throwable t) {
  /* React to exceptional condition; invoked if exception occurs between
     doStartTag() and doEndTag(). */
  h.doCatch(t);
} finally {
  // restore data invariants and release pre-invocation resources
  h.doFinally();
  /* doFinally() is almost always called, unless Throwable error occurs
     during setter method, or Java thread terminates. */
}
```

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 JSP page context, 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.

OC4J JSP Tag Handler Features

This section describes OC4J JSP extended features for tag handler pooling and code generation size reduction. It covers the following topics:

- Disabling or Enabling Runtime or Compile-Time Tag Handler Reuse
- Tag Handler Code Generation

Disabling or Enabling Runtime or Compile-Time Tag Handler Reuse

In Oracle9*i*AS release 2, you can specify that tag handler instances be reused within each JSP page. This is sometimes referred to as *tag handler instance pooling*. As of release 2 (9.0.3), there are two models for this:

- *runtime model*—The logic and patterns of tag handler reuse is determined at runtime, during execution of the JSP pages. Tag handler reuse is within `application` scope.
- *compile-time model*—The logic and patterns of tag handler reuse is determined at compile-time, during translation of the JSP pages. This is an effective way to improve performance for an application with very large numbers of tags within the same page (hundreds of tags, for example).

The JSP `tags_reuse_default` configuration parameter is relevant in either case. See "JSP Configuration Parameters" on page 3-9 for further information about this parameter and how to set it.

Key Points Regarding Tag Handler Reuse

Be aware of the following points about tag handler reuse:

- In the OC4J 9.0.3 implementation, the default `tags_reuse_default` setting is `runtime`, for use of the runtime model.
- If you switch from the runtime model (`tags_reuse_default` value of `runtime`) to the compile-time model (`tags_reuse_default` value of `compiletime` or `compiletime_with_release`), or from the compile-time model to the runtime model, you must re-translate the JSP pages.
- The JSP container also supports tag handler reuse in the JServ environment. In that environment, the default `tags_reuse_default` setting is `none`, for no tag handler reuse.
- Any given tag handler instance processes only one request at a time.

Enabling or Disabling the Runtime Model for Tag Handler Reuse

The runtime model can be enabled in either of two ways:

- Use the default `tags_reuse_default` value of `runtime`. (For backward compatibility, a setting of `true` is also supported and is equivalent to `runtime`.)

or:

- If `tags_reuse_default` has a value of `none`, you can override this in any particular JSP page by setting the `oracle.jsp.tags.reuse` attribute in the JSP page context to `true`. For example:

```
pageContext.setAttribute("oracle.jsp.tags.reuse", new Boolean(true));
```

You can also disable the runtime model in either of two ways:

- Set `tags_reuse_default` to a value of `none`. This also disables the compile-time model. (For backward compatibility, a setting of `false` is also supported and is equivalent to `none`.)

or:

- If `tags_reuse_default` has a value of `runtime`, you can override this in any particular JSP page by setting the `oracle.jsp.tags.reuse` attribute in the JSP page context to `false`. For example:

```
pageContext.setAttribute("oracle.jsp.tags.reuse", new Boolean(false));
```

Notes:

- Remember to retranslate your JSP pages when switching from the compile-time model to the runtime model for tag handler reuse.
- You can use separate `oracle.jsp.tags.reuse` settings in different pages, or even in different sections of the same page.
- The `oracle.jsp.tags.reuse` attribute is ignored with a `tags_reuse_default` setting of `compiletime` or `compiletime_with_release`.

Enabling or Disabling the Compile-Time Model for Tag Handler Reuse

You can switch to the compile-time model for tag-handler reuse in one of two ways:

- Set the `tags_reuse_default` configuration parameter to `compiletime`.

or:

- Set the `tags_reuse_default` configuration parameter to `compiletime_with_release`.

A `compiletime_with_release` setting results in the tag handler `release()` method being called between usages of the same tag handler within the same page. This method releases state information, with details according to the tag handler implementation. For example, if the tag handler is coded so as to assume a release of state information between tag usages, then `compiletime_with_release` would be appropriate. If unsure about the implementation of the tag handler and about which compile-time setting to use, you might consider experimentation.

To disable the compile-time model, set `tags_reuse_default` to a value of `none`. This also disables the runtime model.

Notes:

- Remember to retranslate your JSP pages when switching from the runtime model to the compile-time model for tag handler reuse.
- The page context `oracle.jsp.tags.reuse` attribute is ignored with a `tags_reuse_default` setting of `compiletime` or `compiletime_with_release`.

Tag Handler Code Generation

The Oracle JSP implementation reduces the code generation size for custom tag usage. In addition, there is a JSP configuration flag, `reduce_tag_code`, that you can set to `true` for even further size reduction.

Be aware, however, that when this flag is enabled, the code generation pattern does not maximize tag handler reuse. Although you can still improve performance by setting `tags_reuse_default` to `true` as described in "Disabling or Enabling Runtime or Compile-Time Tag Handler Reuse" on page 8-38, the effect is not maximized when `reduce_tag_code` is also `true`.

See "JSP Configuration Parameters" on page 3-9 for further information about these parameters and how to set them.

Scripting Variables, Declarations, 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. A scripting variable can be defined either through a `<variable>` element in the TLD file of the tag library, for elementary cases, or through a tag-extra-info class, for cases where the logic for the scripting variable is more complex.

This section covers the following topics:

Using Scripting Variables

Objects that are defined explicitly in a custom tag can be referenced in other actions through the JSP page context, 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 scripting elements in the page, according to the declared scope of `myobj`. (See "Scripting Variable Scopes" on page 8-42.) The `id` attribute is a translation-time attribute. You can specify a variable in one of two ways:

- Provide a `<variable>` element for the variable in the TLD file, to specify the name and type of the variable, along with additional information. See "Variable Declaration Through TLD variable Elements" on page 8-42.
- Create a tag-extra-info class to specify the name and type of the variable, along with additional information and related logic. Specify the tag-extra-info class name in a `<tei-class>` element in the TLD file. See "Variable Declaration Through Tag-Extra-Info Classes" on page 8-44.

Generally, the more convenient `<variable>` mechanism will suffice.

The JSP container enters `myobj` into the page context, 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 the foo and bar tags. All that is required is knowledge of the name of the object (myobj).

Note: In the example, id and ref are merely sample attribute names; there are no special predefined semantics for these attribute names. 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 <variable> element or tag-extra-info class of the tag that creates the variable. It can be one of the following int constants:

- NESTED—Use this setting for the scripting variable to be available between the start-tag and end-tag of the action that defines it.
- AT_BEGIN—Use this setting for the scripting variable to be available from the start-tag to the end of the page.
- AT_END—Use this setting for the scripting variable to be available from the end-tag to the end of the page

Variable Declaration Through TLD variable Elements

In the JSP 1.1 specification, use of a scripting variable for a custom tag requires the creation of a tag-extra-info (TEI) class. See "Variable Declaration Through Tag-Extra-Info Classes" on page 8-44. With the JSP 1.2 specification, however, there is a simpler mechanism—a <variable> element in the TLD file where the associated tag is defined. This is sufficient for most cases, where logic related to the variable is simple enough to not require use of a TEI class.

The <variable> element is a subelement under the <tag> element that defines the tag that uses the variable.

You can specify the name of the variable in one of two ways:

- Use a <name-given> subelement under <variable> to specify the variable name directly.

or:

- Use a <name-from-attribute> subelement under <variable> to specify a tag attribute whose value, at translation-time, will specify the variable name.

Along with <name-given> and <name-from-attribute>, the <variable> element has the following subelements:

- The <variable-class> element specifies the class of the variable. The default is java.lang.String.
- The <declare> element specifies whether the variable is to be a newly declared variable, in which case the JSP translator will declare it. The default is true. If false, then the variable is assumed to have been declared earlier in the JSP page through a standard mechanism such as a jsp:useBean action, a JSP scriptlet, a JSP declaration, or some custom action.
- The <scope> element specifies the scope of the variable—NESTED, AT_BEGIN, or AT_END, as described in "Scripting Variable Scopes" on page 8-42. The default is NESTED.

Here is an example that declares two scripting variables for a tag myaction. Note that details within the <tag> element that are not directly relevant to this discussion are omitted:

```
<tag>
   <name>myaction</name>
   ...
   <attribute>
     <name>attr2</name>
     <required>true</required>
   </attribute>
   <variable>
      <name-given>foo_given</name-given>
      <declare>false</declare>
      <scope>AT_BEGIN</scope>
   </variable>
   <variable>
      <name-from-attribute>attr2</name-from-attribute>
      <variable-class>java.lang.Integer</variable-class>
   </variable>
</tag>
```

The name of the first variable is hardcoded as foo_given. By default, it is of type String. It is not to be newly declared, so is assumed to exist already, and its scope is from the start-tag to the end of the page.

The name of the second variable is according to the setting of the required attr2 attribute. It is of type Integer. By default, it is to be newly declared and its scope is NESTED—between the myaction start-tag and end-tag.

See "Tag Library Descriptor Files" on page 8-8 for more information about related TLD syntax.

Variable Declaration Through Tag-Extra-Info Classes

For a scripting variable with associated logic that is at least somewhat complicated, the use of a `<variable>` element in the TLD file to declare the variable might be insufficient. In this case, you can specify details regarding the scripting variable in a subclass of the `javax.servlet.jsp.tagext.TagExtraInfo` abstract class. This manual refers to such a subclass as a *tag-extra-info class*. Tag-extra-info classes support additional validation of tag attributes and provide additional information about scripting variables to the JSP runtime.

The JSP container uses tag-extra-info instances during translation. The TLD file specifies any tag-extra-info classes to use for scripting variables of a given tag. Use `<tei-class>` elements, as in the following example:

```
<tag>
  <name>loop</name>
  <tag-class>examples.ExampleLoopTag</tag-class>
  <tei-class>examples.ExampleLoopTagTEI</tei-class>
  <body-content>JSP</body-content>
  <description>for loop</description>
  <attribute>
     ...
  </attribute>
  ...
</tag>
```

The following are key related classes, also in the `javax.servlet.jsp.tagext` package:

- `TagData`—An instance of this class contains translation-time attribute value information for a tag instance.
- `VariableInfo`—Each instance of this class contains information about a scripting variable that is declared, created, or modified by a tag at runtime.

The key methods of the `TagExtraInfo` class are as follows:

- `boolean isValid(TagData data)`—The JSP translator calls this method for translation-time validation of the tag attributes, passing it a `TagData` instance.
- `VariableInfo[] getVariableInfo(TagData data)`—The JSP translator calls this method during translation, passing it a `TagData` instance. This

method returns an array of `VariableInfo` instances, with one 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 (not a primitive type)
- a boolean value indicating whether the variable is to be newly declared, in which case the JSP translator will declare it
- its scope

Important: As of the OC4J 9.0.3 implementation, you can have the `getVariableInfo()` method 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 to avoid confusion in case there are duplicate class names between packages. Primitive types are not supported.

See "Sample Tag-Extra-Info Class: ExampleLoopTagTEI.java" on page 8-59 for sample code of a tag-extra-info class.

Validation and Tag-Library-Validator Classes

The JSP 1.2 specification adds a feature to optionally associate a "validator" class with each tag library. These classes are referred to as *tag-library-validator* (TLV) classes. The purpose of a TLV class is to validate any JSP page that uses the tag library, verifying that the page adheres to any constraints that you wish to impose through your implementation of the TLV class. Although it is probably typical for a TLV class to check for constraints regarding use of the associated tag library only, there is no limitation—the TLV class can check any aspect of a JSP page.

A tag-library-validator class must be a subclass of the `javax.servlet.jsp.tagext.TagLibraryValidator` class.

This section discusses tag library validation and TLV classes, covering the following topics:

- TLD validator Element
- Key TLV-Related Classes and the validation() Method
- TLV Processing
- Validation Mechanisms

TLD validator Element

To specify a TLV class for a tag library, use a `<validator>` element in the TLD file. The `<validator>` element has the following subelements:

- The `<validator-class>` subelement specifies the TLV class name.
- The `<description>` subelement can be used to provide documentation about the TLV class.
- The `<init-param>` subelement and its own subelements—`<param-name>` and `<param-value>`—can be used to set initialization parameters for the TLV class. This is similar to how `<init-param>` subelements work within `<servlet>` elements in the application deployment descriptor (`web.xml`). There is also an optional `<description>` subelement under the `<init-param>` element.

The following `<validator>` element examples are from the Sun Microsystems *JavaServer Pages Standard Tag Library, Version 1.0* specification.

Example 1 This is an example of a TLV class (`ScriptFreeTLV`) that can disallow JSP declarations, JSP scriptlets, JSP expressions, and runtime expressions according to the settings of its initialization parameters. In this case, JSP expressions and runtime expressions will be allowed, but not JSP declarations or JSP scriptlets.

```
<validator>
   <validator-class>
      javax.servlet.jsp.jstl.tlv.ScriptFreeTLV
   </validator-class>
   <init-param>
      <param-name>allowDeclarations</param-name>
      <param-value>false</param-value>
   </init-param>
   <init-param>
      <param-name>allowScriptlets</param-name>
      <param-value>false</param-value>
   </init-param>
   <init-param>
      <param-name>allowExpressions</param-name>
      <param-value>true</param-value>
   </init-param>
   <init-param>
      <param-name>allowRTExpressions</param-name>
      <param-value>true</param-value>
   </init-param>
</validator>
```

Example 2 This is an example of a TLV class (`PermittedTagLibsTLV`) that allows tag library usage only as specified in its initialization parameter. In addition to the tag library with which the TLV class is associated (the use of which is allowed implicitly), the TLV class allows the libraries specified in a list (with entries separated by white space) in its initialization parameter setting. In this case, it allows only the `core`, `xml`, `fmt`, and `sql` JSTL libraries.

```
<validator>
   <validator-class>
      javax.servlet.jsp.jstl.tlv.PermittedTaglibsTLV
   </validator-class>
   <init-param>
      <param-name>permittedTaglibs</param-name>
      <param-value>
         http://java.sun.com/jstl/core
         http://java.sun.com/jstl/xml
         http://java.sun.com/jstl/fmt
```

```
        http://java.sun.com/jstl/sql
      </param-value>
    </init-param>
</validator>
```

Key TLV-Related Classes and the validation() Method

As the introduction mentions, a TLV class is a subclass of the `javax.servlet.jsp.tagext.TagLibraryValidator` class.

The following related classes are also in the `javax.servlet.jsp.tagext` package:

- `PageData`—An instance of this class is generated by the JSP translator and contains information corresponding to the XML view of the page being translated.
- `ValidationMessage`—An instance of this class contains an error message from a TLV instance, being returned through the TLV `validate()` method.

Here is the key method of a TLV class:

- `ValidationMessage[] validate (String prefix, String uri, PageData page)`

  The JSP container calls this method each time it encounters a `taglib` directive that points to a TLD file that has a `<validator>` element. The method takes as input the tag library prefix, the TLD URI, and the `PageData` object (XML view) of the page. If errors are encountered during validation, the `validate()` method returns an array of validation messages. Because the OC4J JSP container supports the optional `jsp:id` attribute, the `jsp:id` values are included in the validation messages.

  See the next section, "TLV Processing", for more information.

TLV Processing

As each `taglib` directive is encountered in a JSP page during translation, the JSP container searches the associated TLD file for a `<validator>` element that specifies a TLV class. If one is found, the container executes the following steps during the translation. See the preceding section, "Key TLV-Related Classes and the validation() Method", for background information about classes and methods discussed here.

1. The TLV class is instantiated, with initialization parameter settings according to any `<init-param>` subelements of the `<validator>` element.

2. The XML view of the JSP page is exposed to the TLV instance. (See "Details of the JSP XML View" on page 5-15.)
3. The `validate()` method of the TLV instance is called to validate the JSP page. (See the next section, "Validation Mechanisms".) If this method encounters any errors, it returns an array of `ValidationMessage` instances. If there are no errors, the method can return `null` or an empty `ValidationMessage[]` array.

> **Note:** The OC4J JSP container implements an optional JSP 1.2 feature for improved reporting of validation errors—the `jsp:id` attribute. See "The jsp:id Attribute for Error Reporting During Validation" on page 5-16 for information.

4. Each time a custom tag belonging to this library (the library associated with the TLV class) is encountered, it is checked for a tag-extra-info class. If one is specified, then it is instantiated by the JSP container and its `isValid()` method is called to validate the attributes of the tag. The `isValid()` method returns `true` if this validation is successful, or `false` if not.

Validation Mechanisms

The XML view of a JSP page cannot generally be validated against a DTD and does not include a `DOCTYPE` statement. There are various namespace-aware mechanisms that you can use for validation. One mechanism in particular is the W3C XML Schema language. Refer to the W3C Web site for information:

```
http://www.w3.org/XML/
```

More elementary mechanisms may be suitable as well, such as simply verifying that only a certain set of elements are used in a JSP page, or that a certain set of elements are *not* used in a page.

Tag Library Event Listeners

The Sun Microsystems *Java Servlet Specification, Version 2.3* describes the use of the following types of event listeners:

- servlet context listener, implementing interface `javax.servlet.ServletContextListener`
- servlet context attribute listener, implementing interface `javax.servlet.ServletContextAttributeListener`
- HTTP session listener, implementing interface `javax.servlet.http.HttpSessionListener`
- HTTP session attribute listener, implementing interface `javax.servlet.http.HttpSessionAttributeListener`

In servlet 2.3 functionality, you can specify event listeners in the application `web.xml` file. As a result of this, they are registered with the servlet container and notified of relevant state changes. Servlet context listeners, for example, are notified of changes in the application `ServletContext` object, such as application startup or shutdown.

The JSP 1.2 specification, for convenience in packaging and deploying tag libraries, adds support for `<listener>` elements in TLD files. You can use these elements to specify event listeners, as an alternative to specifying them in the `web.xml` file. This section describes the JSP 1.2 features, covering the following topics:

- TLD listener Element
- Activation of Tag Library Event Listeners
- Access of TLD Files for Event Listener Information

TLD listener Element

In a TLD file, each `<listener>` element is at the top level underneath the root `<taglib>` element. The `<listener>` element has one subelement, the required `<listener-class>` element, which specifies the listener class to be instantiated. This would be a class that implements the `ServletContextListener`, `ServletContextAttributeListener`, `HttpSessionListener`, or `HttpSessionAttributeListener` interface.

Following is an example:

```
<taglib>
...
   <listener>
      <listener-class>mypkg.MyServletContextListener</listener-class>
   </listener>
...
</taglib>
```

Activation of Tag Library Event Listeners

When an application starts, the servlet container will make a call to the JSP container to perform the following:

1. Find and access TLD files.
2. Read TLD files to find their `<listener>` elements.
3. Instantiate and register the listeners.

This is a convenient way to manage application-level and session-level resources that are associated with the usage of a particular tag library. The functionality is essentially the same as for servlet context listeners specified in the `web.xml` file.

Notes:

- For event listeners specified in TLD files, the order in which the listeners are registered is undefined, but they are all registered prior to application startup and they are all registered after listeners that are specified in the `web.xml` file.
- If a TLD file is present within the WAR file structure, it will be scanned for listeners, and any listeners will be registered, even if the associated tag library is not actually used in the application.

Access of TLD Files for Event Listener Information

You must take certain standard measures to ensure that the JSP container can access TLD files to find their `<listener>` elements. For general information about TLD file location, accessibility, and packaging, see "Tag Library and TLD Setup and Access" on page 8-16. That section includes information about the OC4J shared tag library directory ("well-known" URI location).

In addition, for any TLD file in the well-known tag library directory, you must specify the tag library in a `<taglib>` element in the application `web.xml` file if you want the application to activate any listeners specified in the TLD file. Without this step, TLD files in the shared directory are not accessed to search for their `<listener>` elements. This is to protect against needless performance impact for any application that does not use a tag library that happens to be in the shared directory.

End-to-End Custom Tag Examples

This section provides complete examples of custom tag usage, including sample JSP pages, tag handler classes, and tag library descriptor files. It includes the following samples:

- Example: Using the IterationTag Interface
- Example: Using the IterationTag Interface and a Tag-Extra-Info Class

> **Note:** These examples are for illustrative purposes only and do not necessarily reflect the most realistic or efficient approaches.

Example: Using the IterationTag Interface

This sample shows the use of a custom tag, `myIterator`, to make the current item in a collection available as a scripting variable. It defines a scripting variable through a `<variable>` element in the TLD file.

For complete information about this example, including unpacking and deploying it, refer to the following Oracle Technology Network Web site:

`http://otn.oracle.com/tech/java/oc4j/htdocs/how-to-jsp-iterationtag.html`

(You must register for membership, but registration is free of charge.)

Sample JSP Page: exampleiterator.jsp

The following JSP page uses the `myIterator` tag:

```
%@ page contentType="text/html;charset=windows-1252"%>
<HTML>
<HEAD>
<TITLE>
OJSP 1.2 IterationTag Sample
</TITLE>
</HEAD>
<%@ taglib uri="/WEB-INF/exampleiterator.tld" prefix="it"%>
<BODY>

<% java.util.Vector vector = new java.util.Vector();
   vector.addElement("One");
   vector.addElement("Two");
   vector.addElement("Three");
```

```
   vector.addElement("Four");
   vector.addElement("Five");
%>
  Collection to Iterate over is <%=vector%> ..... <p>

 <B>Iterating ...</B><br>
 <it:myIterator collection="<%= vector%>" >
      Item  <B><%= item%></B><br>
 </it:myIterator>
</p>
</BODY>
</HTML>
```

Sample Tag Handler Class: MyIteratorTag.java

In this sample tag handler class, `MyIteratorTag`, the `doStartTag()` method checks whether the collection is null. If not, it retrieves the collection object. If the iterator contains at least one element, then `doStartTag()` makes the first item in the collection available as a page-scope object and returns `EVAL_BODY_INCLUDE`. This alerts the JSP container to add the contents of the tag body to the response object and to call the `doAfterBody()` method.

This class extends the tag handler support class `TagSupport`, which implements the `IterationTag` interface.

```
package oracle.taglib;

import java.util.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

/**
 * MyIteratorTag extends TagSupport. The TagSupport class in JSP 1.2 implements
the IterationTag
 */

public class MyIteratorTag extends TagSupport
{
  private Iterator iterator;
  private Collection _collection;

  public void setCollection(Collection collection)
  {
        this._collection = collection;
```

```
  }

  public int doStartTag() throws JspTagException
  {
    if (_collection == null)
    {
      throw new JspTagException("No collection with name "
        + _collection
        + " found");
    }

    iterator = _collection.iterator();
    if (iterator.hasNext())
    {
      pageContext.setAttribute("item", iterator.next());
      return EVAL_BODY_INCLUDE;
    }
    else
    {
       return SKIP_BODY;
    }
  }

  public int doAfterBody()
  {

     if (iterator.hasNext())
     {
      pageContext.setAttribute("item", iterator.next());
      return EVAL_BODY_AGAIN;
     }
     else
     {
      return SKIP_BODY;
     }
  }

}
```

Sample Tag Library Descriptor File: exampleiterator.tld

Here is a sample TLD file to define the `myIterator` tag. This example takes advantage of the JSP 1.2 feature allowing definition of scripting variables directly in TLD files through `<variable>` elements. This TLD file defines the scripting variable `item` of type `java.lang.Object`. (In a JSP 1.1 environment, this would require use of a tag-extra-info class.) The variable is to be newly declared.

The `myIterator` tag has an attribute `collection` to specify the collection. This attribute is required and can be set as a runtime expression. The tag also has a `<body-content>` value of JSP, which means the JSP translator should process and translate the body code.

For JSP 1.2 syntax, be sure to specify the JSP 1.2 tag library DTD path.

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<!DOCTYPE taglib
        PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
        "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">

<taglib>
   <tlib-version>1.0</tlib-version>
   <jsp-version>1.2</jsp-version>
   <short-name>iterate</short-name>
   <description>This tag lib implements new JSP 1.2 IterationTag
                interface</description>
   <tag>
      <name>myIterator</name>
      <tag-class>oracle.taglib.MyIteratorTag</tag-class>
      <body-content>JSP</body-content>
      <attribute>
          <name>collection</name>
          <required>true</required>
          <rtexprvalue>true</rtexprvalue>
      </attribute>
      <variable>
        <name-given>item</name-given>
        <variable-class>java.lang.Object</variable-class>
        <declare>true</declare>
        <!-- default scope: nested -->
        <description>Scripting Variable item</description>
      </variable>
    </tag>
</taglib>
```

Example: Using the IterationTag Interface and a Tag-Extra-Info Class

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. It defines a scripting variable through a tag-extra-info class.

Included in the example are the following:

- JSP source code for a page that uses the tag
- source code for the tag handler class
- source code for the tag-extra-info class
- the TLD file

Note: Sample code here uses extended datatypes in the `oracle.jsp.jml` package. "Extended Type JavaBeans" on page 2-14 has an overview of these types. For more information, refer to the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference*.

Sample JSP Page: exampletag.jsp

Following is a sample JSP page, `exampletag.jsp`, that uses the `loop` tag, specifying that the outer loop be executed five times and the inner loop three times:

```
<%@ taglib uri="/WEB-INF/exampletag.tld" prefix="foo" %>
<% 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 tag handler class extends the standard `TagSupport` class to implement the `IterationTag` interface.
- The `doStartTag()` method returns the integer constant `EVAL_BODY_INCLUDE` 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_AGAIN` if there are more iterations left, and `SKIP_BODY` after the last iteration.
- This class does not define a `doEndTag()` method—the underlying implementation from `TagSupport` is used.

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 TagSupport
{

    String index;
    int count;
    int i;
    JmlNumber ib;

    public ExampleLoopTag() {
      resetAttr();
    }

    public void release() {
      resetAttr();
    }

    private void resetAttr() {
```

```
      index=null;
      count=0;
      i=0;
      ib=null;
    }

    public void setIndex(String index)
    {
      this.index=index;
    }
    public void setCount(String count)
    {
      this.count=Integer.parseInt(count);
    }

    public int doStartTag() throws JspException {
        ib=new JmlNumber();
        pageContext.setAttribute(index, ib);
        i++;
        ib.setValue(i);
        return EVAL_BODY_INCLUDE;
    }

    public int doAfterBody() throws JspException {
        if (i >= count) {
            return SKIP_BODY;
        } else
            pageContext.setAttribute(index, ib);
        i++;
        ib.setValue(i);
        return EVAL_BODY_AGAIN;
    }
}
```

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`, which you must specify as a fully qualified class name.
- The variable is to be newly declared (by the JSP translator).
- 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 Descriptor File: exampletag.tld

This section presents the TLD file for the tag library. In this example, the library consists of only the one tag, `loop`.

This TLD file follows JSP 1.2 syntax, specifying the following for the `loop` tag:

- The tag handler class is `examples.ExampleLoopTag`.
- The tag-extra-info class is `examples.ExampleLoopTagTEI`.
- The `body-content` specification is `JSP`. This means that the JSP translator should process and translate the body code.
- There are attributes `index` and `count`, both required. 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 1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>simple</short-name>
  <description>
       A simple tab library for the examples
  </description>
  <!-- example tag -->
  <!-- for loop -->
  <tag>
    <name>loop</name>
    <tag-class>examples.ExampleLoopTag</tag-class>
    <tei-class>examples.ExampleLoopTagTEI</tei-class>
    <body-content>JSP</body-content>
    <description>for loop</description>
    <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 1.2*, use a runtime support mechanism. They are typically portable, not requiring any particular JSP container.

It is also possible for vendors to support custom tags through vendor-specific functionality in their JSP translators. 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 this section discusses.

General Compile-Time Versus Runtime Considerations

The JSP 1.2 specification describes a runtime support mechanism for custom tag libraries. This mechanism, using an XML-style TLD file to specify the tags, is covered earlier in this chapter.

Creating and using a tag library that adheres to this model generally 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 can 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.

JSP Compile-Time Versus Runtime JML Library

OC4J provides a portable tag library called the JSP Markup Language (JML) library. This library uses the standard JSP 1.2 runtime mechanism. However, the JML tags are also supported through a compile-time mechanism. This is because the tags were first introduced with JSP implementations that preceded the JSP 1.1 specification, which is 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. There are also a few additional tags in that implementation, and some additional expression syntax that is supported.

The *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference* describes both the runtime version and the compile-time version of the JML library.

9

JSP Globalization Support

The JSP container in OC4J provides standard globalization support (also known as National Language Support, or NLS) according to the Sun Microsystems *JavaServer Pages Specification, Version 1.2*, 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 for uniform internal representation of text. Unicode is used as the base character set for conversion to alternative character sets. (The Unicode version depends on the JDK version. You can find the Unicode version through the Sun Microsystems Javadoc for the `java.lang.Character` class.)

This chapter describes key aspects of JSP support for globalization and internationalization. The following topics are covered:

- Content Type Settings
- JSP Support for Multibyte Parameter Encoding

Note: For detailed information about Oracle9*i*AS Globalization Support, see the *Oracle9i Application Server Globalization Support Guide*.

Content Type Settings

This section covers standard ways to statically or dynamically specify the content type for a JSP page. It also discusses an Oracle extension method that enables you to specify a non-IANA (Internet Assigned Numbers Authority) character set for the JSP writer object. The section is organized as follows:

- Content Type Settings in the page Directive
- Dynamic Content Type Settings
- Oracle Extension for the Character Set of the JSP Writer Object

Content Type Settings in the page Directive

The `page` directive has two attributes, `pageEncoding` and `contentType`, that affect the character encoding of the JSP page source (during translation) or response (during runtime). The `contentType` attribute also affects the MIME type of the response. The function of each attribute is as follows:

- You can use `contentType` to set the character encoding of the page source and response, and the MIME type of the response.
- You can use `pageEncoding` to set the character encoding of the page source. The main purpose of this attribute, which was added in the JSP 1.2 specification, is to allow you to set a page source character encoding that is different than the response character encoding. However, this setting also acts as a default for the response character encoding if there is no `contentType` attribute that specifies a character set.

(There is more information about the relationship between `contentType` and `pageEncoding` later in this section.)

Use the following syntax for `contentType`:

```
contentType="TYPE; charset=character_set"
```

or, to set the MIME type while using the default character set:

```
contentType="TYPE"
```

Use the following syntax for `pageEncoding`:

```
pageEncoding="character_set"
```

Use the following syntax to set everything:

```
<%@ page ... contentType="TYPE; charset=character_set"
            pageEncoding="character_set" ... %>
```

TYPE is an IANA MIME type; character_set is an IANA character set. When specifying a character set through the contentType attribute, the space after the semicolon is optional.

Here are some examples of contentType and pageEncoding settings:

```
<%@ page language="java" contentType="text/html" %>
```

or:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" %>
```

or:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
                         pageEncoding="US-ASCII" %>
```

Without any page directive settings, default settings are as follows:

- The default MIME type is text/html for traditional JSP pages; it is text/xml for JSP XML documents.
- The default for the page source character encoding (for translation) is ISO-8859-1 (also known as Latin-1) for traditional JSP pages; it is UTF-8 or UTF-16 for JSP XML documents.
- The default for the response character encoding is ISO-8859-1 for traditional JSP pages; it is UTF-8 or UTF-16 for JSP XML documents.

The determination of UTF-8 versus UTF-16 is according to "Autodetection of Character Encodings" in the XML specification, at http://www.w3.org/TR/REC-xml.html.

Be aware, however, that there is a relationship between pageEncoding and contentType regarding character encodings, as documented in the following table.

	contentType Encoding Is Specified	contentType Encoding Is Not Specified
pageEncoding Is Specified	Page source encoding is according to `pageEncoding`. Response encoding is according to `contentType`.	Page source encoding is according to `pageEncoding`. Response encoding is according to `pageEncoding`.
pageEncoding Is Not Specified	Page source encoding is according to `contentType`. Response encoding is according to `contentType`.	Page source encoding is according to the default. Response encoding is according to the default.

Be aware of the following important usage notes.

- A `page` directive that sets `contentType` or `pageEncoding` should appear as early as possible in the JSP page.
- When a page is a JSP XML document, any `pageEncoding` setting is ignored. The JSP container will instead use the XML encoding declaration of the document. Consider the following example:

```
<?xml version="1.0" encoding="EUC-JP" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="1.2">
<jsp:directive.page contentType="text/html;charset=Shift_Jis" />
<jsp:directive.page pageEncoding="UTF-8" />
...
```

  The effective page encoding would be `EUC-JP`, not `UTF-8`.
- You should use `pageEncoding` only for pages where the byte sequence represents legal characters in the target character set.
- You should use `contentType` only for pages or response output where the byte sequence represents legal characters in the target character set.
- The target character set of the response output (as specified by `contentType`, for example) should be a superset of the character set of the page source. For example, `UTF-8` is the superset of `Big5`, but `ISO-8859-1` is not.
- The parameters of a `page` directive are static. If a page discovers during execution that a different character set specification 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 9-5.

or:

  - Forward the request to another JSP page or to a servlet.

- A traditional JSP page source (not a JSP XML document) written in a character set other than `ISO-8859-1` must set the appropriate character set in a `page` directive (through the `contentType` or `pageEncoding` attribute). The character set for the page encoding cannot be set dynamically, because the JSP container has to be aware of the setting during translation.
- This manual, 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 and Internet Explorer browsers follow the setting you specify for the response parameters.

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 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
```

You should use only character sets from the IANA list, except for any additional Oracle extensions as described in "Oracle Extension for the Character Set of the JSP Writer Object" on page 9-6.

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:

```
void setContentType(java.lang.String contenttype)
```

Important: To use dynamic content type settings in an OC4J environment, you must enable the JSP `static_text_in_chars` configuration parameter. See "JSP Configuration Parameters" on page 3-9 for a description.

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` for traditional JSP pages or `text/xml` for JSP XML documents, and the default character encoding is `ISO-8859-1`.

Set the content type as early as possible in the page, before writing any output to the `JspWriter` object.

The `setContentType()` 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 9-2.

Note: In servlet 2.2 and higher environments, such as OC4J, the `response` object has a `setLocale()` method that takes a `java.util.Locale` object as input and sets the character set based on the specified locale. For example, the following method call results in a character set of `Shift_JIS`:

```
response.setLocale(new Locale("ja", "JP"));
```

For dynamic specification of the character set, the most recent call to `setContentType()` or `setLocale()` takes precedence.

Oracle Extension for the Character Set of the JSP Writer Object

In standard usage, the character set of the content type of the `response` object, as determined by the `page` directive `contentType` parameter or the `response.setContentType()` method, automatically becomes the character set of the JSP writer object as well. The JSP writer object is a `javax.servlet.jsp.JspWriter` instance.

There are some character sets, however, that are not recognized by IANA and therefore cannot be used in a standard content type setting. For this reason, OC4J provides the static `setWriterEncoding()` method of the `oracle.jsp.util.PublicUtil` class:

```
static void setWriterEncoding(JspWriter out, String encoding)
```

You can use this method to specify the character set of the JSP writer directly, overriding the character set of the `response` object. The following example uses `Big5` as the character set of the content type, but specifies `MS950`, a non-IANA Hong Kong dialect of `Big5`, as the character set of the JSP writer:

```
<%@ page contentType="text/html; charset=Big5" %>
<% oracle.jsp.util.PublicUtil.setWriterEncoding(out, "MS950"); %>
```

Note: Use the `setWriterEncoding()` method as early as possible in the JSP page.

JSP Support for Multibyte Parameter Encoding

The Sun Microsystems servlet 2.3 specification has a method, `setCharacterEncoding()`, in the `javax.servlet.ServletRequest` interface. This method is useful in case the default encoding of the servlet container is not suitable for multibyte request parameters and bean property settings, such as for a `getParameter()` call in Java code or a `jsp:setProperty` tag to set a bean property in JSP code.

The `setCharacterEncoding()` method and equivalent Oracle extensions affect 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

This section covers the following topics:

Standard setCharacterEncoding() Method

Effective with the servlet 2.3 specification, the `setCharacterEncoding()` method is specified in the `javax.servlet.ServletRequest` interface as the standard mechanism for specifying a nondefault character encoding for reading HTTP requests. The signature of this method is as follows:

```
void setCharacterEncoding(java.lang.String enc)
                          throws java.io.UnsupportedEncodingException
```

The `enc` parameter is a string specifying the name of the desired character encoding and overrides the default character encoding. Call this method before reading request parameters or reading input through the `getReader()` method (also specified in the `ServletRequest` interface).

There is also a corresponding getter method:

```
String getCharacterEncoding()
```

Overview of Oracle Extensions for Older Servlet Environments

In pre-2.3 servlet environments, the `setCharacterEncoding()` method is not available. For such environments, particularly the JServ servlet 2.0 environment, Oracle provides two alternative mechanisms:

- `oracle.jsp.util.PublicUtil.setReqCharacterEncoding()` static method (preferred)
- `translate_params` configuration parameter (or equivalent code)

For information about these mechanisms, see "Multibyte Parameter Encoding in JServ" on page B-24.

A

Servlet and JSP Technical Background

This appendix provides technical background on servlets and JavaServer Pages. Although this document is written for users who are well grounded in servlet technology, the servlet information here may be a useful refresher for some.

Standard JavaServer Pages interfaces, implemented automatically by generated JSP page implementation classes, are briefly discussed as well. Most readers, however, will not require this information.

The following topics are covered:

- Background on Servlets
- Web Application Hierarchy
- Standard JSP Interfaces and Methods

Note: For more information about servlets and the OC4J servlet container, refer to the *Oracle9iAS Containers for J2EE Servlet Developer's Guide.*

Background on Servlets

Because JSP pages are translated into Java servlets, a brief review of servlet technology may be helpful. Refer to the Sun Microsystems *Java Servlet Specification, Version 2.2* or *Version 2.3* for more information about the concepts discussed here.

For information about the methods this section discusses, refer to Sun Microsystems Javadoc at the following locations (for servlet 2.2 and 2.3, respectively):

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

```
http://java.sun.com/products/servlet/2.3/javadoc/index.html
```

Review of Servlet Technology

In recent years, servlet technology has emerged as a powerful way to extend Web server functionality through dynamic HTML pages. A servlet is a Java program that runs in a Web server (as opposed to an applet, which is a Java program that runs in a client browser). The servlet takes an HTTP request from a browser, generates dynamic content (such as by querying a database), and provides an HTTP response back to the browser.

Prior to servlets, CGI (Common Gateway Interface) technology was used for dynamic content, with CGI programs being written in languages such as Perl and being called by a Web application through the Web server. CGI ultimately proved less than ideal, however, due to its architecture and scalability limitations.

Servlet technology, in addition to improved scalability, offers the well-known Java advantages of object orientation, platform independence, security, and robustness. Servlets can use all standard Java APIs, including the JDBC API (for Java database connectivity, of particular interest to database programmers).

In the Java realm, servlet technology offers advantages over applet technology for server-intensive applications such as those accessing a database. One advantage is that a servlet runs in the server, which is usually a robust machine with many resources, minimizing use of client resources. An applet, by contrast, is downloaded into the client browser and runs there. Another advantage is more direct access to the data. The Web server or data server in which a servlet is running is on the same side of the network firewall as the data being accessed. An applet running on a client machine, outside the firewall, requires special measures (such as signed applets) to allow the applet to access any server other than the one from which it was downloaded.

The Servlet Interface

A Java servlet, by definition, implements the standard `javax.servlet.Servlet` interface. This interface specifies methods to initialize a servlet, process requests, get the configuration and other basic information of a servlet, and terminate a servlet instance.

For Web applications, you can implement the `Servlet` interface by extending the standard `javax.servlet.http.HttpServlet` abstract class. The `HttpServlet` class includes the following methods:

- `init(...)` and `destroy(...)`—to initialize and terminate the servlet, respectively
- `doGet(...)`—for HTTP `GET` requests
- `doPost(...)`—for HTTP `POST` requests
- `doPut(...)`—for HTTP `PUT` requests
- `doDelete(...)`—for HTTP `DELETE` requests
- `service(...)`—to receive HTTP requests and, by default, dispatch them to the appropriate `do`*`XXX`*`()` methods
- `getServletInfo(...)`—for use by the servlet to provide information about itself

A servlet class that subclasses `HttpServlet` must implement some of these methods, as appropriate. Each method takes as input a standard `javax.servlet.http.HttpServletRequest` instance and a standard `javax.servlet.http.HttpServletResponse` instance.

The `HttpServletRequest` instance provides information to the servlet regarding the HTTP request, such as request parameter names and values, the name of the remote host that made the request, and the name of the server that received the request. The `HttpServletResponse` instance provides HTTP-specific functionality in sending the response, such as specifying the content length and MIME type and providing the output stream.

Servlet Containers

Servlet containers, sometimes referred to as *servlet engines*, execute and manage servlets. A servlet container is usually written in Java and is either part of a Web server (if the Web server is also written in Java) or otherwise associated with and used by a Web server.

When a servlet is called (such as when a servlet is specified by URL), the Web server passes the HTTP request to the servlet container. The container, in turn, passes the request to the servlet. In the course of managing a servlet, a simple container performs the following:

- It creates an instance of the servlet and calls its `init()` method to initialize it.
- It calls the `service()` method of the servlet.
- It calls the `destroy()` method of the servlet to discard it when appropriate, so that it can be garbage-collected.

  For performance reasons, it is typical for a servlet container to keep a servlet instance in memory for reuse, rather than destroying it each time it has finished its task. It would be destroyed only for infrequent events, such as Web server shutdown.

If there is an additional servlet request while a servlet is already running, servlet container behavior depends on whether the servlet uses a single-thread model or a multiple-thread model. In a single-thread case, the servlet container prevents multiple simultaneous `service()` calls from being dispatched to a single servlet instance—it may spawn multiple separate servlet instances instead. In a multiple-thread model, the container can make multiple simultaneous `service()` calls to a single servlet instance, using a separate thread for each call, but the servlet developer is responsible for managing synchronization.

Servlet Sessions

Servlets use HTTP sessions to keep track of which user each HTTP request comes from, so that a group of requests from a single user can be managed in a stateful way. Servlet session-tracking is similar in nature to HTTP session-tracking in previous technologies, such as CGI.

HttpSession Interface

In the standard servlet API, each user is represented by an instance of a class that implements the standard `javax.servlet.http.HttpSession` interface. Servlets can set and get information about the session in this `HttpSession` object, which must be of application-level scope.

A servlet uses the `getSession()` method of an `HttpServletRequest` object (which represents an HTTP request) to retrieve or create an `HttpSession` object for the user. This method takes a boolean argument to specify whether a new session object should be created for the user if one does not already exist.

The `HttpSession` interface specifies the following methods to get and set session information:

- `public void setAttribute(String name, Object value)`

  This method binds the specified object to the session, under the specified name.

- `public Object getAttribute(String name)`

  This method retrieves the object that is bound to the session under the specified name (or `null` if there is no match).

Note: Older servlet implementations use `putValue()` and `getValue()` instead of `setAttribute()` and `getAttribute()`, with the same signatures.

Depending on the implementation of the servlet container and the servlet itself, sessions may expire automatically after a set amount of time or may be invalidated explicitly by the servlet. Servlets can manage session lifecycle with the following methods, specified by the `HttpSession` interface:

- `public boolean invalidate()`

  This method immediately invalidates the session and unbinds any objects from it.

- `public boolean setMaxInactiveInterval(int interval)`

  This method sets a timeout interval, in seconds, as an integer.

- `public boolean isNew()`

  This method returns `true` within the request that created the session; it returns `false` otherwise.

- `public boolean getCreationTime()`

  This method returns the time when the session object was created, measured in milliseconds since midnight, January 1, 1970.

- `public boolean getLastAccessedTime()`

  This method returns the time of the last request associated with the client, measured in milliseconds since midnight, January 1, 1970.

Session Tracking

The `HttpSession` interface supports alternative mechanisms for tracking sessions. Each involves some way to assign a *session ID*. A session ID is an intermediate handle that is assigned and used by the servlet container. Multiple sessions by the same user can share the same session ID, if appropriate.

The following session-tracking mechanisms are supported:

- cookies

  The servlet container sends a cookie to the client, which returns the cookie to the server upon each HTTP request. This associates the request with the session ID indicated by the cookie. This is the most frequently used mechanism and is supported by any servlet container that adheres to the servlet 2.2 or higher specification.

- URL rewriting

  The servlet container appends a session ID to the URL path, as in the following example:

  `http://`*`host`*`[:`*`port`*`]/myapp/index.html?jsessionid=6789`

  This is the most frequently used mechanism where clients do not accept cookies.

- SSL Sessions

  SSL (Secure Sockets Layer, used in the HTTPS protocol) includes a mechanism to take multiple requests from a client and define them as belonging to a single session. Some servlet containers use the SSL mechanism for their own session tracking as well.

Servlet Contexts

A *servlet context* is used to maintain state information for all instances of a Web application within any single JVM (that is, for all servlet and JSP page instances that are part of the Web application). This is similar to the way a session maintains state information for a single client on the server; however, a servlet context is not specific to any single user and can handle multiple clients. There is usually one servlet context for each Web application running within a given JVM. You can think of a servlet context as an application container.

Any servlet context is an instance of a class that implements the standard `javax.servlet.ServletContext` interface, with such a class being provided with any Web server that supports servlets.

A `ServletContext` object provides information about the servlet environment (such as name of the server) and allows sharing of resources between servlets in the group, within any single JVM. (For servlet containers supporting multiple simultaneous JVMs, implementation of resource-sharing varies.)

A servlet context maintains the session objects of the users who are running the application and provides a scope for the running instances of the application. Through this mechanism, each application is loaded from a distinct class loader and its runtime objects are distinct from those of any other application. In particular, the `ServletContext` object is distinct for an application, as is the `HttpSession` object for each user of the application.

Beginning with the Sun Microsystems *Java Servlet Specification, Version 2.2,* most implementations can provide multiple servlet contexts within a single host, which is what allows each Web application to have its own servlet context. (Previous implementations usually provided only a single servlet context with any given host.)

The `ServletContext` interface specifies methods that allow a servlet to communicate with the servlet container that runs it, which is one of the ways that the servlet can retrieve application-level environment and state information.

Note: In early versions of the servlet specification, the concept of servlet contexts was not sufficiently defined. Beginning with version 2.1(b), however, the concept was further clarified, and it was specified that an HTTP session object could not exist across multiple servlet context objects.

Application Lifecycle Management Through Event Listeners

The Sun Microsystems *Java Servlet Specification, Version 2.2* first provided limited application lifecycle management through the standard Java event-listener mechanism. HTTP session objects can use event listeners to make objects stored in the session object aware of when they are added or removed. Because the typical reason for removing objects within a session object is that the session has become invalid, this mechanism allows the developer to manage session-based resources. Similarly, the event-listener mechanism also allows the managing of page-based and request-based resources.

The *Java Servlet Specification, Version 2.3* provides additional support for event listeners, defining interfaces you can implement for event listeners that can be informed of changes in the servlet context lifecycle, servlet context attributes, the

HTTP session lifecycle, and HTTP session attributes. See the *Oracle9iAS Containers for J2EE Servlet Developer's Guide* for more information.

Servlet Invocation

A servlet, like an HTML page, can be directly invoked through a URL. The servlet is launched according to how servlets are mapped to URLs in the Web server implementation. Following are the possibilities:

- A specific URL can be mapped to a specific servlet class.
- An entire directory can be mapped so that any class in the directory is executed as a servlet. For example, the special `/servlet` directory can be mapped so that any URL of the form `/servlet/`*`servlet_name`* executes a servlet.
- A file name extension can be mapped so that any URL specifying a file whose name includes that extension executes a servlet.

This mapping would be specified as part of the Web server configuration. In OC4J, this is according to settings in the `global-web-application.xml` file.

A servlet can also be invoked indirectly, like a JSP page, such as through a `jsp:include` or `jsp:forward` tag. See "Invoking a Servlet from a JSP Page" on page 4-2.

Web Application Hierarchy

The entities relating to a Web application (which consists of some combination of servlets and JSP pages) do not follow a simple hierarchy but can be considered in the following order:

1. servlet objects (including page implementation objects)

   There is a servlet object for each servlet and for each JSP page implementation in a running application (and possibly more than one object, depending on whether a single-thread or multiple-thread execution model is used). A servlet object processes request objects from a client and sends response objects back to the client. A JSP page, as with servlet code, specifies how to create the response objects.

   You can think of multiple servlet objects as being within a single request object in some circumstances, such as when one page or servlet "includes" or forwards to another.

   A user will typically access multiple servlet objects in the course of a session, with the servlet objects being associated with the session object.

   Servlet objects, as well as page implementation objects, indirectly implement the standard `javax.servlet.Servlet` interface. For servlets in a Web application, this is accomplished by subclassing the standard `javax.servlet.http.HttpServlet` abstract class. For JSP page implementation classes, this is accomplished by implementing the standard `javax.servlet.jsp.HttpJspPage` interface.

2. request and response objects

   These objects represent the individual HTTP requests and responses that are generated as a user runs an application.

   A user will typically generate multiple requests and receive multiple responses in the course of a session. The request and response objects are not contained in the session, but are associated with the session.

   As a request comes in from a client, it is mapped to the appropriate servlet context object (the one associated with the application the client is using) according to the virtual path of the URL. The virtual path will include the root path of the application.

   A request object implements the standard `javax.servlet.http.HttpServletRequest` interface.

A response object implements the standard `javax.servlet.http.HttpServletResponse` interface.

3. session objects

   Session objects store information about the user for a given session and provide a way to identify a single user across multiple page requests. There is one session object for each user.

   There may be multiple users of a servlet or JSP page at any given time, each represented by their own session object. All these session objects, however, are maintained by the servlet context that corresponds to the overall application. In fact, you can think of each session object as representing an instance of the Web application associated with a common servlet context.

   Typically, a session object will sequentially make use of multiple request objects, response objects, and page or servlet objects, and no other session will use the same objects; however, the session object does not actually contain those objects.

   A session lifecycle for a given user starts with the first request from that user. It ends when the user session terminates (such as when the user quits the application) or there is a timeout.

   HTTP session objects implement the `javax.servlet.http.HttpSession` interface.

   Note: Prior to the 2.1(b) version of the servlet specification, a session object could span multiple servlet context objects.

4. servlet context object

   A servlet context object is associated with a particular path in the server. This is the base path for modules of the application associated with the servlet context, and is referred to as the *application root*.

   There is a single servlet context object for all sessions of an application in any given JVM, providing information from the server to the servlets and JSP pages that form the application. The servlet context object also allows application sessions to share data within a secure environment isolated from other applications.

   The servlet container provides a class that implements the standard `javax.servlet.ServletContext` interface, instantiates this class the first

time a user requests an application, and provides this `ServletContext` object with the path information for the location of the application.

The servlet context object typically has a pool of session objects to represent the multiple simultaneous users of the application.

A servlet context lifecycle starts with the first request (from any user) for the corresponding application. The lifecycle ends only when the server is shut down or otherwise terminated.

For additional introductory information about servlet contexts, see "Servlet Contexts" on page A-6.

5. servlet configuration object

   The servlet container uses a servlet configuration object to pass information to a servlet when it is initialized—the `init()` method of the `Servlet` interface takes a servlet configuration object as input.

   The servlet container provides a class that implements the standard `javax.servlet.ServletConfig` interface and instantiates it as necessary. Included within the servlet configuration object is a servlet context object (also instantiated by the servlet container).

Standard JSP Interfaces and Methods

Two standard interfaces, both in the `javax.servlet.jsp` package, are available to be implemented in code that is generated by a JSP translator:

- `JspPage`
- `HttpJspPage`

`JspPage` is a generic interface that is not intended for use with any particular protocol. It extends the `javax.servlet.Servlet` interface.

`HttpJspPage` is an interface for JSP pages using the HTTP protocol. It extends `JspPage` and is typically implemented directly and automatically by any servlet class generated by a JSP translator.

`JspPage` specifies the following methods for use in initializing and terminating instances of the generated class:

- `jspInit()`
- `jspDestroy()`

If you want any special initialization or termination functionality, you must provide a JSP declaration to override the relevant method, as in the following example:

```
<%! void jspInit()
    {
        ...your implementation code...
    }
%>
```

`HttpJspPage` adds specification for the following method:

- `_jspService()`

Code for this method is typically generated automatically by the translator and includes the following:

- code from scriptlets in the JSP page
- code resulting from any JSP directives
- any static content of the page

(JSP directives provide information for the page, such as specifying the Java language for scriptlets and providing package imports. See "Directives" on page 1-7.)

As with the `Servlet` methods, the `_jspService()` method takes an `HttpServletRequest` instance and an `HttpServletResponse` instance as input.

The `JspPage` and `HttpJspPage` interfaces inherit the following methods from the `Servlet` interface:

- `init()`
- `destroy()`
- `service()`
- `getServletConfig()`
- `getServletInfo()`

Refer back to "The Servlet Interface" on page A-3 for a discussion of the `Servlet` interface and its key methods.

B

The Apache JServ Environment

The primary Web application environment supplied with the Oracle9*i* Application Server is Oracle9*i*AS Containers for J2EE (OC4J). In addition, as of Oracle9*i*AS release 2, an Apache JServ servlet environment is provided. (In fact, JServ was the primary servlet environment in earlier releases of the Oracle9*i* Application Server.)

For those who use the JServ environment (presumably for backward compatibility), there are special considerations relating to servlet and JSP usage, as with any servlet 2.0 environment. This appendix covers these considerations.

Specifically, the following topics are discussed:

- Getting Started in a JServ Environment
- Considerations for the JServ Environment
- JSP Application and Session Support for JServ
- Samples Using globals.jsa for Servlet 2.0 Environments

Getting Started in a JServ Environment

This section provides information about configuring JServ to run JSP pages, covering the following topics:

Adding Files to the Apache 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 OC4J for servlet 2.2 or higher versions of `javax.servlet.*` packages) is in your classpath in a JServ environment, `jsdk.jar` must precede it.

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.

JSP Configuration Parameters for JServ

This section describes the configuration parameters supported by the Oracle `JspServlet` for the JServ environment.

For information about JSP configuration parameters for OC4J, see "JSP Configuration Parameters" on page 3-9.

Configuration Parameter Summary Table for JServ

Table 9–1 summarizes the configuration parameters supported by the original Oracle JSP front-end servlet, `oracle.jsp.JspServlet`. This is the front-end used for the JServ environment. (OC4J uses the front-end servlet `oracle.jsp.runtimev2.JspServlet`.) For each parameter, the table notes the following:

- the correspondence (if any) to OC4J configuration parameters
- equivalent or related `ojspc` options for pre-translation
- a brief description of the option
- the default value
- whether it is used at compile-time or runtime

Notes:

- See "The ojspc Pre-Translation Utility" on page 7-13 for information about the equivalent `ojspc` options.
- The `main_mode`, `precompile_check`, and `static_text_in_chars` parameters, offered for OC4J, are not available for JServ environments. For JServ, however, outputting static text as characters is the default.

Table 9–1 JSP Configuration Parameters, JServ Environment

Parameter	Relation to Config Params in OC4J	Related ojspc Options	Description	Default	Runtime / Compile-Time
alias_translation	not applicable in OC4J	(n/a)	Set this boolean to `true` to work around JServ limitations in directory aliasing for JSP page references.	false	both
bypass_source	migrated to `main_mode` flag, `justrun` setting	(n/a)	Set this boolean to `true` for the JSP container to ignore `FileNotFound` exceptions on `.jsp` source. Uses pre-translated and compiled code when source is not available.	false	runtime
classpath	not applicable in OC4J	-addclasspath (related, but different functionality)	This is for additional classpath entries for JSP class loading.	null (no addl. path)	both
debug_mode	same	(n/a)	Set this boolean to `true` for the JSP container to print the stack trace when a runtime exception occurs.	false	runtime
developer_mode	migrated to `main_mode` flag	(n/a)	Set this boolean to `false` to *not* check timestamps to see if page retranslation and class reloading is necessary when a page is requested.	true	runtime
emit_debuginfo	same	-debug	Set this boolean to `true` to generate a line map to the original `.jsp` file for debugging.	false	compile-time

Table 9–1 JSP Configuration Parameters, JServ Environment (Cont.)

Parameter	Relation to Config Params in OC4J	Related ojspc Options	Description	Default	Runtime / Compile-Time
external_resource	same	-extres	Set this boolean to `true` for the JSP translator to place all static content of the page into a separate Java resource file during translation.	false	compile-time
external_resource_timeout	equivalent functionality in `jsp-timeout`	(n/a)	Set this for a timeout value to free external resources (relevant if `external_resource` is enabled).	0 (no timeout)	runtime
extra_imports	same	-extraImports	Use this to add imports beyond the JSP defaults.	null	compile-time
forgive_dup_dir_attr	same	-forgiveDupDirAttr	Set this boolean to `true` to avoid JSP 1.2 translation errors if you have duplicate settings for the same directive attribute within a single JSP translation unit.	false	compile-time
javaccmd	same	-noCompile	This is for a Java compiler command line—`javac` options, or an alternative Java compiler to run in a separate JVM. Set it to `null` for JDK `javac` with default options.	null	compile-time
no_tld_xml_validate	same	-noTldXmlValidate	Set this boolean to `true` to *not* perform XML validation of TLD files. By default, validation of TLD files is performed.	false	compile-time

Table 9–1 JSP Configuration Parameters, JServ Environment (Cont.)

Parameter	Relation to Config Params in OC4J	Related ojspc Options	Description	Default	Runtime / Compile-Time
old_include_from_top	same	-oldIncludeFromTop	Set this boolean to `true` for page locations in nested `include` directives to be relative to the top-level page, for backward compatibility with Oracle JSP behavior prior to Oracle9*i*AS release 2.	false	compile-time
reduce_tag_code	same	-reduceTagCode	Set this boolean to `true` for further reduction in the size of generated code for custom tag usage.	false	compile-time
req_time_introspection	same	-reqTimeIntrospection	Set this boolean to `true` to enable request-time JavaBean introspection when compile-time introspection is not possible.	false	compile-time
send_error	not applicable in OC4J	(n/a)	Set this boolean to `true` to output standard "404" errors for file-not-found, and "500" errors for compilation failure (instead of outputting customized messages).	false	runtime
session_sharing (for use with `globals.jsa`)	not applicable in OC4J	(n/a)	For applications using `globals.jsa`, set this boolean to `true` for JSP session data to be propagated to the underlying servlet session.	true	runtime

Table 9–1 JSP Configuration Parameters, JServ Environment (Cont.)

Parameter	Relation to Config Params in OC4J	Related ojspc Options	Description	Default	Runtime / Compile-Time
sqljcmd	same	-S	This is for a SQLJ command line—`sqlj` options, or alternative SQLJ translator to run in a separate JVM. Set this to `null` for the Oracle SQLJ version provided with OC4J, with default option settings.	null	compile-time
tags_reuse_default	same	(n/a)	This boolean specifies a default setting for JSP tag handler pooling (`true` to enable by default, `false` to disable by default). This default setting can be overridden for any particular JSP page.	false	runtime
translate_params	not applicable in OC4J	(n/a)	Set this boolean to `true` to override servlet containers that do not perform multibyte encoding.	false	runtime

Table 9–1 JSP Configuration Parameters, JServ Environment (Cont.)

Parameter	Relation to Config Params in OC4J	Related ojspc Options	Description	Default	Runtime / Compile-Time
unsafe_reload	not applicable in OC4J	(n/a)	Set this boolean to `true` to *not* restart the application and sessions whenever a JSP page is retranslated and reloaded.	false	runtime
well_known_taglib_loc	same	(n/a)	This specifies a directory where tag library JAR files can be placed for sharing across multiple Web applications. The default location is `j2ee/home/jsp/lib/taglib/` under the `[Oracle_Home]` directory.	per previous column	compile-time
xml_validate	same	-xmlValidate	Set this boolean to `true` for XML validation to be performed on the `web.xml` file.	false	compile-time

Configuration Parameter Descriptions for JServ

This section describes configuration parameters for the JServ environment in more detail.

alias_translation (boolean; default: `false`)

This parameter allows the OC4J JSP container to work around limitations in the way JServ handles directory aliasing. For information about the current limitations, see "JServ Directory Alias Translation" on page B-21.

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 JSP container throws 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 JSP container to proceed and execute the page implementation class even if it cannot find the page source.

If `bypass_source` is enabled, the container still checks 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 7-40.)
- 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 JSP default classpath for use during translation, compilation, or execution of JSP pages.

Overall, the 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 that are loaded through the path specified in the `classpath` setting are loaded by the JSP class loader, not by the system class loader. During JSP execution, classes that are loaded by the JSP class loader cannot access (or be accessed by) classes that are loaded by the system class loader or by any other class loader.

Notes:

- Runtime automatic class reloading applies only to classes in the JSP classpath. This includes paths specified through this `classpath` parameter. (See "Dynamic Page Retranslation and Class Reloading" on page 6-17 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 7-20.

debug_mode (boolean; default: `false`)

This flag has the same use in JServ as in OC4J. See "JSP Configuration Parameter Descriptions" on page 3-12.

developer_mode (boolean; default: `true`)

Set this flag to `false` to instruct the 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 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 translator 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 reexecute 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 JSP container checks to see if such classes have been modified since being loaded by the JSP class loader.

Oracle generally recommends setting `developer_mode` to `false`, particularly in a production environment where code is not likely to change and where performance is a significant issue.

Also see "Dynamic Page Retranslation and Class Reloading" on page 6-17.

emit_debuginfo (boolean; default: `false`)

This has the same use in JServ as in OC4J. See "JSP Configuration Parameter Descriptions" on page 3-12.

external_resource (boolean; default: `false`)

This has the same use in JServ as in OC4J. See "JSP Configuration Parameter Descriptions" on page 3-12.

external_resource_timeout (integer; default: `0`, no timeout)

If `external_resource` is enabled, you can use `external_resource_timeout` to specify a timeout value, in seconds, for JSP page resources. If a JSP page has not been accessed in this amount of time, then its external resources will be freed. They will be reloaded the next time the page is requested.

Note: In OC4J, the `jsp-timeout` flag frees external resources, along with the page as a whole. This is an attribute of the `<orion-web-app>` element in the `global-web-application.xml` file or `orion-web.xml` file. See "OC4J Configuration Parameters for JSP" on page 3-21.

---

extra_imports (import list; default: `null`)

As described in "Default Package Imports" on page 3-5, as of Oracle9*i*AS release 2 (9.0.3) the OC4J JSP container has a smaller default list of packages that are imported into each JSP page. This is in accordance with the JSP specification. You can avoid updating your code, however, by specifying package names or fully qualified class names for any additional imports through the `extra_imports` configuration parameter. See "Setting JSP Parameters in JServ" on page B-15 for general syntax, and be aware that the list of names must be quoted and space-delimited, as in the following example:

```
servlet.oracle.jsp.JspServlet.initArgs=extra_imports='java.util.* java.io.*'
```

forgive_dup_dir_attr (boolean; default: `false`)

This has the same use in JServ as in OC4J. See "JSP Configuration Parameter Descriptions" on page 3-12.

javaccmd (compiler executable and options; default: `null`)

This has the same use in JServ as in OC4J. See "JSP Configuration Parameter Descriptions" on page 3-12.

no_tld_xml_validate (boolean; default: `false`)

This has the same use in JServ as in OC4J. See "JSP Configuration Parameter Descriptions" on page 3-12.

old_include_from_top (boolean; default: `false`)

This has the same use in JServ as in OC4J. See "JSP Configuration Parameter Descriptions" on page 3-12.

reduce_tag_code (boolean; default: `false`)

This has the same use in JServ as in OC4J. See "JSP Configuration Parameter Descriptions" on page 3-12.

req_time_introspection (boolean; default: `false`)

This has the same use in JServ as in OC4J. See "JSP Configuration Parameter Descriptions" on page 3-12.

send_error (boolean; default: `false`)

Set this flag to `true` to direct the JSP container to output generic "404" messages for file-not-found conditions, and generic "500" messages for compilation errors.

This is in contrast 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 you use a `globals.jsa` file 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 `true` (default) 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 "JSP Application and Session Support for JServ" on page B-32.

sqljcmd (SQLJ translator executable and options; default: `null`)

This has the same use in JServ as in OC4J. See "JSP Configuration Parameter Descriptions" on page 3-12.

tags_reuse_default (boolean; default: `false`)

This has the same use in JServ as in OC4J, but in JServ is `false` by default. See "JSP Configuration Parameter Descriptions" on page 3-12.

translate_params (boolean; default: `false`)

Note: It is preferable to use the `PublicUtil.setReqCharacterEncoding()` method instead of the `translate_params` parameter. See "The setReqCharacterEncoding() Method" on page B-24.

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 JSP container encodes request parameters and bean property settings. Otherwise, it 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 "Multibyte Parameter Encoding in JServ" on page B-24.

unsafe_reload (boolean; default: `false`)

By default, the JSP container restarts the application and sessions whenever a JSP page is dynamically retranslated and reloaded (which occurs when there is 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 to *not* restart the application after dynamic retranslations and reloads. This avoids having existing sessions become invalid. A `true` setting is appropriate for deployment environments. The `false` (default) setting is appropriate for development environments.

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).

well_known_taglib_loc (directory path; default: `j2ee/home/jsp/lib/taglib/`)

This has the same use in JServ as in OC4J. See "JSP Configuration Parameter Descriptions" on page 3-12.

xml_validate (boolean; default: `false`)

This has the same use in JServ as in OC4J. See "JSP Configuration Parameter Descriptions" on page 3-12.

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 Apache 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 -ser2class=true,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-delimited, 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.

Using ojspc for JServ

Using `ojspc` for a JServ environment requires a different command—use `ojspc_jserv` instead of `ojspc`. This executes the same class, `oracle.jsp.tool.Jspc`, but is set up with a classpath appropriate for JServ.

The `ojspc -staticTextInChars` option has no effect for JServ, because in a JServ environment static text is output as characters by default. You cannot disable this.

Important: To use `ojspc_jserv` (as with `ojspc`), you must be using a Sun Microsystems JDK (version 1.1.8 or higher) and you must have `tools.jar` (for JDK 2.0 or higher) or `classes.zip` (for JDK 1.1.8) in your classpath.

Considerations for the JServ Environment

There are special considerations in running JSP pages in the JServ environment, because it is a servlet 2.0 environment. The servlet 2.0 specification lacks support for some significant features that are available in servlet 2.2 and higher environments.

For information about how to configure a JServ environment for JSP pages, see "Getting Started in a JServ Environment" on page B-2.

This section discusses the following considerations for the JServ environment:

- The mod_jserv Apache Mod
- JSP Container Features for Application Root Support in JServ
- Overview of Application and Session Framework for JServ
- JSP and Servlet Session Sharing in JServ
- Dynamic Includes and Forwards in JServ
- JServ Directory Alias Translation
- JSP Security Considerations in JServ
- Multibyte Parameter Encoding in JServ

The mod_jserv Apache Mod

The `mod_jserv` component, supplied by Apache, delegates HTTP requests to JSP pages or servlets running in the JServ servlet container in a middle-tier JVM. 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 the proprietary Apache JServ protocol (AJP) over TCP/IP. The `mod_jserv` component can delegate requests to multiple JVMs in a pool for load balancing.

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

JSP Container Features for Application Root Support in JServ

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. By default, JSP pages and servlets running in the JServ environment of the Oracle9*i* Application Server are routed through the

Apache `mod_jserv` module provided with JServ, and use the Apache JServ doc root. This is typically some `.../htdocs` directory. In addition, it is possible to specify "virtual" doc roots through `alias` settings in the `httpd.conf` configuration file.

The OC4J JSP container does, however, offer additional functionality regarding doc roots and application roots in the JServ environment. Through the OC4J JSP `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 B-32 for more information.

The application root directory can be located in any of the following locations, listed in the order they are searched:

1. the Web server directory to which the application is mapped
2. the Web server document root directory
3. the directory containing the `globals.jsa` file

Overview of Application and Session Framework for JServ

Because the concept of a Web application is not well defined in the servlet 2.0 specification, in JServ there is only one servlet context for each servlet container. Additionally, there is only one session object for each servlet container.

OC4J, however, supports a special application framework for use in the JServ environment. It accomplishes this through a file, `globals.jsa`, that you can use as an application marker. This allows distinct servlet contexts and session objects for each application.

For more information, see "Distinct Applications and Sessions Through globals.jsa" on page B-33.

JSP and Servlet Session Sharing in JServ

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 JSP container in a JServ environment, is in the same zone as the servlet or servlets that you want your JSP pages to share a session with. 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, when you use a `globals.jsa` file for an application, presumably in a servlet 2.0 environment such as JServ, 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 `true` (default) setting of the JSP `session_sharing` configuration 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 "JSP Application and Session Support for JServ" on page B-32.

Also see these sections for related information:

- "JSP Application and Session Support for JServ" on page B-32
- "JSP Configuration Parameters for JServ" on page B-4
- "Setting JSP Parameters in JServ" on page B-15

Dynamic Includes and Forwards in JServ

JSP dynamic includes (using the `jsp:include` tag) and forwards (using the `jsp:forward` tag) rely on request dispatcher functionality that is present in servlet 2.2 and higher environments, but not in servlet 2.0 environments.

The OC4J 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 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, `TestServlet`:

```
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 2.2 or higher environment.

Notes:

- 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 `jsp:include` or `jsp:forward` action 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.)

JServ Directory Alias Translation

Apache JServ 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 results 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 Apache JServ `getRealPath()` method returns an incorrect value when processing a file under an alias directory.

The OC4J JSP container supports an Apache-specific configuration parameter, `alias_translation`, that works around this limitation when you set it to `true`. (The default setting is `false`.)

Be aware that setting `alias_translation` to `true` also results in the alias directory becoming the application root. Therefore, in a `jsp:include` or `jsp:forward` tag 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/"
```

And assume there is a JSP page located as follows:

```
/private/foo/xxx.jsp
```

The following `jsp:include` tag 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 possible to forward to or include pages or HTML files only within the same application (according to the servlet 2.2 and 2.3 specifications).

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 B-15.

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 the following 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 arrange it so that the physical directories do not have the same names as the alias directories:

  ```
  Alias /foo/bar1  "/path/to/my/dir/x_bar1"
  Alias /foo/bar2  "/path/to/my/dir/y_bar2"
  ```

  Note the use of `x_bar1` instead of `bar1` and `y_bar2` instead of `bar2`. In the problematic example earlier, the first alias used `bar1`, which is the same as the directory name, and the second alias used `bar2`, which is the same as the directory name.

JSP Security Considerations in JServ

In a JServ environment, be aware of the following security considerations:

- It is highly advisable for access to be denied to any `_pages` directory. By default, access is already denied to the default `_pages` directory. In addition, if you are using aliases, you should deny access to any `_pages` directory under each alias. See "Generated Files and Locations" on page 7-6 for general information about the `_pages` directory.
- It is also highly advisable for access to be denied to the `globals.jsa` file. Access is already denied by default. For information about `globals.jsa`, see "JSP Application and Session Support for JServ" on page B-32.

Multibyte Parameter Encoding in JServ

This section describes Oracle extensions to support multibyte request parameters and bean property settings in a JServ or other servlet 2.0 environment, such as for a `getParameter()` call in Java code or for a `jsp:setProperty` tag to set a bean property in JSP code. There are two mechanisms for this:

- `oracle.jsp.util.PublicUtil.setReqCharacterEncoding()` static method (preferred)
- `translate_params` configuration parameter (or equivalent code)

The discussion of `translate_params` is followed by a discussion of how to migrate away from its use when you move to an OC4J environment.

For general information about multibyte parameter encoding, see "JSP Support for Multibyte Parameter Encoding" on page 9-8.

The setReqCharacterEncoding() Method

For pre-2.3 servlet environments, 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` tag 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, with the following signature:

```
public static void setReqCharacterEncoding
                  (HttpServletRequest req, String encoding)
                  throws java.io.UnsupportedEncodingException
```

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(request, "EUC-JP");
```

Note: The `setReqCharacterEncoding()` method is forward-compatible with the method `request.setCharacterEncoding()` of the servlet 2.3 API.

JSP translate_params Configuration Parameter

Set this boolean flag to `true` to override servlet containers that do not encode multibyte (globalization support) request parameters or bean property settings. (The default setting is `false`.) With a `true` setting, the JSP container decodes and encodes request parameters and bean property settings. Otherwise, it returns the parameters from the servlet container unchanged.

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 may cause incorrect results. It is known, however, that JServ 1.1 does *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 B-26.)

Effect of translate_params in Overriding Non-Multibyte Servlet Containers

Setting `translate_params` to `true` overrides insufficient functionality of servlet containers that cannot decode and encode multibyte request parameters and bean property settings. (For information about how to set JSP configuration parameters, see "Setting JSP Parameters in JServ" on page B-15.)

When this flag is enabled, the 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.

Code Equivalent to the translate_params Configuration Parameter

There may be situations where you cannot use or do not want to use the `translate_params` configuration parameter. It is useful to be aware of equivalent functionality that you can implement 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:

- It sets XXYYZZ as the parameter name to search for. (Presume XX, YY, and ZZ are three Japanese characters.)
- It 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.)

- It 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.)
- It 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 setting the JSP configuration parameter `translate_params` to `true`.

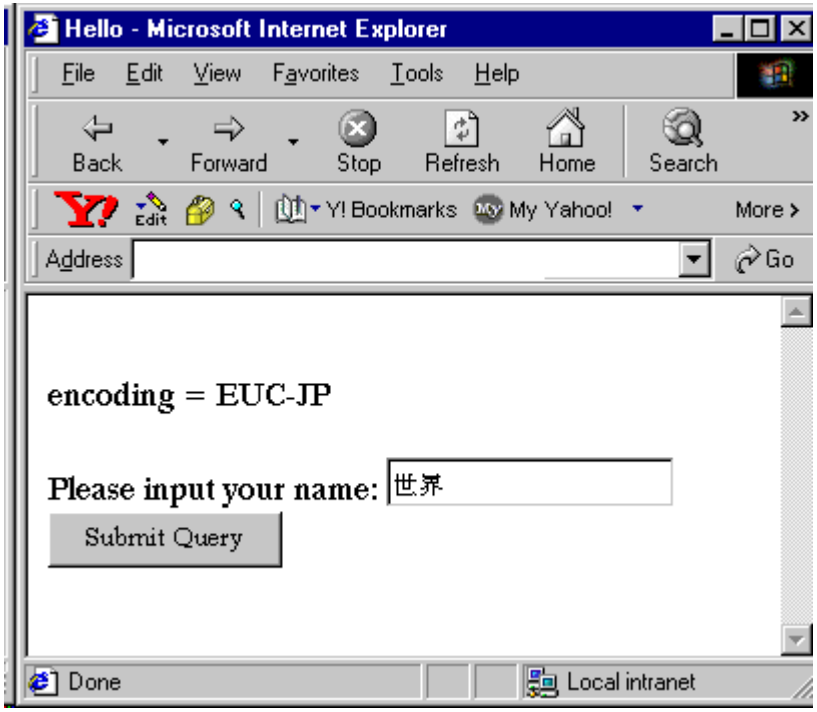
Presume `XXYY` is the parameter name (something equivalent to "user name" in Japanese) and `AABB` is the default value (also in Japanese).

(See the next section for a sample that has the code equivalent of the `translate_params` functionality, thereby not depending 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:



and the sample output:

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 may 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 B-26.

```
<%@ 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>
```

Migration Away from translate_params

The *global includes* functionality in the OC4J JSP container, described in "Oracle JSP Global Includes" on page 7-9, is useful in migrating applications that have previously used `translate_params` for globalization.

In this case, the globally included file can consist of a scriptlet similar to one of the following to achieve functionality that is equivalent to that of `translate_params`.

- Hardcode the request character set:

```
<% request.setCharacterEncoding("desired_charset"); %>
```

or:

- Use the character set of the response as the character set of the request, where the character set of the response is specified in a JSP `page` directive:

```
<% request.setCharacterEncoding(response.getCharacterEncoding()); %>
```

or:

- Use the character set of the response as the character set of the request, where the character set of the response is determined dynamically by Java logic:

```
<% String yourCharSet = yourLogicToDetermineCharset();
   response.setContentType("text/html; charset="+yourCharSet);
   request.setCharacterEncoding(response.getCharcterEncoding());
   // NOTE: The relative ordering of response.setContentType()
   // and request.setCharacterEncoding() is important.
%>
```

JSP Application and Session Support for JServ

OC4J supports a file, `globals.jsa`, as a mechanism for implementing the JSP specification in a servlet 2.0 environment. Web applications and servlet contexts are not fully defined in the servlet 2.0 specification.

This section discusses the `globals.jsa` mechanism and covers the following topics, including information about migrating away from `globals.jsa` when you move to an OC4J environment:

- Overview of globals.jsa Functionality
- Overview of globals.jsa Syntax and Semantics
- The globals.jsa Event-Handlers
- Global Declarations and Directives
- Migration from globals.jsa

For sample applications, see "Samples Using globals.jsa for Servlet 2.0 Environments" on page B-46.

Important:

- For security reasons, it is highly advisable for access to be denied to the `globals.jsa` file. This is already the case by default.
- 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 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 a 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 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 will automatically be loaded by the JSP classloader without the need for specific configuration.

Note: 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.

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 for each 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 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 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.

A `globals.jsa` file supports 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 JSP container unloads an application.

For more information, see "The globals.jsa Event-Handlers" on page B-37.

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-tag and end-tag) 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 declared again 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 B-41.

For details about each of the four event handlers, see "The globals.jsa Event-Handlers" on page B-37.

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.

The application_OnStart Event Handler

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 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)—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 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 B-46. 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>
```

The application_OnEnd Event Handler

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 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)—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 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 B-46. 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>
```

The session_OnStart Event Handler

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 JSP container creates a new session in response to a JSP page request. This occurs for each client, 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 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>
```

The session_OnEnd Event Handler

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 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)—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 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
- The globals.jsa Structure
- 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 that is declared in a `globals.jsa` file (such as by a `jsp:useBean` tag) need not be declared again 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 the standard `jsp:useBean` tag or the JML `useVariable` tag. Globally declared objects must be of either `session` or `application` scope (not `page` or `request` scope).

Nested tags are supported. Thus, a `jsp:setProperty` tag can be nested in a `jsp:useBean` tag. (A translation error occurs if `jsp:setProperty` is used outside a `jsp:useBean` tag.)

The globals.jsa Structure

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 B-52.

```
<%-- 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 --%>
```

Migration from globals.jsa

The OC4J JSP front-end servlet in Oracle9*i*AS release 2 no longer supports `globals.jsa`. If an existing application uses `globals.jsa`, you should migrate away from this usage. The following substitutions for `globals.jsa` functionality are recommended:

- Instead of using `globals.jsa` as an application marker, use standard WAR packaging to denote the application structure.
- Instead of using `globals.jsa` start-session, end-session, start-application, and end-application events, use standard servlet 2.3 listener functionality. For example, equivalent capabilities are offered through the standard `javax.servlet.ServletContextListener` and `javax.servlet.http.HttpSessionListener` interfaces.
- Instead of using `globals.jsa` for global variable declarations, make the declarations in a single source file and use "global include" functionality of the OC4J JSP engine. See "Oracle JSP Global Includes" on page 7-9.

If you cannot migrate your code immediately, an application that uses `globals.jsa` can still run in OC4J if you use the previous `oracle.jsp.JspServlet` front-end servlet instead of the `oracle.jsp.runtimev2.JspServlet` front-end. You can specify this in the `<servlet>` element in the application `web.xml` file, which overrides definitions in the OC4J `global-web-application.xml` file. This should be for short-term use only, however, given that the `runtimev2` front-end servlet has improved features, supports additional configuration parameters, and offers improved performance.

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 "JSP Application and Session Support for JServ" on page B-32.

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, however. In this case, `globals.jsa` cannot function as an application marker. But 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 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, users are identified by their IP addresses.

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).

Note: This sample uses the `setAttribute()` method specified in the `ServletContext` interface in servlet 2.1 or higher environments. To use this feature in JServ, include the servlet 2.2 or servlet 2.3 JAR file—`[Oracle_Home]/lib/servlet.jar`—in your classpath.

The 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();
        pageContext.setAttribute("today", today, PageContext.APPLICATION_SCOPE);
        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();
                        pageContext.setAttribute(
                            "cachedNumbers", cachedNumbers, PageContext.APPLICATION_SCOPE);
                }
                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>
```

The 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 it's 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;
                pageContext.setAttribute("today", today, PageContext.APPLICATION_SCOPE);
        }

        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.

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, because 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*.

The 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>
```

The 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 B-50, 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 must provide `jsp:useBean` tags to locally declare the beans they access.)

Declaring the beans globally results in implicit declaration in all event handlers and the JSP page.

The 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>
```

The 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>
```

C

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

Under the terms of the Apache license, Oracle is required to provide the following notices. However, the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the Apache software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Apache.

The Apache Software License

```
/* ====================================================================
 * The Apache Software License, Version 1.1
 *
 * Copyright (c) 2000 The Apache Software Foundation.  All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. The end-user documentation included with the redistribution,
 *    if any, must include the following acknowledgment:
 *       "This product includes software developed by the
 *        Apache Software Foundation (http://www.apache.org/)."
 *    Alternately, this acknowledgment may appear in the software itself,
 *    if and wherever such third-party acknowledgments normally appear.
 *
 * 4. The names "Apache" and "Apache Software Foundation" must
 *    not be used to endorse or promote products derived from this
 *    software without prior written permission. For written
 *    permission, please contact apache@apache.org.
 *
 * 5. Products derived from this software may not be called "Apache",
 *    nor may "Apache" appear in their name, without prior written
```

```
*    permission of the Apache Software Foundation.
*
* THIS SOFTWARE IS PROVIDED ‘‘AS IS’’ AND ANY EXPRESSED OR IMPLIED
* WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
* OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
* DISCLAIMED.  IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
* LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
* USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
* ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
* OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
* OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
* ====================================================================
*
* This software consists of voluntary contributions made by many
* individuals on behalf of the Apache Software Foundation.  For more
* information on the Apache Software Foundation, please see
* <http://www.apache.org/>.
*
* Portions of this software are based upon public domain software
* originally written at the National Center for Supercomputing Applications,
* University of Illinois, Urbana-Champaign.
*/
```

Apache JServ

Under the terms of the Apache license, Oracle is required to provide the following notices. However, the Oracle program license that accompanied this product determines your right to use the Oracle program, including the Apache software, and the terms contained in the following notices do not change those rights. Notwithstanding anything to the contrary in the Oracle program license, the Apache software is provided by Oracle "AS IS" and without warranty or support of any kind from Oracle or Apache.

Apache JServ Public License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistribution of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistribution in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- All advertising materials mentioning features or use of this software must display the following acknowledgment:

  This product includes software developed by the Java Apache Project for use in the Apache JServ servlet engine project (http://java.apache.org/).

- The names "Apache JServ", "Apache JServ Servlet Engine" and "Java Apache Project" must not be used to endorse or promote products derived from this software without prior written permission.
- Products derived from this software may not be called "Apache JServ" nor may "Apache" nor "Apache JServ" appear in their names without prior written permission of the Java Apache Project.
- Redistribution of any form whatsoever must retain the following acknowledgment:

  This product includes software developed by the Java Apache Project for use in the Apache JServ servlet engine project (http://java.apache.org/).

THIS SOFTWARE IS PROVIDED BY THE JAVA APACHE PROJECT "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE JAVA

APACHE PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X